🚀 AI 성능 엔지니어링

AI Systems Performance Engineering

GPU 최적화, 분산 학습, 추론 스케일링, 풀스택 성능 튜닝

— O'Reilly 도서 한국어 문서 모음 —

저자: Chris Fregly | 출판: O'Reilly Media, November 2025
한국어 번역: Claude Code (Anthropic)

목차

AI 성능 엔지니어링

업데이트: *이 내용을 직접 실습하는 강좌에 관심이 있으신가요?*

*관심이 있으시다면, 이 **양식**을 작성하여 관심을 표명하고 알림을 받으세요.*

이 저장소에 대하여

O'Reilly 도서를 위한 AI 시스템 성능 엔지니어링 코드, 도구 및 자료입니다. GPU 최적화, 분산 학습, 추론 스케일링, 그리고 현대 AI 워크로드를 위한 풀스택 성능 튜닝을 다룹니다.

채팅으로 이 책과 직접 대화하세요!

바로 **코드**로 이동하기.

> O'Reilly 도서 - 2025년 11월 Amazon에서 구매하기

AI 시스템 성능 엔지니어링 도서

현대 AI 시스템은 단순한 FLOP 성능 이상을 요구합니다. 하드웨어, 소프트웨어, 알고리즘 전반에 걸쳐 굿풋(goodput) 기반의 프로파일 우선 엔지니어링이 필요합니다. 이 실습 중심 가이드는 GPU, 인터커넥트, 런타임 스택을 효율적이고 신뢰할 수 있는 학습 및 추론 파이프라인으로 전환하는 방법을 보여줍니다.

Nsight와 PyTorch 프로파일러로 실제 병목을 진단하고, 대역폭과 메모리를 최대한 활용하며, 컴파일러 스택(PyTorch + OpenAI Triton)을 사용하여 고성능 커널을 만드는 방법을 배웁니다. 서빙 측면에서는 vLLM/SGLang, TensorRT-LLM, NVIDIA Dynamo를 활용한 고처리량 추론을 마스터하며, 분리된 프리필/디코드 및 페이지드 KV 캐시를 포함하여 예산 내에서 랙 전체로 확장하는 방법을 익힙니다.

실습 위주의 경험적 방법론과 케이스 스터디, 프로파일링 데이터를 활용하여, AI/ML 엔지니어, 시스템 엔지니어, 연구자, 그리고 대규모 학습/추론을 구축하거나 운영하는 플랫폼 팀에게 유용한 책입니다. 현대 NVIDIA GPU를 위한 수천 줄의 PyTorch 및 CUDA C++ 코드 예제가 포함되어 있습니다.

- 굿풋을 위해 프로파일링하기 - 단순 사용률이 아닌, Nsight Systems/Compute와 PyTorch 프로파일러를 사용하여 실제 병목 지점을 찾습니다.
- 메모리와 대역폭 최대화 - 레이아웃, 캐싱, 데이터 이동을 최적화하여 GPU를 지속적으로 바쁘게 유지합니다.
- 컴파일러로 튜닝하기 - PyTorch 컴파일러 스택과 Triton을 활용하여 C++ 보일러플레이트 없이 고성능 커널을 생성합니다.
- 안정적으로 학습 확장하기 - 병렬화 전략(DP, FSDP, TP, PP, CP, MoE)을 적용하고 계산/통신을 오버랩하여 버블을 최소화합니다.
- 조 단위 파라미터 모델 효율적으로 서빙하기 - vLLM, SGLang, TensorRT-LLM, NVIDIA Dynamo를 분리된 프리필/디코드 및 KV 캐시 이동과 함께 활용합니다.
- 토큰당 비용 절감 - 단순 최대 속도가 아닌 성능/전력, 달러당 처리량을 엔지니어링합니다.
- AI 지원 최적화 채택 - AI가 시스템의 수동 조정 한계를 넘어설 때 커널 합성 및 튜닝을 돕도록 합니다.
- 확신을 가지고 출시하기 - 200개 이상의 항목으로 구성된 체크리스트를 적용하여 팀 전반에서 성과를 재현하고 회귀를 방지합니다.

저자 소개

Chris Fregly는 Netflix, Databricks, Amazon Web Services(AWS)에서 혁신을 이끌어 온 성능 엔지니어이자 AI 제품 리더입니다. 그는 AI/ML 제품 구축, 시장 진출 이니셔티브 확장, 대규모 생성형 AI 및 분석 워크로드의 비용 절감에 집중한 성능 중심 엔지니어링 팀을 이끌었습니다.

Chris는 두 권의 다른 O'Reilly 도서의 저자이기도 합니다: Data Science on AWS와 Generative AI on AWS. 또한 O'Reilly 강좌 "NVIDIA GPU로 프로덕션에서 고성능 AI"와 Andrew Ng과 함께하는 DeepLearning.ai 강좌 "대규모 언어 모델로 생성형 AI"의 제작자이기도 합니다.

그의 작업은 커널 수준 튜닝, 컴파일러 기반 가속, 분산 학습, 고처리량 추론에 걸쳐 있습니다. Chris는 AI 성능 엔지니어링이라는 월간 밋업을 주최합니다.

200개 이상의 항목으로 구성된 성능 체크리스트

이 도서는 전체 수명주기를 다루는 현장 검증된 최적화를 담은 **200개 이상의 성능 체크리스트**와 함께 제공됩니다. 즉시 적용할 수 있습니다:

- ✅ 성능 튜닝 마인드셋 및 비용 최적화
- ✅ 재현성 및 문서화 모범 사례
- ✅ 시스템 아키텍처 및 하드웨어 계획
- ✅ 운영 체제 및 드라이버 최적화
- ✅ GPU 프로그래밍 및 CUDA 튜닝
- ✅ 분산 학습 및 네트워크 최적화
- ✅ 효율적인 추론 및 서빙
- ✅ 전력 및 열 관리
- ✅ 최신 프로파일링 도구 및 기법
- ✅ 아키텍처별 최적화

링크

- **도서**: Amazon에서 AI 시스템 성능 엔지니어링
- **밋업**: AI 성능 엔지니어링
- **YouTube**: AI 성능 엔지니어링 채널

> *AI 성능 엔지니어링 커뮤니티를 위해 샌프란시스코에서 제작*

주요 집중 영역

- **GPU 아키텍처, PyTorch, CUDA, OpenAI Triton 프로그래밍**
- **분산 학습 및 추론**
- **메모리 최적화 및 프로파일링**
- **PyTorch 성능 튜닝**
- **다중 노드 확장 전략**

도서 챕터

챕터 1: 소개 및 AI 시스템 개요

- AI 시스템 성능 엔지니어
- 벤치마킹 및 프로파일링
- 분산 학습 및 추론 확장
- 리소스의 효율적인 관리

- 팀 간 협업
- 투명성과 재현성

챕터 2: AI 시스템 하드웨어 개요

- CPU와 GPU "슈퍼칩"
- NVIDIA Grace CPU 및 Blackwell GPU
- NVIDIA GPU 텐서 코어와 트랜스포머 엔진
- 스트리밍 멀티프로세서, 스레드, 워프
- 초대규모 네트워킹
- NVLink와 NVSwitch
- 다중 GPU 프로그래밍

챕터 3: OS, Docker, Kubernetes 튜닝

- 운영 체제 구성
- GPU 드라이버 및 소프트웨어 스택
- NUMA 인식 및 CPU 피닝
- 컨테이너 런타임 최적화
- 토폴로지 인식 오케스트레이션을 위한 Kubernetes
- 메모리 격리 및 리소스 관리

챕터 4: 분산 네트워크 통신 튜닝

- 통신과 계산 오버랩
- 분산 다중 GPU 통신을 위한 NCCL
- NCCL의 토폴로지 인식
- 분산 데이터 병렬 전략
- NVIDIA 추론 전송 라이브러리 (NIXL)
- In-Network SHARP 집계

챕터 5: GPU 기반 스토리지 I/O 최적화

- 빠른 스토리지와 데이터 지역성
- NVIDIA GPUDirect 스토리지
- 분산 병렬 파일 시스템
- NVIDIA DALI를 활용한 멀티모달 데이터 처리
- 고품질 LLM 데이터셋 생성

챕터 6: GPU 아키텍처, CUDA 프로그래밍, 점유율 극대화

- GPU 아키텍처 이해
- 스레드, 워프, 블록, 그리드
- CUDA 프로그래밍 리뷰
- GPU 메모리 계층 구조 이해
- 높은 점유율과 GPU 활용률 유지
- 루프라인 모델 분석

챕터 7: GPU 메모리 접근 패턴 프로파일링 및 튜닝

- 병합된 vs 비병합 글로벌 메모리 접근
- 벡터화된 메모리 접근
- 공유 메모리를 이용한 타일링 및 데이터 재사용
- 워프 셔플 인트린식
- 비동기 메모리 프리페칭

챕터 8: 점유율 튜닝, 워프 효율성, 명령어 수준 병렬성

- GPU 병목 프로파일링 및 진단
- Nsight Systems 및 Compute 분석
- 점유율 튜닝
- 워프 실행 효율성 향상
- 명령어 수준 병렬성 노출

챕터 9: CUDA 커널 효율성 및 산술 강도 향상

- 다단계 마이크로 타일링
- 커널 퓨전
- 혼합 정밀도 및 텐서 코어
- 최적 성능을 위한 CUTLASS 활용
- 인라인 PTX 및 SASS 튜닝

챕터 10: 커널 내부 파이프라이닝 및 협력 스레드 블록 클러스터

- 커널 내부 파이프라이닝 기법
- 워프 특화 생산자-소비자 모델
- 영구 커널 및 메가커널
- 스레드 블록 클러스터 및 분산 공유 메모리
- 협력 그룹

챕터 11: 커널 간 파이프라이닝 및 CUDA 스트림

- 스트림을 사용한 계산과 데이터 전송 오버랩
- 스트림 순서 메모리 할당자
- 이벤트를 통한 세밀한 동기화
- CUDA 그래프를 통한 제로 오버헤드 실행

챕터 12: 동적 및 디바이스 측 커널 오케스트레이션

- 원자적 작업 큐를 통한 동적 스케줄링
- CUDA 그래프로 반복 커널 실행 일괄 처리
- 동적 병렬성
- NVSHMEM을 통한 다중 GPU 오케스트레이션

챕터 13: PyTorch 프로파일링, 튜닝, 확장

- NVTX 마커 및 프로파일링 도구
- PyTorch 컴파일러 (torch.compile)
- PyTorch에서 메모리 프로파일링 및 튜닝
- PyTorch Distributed로 확장
- HTA를 활용한 다중 GPU 프로파일링

챕터 14: PyTorch 컴파일러, XLA, OpenAI Triton 백엔드

- PyTorch 컴파일러 심층 분석
- OpenAI Triton으로 커스텀 커널 작성
- PyTorch XLA 백엔드
- 고급 Triton 커널 구현

챕터 15: 다중 노드 추론 병렬성 및 라우팅

- 분리된 프리필 및 디코드 아키텍처
- MoE 모델을 위한 병렬성 전략
- 추측 및 병렬 디코딩 기법
- 동적 라우팅 전략

챕터 16: 대규모 추론 프로파일링, 디버깅, 튜닝

- 성능 프로파일링 및 튜닝 워크플로우
- 동적 요청 배칭 및 스케줄링
- 시스템 수준 최적화
- 실시간 추론을 위한 양자화 접근법
- 애플리케이션 수준 최적화

챕터 17: 분리된 프리필 및 디코드 확장

- 프리필-디코드 분리의 이점
- 프리필 워커 설계
- 디코드 워커 설계
- 분리된 라우팅 및 스케줄링 정책
- 확장성 고려 사항

챕터 18: 고급 프리필-디코드 및 KV 캐시 튜닝

- 최적화된 디코드 커널 (FlashMLA, ThunderMLA, FlexDecoding)
- KV 캐시 활용 및 관리 튜닝
- 이기종 하드웨어 및 병렬성 전략
- SLO 인식 요청 관리

챕터 19: 동적 및 적응형 추론 엔진 최적화

- 적응형 병렬성 전략
- 동적 정밀도 변경
- 커널 자동 튜닝
- 런타임 튜닝을 위한 강화학습 에이전트
- 적응형 배칭 및 스케줄링

챕터 20: AI 지원 성능 최적화

- AlphaTensor AI 발견 알고리즘
- 자동화된 GPU 커널 최적화
- 자기 개선 AI 에이전트
- 수백만 GPU 클러스터를 향한 확장

커뮤니티 리소스

20개 이상의 도시에서 10만 명 이상의 회원을 보유한 월간 밋업:

- YouTube 채널
- 밋업 그룹

최근 세션:

- 동적 적응형 RL 추론 CUDA 커널 튜닝
- 고성능 에이전틱 AI 추론 시스템
- PyTorch 모델 최적화

월간 밋업 요약

- **2026년 2월 16일** – YouTube 공개 예정 & 슬라이드: Verda의 Riccardo Mereu가 발표하는 NVFP4 저정밀 수치
- **2026년 1월 19일** – YouTube & 슬라이드: Chaim Rand의 NVIDIA Nsight Systems 프로파일러를 통한 데이터 전송 최적화
- **2025년 11월 17일** – YouTube & 슬라이드: Abdul Dakkak의 NVIDIA/AMD GPU 및 Modular 플랫폼을 이용한 광속 추론
- **2025년 10월 20일** – YouTube: AI 기반 GPU 커널 최적화 + nbdistributed를 활용한 분산 PyTorch
- **2025년 9월 15일** – YouTube: 동적 적응형 RL 추론 커널 튜닝 심층 분석
- **2025년 8월 18일** – YouTube: 다중 GPU 오케스트레이션 전략 및 Nsight 프로파일링 케이스 스터디
- **2025년 7월 21일** – YouTube: FlashMLA, ThunderMLA, FlexDecoding 커널 워크스루와 라이브 Nsight Compute 데모
- **2025년 6월 16일** – 슬라이드: 분리된 추론 라우팅을 다루는 고성능 에이전틱 AI 추론 시스템
- **2025년 5월 19일** – YouTube & PyTorch 데이터 로더 최적화: Torch.compile 파이프라인, 데이터 로더 처리량 튜닝, 크로스 아키텍처 CUDA/ROCm 커널
- **2025년 4월 21일** – YouTube & AI 성능 엔지니어링 밋업 슬라이드: 종합적인 GPU 성능 플레이북 및 PyTorch 모델 최적화 워크샵

기여하기

기여를 환영합니다! 코드, 문서, 성능 개선에 대한 가이드라인은 `CONTRIBUTING.md` 를 참조하세요.

라이선스

챕터 01

챕터 1 – 성능 기초

요약

간단한 학습 루프 굿풋 벤치마크와 소규모 CUDA GEMM 케이스 스터디를 통해 벤치마킹 기초를 확립합니다. 이후의 최적화를 반복 가능한 측정, 동등한 워크로드, 검증 가능한 출력에 기반하게 하는 것이 목표입니다.

학습 목표

- 공유 하네스로 최소한의 PyTorch 학습 루프를 프로파일링하고 처리량 대 지연 시간을 분석합니다.
- 알고리즘 워크로드를 변경하지 않고 기본 최적화(FP16 + 퓨전 마이크로배치)를 적용합니다.
- 배치형 vs 스트라이드형 GEMM 커널을 비교하여 산술 강도를 이해합니다.

디렉토리 구조

경로	설명
`baseline_performance.py`, `optimized_performance.py`	FP32 이거(eager) vs FP16 + 퓨전 마이크로배치(배치 퓨전)를 비교하는 굿풋 중심 학습 루프 쌍.
`baseline_gemm.cu`, `optimized_gemm_batched.cu`, `optimized_gemm_strided.cu`	실행 분할 상각 및 메모리 병합을 설명하는 CUDA GEMM 변형(단일, 배치, 스트라이드).
`compare.py`, `workload_config.py`, `arch_config.py`, `expectations_{hardware_key}.json`	하네스 진입점, 워크로드 형태, 아키텍처 오버라이드, 저장된 기대값 임계치.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch01/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch01
python -m cli.aisp bench run --targets ch01 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python compare.py`가 기본 마이크로배치 크기에서 optimized_performance가 baseline 대비 토큰/초)=2배를 달성했다고 보고합니다.
- `make && ./baseline_gemm_sm100` vs `./optimized_gemm_batched_sm100` 실행이 실행 횟수와 총 런타임에서 상당한 감소를 보입니다.

참고 사항

- `requirements.txt`는 헬퍼 스크립트에서 사용하는 경량 추가 의존성(Typer, tabulate)을 고정합니다.
- `Makefile`은 빠른 비교를 위해 SM별 접미사를 붙인 CUDA GEMM 바이너리를 빌드합니다.

챕터 02

챕터 2 – GPU 하드웨어 아키텍처

요약

Blackwell 시대 시스템을 위한 아키텍처 인식 도구를 제공합니다. SM 및 메모리 사양 쿼리, NVLink 처리량 검증, CPU-GPU 일관성 실험 등을 통해 최적화가 측정된 하드웨어 한계에 기반하게 합니다.

학습 목표

- 성능 연구 전에 GPU, CPU, 패브릭 기능을 쿼리하고 기록합니다.
- 전용 마이크로벤치마크를 사용하여 NVLink, PCIe, 메모리 대역폭 상한을 측정합니다.
- 제로 카피 버퍼가 도움이 되는지 해가 되는지 알기 위해 Grace-Blackwell 일관성 경로를 검증합니다.
- 아키텍처별 튜닝 레버를 강조하기 위해 베이스라인 vs 최적화된 cuBLAS 호출을 비교합니다.

디렉토리 구조

경로	설명
`hardware_info.py` , `cpu_gpu_topology_aware.py`	GPU 기능, NUMA 레이아웃, NVLink/NVSwitch 연결, 친화성 힌트를 기록하는 시스템 스캐너.
`nvlink_c2c_bandwidth_benchmark.py` , `baseline_memory_transfer.py` , `optimized_memory_transfer.py` , `memory_transfer_pcie_demo.cu` , `memory_transfer_nvlink_demo.cu` , `memory_transfer_zero_copy_demo.cu` , `baseline_memory_transfer_multigpu.cu` , `optimized_memory_transfer_multigpu.cu`	NVLink, PCIe, 일관성 메모리 성능을 정량화하는 피어-투-피어 및 제로 카피 실험.
`cpu_gpu_grace_blackwell_coherency.cu` , `cpu_gpu_grace_blackwell_coherency_sm121`	명시적 전송 vs 공유 매핑을 비교하는 Grace-Blackwell 캐시 일관성 샘플.
`baseline_cublas.py` , `optimized_cublas.py`	TF32, 텐서 연산 수학, 스트림 친화성을 토글하여 아키텍처 노브를 강조하는 cuBLAS GEMM 벤치마크 쌍.
`compare.py` , `Makefile` , `expectations_{hardware_key}.json`	하네스 드라이버, CUDA 빌드 규칙, 자동화된 합격/불합격 검사를 위한 기대값 파일.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch02/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch02
python -m cli.aisp bench run --targets ch02 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations` 를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json` 으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations` 로 갱신하세요.

검증 체크리스트

- `python hardware_info.py` 가 시스템의 모든 GPU에 대해 정확한 디바이스 이름, SM 수, HBM 크기를 기록합니다.
- `python nvlink_c2c_bandwidth_benchmark.py --gpus 0 1` 이 NVLink 연결 쌍에서 단방향 ~250 GB/s를 유지합니다. 불일치는 토폴로지 또는 드라이버 문제를 나타냅니다.
- 일관성 샘플을 실행하면 제로 카피가 소형 전송(수 MB 미만)에 유리하고, 대형 전송은 명시적 H2D 복사를 선호한다는 것이 문서화된 임계치와 일치함을 보여줍니다.

참고 사항

- Grace 전용 일관성 테스트는 GB200/GB300 노드가 필요합니다. PCIe 전용 호스트에서는 바이너리가 no-op입니다.
- `Makefile` 은 CUDA 및 CPU 도구를 모두 빌드하므로 챕터를 벗어나지 않고 결과를 비교할 수 있습니다.

챕터 03

챕터 3 – 시스템 튜닝

요약

커널 수준 최적화 전에 GPU 워크로드를 지속적으로 공급하는 호스트 수준 변경 사항인 NUMA 피닝, 거버너 조정, 컨테이너 설정, Kubernetes 매니페스트를 다룹니다.

학습 목표

- GPU 파이프라인을 스로틀하는 CPU 및 메모리 친화성 문제를 진단합니다.
- 공유 클러스터에서 지속적인 GPU 처리량을 위해 Docker 및 Kubernetes 환경을 강화합니다.
- 실험실 머신이 일관성을 유지하도록 셀 스크립트를 통해 반복 가능한 시스템 튜닝을 자동화합니다.
- 호스트 수준 수정이 GEMM 처리량을 높이고 실행 지연 시간을 줄이는 방법을 정량화합니다.

디렉토리 구조

경로	설명
`baseline_numa_unaware.py`, `optimized_numa_unaware.py`, `bind_numa_affinity.py`, `numa_topology_script.sh`	데이터 로더, NCCL 랭크, GPU 컨텍스트를 올바른 CPU 소켓에 바인딩하기 위한 NUMA 진단 및 피닝 헬퍼.
`baseline_docker.py`, `optimized_docker.py`, `docker_gpu_optimized.dockerfile`, `system_tuning.sh`, `gpu_setup_commands.sh`	영속 모드, 대용량 페이지, IRQ 스티어링, MIG 가시성을 토글하는 컨테이너 구성 및 호스트 설정 스크립트.
`baseline_kubernetes.py`, `optimized_kubernetes.py`, `kubernetes_mig_pod.yaml`, `kubernetes_topology_pod.yaml`	다중 테넌트 플릿을 위한 토폴로지 인식 스케줄링 및 MIG 파티셔닝을 보여주는 Kubernetes 매니페스트.
`cpu_gpu_numa_optimizations.sh`, `system_tuning.sh`, `gpu_setup_commands.sh`	CPU 거버너, cgroup 제한, 영속 모드, 드라이버 설정을 벤치마크 하네스와 정렬하는 워크플로우 스크립트.
`baseline_gemm.py`, `optimized_gemm.py`, `train.py`	시스템 튜닝 변경의 영향을 측정 가능한 FLOP/s로 표면화하는 간단한 GEMM + 학습 루프.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`	하네스 진입점, Python 의존성, 회귀 임계치.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch03/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch03
python -m cli.aisp bench run --targets ch03 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `bind_numa_affinity.py` 전후에 `python baseline_numa_unaware.py --diagnostics`를 실행하여 교차 소켓 메모리 트래픽이 거의 제로로 떨어지는지 확인합니다.
- `python optimized_docker.py --image docker_gpu_optimized.dockerfile`이 GPU 클록이 고정된 상태에서 호스트 실행과 동일한 처리량을 유지해야 합니다.
- `python compare.py --examples gemm`이 `system_tuning.sh` 적용 후 optimized_gemm이 측정된 호스트 피크와 일치함을 보여줍니다.

참고 사항

- `cpu_gpu_numa_optimizations.sh`는 재부팅 후 안전하게 재실행할 수 있습니다. irqbalance 피닝 및 거버너 설정을 재적용합니다.
- Kubernetes 매니페스트는 외부 저장소를 참조하지 않고 NVLink/NVSwitch 친화성에 필요한 어노테이션을 문서화합니다.

챕터 04

챕터 4 – 다중 GPU 분산

요약

NVLink/NVSwitch 패브릭 인식, NCCL 튜닝, NVSHMEM 집합 연산, 대칭 메모리 패턴을 사용하여 여러 Blackwell GPU에 걸쳐 학습 및 추론을 확장하는 방법을 시연합니다.

학습 목표

- 오버랩 유무에 따른 데이터 병렬 및 텐서 병렬 학습 루프를 벤치마크합니다.
- 로컬 및 분리된 GPU를 혼합할 때 NVLink 대역폭과 토폴로지 효과를 정량화합니다.
- GPU 동기화에서 호스트 개입을 줄이기 위해 NVSHMEM 파이프라인을 실험합니다.
- KV 캐시 복제 및 옵티마이저 상태 샤딩을 단순화하기 위해 대칭 메모리 풀을 채택합니다.

디렉토리 구조

경로	설명
`baseline_dataparallel.py`, `optimized_dataparallel.py`	단일 GPU DataParallel 안티 패턴 vs 직접 GPU 실행.
`baseline_dataparallel_multigpu.py`, `optimized_dataparallel_multigpu.py`	다중 GPU DataParallel vs 사전 스테이징된 샤드를 사용한 수동 그래디언트 감소.
`baseline_no_overlap.py`, `optimized_no_overlap.py`	allreduce 지연 시간을 숨기기 위해 계산/통신 동시성을 스테이징하고 마이크로배치를 파이프라인 처리하는 오버랩 연구.
`baseline_nvlink.py`, `optimized_nvlink.py` 외 여러 NVLink 관련 파일	피어 대역폭과 토폴로지 효과(단일 및 다중 GPU)를 검증하는 NVLink 실습.
`baseline_continuous_batching.py`, `optimized_continuous_batching.py` 외 여러 배칭/분리 파일	풀링 및 원격 KV 재사용을 보여주는 연속 배칭 + 분리 추론 데모.
`baseline_gradient_compression_fp16.py`, `optimized_gradient_compression_fp16.py` 외 여러 그래디언트 압축 파일	소형 버킷 vs 전체 버퍼 압축을 비교하는 그래디언트 압축 all-reduce 벤치마크(단일 GPU 및 다중 GPU FP16/INT8 경로).
`baseline_pipeline_parallel.py`, `optimized_pipeline_parallel_1f1b.py` 외 여러 파이프라인/텐서 병렬 파일	파이프라인/텐서 병렬 및 torchcomms 오버랩 연구(단일 및 다중 GPU).
`baseline_nvshmem_pipeline_parallel_multigpu.py`, `optimized_nvshmem_pipeline_parallel_multigpu.py` 외	디바이스 주도 동기화의 이점을 강조하는 NVSHMEM 파이프라인 및 학습 샘플.
`baseline_symmetric_memory_perf.py`, `optimized_symmetric_memory_perf.py` 외	KV 캐시 및 옵티마이저 샤드를 위한 대칭 메모리 유틸리티 및 성능 프로브.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`, `bandwidth_benchmark_suite_multigpu.py`, `nccl_benchmark.py`	하네스 드라이버 및 토폴로지 bring-up을 위한 독립형 NCCL/NVLink 스위퍼.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch04/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch04
python -m cli.aisp bench run --targets ch04 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python compare.py --examples dataparallel_multigpu`가 최적화된 쌍이 더 낮은 지연 시간으로 계산과 통신을 오버랩하는 것을 보여줍니다.
- `python bandwidth_benchmark_suite_multigpu.py --profile minimal`이 연결된 GPU 쌍에서 ≥250 GB/s 링크를 표면화하고 느린 홉을 강조합니다.
- NVSHMEM 샘플이 `NVSHMEM_SYMMETRIC_SIZE`가 워크로드를 수용할 크기로 설정되면 일관된 출력을 생성합니다. 잘못된 구성은 명확한 오류를 발생시킵니다.

참고 사항

- `symmetric_memory_*` 헬퍼는 NVSwitch 패널티 없이 GPU 간 KV 캐시 라인을 풀링하기 위한 사용자 공간 할당자를 보유합니다.
- 다중 노드 테스트를 시작하기 전에 `nccl_blackwell_config.py`를 사용하여 NCCL 환경 변수(최소 NRings, IB 매핑)를 초기화하세요.

챕터 05

챕터 5 – 스토리지 및 I/O 최적화

요약

GPU를 효율적으로 공급하는 것에 초점을 맞춥니다. DataLoader 워커 튜닝, 전처리 벡터화, 계산과 I/O 오버랩, NVMe 트래픽이 병목이 될 때 GPUDirect 스토리지 채택을 다룹니다.

학습 목표

- 하네스 메트릭을 통해 I/O 지연을 감지하고 GPU를 바쁘게 유지하도록 파이프라인을 재구성합니다.
- 대규모 배치 학습을 위해 PyTorch DataLoader 파라미터(워커, 프리페치, 핀 메모리)를 튜닝합니다.
- GPUDirect 스토리지 경로 vs 전통적인 CPU 중재 읽기를 평가합니다.
- 원격 스토리지 및 분산 데이터 읽기 전략을 벤치마크합니다.

디렉토리 구조

경로	설명
`baseline_storage_cpu.py`, `optimized_storage_cpu.py`	워커 수, 핀 메모리, 캐싱 전략을 다루는 단일 노드 데이터 로더 비교.
`baseline_vectorization.py`, `optimized_vectorization.py`	전처리에서 Python 루프를 제거하는 벡터화된 파싱 및 메모리 맵 예제.
`baseline_ai.py`, `optimized_ai.py`, `storage_io_optimization.py`	스트리밍 읽기 및 프리페치와 함께 계산을 오버랩하는 LLM 스타일 토큰 파이프라인.
`baseline_distributed.py`, `optimized_distributed.py`	단일 GPU 합산 vs 선택적 분산 all-reduce 폴백.
`baseline_distributed_multigpu.py`, `optimized_distributed_multigpu.py`	다중 GPU 감소 베이스라인(CPU 스테이징) vs GPU 측 reduce_add.
`gds_cufile_minimal.py`, `gpudirect_storage_example.py`	cuFile 설정, 버퍼 정렬, 처리량을 검증하는 GPUDirect 스토리지 샘플.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`	하네스 진입점 및 회귀를 감지하기 위한 기대값 기준선.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch05/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch05
python -m cli.aisp bench run --targets ch05 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python baseline_storage_cpu.py --inspect`가 CPU 대기 시간 〉 GPU 시간을 노출합니다. `optimized_storage_cpu.py`는 〉=80% GPU 활용률로 비율을 역전시킵니다.
- `python gds_cufile_minimal.py --bytes 1073741824`가 `/etc/cufile.json`이 구성되고 NVMe가 GPUDirect 지원을 알릴 때 멀티 GB/s 처리량을 유지합니다.
- `python compare.py --examples ai`가 optimized_ai가 크리티컬 패스에서 CPU 측 전처리를 제거함을 보여줍니다.

참고 사항

- `libcufile.so`를 사용할 수 없을 때 GPUDirect 스크립트가 호스트 중재 읽기로 폴백하여 개발 랩톱에서도 안전하게 실행할 수 있습니다.
- `requirements.txt`는 데이터셋 심에 필요한 제한된 추가 의존성(`lmdb` 등)을 캡처합니다.

챕터 06

챕터 6 – CUDA 프로그래밍 기초

요약

Python에서 CUDA C++로 이동합니다. 첫 번째 커널 작성, 점유율 분석, 메모리 레이아웃 제어, Blackwell 디바이스에서 ILP, 실행 경계, 통합 메모리 실험을 다룹니다.

학습 목표

- 하네스 워크로드를 미러링하는 커스텀 커널을 작성하고 실행합니다.
- 점유율, 실행 경계, 레지스터 압력이 상호 작용하는 방식을 이해합니다.
- ILP와 벡터화된 메모리 연산을 사용하여 스레드당 처리량을 높입니다.
- Blackwell GPU에서 통합 메모리 및 할당자 튜닝을 검증합니다.

디렉토리 구조

경로	설명
`my_first_kernel.cu`, `simple_kernel.cu`, `baseline_add_cuda.cu`, `optimized_add_cuda_parallel.cu` 외	CUDA 빌드 체인 및 실행 파라미터를 검증하는 Hello-world 커널 및 Python 래퍼.
`baseline_add_tensors_cuda.cu`, `optimized_add_tensors_cuda.cu` 외	자동 핀 메모리 스테이징 및 정확성 검사가 있는 텐서 지향 덧셈.
`baseline_attention_ilp.py`, `baseline_gemm_ilp.py`, `optimized_gemm_ilp.py`, `ilp_low_occupancy_vec4_demo.cu` 외	루프 언롤링, 레지스터, 벡터 너비를 조작하는 명령어 수준 병렬성(ILP) 연구.
`baseline_bank_conflicts.cu`, `optimized_bank_conflicts.cu`, `baseline_launch_bounds*.{py,cu}`, `optimized_launch_bounds*.{py,cu}`	공유 메모리 레이아웃과 CTA 크기를 강조하는 뱅크 충돌 및 실행 경계 실습.
`baseline_autotuning.py`, `optimized_autotuning.py`, `memory_pool_tuning.cu`, `stream_ordered_allocator/`	단편화 및 스트림 순서를 제어하는 자동 튜닝 하네스 및 할당자 실험.
`unified_memory.cu`, `occupancy_api.cu`, `baseline_quantization_ilp.py`, `optimized_quantization_ilp.py`	통합 메모리 데모, 점유율 계산기 샘플, 양자화 중심 ILP 워크로드.
`compare.py`, `Makefile`, `expectations_{hardware_key}.json`, `workload_config.py`	하네스 진입점, 빌드 스크립트, 기대값 기준선, 워크로드 설정.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch06/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch06
python -m cli.aisp bench run --targets ch06 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `nvcc -o baseline_add_cuda_sm121 baseline_add_cuda.cu` vs 최적화된 벡터화 버전이 Nsight Compute로 검사했을 때 명확한 대역폭 차이를 보입니다.
- `python optimized_autotuning.py --search`가 큐레이션된 프리셋과 동일한 스케줄로 수렴하고 `artifacts/` 아래에 점수 테이블을 기록합니다.
- `python compare.py --examples ilp`가 최적화된 ILP 커널이 동일한 출력으로 더 높은 명령어당 바이트를 달성함을 확인합니다.

참고 사항

- `arch_config.py`는 SM별 컴파일 플래그(예: 지원되지 않는 GPU에서 파이프라인 비활성화)를 강제하여 구형 하드웨어에서 우아하게 실패합니다.
- `cuda_extensions/`의 CUDA 확장은 인터랙티브 프로토타이핑을 위해 노트북으로 직접 가져올 수 있습니다.

챕터 07

챕터 7 – 메모리 접근 패턴

요약

메모리 레이아웃이 성능을 어떻게 결정하는지 가르칩니다. 병합된 복사, 타일드 행렬 곱셈, 비동기 프리페치, TMA 전송, 룩업 집약적인 워크로드를 위한 공유 메모리 스테이징을 다룹니다.

학습 목표

- 스칼라, 병합된, 벡터화된 메모리 이동 간의 성능 차이를 측정합니다.
- 공유 메모리 타일링, TMA, 비동기 복사를 사용하여 텐서 코어를 포화 상태로 유지합니다.
- 룩업 집약적인 워크로드를 분석하고 캐시 스래싱 접근 패턴을 완화합니다.
- 전치 및 gather/scatter 패널티를 정량화하여 레이아웃 변경을 정당화합니다.

디렉토리 구조

경로	설명
`baseline_copy_scalar.cu`, `baseline_copy_uncoalesced.cu`, `optimized_copy_uncoalesced_coalesced.cu`, `optimized_copy_scalar_vectorized.cu` 외	병합, 벡터 너비, 워프 수준 효율성을 강조하는 복사 커널.
`baseline_hbm_copy.cu`, `baseline_hbm_peak.cu`, `optimized_hbm_copy.cu`, `optimized_hbm_peak.cu` 외	CUDA 및 Python 하네스를 갖춘 HBM 최대 대역폭 프로브.
`baseline_async_prefetch.cu`, `optimized_async_prefetch.cu`, `baseline_tma_copy.cu`, `optimized_async_prefetch.py` 외	글로벌 메모리 페치와 계산을 오버랩하는 Async/TMA 샘플.
`baseline_matmul.cu`, `baseline_matmul.py`, `optimized_matmul_tiled.py`, `optimized_matmul_tiled.cu`	나이브한 글로벌 메모리 접근과 공유 메모리 타일링 및 워프 수준 재사용을 비교하는 행렬 곱셈 구현.
`baseline_lookup.cu`, `baseline_lookup.py`, `optimized_lookup.cu`, `lookup_pytorch.py`	테이블을 더 나은 지역성을 위해 재구성하는 방법을 보여주는 캐시 민감 룩업 워크로드.
`baseline_transpose.cu`, `baseline_transpose.py`, `optimized_transpose_padded.py` 외	뱅크 충돌을 최소화하는 방법을 보여주는 전치 및 gather/scatter 실험.
`compare.py`, `Makefile`, `expectations_{hardware_key}.json`, `memory_access_pytorch.py`	하네스 진입점, 빌드 레시피, 기대값 임계치, PyTorch 검증 스크립트.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch07/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch07
python -m cli.aisp bench run --targets ch07 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python baseline_hbm_copy.py --bytes 1073741824`가 `optimized_hbm_copy.py`보다 눈에 띄게 낮은 GB/s를 보고하여 벡터화 및 비동기 복사가 작동함을 증명합니다.
- `python compare.py --examples async_prefetch`가 optimized_async_prefetch가 정확도를 유지하면서 총 커널 수를 줄임을 보여줍니다.
- `optimized_matmul_tiled.cu`의 Nsight Compute 캡처가 최소한의 뱅크 충돌로 >80% 공유 메모리 대역폭 활용률을 달성합니다.

참고 사항

- Python 행렬 곱셈 래퍼를 사용할 때 `TORCH_COMPILE_MODE`를 토글하여 원시 CUDA 커널과 함께 퓨전 이점을 확인하세요.
- HBM 도구는 현실적인 참조 상한을 제공하기 위해 `benchmark_peak_results_*.json`이 있을 때 실제 최대값을 읽습니다.

챕터 08

챕터 8 – 점유율 및 파이프라인 튜닝

요약

리소스 균형에 집중합니다. 이중 버퍼링, 루프 언롤링, 비동기 파이프라인을 통해 TMEM 지연 시간을 숨기면서 SM을 가득 채우도록 블록 크기, 레지스터, 공유 메모리를 조정합니다.

학습 목표

- 점유율을 명시적으로 튜닝하고 레지스터 수가 상주 CTA를 어떻게 제한하는지 관찰합니다.
- 이중 버퍼링과 비동기 스테이징을 적용하여 DRAM 페치와 계산을 오버랩합니다.
- 타일링, 루프 언롤링, AI별 임계값을 사용하여 지연 시간과 처리량을 제어합니다.
- 공유 하네스를 사용하여 파이프라인된 스케줄이 SM/TMEM 활용률을 어떻게 변화시키는지 측정합니다.

디렉토리 구조

경로	설명
`baseline_occupancy_tuning.py`, `optimized_occupancy_tuning.py`, `occupancy_tuning_tool.py`, `occupancy_api_example.cu`, `occupancy_tuning.cu`	CTA 형태, 레지스터 캡, API 계산 한계를 튜닝하는 점유율 연구(빠른 프리셋 탐색을 위한 스윕 도구 포함).
`baseline_ai_optimization.py`, `optimized_ai_optimization.py`, `ai_optimization_kernels.cu`, `independent_ops.cu`	파이프라인 및 점유율 트레이드오프를 강조하기 위해 독립적인 ops를 스테이징하는 AI 커널 스케줄링 샘플.
`baseline_hbm_cuda.cu`, `baseline_hbm.py`, `optimized_hbm.py`, `optimized_hbm_cuda_vectorized.cu` 외	스칼라, 벡터화, 비동기 페치 패턴을 비교하는 HBM 스트리밍 워크로드.
`baseline_loop_unrolling.cu`, `baseline_loop_unrolling.py`, `optimized_loop_unrolling.cu`, `optimized_loop_unrolling.py`, `loop_unrolling_kernels.cu`	다양한 ILP 레짐을 목표로 하는 루프 언롤링 케이스 스터디.
`baseline_threshold.py`, `baseline_thresholdtma.py`, `optimized_threshold.py`, `optimized_thresholdtma.py`, `threshold_kernels.cu`	스칼라, 벡터화, TMA 기반 파이프라인으로 구현된 임계값 연산자.
`baseline_tiling.py`, `baseline_tiling_tcgen05.py`, `optimized_tiling.py`, `optimized_tiling_tcgen05.py`, `tiling_kernels.cu`	tcgen05 행렬 곱셈을 위한 타일 스케줄러(tcgen05가 없을 때의 안전한 폴백 포함).
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`	하네스 진입점, 의존성, 회귀 임계치.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch08/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch08
python -m cli.aisp bench run --targets ch08 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `optimized_thresholdtma.py`에 대한 Nsight Compute 트레이스가 최소한의 유휴 사이클로 TMA 로드가 오버랩되는 것을 보여야 합니다.
- `python -m cli.aisp tools occupancy-tuning`이 점유율 튜닝 마이크로벤치마크에 대한 프리셋 타이밍 + 속도 향상을 출력합니다.
- `python compare.py --examples threshold`가 TMA 기반 커널이 스칼라 참조 구현에 비해 지연 시간을 줄임을 확인합니다.

참고 사항

- `arch_config.py`는 GPU별로 tcgen05 낮추기를 활성화/비활성화하는 토글을 노출하여 동일한 스크립트가 SM100과 SM121에서 작동합니다.
- `build/`는 구성별 CUDA 오브젝트 파일을 캐시합니다. 도구 체인을 조정할 때 `python cleanup.py --include-build`로 정리하세요.

챕터 09

챕터 9 – 산술 강도 및 커널 퓨전

요약

루프라인을 따라 워크로드를 이동하는 방법을 탐구합니다. 타일링으로 산술 강도를 높이고, 메모리 바운드 커널을 퓨전하며, Blackwell 텐서 코어를 위해 구축된 CUTLASS/Triton/인라인 PTX 경로를 배포합니다.

학습 목표

- 계산 바운드 vs 메모리 바운드 동작을 분리하고 커널을 그에 맞게 조정합니다.
- 레지스터 압력과 데이터 재사용을 균형 있게 유지하는 마이크로 타일링 스케줄을 설계합니다.
- 커스텀 CUDA 폴백을 유지하면서 빠른 반복을 위해 CUTLASS와 Triton을 활용합니다.
- 중복 메모리 접근을 제거하기 위해 리덕션 집약적인 커널(예: norm + activation)을 퓨전합니다.

디렉토리 구조

경로	설명
`baseline_compute_bound.py`, `optimized_compute_bound.py`, `baseline_memory_bound.py`, `optimized_memory_bound.py`	계산 vs 대역폭 상한을 분리하고 튜닝 전략을 시연하는 참조 커널.
`baseline_micro_tiling_matmul.cu`, `optimized_micro_tiling_matmul.cu` 외	명시적 레지스터 블로킹 및 cp.async 프리페치가 있는 마이크로 타일링 행렬 곱셈.
`baseline_cutlass_gemm.cu`, `optimized_cutlass_gemm.cu` 외	수동 튜닝 커널과 벤더 라이브러리를 비교하기 위한 라이브러리 GEMM 기준선.
`baseline_cublaslt_gemm.cu`, `optimized_cublaslt_gemm.cu`, `tcgen05_pipelined.cu` 외	tcgen05 낮추기 및 점유율 튜닝을 보여주는 cuBLASLt 기반 행렬 곱셈 및 tcgen05 파이프라인 커널.
`baseline_cute_dsl_nvfp4_gemm.cu`, `optimized_cute_dsl_nvfp4_gemm.cu` 외	베이스라인 vs TMA 워프 특화 스케줄이 있는 CuTe-DSL 영감의 NVFP4 GEMM 쌍(대회 형태).
`baseline_cutlass_gemm_fp4.cu`, `optimized_cutlass_gemm_fp4.cu` 외	스케줄링을 분리하는 CUTLASS NVFP4 GEMM 쌍: 자동 스케줄링 vs 명시적 `KernelTmaWarpSpecialized1SmNvf4Sm100`.
`baseline_fused_l2norm.cu`, `optimized_fused_l2norm.cu` 외	L2 norm + 스케일링을 수치적으로 안정적으로 병합하는 퓨전 예제.
`baseline_triton.py`, `optimized_triton.py`	빠른 프로토타이핑 및 Blackwell에서 컴파일러 생성 PTX 검증을 위한 Triton 대응물.
`baseline_tcgen05_tma_pipeline.py`, `optimized_tcgen05_tma_pipeline.py`, `two_stage_pipeline.cu`	스테이징된 TMA 로드 및 인라인 PTX 훅을 강조하는 생산자/소비자 파이프라인.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`	모든 예제에 대한 하네스 훅 및 회귀 임계치.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch09/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch09
python -m cli.aisp bench run --targets ch09 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python baseline_compute_bound.py --summaries`가 `baseline_memory_bound.py`보다 훨씬 높은 산술 강도를 보고하여 루프라인 플롯과 일치합니다.
- `python optimized_cublaslt_gemm.py --sizes 4096 4096 8192`가 동일한 디바이스에서 `baseline_cublaslt_gemm.py`에 비해 처리량을 개선합니다.
- `python compare.py --examples fused_l2norm`이 퓨전 전후에 수치적으로 동일한 출력을 확인합니다.

참고 사항

- `inline_ptx_example.cu`는 아키텍처 가드를 통해 tcgen05 인트린식을 안전하게 래핑하는 방법을 시연합니다.
- `baseline_cute_dsl_nvfp4_gemm.cu` / `optimized_cute_dsl_nvfp4_gemm.cu`는 CuTe DSL NVFP4 커널 워크스루에서 영감을 받았습니다.
- `requirements.txt`는 커널이 PyTorch 2.10-dev 기능을 추적하도록 Triton 나이틀리 피닝을 포함합니다.

챕터 10

챕터 10 – 텐서 코어 파이프라인 및 클러스터 기능

요약

Blackwell에서 텐서 코어 친화적인 스케줄링을 적용합니다. 워프 특화, TMA 기반 파이프라인, 영구 커널, DSMEM 및 NVLink-C2C 인식이 있는 스레드 블록 클러스터를 다룹니다.

학습 목표

- 워프 특화와 cp.async/TMA를 사용하여 텐서 코어를 포화 상태로 유지합니다.
- 반복에 걸쳐 실행 오버헤드를 분할 상각하는 영구 행렬 곱셈을 프로토타입합니다.
- DSMEM 유무에 관계없이 스레드 블록 클러스터를 실습하여 하드웨어 한계를 이해합니다.
- PyTorch, Triton, CUDA 커널을 결합하면서 기대값을 동기화합니다.

디렉토리 구조

경로	설명
`baseline_attention.py`, `optimized_attention.py`, `baseline_flash_attention.py`, `optimized_flash_attention.py`, `analyze_scaling.py`	현대 디코더 모델을 위한 이거(eager), 퓨전, `torch.compile` 경로에 걸친 어텐션 워크로드.
`baseline_batch.py`, `optimized_batch.py`, `baseline_matmul.py`, `optimized_matmul.py`, `baseline_matmul_tcgen05.py`, `optimized_matmul_tcgen05.py`	tcgen05 낮추기, 레지스터 타일링, PyTorch 통합을 시연하는 텐서 코어 행렬 곱셈 변형.
`baseline_tcgen05_warp_specialization.py`, `optimized_tcgen05_warp_specialization.py`, `tcgen05_warp_specialized.cu`	전용 생산자/소비자 워프가 있는 워프 특화 tcgen05 GEMM.
`baseline_tcgen05_warp_specialization_cutlass.py`, `optimized_tcgen05_warp_specialization_cutlass.py` 외	CUTLASS 워프 특화 메인루프 비교(1-SM 워프 특화 vs 2-SM 워프 그룹 타일).
`warpgroup_specialization_demo.py`, `tcgen05_warpgroup_specialized.cu`	2-SM 타일을 사용하는 CUTLASS 워프그룹 배열 메인루프 데모.
`baseline_double_buffered_pipeline.{py,cu}`, `optimized_double_buffered_pipeline.{py,cu}` 외	cp.async, TMA, 수동 이중 버퍼링을 혼합하는 비동기 파이프라인 샘플.
`baseline_cluster_group*.{py,cu}`, `optimized_cluster_group*.{py,cu}` 외	DSMEM 활성화 및 DSMEM 없는 스레드 블록 클러스터를 다루는 클러스터 커널 제품군.
`baseline_cluster_multicast.py`, `optimized_cluster_multicast.py` 외	CUDA 바이너리 하네스 벤치마크로 래핑된 클러스터 멀티캐스트 GEMM 예제.
`baseline_cooperative_persistent.{py,cu}`, `optimized_cooperative_persistent.{py,cu}` 외	안정적인 처리량을 위해 협력 그룹과 TMA 스트림을 결합하는 영구 커널.
`baseline_flash_attn_tma_micro_pipeline.{py,cu}`, `optimized_flash_attn_tma_micro_pipeline.{py,cu}` 외	Triton, CUDA, 인라인 PTX를 혼합하는 마이크로 파이프라인 및 워프 특화 연구.
`compare.py`, `workload_config.py`, `demo_both_examples.sh`, `profile.sh`, `requirements_cufile.txt`	하네스 진입점, 워크로드 다이얼, 데모 러너, Nsight 자동화, 선택적 cuFile 의존성.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch10/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch10
python -m cli.aisp bench run --targets ch10 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- 클러스터 활성화 커널은 DSMEM 지원이 없는 하드웨어에서 빠르게 실패하고, DSMEM 없는 변형은 여전히 실행됩니다. 이를 사용하여 클러스터 기능 플래그를 확인하세요.
- `python optimized_flash_attn_tma_micro_pipeline.py --profile`이 베이스라인 스크립트보다 더 적은 커널 실행과 더 높은 달성 FLOP/s를 생성합니다.
- `bash demo_both_examples.sh`가 CUDA 메모리 파이프라인 및 GDS 데모를 실행하여 실행 분할 상각 및 I/O 오버랩을 강조합니다.

참고 사항

- `cufile_gds_example.py`는 I/O 집약적인 학습 루프를 위한 텐서 코어 파이프라인에 GPUDirect 스토리지를 통합하는 방법을 시연합니다.
- `requirements_cufile.txt`는 선택적 `cufile` 휠을 보유합니다. GPUDirect 스토리지가 활성화된 호스트에만 설치하세요.
- CUTLASS 스타일 워프 특화 쌍은 성능 비교를 위해 `sm100_mma_array_warpspecialized`와 정렬된 참조 구현을 제공합니다.

챕터 11

챕터 11 – 스트림 및 동시성

요약

CUDA 스트림, 순서화된 시퀀스, Hyper-Q, 워프 특화 파이프라인, 적응형 스케줄링을 사용하여 Blackwell에서 계산, 메모리, 통신을 오버랩하는 방법을 설명합니다.

학습 목표

- 여러 CUDA 스트림을 사용하여 우선순위 작업을 굶주리게 하지 않고 독립적인 커널을 오버랩합니다.
- KV 캐시 업데이트 및 스트림 순서 메모리 풀에 대한 순서화 제약을 제어합니다.
- DSMEM을 통해 데이터를 공유하는 워프 특화 멀티스트림 커널을 벤치마크합니다.
- 런타임 텔레메트리를 기반으로 스트림 사용을 조정하는 적응형 정책을 도입합니다.

디렉토리 구조

경로	설명
`baseline_streams.py`, `optimized_streams.py`, `streams_overlap_demo.cu`, `streams_ordered_demo.cu`, `streams_warp_specialized_demo.cu`, `stream_overlap_base.py`	직렬화된 실행과 오버랩된 워크로드를 비교하는 핵심 스트림 오버랩 데모.
`baseline_stream_ordered.py`, `baseline_stream_ordered_kv_cache.py`, `optimized_stream_ordered.py`, `optimized_stream_ordered_kv_cache.py`	오버랩을 활성화하면서 결정적인 업데이트를 보장하는 스트림 순서 할당자 및 KV 캐시 예제.
`baseline_gemm_streams.py`, `optimized_gemm_streams.py`, `baseline_tensor_cores_streams.py`, `optimized_tensor_cores_streams.py`	수학 vs I/O 단계를 분리하기 위해 여러 스트림에 걸쳐 텐서 코어 커널을 스케줄링하는 GEMM 파이프라인.
`baseline_distributed_streams.py`, `optimized_distributed_streams.py`, `baseline_adaptive_streams.py`, `optimized_adaptive_streams.py`	대규모 시스템에서 NCCL, 계산, I/O 작업을 균형 있게 조정하는 적응형 스트리밍 컨트롤러.
`baseline_warp_specialization_multistream.*`, `optimized_warp_specialized_multistream.*`, `warp_specialized_cluster_pipeline_multistream.cu`	DSMEM 사용 및 스트림별 특화를 시연하는 워프 특화 멀티스트림 커널.
`compare.py`, `Makefile`, `expectations_{hardware_key}.json`	동시성 회귀를 위한 하네스 드라이버 및 기대값 데이터.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch11/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch11
python -m cli.aisp bench run --targets ch11 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_streams.py --trace`가 Nsight Systems에서 오버랩되는 NVTX 범위를 캡처하여 동시성이 활성화되었음을 증명합니다.
- `python optimized_stream_ordered_kv_cache.py --validate`가 캐시 업데이트 간 유휴 갭을 줄이면서 베이스라인 출력과 일치합니다.
- 워프 특화 멀티스트림 커널이 지원되지 않는 하드웨어(DSMEM 없음)를 즉시 표시하여 자동 폴백을 방지합니다.

참고 사항

- `warp_specialized_triton.py`는 컴파일러 생성 스케줄을 비교할 수 있도록 CUDA 동시성 데모에 대한 Triton 유사물을 제공합니다.
- `kv_prefetch_pipeline_enhanced_demo.cu`는 이 디렉토리에 번들된 DSMEM 커널을 기반으로 하여 전체 파이프라인을 로컬에서 연구할 수 있습니다.

챕터 12

챕터 12 – CUDA 그래프 및 동적 워크로드

요약

현대적인 CUDA 그래프 기능인 조건부 캡처, 그래프 메모리 튜닝, 동적 병렬성, 작업 큐를 다루어 실행당 오버헤드 없이 불규칙적인 워크로드를 성능 있게 유지합니다.

학습 목표

- 안정적인 워크로드를 CUDA 그래프로 캡처하고 이거(eager) 실행과의 차이를 연구합니다.
- 적응형 파이프라인을 위해 조건부 노드와 그래프 메모리 풀을 사용합니다.
- CPU 개입을 줄이기 위해 디바이스 측 실행(동적 병렬성)을 실험합니다.
- GPU 상주 작업 큐 및 불균등 파티션 스케줄러를 구현합니다.

디렉토리 구조

경로	설명
`baseline_cuda_graphs.py`, `optimized_cuda_graphs.py`, `baseline_cuda_graphs_conditional*.cu`, `optimized_cuda_graphs_conditional*.cu`	단순 재생에서 조건부 및 DSM 인식 실행으로 발전하는 그래프 캡처 데모.
`baseline_graph_bandwidth.{py,cu}`, `optimized_graph_bandwidth.{py,cu}`, `baseline_kernel_launches.py`, `optimized_kernel_launches.py`	그래프가 CPU 오버헤드를 줄이는 방법을 보여주는 실행 및 대역폭 중심 연구.
`baseline_dynamic_parallelism_host.cu`, `baseline_dynamic_parallelism_device.cu`, `optimized_dynamic_parallelism_host.cu`, `optimized_dynamic_parallelism_device.cu` 외	동적 병렬성이 도움이 되거나 해가 되는 시기를 보여주는 디바이스 측 실행 샘플.
`baseline_work_queue.{py,cu}`, `optimized_work_queue.{py,cu}`, `work_queue_common.cuh`	NVTX 계측을 포함한 불규칙적인 배치 크기를 위한 GPU 작업 큐.
`baseline_uneven_partition.cu`, `optimized_uneven_partition.cu`, `baseline_uneven_static.cu`, `optimized_uneven_static.cu`	런타임에 CTA 할당을 재균형하는 불균등 워크로드 파티셔너.
`baseline_kernel_fusion.py`, `optimized_kernel_fusion.py`, `kernel_fusion_cuda_demo.cu`	CPU 동기화를 완전히 제거할 수 있도록 그래프 캡처 내의 커널 퓨전 실습.
`compare.py`, `cuda_extensions/`, `expectations_{hardware_key}.json`	하네스 진입점, 확장 스텁, 기대값 임계치.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch12/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch12
python -m cli.aisp bench run --targets ch12 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_cuda_graphs.py --iterations 100`이 출력을 일치시키면서 베이스라인보다 낮은 벽 시계 시간을 보고해야 합니다.
- 디바이스 측 동적 병렬성 샘플이 지원되지 않는 하드웨어에서 경고를 내보내어 해당 기능이 있는 GPU의 데이터만 신뢰합니다.
- `python optimized_work_queue.py --trace`가 베이스라인의 뒤처짐과 비교했을 때 CTA 간 균형 잡힌 dequeue 시간을 노출합니다.

참고 사항

- `cuda_graphs_workload.cuh`는 자체 커널을 래핑할 때 사용할 수 있는 재사용 가능한 그래프 캡처 헬퍼를 보유합니다.
- `helper_*.cu` 파일은 동적 병렬성 케이스 스터디를 위한 호스트/디바이스 접착 코드를 포함합니다. 새 실험을 부트스트래핑할 때 복사하세요.

챕터 13

챕터 13 – PyTorch 프로파일링 및 메모리 튜닝

요약

PyTorch 중심 최적화에 초점을 맞춥니다. 컴파일된 autograd, 메모리 프로파일링, FSDP/컨텍스트/전문가 병렬성, 동일한 하네스 인프라를 기반으로 하는 FP8/양자화 워크플로우를 다룹니다.

학습 목표

- PyTorch 학습 루프를 엔드 투 엔드로 프로파일링하여 굿풋, 메모리, 커널 트레이스를 캡처합니다.
- 오버헤드를 줄이기 위해 `torch.compile`, 지역 컴파일, 커스텀 할당자를 적용합니다.
- 단편화를 제거하기 위해 DataLoader, KV 캐시, 옵티마이저 상태를 튜닝합니다.
- Transformer Engine 통합으로 FP8/양자화 학습 레시피를 실습합니다.

디렉토리 구조

경로	설명
`baseline_training_standard.py`, `optimized_training_standard.py`, `train.py`, `train_deepseek_v3.py`, `train_deepseek_coder.py`	이거(eager) vs 컴파일 경로와 DeepSeek 영감 구성을 보여주는 참조 학습 루프.
`baseline_dataloader_default.py`, `optimized_dataloader_default.py`, `baseline_memory_profiling.py`, `optimized_memory_profiling.py`, `memory_profiling.py`	할당자 통계를 읽고 누수를 수정하는 방법을 설명하는 DataLoader/메모리 연구.
`baseline_attention_standard.py`, `optimized_attention_standard.py` 외 다수의 어텐션/행렬 곱셈 파일	장문 컨텍스트 Flash SDP를 포함하여 순수하게 PyTorch 내에서 튜닝된 어텐션 및 행렬 곱셈 마이크로벤치마크.
`baseline_context_parallel_multigpu.py`, `optimized_context_parallel_multigpu.py`, `context_parallel_benchmark_common.py`	랭크 간 all-gather vs 링 스타일 스트리밍을 비교하는 컨텍스트 병렬 어텐션 벤치마크.
`baseline_expert_parallel_multigpu.py`, `optimized_expert_parallel_multigpu.py`, `expert_parallel_common.py`	반복당 리스트 할당 vs 사전 할당된 all_to_all_single을 비교하는 전문가 병렬 all-to-all 벤치마크.
`context_parallelism.py`, `fsdp_example.py`	단일 GPU를 넘어 확장하기 위한 컨텍스트 및 FSDP 샤딩 데모(도구, 벤치마크 타겟 아님).
`baseline_precisionfp8*.py`, `optimized_precisionfp8*.py`, `baseline_precisionmixed.py`, `optimized_precisionmixed.py`, `compiled_autograd.py`	Transformer Engine 및 컴파일된 autograd 레시피를 다루는 정밀도 관리 제품군.
`baseline_quantization.py`, `optimized_quantization.py`, `baseline_kv_cache_naive.py`, `optimized_kv_cache_naive.py`, `optimized_kv_cache_naive_pool.py`	추론/학습 메모리 절약을 위한 양자화 및 KV 캐시 파이프라인.
`compare.py`, `compare_perf.py`, `requirements.txt`, `expectations_{hardware_key}.json`, `workload_config.py`	하네스 진입점, 성능 비교 헬퍼, 의존성, 회귀 기준선.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch13/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch13
python -m cli.aisp bench run --targets ch13 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python compare.py --examples training_standard`가 최적화된 학습 실행이 동일한 메트릭으로 더 높은 굿풋을 생성함을 보여줍니다.
- `python optimized_precisionfp8_te.py --validate`가 최대 오류 허용치가 적용된 Transformer Engine 캘리브레이션 및 NVFP8 실행을 확인합니다.
- `python memory_profiling.py --dump`와 최적화된 변형이 권장 파라미터 적용 후 할당자 단편화가 감소함을 시연합니다.

참고 사항

- `custom_allocator.py`는 단편화를 디버깅할 때 다른 챕터에서 재사용할 수 있는 독립형 torch 할당자 심(shim)을 포함합니다.
- `compiled_autograd.py`는 부분 그래프 캡처에 대한 튜토리얼 역할도 하며, 여기의 README는 이를 직접 참조합니다.

챕터 14

챕터 14 – 컴파일러 및 Triton 최적화

요약

컴파일러 기반 가속을 강조합니다. `torch.compile` 워크플로우, Triton 커널, CUTLASS/TMA 실험, 양자화 인식 통신을 다루며, 모두 공유 하네스를 통해 검증됩니다.

학습 목표

- 컴파일 시간과 안정적인 상태 이득을 추적하면서 대규모 모델에 `torch.compile` 모드를 채택합니다.
- 커스텀 CUDA와 경쟁하는 Triton 커널(TMA 스케줄 포함)을 작성합니다.
- FlexAttention 및 지역 컴파일 전략을 엔드 투 엔드로 프로파일링합니다.
- 회귀 없이 양자화를 NCCL 및 파이프라인 오버랩과 혼합합니다.

디렉토리 구조

경로	설명
`baseline_model_eager.py`, `optimized_model_eager.py`, `torch_compile_large_model.py`, `torch_compiler_examples.py`, `training_large_model_1_5x.py`	컴파일 모드, 가드레일, 대규모 모델 건전성 테스트를 보여주는 모델 규모 예제.
`baseline_cutlass.py`, `optimized_cutlass.py`, `triton_examples.py`, `triton_tma_blackwell.py`, `triton_fp8_advanced.py`, `triton_nvshmem_example.py`	CUTLASS vs Triton 비교 및 고급 TMA/NVSHMEM Triton 커널.
`baseline_flex_attention.py`, `optimized_flex_attention.py`, `baseline_flex_attention_sparse.py`, `optimized_flex_attention_sparse.py`, `flex_attention_sparse_demo.py`	커스텀 스코어 모드, 마스크, 희소성, 컴파일 속도 향상을 검증하는 FlexAttention 워크로드.
`baseline_nccl_quantization.py`, `optimized_nccl_quantization.py`, `deepseek_innovation_l2_bypass.py`	양자화 인식 통신 및 DeepSeek 영감의 L2 바이패스 실험.
`baseline_regional_triton.py`, `optimized_regional_triton.py`, `inspect_compiled_code.py`, `benchmark_tma_configs.py`	자동 생성 커널 자동 튜닝을 위한 지역 컴파일 및 TMA 파라미터 스윕.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`, `train.py`, `transformer.py`	하네스 진입점, 모델 정의, 의존성 핀.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch14/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch14
python -m cli.aisp bench run --targets ch14 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_model_eager.py --profile minimal`이 베이스라인 대비 안정적인 상태 처리량 이득에 이어 컴파일 시간 요약을 생성합니다.
- `python triton_tma_blackwell.py --validate`가 Triton과 CUDA 출력을 비교하여 TMA 스케줄링 로직을 재확인합니다.
- `python compare.py --examples flex_attention`이 컴파일된 경로가 정확도를 변경하지 않고 커널 실행 횟수를 크게 줄임을 보여줍니다.

참고 사항

- `inspect_compiled_code.py`가 모든 타겟의 Triton/PTX/그래프 캡처를 덤프합니다. 새 워크로드를 내성하기 위해 헬퍼를 편집하세요.
- `requirements.txt`는 컴파일러 기능이 CUDA 13 도구 체인과 정렬되도록 나이틀리 Triton + PyTorch 휠을 포함합니다.

챕터 15

챕터 15 – 분리된 추론 및 KV 관리

요약

대규모 추론 관련 사항을 다룹니다. 분리된 계산/스토리지, NVLink를 통한 KV 캐시 풀링, 연속 배칭, 전문가 혼합(MoE) 서빙 패턴을 포함합니다.

학습 목표

- 단일 vs 분리된 추론 경로를 벤치마크하고 패브릭 비용을 정량화합니다.
- 로컬 및 원격 HBM 풀을 우아하게 아우르는 KV 캐시 관리자를 설계합니다.
- 디코드 처리량을 높게 유지하기 위해 연속 배칭 및 큐잉을 구현합니다.
- 최적화된 통신과 라우팅을 짝지어 MoE 모델을 효율적으로 서빙합니다.

디렉토리 구조

경로	설명
`baseline_inference_monolithic.py`, `optimized_inference_monolithic.py`	분리 전 기준선을 확립하는 단일 박스 추론 루프.
`disaggregated_inference_multigpu.py`	프리필/디코드 풀 위에 추측 디코딩을 레이어링하는 분리 추론 데모.
`baseline_disaggregated_inference.py`, `optimized_disaggregated_inference.py` 외 다수의 분리 파일	원격 프리필, 디코드 오버랩, NVLink 풀링을 모델링하는 분리 파이프라인(단일 및 다중 GPU).
`baseline_kv_cache_management.py`, `optimized_kv_cache_management.py`, `kv_cache_management_math.py` 외	로컬 전용, 수학 전용, NVLink 풀링 변형이 있는 KV 캐시 오케스트레이션 유틸리티.
`baseline_continuous_batching.py`, `optimized_continuous_batching.py`	TTFT 인식 큐잉을 위한 단일 GPU 연속 배칭 스케줄러.
`baseline_continuous_batching_multigpu.py`, `optimized_continuous_batching_multigpu.py`	확장된 큐잉 처리량을 위한 다중 GPU 연속 배칭 스케줄러.
`baseline_moe_inference.py`, `optimized_moe_inference.py`	라우터 부하와 통신 제어를 짝지우는 추론 특화 MoE 워크로드.
`baseline_moe_overlap.py`, `optimized_moe_overlap_shared_expert.py`, `baseline_wide_ep.py`, `optimized_wide_ep.py` 외	오버랩, 패킹/언패킹, 토폴로지 인식 라우팅 디스패치를 설명하는 MoE 전문가 병렬 마이크로벤치마크.
`compare.py`, `requirements.txt`, `expectations_{hardware_key}.json`, `Makefile`	추론 중심 검증을 위한 하네스 진입점 및 의존성.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch15/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch15
python -m cli.aisp bench run --targets ch15 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python -m cli.aisp bench run --targets ch15:disaggregated_inference_multigpu --profile minimal --ncu-replay-mode kernel`이 정확도 동등성을 유지하면서 베이스라인에 비해 패브릭 지연을 줄임을 보여줍니다.
- `python optimized_kv_cache_management.py --validate`가 제거 및 승격 정책이 디코드 지연 시간을 예산 내로 유지함을 확인합니다.
- `python compare.py --examples continuous_batching`과 `python compare.py --examples continuous_batching_multigpu`가 최적화된 스케줄링이 단순한 큐 드레이닝보다 토큰/초를 높임을 보여줍니다.

참고 사항

- `disaggregated_inference_multigpu.py`는 순수 시뮬레이션 모드로 실행할 수 있습니다. 하드웨어가 NVLink 풀링을 위해 배선되지 않은 경우 `--simulate-network`를 설정하세요.
- 원하는 GPU 수에서 분리 파이프라인을 실행하려면 `torchrun --nproc_per_node <num_gpus>`를 사용하세요(기본값은 모든 가시 GPU, 짝수 개).
- `Makefile`은 다중 노드 디코드 실험에 필요한 MPI/UCX 타겟을 래핑합니다.

챕터 16

챕터 16 – 프로덕션 추론 최적화

요약

실제 추론 서비스에 초점을 맞춥니다. 페이지드 어텐션, Flash SDP, FP8 서빙, 텔레메트리 훅, 스케줄러, Blackwell 친화적 부하 테스트 하네스를 다룹니다.

학습 목표

- 모델을 배포하기 전에 대형 디코더 워크로드를 프로파일링하여 핫스팟을 파악합니다.
- 지연 시간 목표를 달성하기 위해 페이지드 어텐션, Flash SDP, 조각 컴파일을 채택합니다.
- 서빙 루프에 FP8 양자화, 대칭 메모리, 캐시 모니터링을 통합합니다.
- 당혹감 검사를 통해 정확도를 검증하면서 프로덕션 부하(다중 노드, MoE)를 시뮬레이션합니다.

디렉토리 구조

경로	설명
`inference_optimizations_blackwell.py`, `inference_profiling.py`, `inference_server_load_test.py`, `inference_serving_multigpu.py`	다중 GPU 추론 배포를 프로파일링하고 부하 테스트하는 최상위 오케스트레이션 스크립트.
`baseline_flash_sdp.py`, `optimized_flash_sdp.py`, `baseline_paged_attention.py`, `optimized_paged_attention.py`	나이브한 구현과 Flash/페이지드 변형을 비교하는 어텐션 커널.
`baseline_piece_graphs.py`, `optimized_piece_graphs.py`, `baseline_regional_compilation.py`, `optimized_regional_compilation.py`	안정적인 저지연 디코드를 위한 조각별 그래프 캡처 및 지역 컴파일.
`fp8_transformer_engine.py`, `test_fp8_quantization_real.py`, `symmetric_memory_inference.py`, `multi_gpu_validation.py`	정확도와 NVLink 효율성을 보장하기 위한 서빙 시간 FP8 및 대칭 메모리 검증.
`moe_performance_benchmark.py`, `synthetic_moe_inference_benchmark.py`, `moe_workload.py`	라우터 배치 및 전문가별 배칭을 스트레스 테스트하는 MoE 추론 하네스.
`cache_monitoring.py`, `dcgm_prometheus_exporter.py`, `scheduler.py`, `perplexity_eval.py`	추론 파이프라인에 연결된 텔레메트리, 스케줄링, 정확도 유틸리티.
`compare.py`, `requirements.txt`, `Makefile`, `expectations_{hardware_key}.json`	추론 중심 검증을 위한 하네스 진입점 및 의존성.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch16/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch16
python -m cli.aisp bench run --targets ch16 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_paged_attention.py --profile minimal` 이 베이스라인 스크립트에 비해 페이지 폴트가 적고 처리량이 향상됨을 나타냅니다.
- `python symmetric_memory_inference.py --validate` 가 NVLink 기반 KV 복제본이 최소한의 왜곡으로 동기화를 유지함을 확인합니다.
- `python inference_server_load_test.py --duration 120` 이 스케줄러를 실습하고 워밍업 후 안정적인 TTFT/TPOT 메트릭을 보고해야 합니다.

참고 사항

- `dcgm_prometheus_exporter.py`가 추가 설정 없이 Prometheus/Grafana가 소비할 수 있는 GPU별 메트릭을 내보냅니다.
- `cache_monitoring.py`는 실행 간 할당자 상태를 정상 확인하기 위해 독립적으로 실행할 수 있습니다.

챕터 17

챕터 17 – 동적 라우팅 및 하이브리드 서빙

요약

라우터 설계, 분리된 추론, 프로파일링 기율을 혼합하여 Blackwell 클러스터가 활용률을 희생하지 않고 프리필/디코드 풀, MoE 전문가, 파이프라인 스테이지 간에 쿼리를 라우팅할 수 있게 합니다.

학습 목표

- TTFT, TPOT, KV 지역성 메트릭에 반응하는 동적 라우터를 구현합니다.
- 현실적인 합성 부하 하에서 완전한 추론 스택(프리필 + 디코드)을 프로파일링합니다.
- 장문 컨텍스트 워크로드를 위해 파이프라인 병렬성과 라우팅 로직을 혼합합니다.
- 라우팅 실험실에 특화된 프로파일링 단계(루프라인, Nsight)를 문서화합니다.

디렉토리 구조

경로	설명
`baseline_dynamic_routing.py`, `optimized_dynamic_routing.py`, `dynamic_routing.py`, `early_rejection.py`	정적 휴리스틱에서 텔레메트리 기반 입장 및 거부 정책으로 발전하는 라우팅 컨트롤러.
`baseline_inference_full.py`, `optimized_inference_full.py` 외 다수의 프리필/디코드 분리 파일	분리된 프리필 및 디코드 풀을 모델링하는 엔드 투 엔드 추론 흐름(오버랩 중심, 배치 핸드오프, TTFT 중심, 장문 출력 TPOT 중심 다중 GPU 쌍 포함).
`baseline_pipeline_parallelism.py`, `optimized_pipeline_parallelism.py`	계산 및 KV 전송 스케줄링을 결합하는 파이프라인 병렬 워크로드.
`baseline_moe_router_uniform.py`, `optimized_moe_router_uniform_topology.py`	균일 vs 토폴로지 인식 라우팅을 비교하는 MoE 라우터 벤치마크 쌍.
`moe_router_uniform_demo.py`, `moe_router_topology_demo.py`	균일 vs 토폴로지 인식 전문가 선택을 비교하는 MoE 라우팅 데모(비 벤치마크).
`baseline_routing_static.py`, `optimized_routing_static.py`	정적/동적 샤딩 결정을 위한 라우터 변형(비교 가능한 벤치마크).
`baseline_memory.py`, `optimized_memory.py`, `blackwell_profiling_guide.py`	라우팅 워크로드에 맞는 메모리 바운드 케이스 스터디 및 프로파일링 가이드.
`compare.py`, `Makefile`, `expectations_{hardware_key}.json`, `dynamo_config.yaml`	하네스 진입점, 빌드 규칙, 기대값 기준선, Dynamo 구성 파라미터.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch17/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch17
python -m cli.aisp bench run --targets ch17 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_dynamic_routing.py --trace`가 베이스라인의 진동보다 더 빠르게 안정화되는 TTFT/TPOT 추세를 기록합니다.
- `python optimized_pipeline_parallelism.py --profile minimal`이 더 적은 유휴 버블로 프리필/디코드 세그먼트가 오버랩됨을 보여줍니다.
- `python -m cli.aisp tools roofline`이 최신 캡처를 사용하여 문서화된 루프라인 포인트를 재현합니다.

참고 사항

- `blackwell_profiling_guide.py`는 라우팅 집약적인 워크로드에 대한 Nsight Systems/Compute 캡처 및 루프라인 vs 점유율 병목 해석을 안내합니다.
- 분리된 프리필/디코드 베이스라인은 나이브한 스케줄링을 모델링하기 위해 요청별 블로킹 핸드오프와 요청별 동기화/배리어를 사용합니다. 최적화된 대응물은 오버랩하거나 처리량을 높이기 위해 그룹별 배치 또는 연속적인 KV/시드 슬랩을 전송합니다.

챕터 18

챕터 18 – 고급 어텐션 및 디코딩

요약

현대적인 디코더 기법인 FlexAttention, FlexDecoding, 추측 및 페이지드 어텐션 워크플로우를 PyTorch와 CUDA/Triton 모두에서 구현하여 실제 하드웨어에서 커널을 검증하면서 빠르게 반복할 수 있습니다.

학습 목표

- 커스텀 마스크, 스코어 모드, KV 캐시 통합으로 FlexAttention/FlexDecoding 워크로드를 프로토타입합니다.
- 더 낮은 지연 시간을 위해 추가 계산을 거래하는 추측 디코딩 파이프라인을 평가합니다.
- Blackwell tmem 제한에 맞게 조정된 텐서 코어 최적화 어텐션 커널을 테스트합니다.
- 제공된 러너를 사용하여 서빙 프레임워크(vLLM)와의 통합 지점을 검증합니다.

디렉토리 구조

경로	설명
`baseline_flexdecoding.py`, `optimized_flexdecoding.py`, `optimized_flexdecoding_graphs.py`, `v1_engine_loop.py`, `v1_engine_loop_common.py`	FlexDecoding 벤치마크 및 V1 폴링 루프 정확성 도구 (벤치마크 쌍 아님).
`baseline_tensor_cores.py`, `optimized_tensor_cores.py`, `flashmla_kernel.cu`, `warp_specialized_triton.py`	빠른 검증을 위한 텐서 코어 어텐션 커널 및 Triton 동등물.
`flex_attention_native.py`, `flex_attention_enhanced.py`, `flex_attention_large_model.py`, `kv_cache_integration_example.py`	소형 크기에서 KV 캐시 재사용이 있는 대형 모델까지의 FlexAttention 예제.
`baseline_vllm_v1_integration.py`, `optimized_vllm_v1_integration.py`, `baseline_vllm_decode_graphs.py`, `optimized_vllm_decode_graphs.py`, `configs/`, `spec_configs/`, `workload_config.py`	vLLM 또는 커스텀 하네스를 통해 워크로드를 밀어넣기 위한 서빙 통합 및 구성 프리셋.
`speculative_decode/spec_config_sweep.py`	추측 디코딩 구성을 스윕하고 지연 시간/처리량 트레이드오프를 요약하는 도구.
`compare.py`, `expectations_{hardware_key}.json`, `test_flex_attention.py`	하네스 진입점, 회귀 임계치, FlexAttention API를 위한 pytest 커버리지.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch18/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch18
python -m cli.aisp bench run --targets ch18 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_flexdecoding.py --profiling`이 디코딩된 토큰을 일치시키면서 베이스라인보다 훨씬 적은 커널과 낮은 지연 시간을 보고합니다.
- `python run_vllm_decoder.py --spec-config spec_configs/draft_and_verify.json`이 네이티브 FlexAttention 경로와 정확도 동등성으로 완료됩니다.
- `python test_flex_attention.py`가 로컬에서 통과하여 마스크/스코어-모드 헬퍼가 올바르게 연결되었음을 확인합니다.

참고 사항

- `flex_attention` 스크립트는 코드를 편집하지 않고도 형태를 스윕할 수 있도록 `BLOCK_SIZE`, `DOC_SPAN`, `SEQ_LEN`과 같은 환경 변수를 허용합니다.
- `flashmla_kernel.cu`는 SM121 하드웨어에서 컴파일 상태를 유지하기 위해 Blackwell 특화 텐서 메모리 가드를 포함합니다.

챕터 19

챕터 19 – 저정밀 학습 및 메모리 시스템

요약

NVFP4/FP8 워크플로우, KV 캐시 양자화, 메모리 이중 버퍼링, 적응형 할당자를 탐구하여 저정밀 실험이 수치적으로 안전하게 유지되면서 HBM의 모든 바이트를 짜냅니다.

학습 목표

- 캘리브레이션 및 검증 훅이 있는 FP4/FP6/FP8 학습 루프를 벤치마크합니다.
- 정밀도 제약을 존중하면서 KV 캐시 프리페치와 계산을 오버랩합니다.
- 드리프트 없이 실행 중에 형식을 전환하는 동적 양자화 캐시를 구현합니다.
- 단편화된 메모리 풀을 모니터링하고 재균형하는 할당자 헬퍼를 설계합니다.

디렉토리 구조

경로	설명
`baseline_nvfp4_training.py`, `optimized_nvfp4_training.py`, `native_fp4_quantization.py`, `native_fp6_quantization.py`, `native_fp8_training.py`	자동 캘리브레이션으로 FP8과 NVFP4 사이를 전환하는 학습 및 양자화 레시피.
`baseline_memory_double_buffering.py`, `optimized_memory_double_buffering.py`, `memory_allocator_with_monitoring.py`, `dynamic_memory_allocator.py`, `_allocator_worker.py`	이중 버퍼링, 계측, 적응형 워커 풀을 포함하는 메모리 관리 헬퍼.
`baseline_kv_prefetch_overlap.cu`, `optimized_kv_prefetch_overlap.cu`, `kv_prefetch_overlap_sm121` 바이너리	cp.async 파이프라인을 사용할 때 양자화된 KV 프리페치가 계산과 오버랩될 수 있음을 증명하는 CUDA 커널.
`baseline_dynamic_quantized_cache.py`, `optimized_dynamic_quantized_cache.py`, `dynamic_quantized_cache.py`, `token_precision_switching.py`, `dynamic_precision_switching.py`	정확도 예산에 따라 정밀도를 동적으로 전환하는 양자화 캐시 관리.
`baseline_fp4_hardware_kernel.cu`, `optimized_fp4_hardware_kernel.cu`, `fp8_hardware_kernel.cu`, `custom_allocator_retry.py`, `adaptive_parallelism_strategy.py`, `adaptive_parallelism_worker_pool.py`	이기종 정밀도 플릿을 위한 하드웨어 수준 커널 및 적응형 스케줄링 헬퍼.
`compare.py`, `arch_config.py`, `expectations_{hardware_key}.json`	하네스 진입점, 아키텍처 토글, 저장된 기대값 데이터.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch19/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch19
python -m cli.aisp bench run --targets ch19 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python optimized_nvfp4_training.py --calibrate`가 FP8로 워밍업한 후 NVFP4로 전환하여 베이스라인의 정확도 임계치와 일치합니다.
- `python optimized_dynamic_quantized_cache.py --trace`가 제한된 오류로 정밀도 전환을 기록하여 토큰 수준 전환의 정확성을 확인합니다.
- `nvcc -o optimized_kv_prefetch_overlap_sm121 optimized_kv_prefetch_overlap.cu`와 베이스라인 바이너리가 Nsight Compute에서 측정 가능한 오버랩 개선을 보여줍니다.

참고 사항

- `arch_config.py`는 디바이스별 `ENABLE_NVFP4` / `ENABLE_TF32` 토글을 노출하여 정밀도 레시피를 쉽게 비교할 수 있습니다.
- `validate_quantization_performance.py`는 이익 증명 보고를 위해 정확도 vs 처리량 수치를 CSV 형식으로 집계합니다.

챕터 20

챕터 20 – 종합 케이스 스터디

요약

커널, 메모리, 파이프라인, 추론 최적화를 종합적인 케이스 스터디로 결합합니다. 베이스라인 파이프라인을 가져와 단계적인 개선을 적용하고, 모든 주요 서브시스템에 대한 이익 증명 아티팩트를 캡처합니다.

학습 목표

- 메모리, 파이프라인, KV 캐시 최적화를 연결하여 누적 영향을 확인합니다.
- 베이스라인 vs 튜닝된 엔드 투 엔드 실행을 비교하는 자동 보고서를 생성합니다.
- AI 커널 생성기를 통해 새 커널을 프로토타입하고 하네스에 슬롯합니다.
- 워크로드별 허용 테스트로 개선 사항을 검증합니다.

디렉토리 구조

경로	설명
`baseline_multiple_unoptimized.py`, `optimized_multiple_unoptimized.py`, `ai_kernel_generator.py`, `inductor_guard.py`	여러 병목을 쌓는 복합 워크로드 및 후보 커널을 안전하게 생성하는 헬퍼.
`baseline_pipeline_sequential.py`, `optimized_pipeline_sequential.py`, `baseline_end_to_end_bandwidth.py`, `optimized_end_to_end_bandwidth.py`	최적화가 스테이지 간에 어떻게 상호 작용하는지 보여주는 파이프라인 및 대역폭 케이스 스터디.
`baseline_integrated_kv_cache.py`, `optimized_integrated_kv_cache.py`	할당자, 오버랩, NVLink 풀링 트릭을 병합하는 통합 KV 캐시 데모.
`baseline_memory_standard.py`, `optimized_memory_standard.py`	시스템 수준에서 할당자 변경을 검증하는 메모리 중심 하네스.
`baseline_training_single.py`, `optimized_training_single.py`, `test.cu`, `Makefile`	단일 디바이스 학습 케이스 스터디 및 최종 보고서에서 사용된 CUDA 커널.
`compare.py`, `arch_config.py`, `expectations_{hardware_key}.json`	하네스 드라이버, 아키텍처 설정, 기대값 기준선.

벤치마크 실행

빠른 비교를 위해 벤치마크 하네스를 사용하거나, 반복 가능한 아티팩트 캡처가 필요할 때 Typer CLI를 사용하세요.

```
python ch20/compare.py --profile none
python -m cli.aisp bench list-targets --chapter ch20
python -m cli.aisp bench run --targets ch20 --profile minimal
```

- Nsight 트레이스를 캡처할 때 워크로드별로 `--profile` 또는 `--iterations`를 오버라이드하세요.
- 기대값 기준선은 각 챕터 옆에 `expectations_{hardware_key}.json`으로 저장됩니다. 새 하드웨어 검증 후 `--update-expectations`로 갱신하세요.

검증 체크리스트

- `python compare.py`가 각 최적화된 변형이 저장된 기대값을 충족하거나 초과함을 보여주는 스테이지별 요약을 내보냅니다.
- `python ai_kernel_generator.py --emit test.cu`가 `nvcc`로 컴파일되고 수동 편집 없이 하네스에 통합되는 CUDA 커널을 생성합니다.
- `python optimized_pipeline_sequential.py --trace`가 전체 파이프라인을 아우르는 매끄러운 NVTX 범위를 보여주어 오버랩 성공을 시연합니다.

참고 사항

- `inductor_guard.py`는 기능 플래그 뒤에 실험적 커널을 게이팅하기 위한 편의 토글을 제공합니다.
- `ai_kernel_generator.py`는 재현성을 위해 생성된 코드를 `artifacts/`에 기록합니다. 이익 증명 번들과 함께 로그를 캡처하세요.

> 📌 **부록**은 원문(영어) 그대로 포함되어 있습니다. O'Reilly 도서에서 발췌한 200개 이상의 성능 체크리스트입니다.

Appendix

AI Systems Performance Checklist (200+ Items)

These are derviced from the 2025-2026 O'Reilly book, AI Systems Performance Engineering, by Chris Fregly, O'Reilly Media.

This extensive checklist covers both broad process-level best practices and detailed, low-level tuning advice for AI systems performance engineers. Each of these checklist items serves as a practical reminder to squeeze maximum performance and efficiency out of AI systems. Use this guide when debugging, profiling, analyzing, and tuning one's AI systems. By systematically applying these tips—from low-level OS and CUDA tweaks up to cluster-scale optimizations—an AI systems performance engineer can achieve both lightning-fast execution and cost-effective operation on modern NVIDIA GPU hardware using many AI software frameworks, including CUDA, PyTorch, OpenAI's Triton, TensorFlow, Keras, and JAX. The principles in this checklist will also apply to future generations of NVIDIA hardware, including their GPUs, ARM-based CPUs, CPU-GPU superchips, networking gear, and rack systems.

Performance Tuning and Cost Optimization Mindset

A pragmatic, documented loop—quick wins before deep work—turns engineering time into measurable ROI. Start by targeting the biggest runtime and cost drivers, and always profile before and after to verify impact. Combine auto-tuning, framework upgrades, cloud pricing levers, and utilization dashboards for high-ROI wins, documenting results and favoring simple, maintainable fixes. Tune throughput-sensitive hyperparameters when accuracy allows. Here are some tips on the performance tuning and cost optimization mindset:

Global GPU Optimization Principles

- **Centralize workload knobs** so baseline and optimized runs use the same tensor shapes, precision, and batch math; smoke vs. full passes become a config flip instead of a code edit.
- **Align workloads before timing** by clamping RNG seeds, preprocessing, and data staging; measured deltas should come from the optimization, not a lucky batch.
- **Measure the steady state**: compile extensions up front, run warmups, and place CUDA events on the execution stream so first-iteration noise never pollutes reported numbers.
- **Track throughput and latency together**—tokens/s or GB/s next to milliseconds—to expose regressions even when one metric improves.
- **Amortize expensive prep** by caching autotuned tile sizes, compiled graphs, NCCL communicators, and memory pools; retune only when shapes cross a new regime.
- **Exploit structured sparsity** where accuracy allows: 2:4 pruning unlocks sparse Tensor Core paths on Ampere+/Blackwell for free FLOPs.
- **Validate aggressively** with FP32/CPU references, checksums, or tolerance asserts so fast paths never trade correctness for speed.
- **Stage and overlap**: chunk activations, use double buffering, and overlap transfers with compute via CUDA streams or CUDA Graph replay to keep SMs and NVLink saturated.
- **Model realistic baselines**: show genuine inefficiencies (host staging, undersized grids, redundant syncs) instead of sleep loops so improvements stay believable.
- **Log the environment every run** (GPU, driver, CUDA toolkit, compiler flags, git SHA, benchmark command) and archive raw artifacts so anyone can replay the exact conditions.
- **Budget for iteration**: optimizations are investments—start with low-hanging fruit such as AMP or data loader fixes, then escalate to custom kernels only when the ROI pencils out.

Close the loop with automated autotuners

Feed profiler metrics into reinforcement-learning or Bayesian autotuners that can sweep tile sizes, streams, and compiler flags during scheduled "tuning windows." Cache the best configs per `(device, shape)` pair and set up alerts if a new driver or workload forces a retune. This keeps kernels near their hardware limits even as models and GPUs evolve.

Optimize the expensive first

Use the 80/20 rule. Find the top contributors to runtime and focus on those. If 90% of the time is in a couple of kernels or a communication phase, it's better to optimize those deeply than to microoptimize something taking 1% of the time. Each chapter's techniques should be applied where they matter most. For example, if your training is 40% data loading, 50% GPU compute, and 10% communication, then first fix data loading, as you can maybe halve the overhead. Then look at GPU kernel optimization.

Profile before and after

Whenever you apply an optimization, measure its impact. This sounds obvious, but often tweaks are made based on theory and might not help—or even hurt— in practice. Consider a scenario where your workload is not memory-limited, but you decide to try enabling activation checkpointing for your training job. This may actually slow down the job by using extra compute to reduce memory. In other words, always compare key metrics like throughput, latency, and utilization before and after making changes. Use the built-in profilers for simple timing, such as average iteration time over 100 iterations.

Embrace adaptive autotuning feedback loops

Implement advanced autotuning frameworks that leverage real-time performance feedback—using techniques like reinforcement learning or Bayesian optimization—to dynamically adjust system parameters. This approach enables your system to continuously fine-tune settings in response to changing workloads and operating conditions.

Budget for optimization time

Performance engineering is an iterative investment. There are diminishing returns—pick the low-hanging fruit like enabling AMP and data prefetch. These might give 2" easily. Harder optimizations like writing custom kernels might give smaller increments. Always weigh the engineering time versus the gain in runtime and cost saved. For large recurring jobs like training a flagship model, even a 5% gain can justify weeks of tuning since it saves maybe millions. For one-off or small workloads, focus on bigger wins and be pragmatic.

Stay updated on framework improvements

Many optimizations we discussed, such as mixed precision, fused kernels, and distributed algorithms, continue to be improved in deep learning frameworks and libraries. Upgrading to the latest PyTorch or TensorFlow can sometimes yield immediate speedups as they incorporate new fused ops or better heuristics. Leverage these improvements, as they are essentially free gains. Read release notes for performance-related changes.

Codesign collaboratively with vendors and community members

Stay connected with hardware vendors and the broader performance engineering community to align software optimizations with the latest hardware architectures. This codesign approach can reveal significant opportunities for performance gains by tailoring algorithms to leverage emerging hardware capabilities. Regularly review vendor documentation, participate in forums, and test beta releases of drivers or frameworks. These interactions often reveal new optimization opportunities and best practices that can be integrated into your systems. Integrating new driver optimizations, library updates, and hardware-specific tips can provide additional, sometimes significant, performance gains.

Leverage cloud flexibility for cost

If running in cloud environments, use cheaper spot instances or reserved instances wisely. They can drastically cut costs, but you may lose the spot instances with a few minutes' notice. Also consider instance types, as sometimes a slightly older GPU instance at a fraction of the cost can deliver better price/performance if your workload doesn't need the absolute latest. Our discussions on H800 versus H100 showed it's possible to do great work on second-best hardware with effort. In the cloud, you can get similar trade-offs. Evaluate cost/performance by benchmarking on different instance configurations, including number of CPUs, CPU memory, number of GPUs, GPU memory, L1/L2 caches, unified memory, NVLink/NVSwitch interconnects, network bandwidth and latency, and local disk configuration. Calculate metrics like throughput per dollar to guide your optimization decisions.

Monitor utilization metrics

Continuously monitor GPU utilization, SM efficiency, memory bandwidth usage, and, for multinode, network utilization. Set up dashboards using DCGM exporter, Prometheus, etc., so you can catch when any resource is underused. If GPUs are at 50% utilization, dig into why. It's likely data waiting/stalling and slow synchronization communication. If the network is only 10% utilized but the GPU waits on data, maybe something else like a lock is the issue. These metrics help pinpoint which subsystem to focus on.

Iterate and tune hyperparameters for throughput

Some model hyperparameters, such as batch size, sequence length, and number of MoE active experts, can be tuned for throughput without degrading final accuracy. For example, larger batch sizes give better throughput but might require tuning the learning rate schedule to maintain accuracy. Don't be afraid to adjust these to find a sweet spot of speed and accuracy. This is part of performance engineering too—sometimes the model or training procedure can be adjusted for efficiency, like using activation checkpointing or more steps of compute for the same effective batch. You might tweak the training learning rate schedule to compensate for this scenario.

Document and reuse

Keep notes of what optimizations you applied and their impact. Document in code or in an internal wiki-like shared knowledge-base system. This builds a knowledge base for future projects. Many tips are reusable patterns, like enabling overlapping and particular environment variables that help on a cluster. Having this history can save time when starting a new endeavor or when onboarding new team members into performance tuning efforts.

Balance optimizations with complexity

Aim for the simplest solution that achieves needed performance. For example, if native PyTorch with torch.compile meets your speed target, you might not need to write custom CUDA kernels. This will help avoid extra maintenance. Over- optimizing with highly custom code can make the system brittle. There is elegance in a solution that is both fast and maintainable. Thus, apply the least- intrusive optimization that yields the required gain, and escalate to more involved ones only as needed.

Optimize AI-driven performance

Leverage machine learning models to analyze historical telemetry data and predict system bottlenecks, enabling automated adjustments of parameters in real time to optimize resource allocation and throughput.

Reproducibility and Documentation Best Practices

Performance wins don't stick unless they're reproducible, versioned, and continuously checked, or they'll regress quietly over time. Treat docs, CI benchmarks, and shared knowledge as the glue that preserves speedups and accelerates onboarding and audits. Lock down versions, configs, and benchmarks in source control so experiments are repeatable and regressions traceable. Bring performance checks into CI/CD, instrument end-to-end monitoring and alerts, and pair optimization with security and thorough documentation to create a durable, auditable practice. The following is a list of tips to improve reproducibility and documentation:

Rigorous version control

Maintain comprehensive version control for all system configurations, framework/driver versions, OS settings, optimization scripts, and benchmarks. Use Git (or a similar system) to track changes and tag releases. This way, experiments can be reproduced exactly—and performance regressions can be easily identified.

Continuous integration for performance regression

Integrate automated performance benchmarks and real-time monitoring into

your CI/CD pipelines. This ensures that each change—from code updates to configuration changes—is validated against a set of performance metrics, helping catch regressions early and maintaining consistent and measurable performance gains. Adopt industry-standard benchmarks, such as MLPerf, to establish a reliable performance baseline and track improvements over time.

End-to-end workflow optimization

Ensure that optimizations are applied holistically across the entire AI pipeline— from data ingestion and preprocessing through training and inference deployment. Coordinated, cross-system tuning can reveal synergies that isolated adjustments might miss, resulting in more significant overall performance gains.

Automated monitoring and diagnostics

Deploy end-to-end monitoring solutions that collect real-time metrics across hardware, network, and application layers. Integrate these with dashboards, such as Prometheus/Grafana, and configure automated alerts to promptly detect anomalies, such as sudden drops in GPU utilization or spikes in network latency.

Fault tolerance and automated recovery

Incorporate fault tolerance into your system design by using distributed checkpointing, redundant hardware configurations, and dynamic job rescheduling. This strategy minimizes downtime and maintains performance even in the face of hardware or network failures.

Compiler and build optimizations

Leverage aggressive compiler flags and profile-guided optimizations during the build process to extract maximum performance from your code. Regularly update and tune your build configurations, and verify the impact of each change through rigorous benchmarking to ensure optimal execution. Security, compliance, and performance Integrate and codesign security, compliance, and performance. Regularly audit configurations, enforce access controls, and maintain industry-standard safeguards, including encryption, secure data channels, zero-trust networking, hardware security modules (HSMs), and secure enclaves. And make sure that performance tuning never compromises system security. Similarly, make sure security doesn't incur unnecessary performance overhead.

Comprehensive documentation and knowledge sharing

Maintain detailed records of all optimization steps, system configurations, and performance benchmarks. Develop an internal knowledge base to facilitate team collaboration and rapid onboarding, ensuring that best practices are preserved and reused across projects.

Future-proofing and scalability planning

Design modular, adaptable system architectures that can easily incorporate emerging hardware and software technologies. Continuously evaluate scalability requirements and update your optimization strategies to sustain competitive performance as your workload grows.

System Architecture and Hardware Planning

Your hardware, interconnects, and data paths set the ceiling for performance and cost-efficiency—no software tweak can outrun a starved GPU. Plan for goodput per dollar/watt by matching accelerators, CPU/DRAM/I/O, and cooling/power to the workload to avoid bottlenecks from the start. Specifically, design for goodput—useful work per dollar/watt—and not just raw FLOPS. Match accelerators and interconnects to workload, right-size CPU/ memory/I/O to keep GPUs fed, keep data local, and plan power/cooling so hardware sustains peak clocks. Evaluate scaling efficiency before adding more GPUs. Here are some tips for optimizing system architecture and improving hardware planning efficiency:

Design for goodput and efficiency

Treat useful throughput as the goal. Every bit of performance gained translates to massive cost savings at scale. Focus on maximizing productive work per dollar/ watt—and not just raw FLOPS.

Choose the right accelerator

Prefer modern GPUs like the Blackwell GB200/GB300 for superior performance- per-watt and memory capacity. Newer architectures offer features like native FP8 and FP4 precision support—along with much faster interconnects. These produce big speedups over older-generation GPUs and systems.

Leverage high-bandwidth interconnects

Use systems with NVLink/NVSwitch, such as GB200/GB300 NVL72, instead of PCIe-only connectivity for multi-GPU workloads. NVLink 5 provides up to 1.8 TB/s bidirectional GPU-to-GPU bandwidth (over 14" PCIe Gen5), enabling near-linear scaling across GPUs. NVLink Switch domains can be scaled with second-level switches to connect up to 576 GPUs in one NVLink domain. This enables hierarchical collectives that stay on NVLink as long as possible before falling back to the inter-rack fabric.

Balance CPU/GPU and memory ratios

Provision enough CPU cores, DRAM, and storage throughput per GPU. For example, allocate ~1 fast CPU core per GPU for data loading and networking tasks. Ensure system RAM and I/O can feed GPUs at required rates on the order of hundreds of MB/s per GPU to avoid starvation.

Plan for data locality

If training across multiple nodes, minimize off-node communication. Whenever possible, keep tightly coupled workloads on the same NVLink/NVSwitch domain to exploit full bandwidth, and use the highest-speed interconnect that you have access to. Ideally, this is NVLink for intranode and intra-rack communication and InfiniBand for inter-rack communication.

Avoid bottlenecks in the chain

Identify the slowest link—be it CPU, memory, disk, or network—and scale it up. For instance, if GPU utilization is low due to I/O, invest in faster storage or caching rather than more GPUs. An end-to-end design where all components are well-matched prevents wasted GPU cycles.

Choose an appropriate cluster size

Beware of diminishing returns when adding GPUs. Past a certain cluster size, overheads can grow—ensure the speedup justifies the cost. It's often better to optimize utilization on N GPUs by reaching 95% usage, for example, before scaling to 2N GPUs.

Design for cooling and power

Ensure the data center can handle GPU thermal and power needs. High- performance systems like GB200/GB300 have very high TDP. Provide adequate cooling (likely liquid-based) and power provisioning so the GPUs can sustain boost clocks without throttling.

Unified CPU-GPU "Superchip" Architecture

Unified memory and on-package links let you fit larger models and cut copy overhead when you place the right data in the right tier. Using Grace for preprocessing and HBM for "hot" tensors turns the superchip into a tightly coupled engine with fewer stalls. On Grace Blackwell Superchips, treat CPU and GPU as a shared-memory complex. Keep hot weights/activations in HBM and overflow or infrequent data in Grace LPDDR via NVLink-C2C. Use the on-package Grace CPU for preprocessing/orchestration and prefetch or pipeline-managed memory to hide latency for ultralarge models. Take advantage of the superchip architecture as follows:

Utilize unified CPU-GPU memory

Exploit the Grace Blackwell (GB200/GB300) Superchip's unified memory space. Two Blackwell GPUs and a 72-core Grace CPU share a coherent memory pool with NVLink-C2C (900 GB/s). Use the CPU's large memory (e.g., 480 GB LPDDR5X) as an extension for oversize models while keeping "hot" data in the GPUs' HBM for speed.

Place data for locality

Even with unified memory, prioritize data placement. Put model weights, activations, and other frequently accessed data on GPU HBM3e (which has much higher local bandwidth), and let infrequently used or overflow data reside in CPU RAM. This ensures the 900 GB/s NVLink-C2C link isn't a bottleneck for critical data. Take advantage of CPU-GPU direct memory access when available Use the GPU's ability to directly access CPU memory on combined CPU-GPU superchips like the GB200 and GB300. GPUs can read and write Grace LPDDR memory coherently over NVLink-C2C without staging over host PCIe. Bandwidth and latency are still lower than HBM, so prefetch managed pointers, stage data, and pipeline transfers to hide latency. As such, it's recommended to keep hot activations and KV cache in HBM and use CPU memory as a lower-tier cache with explicit prefetch.

Use the Grace CPU effectively

The on-package Grace CPU provides 72 high-performance cores—utilize themOffload data preprocessing, augmentation, and other CPU-friendly tasks to these cores. They can feed the GPUs quickly using NVLink-C2C, essentially acting as an extremely fast I/O and compute companion for the GPU.

Plan for ultralarge models

For trillion-parameter model training that exceeds GPU memory, GB200/GB300 systems allow you to train using CPU memory as part of the model's memory pool. Prefer framework caching allocators and use cudaMallocAsync in custom code to minimize fragmentation and enable graph capture. Use CUDA Unified Memory or managed memory APIs to handle overflow gracefully, and consider explicit prefetching (e.g., cudaMemPrefetchAsync) of upcoming layers from CPU # GPU memory to hide latency.

Superchip-optimized algorithms

SuperOffload is an example of a superchip-optimized set of algorithms focused on improving efficiency of offload and tensor cast/copy strategies. Innovations include speculation-then-validation (STV), heterogeneous optimizer computation, and an

ARM-based CPU optimizer. Designed specifically for NVIDIA superchips (e.g., Grace Hopper, Grace Blackwell, Vera Rubin), SuperOffload increases token-processing throughput and chip utilization relative to traditional offload strategies.

Multi-GPU Scaling and Interconnect Optimizations

Scaling pays only when communication is fast and topology-aware—otherwise added GPUs just wait on one another. Lean on NVLink/NVSwitch bandwidth, modern collectives, and fabric-aware placement to approach linear speedups. Specifically, exploit NVLink/NVSwitch domains (e.g., NVL72) for near-linear scaling, and choose parallelism strategies that fit the fabric. Use topology-aware placement, updated NCCL collectives (e.g., PAT), and telemetry to verify you're using the ~1.8 TB/s bidirectional throughput per-GPU bandwidth effectively. Plan hierarchical communications as you expand. The following are tips on utilizing multi-GPU scaling through interconnect and topology optimizations:

Design for high-speed all-to-all topology

On NVL72 NVSwitch clusters with 72 fully interconnected GPUs, for example, any GPU can communicate with any other at full NVLink 5 speed. At the fabric level, the NVLink Switch domain is nonblocking. Application-level throughput can vary with concurrent traffic and path scheduling, so verify behavior using DCGM NVLink counters and Nsight Systems traces before assuming per-pair saturation. Take advantage of this topology by using parallelization strategies, such as data parallel, tensor parallel, and pipeline parallelism, that would be bottlenecked on lesser interconnects.

Utilize topology-aware scheduling

Always colocate multi-GPU jobs within an NVLink Switch domain if possible. Keeping all GPUs of a job on the NVL72 fabric means near-linear scaling for communication-heavy workloads. Mixing GPUs across NVLink domains or standard networks will introduce bottlenecks and should be avoided for tightly coupled tasks.

Leverage unprecedented bandwidth

Recognize that NVLink 5 has 900 GB/s per GPU in each direction, which doubles the per-GPU bandwidth versus the previous generation. An NVL72 rack provides 130 TB/s total bisection bandwidth. This drastically reduces communication wait times, as even tens of gigabytes of gradient data can be all-reduced in a few milliseconds at 1.8 TB/s. Design training algorithms, such as gradient synchronization and parameter sharding, to fully exploit this relatively free communication budget.

Embrace modern collective algorithms

Use the latest NVIDIA NCCL library optimized for NVSwitch. Specifically, enable the parallel aggregated tree (PAT) algorithm, which was introduced for NVLink Switch topologies. This further reduces synchronization time by taking advantage of the NVL72 topology to perform reductions more efficiently than other tree/ring algorithms.

Consider fine-grained parallelism

With full-bandwidth all-to-all connectivity, consider fine-grained model parallelism that wasn't feasible before. For example, layer-wise parallelism or tensor parallelism across many GPUs can be efficient when each GPU has 1.8 TB/s bidirectional throughput to every other. Previously, one might avoid excessive cross-GPU communication, but NVL72 allows aggressive partitioning of work without hitting network limits.

Monitor for saturation

Although NVL72 is extremely fast, keep an eye on link utilization in profiling. If your application somehow saturates the NVSwitch using extreme all-to-all operations, for example, you might need to throttle communication by aggregating gradients, etc. Use NVIDIA's tools or NVSwitch telemetry to verify that communications are within the NVLink capacity, and adjust patterns if needed. For instance, you can stagger all-to-all exchanges to avoid network contention. DCGM exposes NVLink counters that can help verify link balance and detect hotspots during collectives.

Plan for future expansion

Be aware that NVLink Switch can scale beyond a single rack—up to 576 GPUs in one connected domain using second-level switches. If you operate at that ultrascale, plan hierarchical communication using local NVL72 inter-rack collectives first, then use inter-rack interconnects only when necessary. This helps to maximize intra-rack NVLink usage first. This ensures you're using the fastest links before resorting to inter-rack InfiniBand hops.

Identify opportunities for federated and distributed optimizations

For deployments that span heterogeneous environments, such as multicloud or edge-to-cloud setups, adopt adaptive communication protocols and dynamic load balancing strategies. This minimizes latency and maximizes throughput across distributed systems, which ensures robust performance even when resources vary in capability and capacity.

Operating System and Driver Optimizations

OS jitter, NUMA misses, and driver mismatches quietly drain throughput and create variability you can't tune around. Hardening the stack (huge pages, affinities, consistent CUDA/driver, persistence) creates a stable, high-performance baseline. Run a lean, HPC-tuned Linux. Set NUMA/IRQ affinities and enable THP and high memlock. Keep NVIDIA drivers/CUDA consistent across nodes. Isolate system jitter, tune CPU libraries/storage, set container limits correctly, and keep BIOS/firmware/ NVSwitch fabric up to date for predictable throughput. Here are some host, OS, and container optimizations that you should explore in your environment:

Prioritize critical host threads

Pin latency-sensitive work (data-loader processes, NCCL launchers, RPC handlers) to isolated CPU cores with higher scheduling priority. Reserve sibling cores for background daemons so kernel housekeeping never preempts threads feeding the GPU. Pair this with IRQ affinity and cgroup limits so `ksoftirqd` or noisy neighbors don't steal cycles during all-reduce phases.

Use a Linux kernel tuned for HPC

Ensure your GPU servers run a recent, stable Linux kernel configured for high- performance computing. Disable unnecessary background services that consume CPU or I/O. Use the "performance" CPU governor—versus "on-demand" or "power-save"—to keep CPU cores at a high clock for feeding GPUs.

Disable swap for performance-critical workloads

Disable swap on training servers to avoid page thrashing, or, if swap must remain enabled, lock critical buffers using mlock or cudaHostAlloc to ensure they stay in RAM.

Avoid memory fragmentation with aggressive preallocation

Preallocate large, contiguous blocks of memory for frequently used tensors to reduce runtime allocation overhead and fragmentation. This proactive strategy ensures more stable and efficient memory management during long training runs.

Optimize environment variables for CPU libraries

Fine-tune parameters, such as OMP_NUM_THREADS and MKL_NUM_THREADS, to better match your hardware configuration. Adjusting these variables can reduce thread contention and improve the parallel efficiency of CPU-bound operations.

Design for NUMA awareness

For multi-NUMA servers, pin GPU processes/threads to the CPU of the local NUMA node. Use tools like numactl or taskset to bind each training process to the CPU nearest its assigned GPU. Similarly, bind memory allocations to the local NUMA node (numactl --membind) so host memory for GPU DMA comes from the closest RAM. This avoids costly cross-NUMA memory traffic that can halve effective PCIe/NVLink bandwidth.

Utilize IRQ affinity for network and GPU tasks

Explicitly bind NIC interrupts to CPU cores on the same NUMA node as the NIC, and similarly pin GPU driver threads to dedicated cores—including those from long-running services like the nvidia-persistence service daemon. This strategy minimizes cross-NUMA traffic and stabilizes performance under heavy loads.

Enable transparent hugepages

Turn on transparent hugepages (THP) in always or madvise mode so that large memory allocations use 2 MB pages. This reduces TLB thrashing and kernel overhead when allocating tens or hundreds of GBs of host memory for frameworks. Verify THP is active by checking for /sys/kernel/mm/transparent_huge page/enabled. With THP enabled, your processes are using hugepages for big allocations. Consider disabling THB (or using madvise) if your workload is latency-critical and you observe jitter.

Increase max locked memory

Configure the OS to allow large pinned (aka page-locked) allocations. GPU apps often pin memory for faster transfers—set ulimit -l unlimited or a high value so your data loaders can allocate pinned buffers without hitting OS limits. This prevents failures or fallbacks to pageable memory, which would slow down GPU DMA.

Use the latest NVIDIA driver and CUDA stack

Keep NVIDIA drivers and CUDA runtime up-to-date (within a tested stable version) on all nodes. New drivers can bring performance improvements and are required for new GPUs' compute capabilities. Make sure all nodes have the same driver/CUDA versions to avoid any mismatches in multinode jobs. Enable persistence mode on GPUs at boot (nvidia-smi -pm 1) so the driver stays loaded and GPUs don't incur re-init delays. Update the NVIDIA driver and toolkit on all nodes to inherit bug fixes and performance improvements.

Enable GPU persistence when using a MIG configuration

With persistence mode enabled, the GPU remains "warm" and ready to use, reducing startup latency for jobs. This is especially crucial if using a Multi- Instance GPU (MIG) partitioning—without persistence, MIG configurations would reset on every job, but keeping the driver active preserves the slices. Always configure persistence mode when using MIG.

Isolate system tasks

Dedicate a core—or small subset of cores—on each server for OS housekeeping, such as interrupt handling and background daemons. This way, your main CPU threads feeding the GPU are not interrupted. This can be done using CPU isolation or cgroup pinning. Eliminating OS jitter ensures consistent throughput.

Optimize system I/O settings

If your workload does a lot of logging or checkpointing, mount filesystems with options that favor throughput. Consider using noatime for data disks and increase filesystem read-ahead for streaming reads. Ensure the disk scheduler is set appropriately to use mq-deadline or noop for NVMe SSDs to reduce latency variability.

Perform regular maintenance

Keep BIOS/firmware updated for performance fixes. Some BIOS updates improve PCIe bandwidth or fix input - output memory management unit (IOMMU) issues for GPUs. Also, periodically check for firmware updates for NICs and NVSwitch/Fabric if applicable, as provided by NVIDIA, such as Fabric Manager upgrades, etc. Minor firmware tweaks can sometimes resolve obscure bottlenecks or reliability issues.

Tune Docker and Kubernetes configurations for maximum performance

When running in containers, add options, such as --ipc=host for shared memory, and set --ulimit memlock=-1 to prevent memory locking issues. This guarantees that your containerized processes access memory without OS-imposed restrictions.

GPU Resource Management and Scheduling

Smarter placement and partitioning raise utilization without buying new hardware— and protect predictability for mixed workloads. Respect topology, use MPS/MIG where appropriate, and control clocks/power to minimize contention and tail latency. Schedule with GPU/NUMA/NVLink topology in mind, and use MPS or MIG to raise utilization for smaller jobs while retaining ECC and persistence for reliability. Lock clocks or power limit for stability when needed, avoid CPU oversubscription, and pack jobs intelligently to maximize ROI without contention. Here are some GPU resource management and scheduling tips:

Topology-aware job scheduling

Ensure that orchestrators like Kubernetes and SLURM are scheduling containers on nodes that respect NUMA and NVLink boundaries to minimize cross- NUMA and cross-NVLink-domain memory accesses. This alignment reduces latency and improves overall throughput.

Multi-Process Service (MPS)

Enable NVIDIA MPS when running multiple processes on a single GPU to improve utilization. MPS allows kernels from different processes to execute concurrently on the GPU instead of time-slicing. This is useful if individual jobs

don't fully saturate the GPU—for example, running 4 training tasks on one GPU with MPS can overlap their work and boost overall throughput.

Multi-Instance GPU (MIG)

Use MIG to partition high-end GPUs into smaller instances for multiple jobs. If you have many light workloads like inferencing small models or running many experiments, you can slice a GPU to ensure guaranteed resources for each job. For instance, modern GPUs can be split into 7 MIG slices. Do not use MIG for tightly coupled parallel jobs, as those benefit from full GPU access. Deploy MIG for isolation and maximizing GPU ROI when jobs are smaller than a full GPU.

Persistence for MIG

Keep persistence mode on to maintain MIG partitions between jobs. This avoids repartitioning overhead and ensures subsequent jobs see the expected GPU slices without delay. Configure MIG at cluster boot and leave it enabled so that scheduling is predictable, as changing MIG config on the fly requires resetting the GPU, which can disrupt running jobs. Plan for maintenance windows as MIG device partitions are not persisted by the GPU across reboot. Use NVIDIA's MIG Manager to automatically recreate the desired layout on boot.

GPU clock and power settings

Consider locking GPU clocks to a fixed high frequency with nvidia-smi -lgc/- lmc if you need run-to-run consistency. By default, GPUs use auto boost, which is usually optimal, but fixed clocks can avoid any transient downclocking. In power-constrained scenarios, you might slightly underclock or set a power limit to keep GPUs in a stable thermal/power envelope—this can yield consistent performance if occasional throttling was an issue.

ECC memory

Keep ECC enabled on data center GPUs for reliability unless you have a specific reason to disable it. The performance cost is minimal—on the order of a few percent loss in bandwidth and memory—but ECC catches memory errors that could otherwise corrupt a long training job. Most server GPUs ship with ECC on by default. Leave it on to safeguard multiweek training.

Job scheduler awareness

Integrate GPU topology into your job scheduler, such as SLURM and Kubernetes. Configure the scheduler to allocate jobs on the same node or same NVSwitch group when low-latency coupling is needed. Use Kubernetes device plugins or SLURM Gres to schedule MIG slices for smaller jobs. A GPU-aware scheduler prevents scenarios like a single job spanning distant GPUs and suffering bandwidth issues.

CPU oversubscription

When scheduling jobs, account for the CPU needs of each GPU task, such as data loading threads, etc. Don't pack more GPU jobs on a node than the CPUs can handle. It's better to leave a GPU idle than to overload the CPU such that all GPUs become underfed. Monitor CPU utilization per GPU job to inform scheduling decisions.

Use NVIDIA Fabric Manager for NVSwitch

On systems with NVSwitch, the GB200/GB300 NVL72 racks ensure NVIDIA Fabric Manager is running. It manages the NVSwitch topology and routing. Without it, multi-GPU communication might not be fully optimized or could even fail for large jobs. The Fabric Manager service typically runs by default on NVSwitch-equipped servers, but you should double-check that it's enabled and running—especially after driver updates.

Job packing for utilization

Maximize utilization by intelligently packing jobs. For example, on a 4-GPU node, if you have two 2-GPU jobs that don't use much CPU, running them together on the same node can save resources and even use the faster NVLink for communication if running together inside the same compute node or NVLink- enabled rack. Conversely, avoid colocating jobs that collectively exceed the memory or I/O capacity of the node. The goal is high hardware utilization without contention.

I/O Optimization

If data can't keep up, GPUs idle—often the largest, cheapest speedups come from fixing input, not math. Parallelism, pinned memory, async transfers, and fast storage ensure the model is continuously fed. Keep the GPUs fed by parallelizing data loaders, using pinned memory and async transfers, and storing data on fast NVMe—preferably with GPUDirect Storage. Stripe, cache, and compress wisely. Measure end-to-end throughput so I/O scales with cluster size, and write checkpoints/logs asynchronously. Here are some tips on I/O optimizations for your data pipeline:

Load data in parallel

Use multiple workers/threads to load and preprocess data for the GPUs. The default of one to two data loader workers may be insufficient. Profile and increase the number of data loader processes/threads using PyTorch Data Loader(num_workers=N), for example, until the data input is no longer the bottleneck. High core-count CPUs exist to feed those GPUs, so make sure you utilize them.

Pin host memory for I/O

Enable pinned (aka page-locked) memory for data transfer buffers. Many frameworks have an option like PyTorch's pin_memory=True for its DataLoader to allocate host memory that the GPU can DMA from directly. Using pinned memory significantly improves H2D copy throughput. Combine this with asynchronous transfers to overlap data loading with computation.

Overlap compute and data transfers

Pipeline your input data. While the GPU is busy computing on batch N, load and prepare batch N+1 on the CPU and transfer it in the background using CUDA streams and nonblocking cudaMemcpyAsync. This double buffering hides latency —the GPU ideally never waits for data. Use dedicated streams for H2D copies, augmentation kernels, and the main training loop, then fence them once per iteration via CUDA events. In PyTorch, copy tensors with non_blocking=True so the transfer happens in parallel with compute. The CPU stays productive preparing future batches while the GPU trains on the current one.

Use fast storage (NVMe/SSD)

Store training data on fast local NVMe SSDs or a high-performance parallel filesystem. Spinning disks will severely limit throughput. If available, enable GPUDirect Storage (GDS) so that GPUs can stream data directly from NVMe or network storage—bypassing the CPU. This further reduces I/O latency and CPU load when reading large datasets. For large datasets, consider each node having a local copy or shard of the data. If using network storage, prefer a distributed filesystem like Lustre with striping or an object store that can serve many clients in parallel.

Enable GPUDirect Storage end-to-end

GDS only pays off when every layer is configured: recent NVIDIA drivers, a GDS-enabled filesystem or NVMe stack, and application code that issues large, aligned reads. Batch small files into tar/record containers, align buffers to 4 KB, and issue deep I/O queues so the DMA engines stay busy. Monitor `nvidia-fs` stats and compare GPU-side throughput (via Nsight Systems) against CPU paths to ensure the zero-copy path is really engaged.

Overlap checkpointing with compute

Write checkpoints asynchronously so the training loop never blocks on disk. Stage weights and optimizer states into pinned host buffers (or a background GPU stream) while the next iteration runs, then flush them to storage via dedicated I/O threads. Track checkpoint bandwidth/latency metrics; if a flush falls behind, throttle frequency or compress on the GPU before shipping data over PCIe.

Tune I/O concurrency and striping

Avoid bottlenecks from single-file access. If one large file is used by all workers, stripe it across multiple storage targets or split it into chunks so multiple servers can serve it. For instance, break datasets into multiple files and have each data loader worker read different files simultaneously. This maximizes aggregate bandwidth from the storage system.

Optimize small files access

If your dataset consists of millions of small files, mitigate metadata overhead. Opening too many small files per second can overwhelm the filesystem's metadata server. Solutions pack small files into larger containers, such as tar or RecordIO files; use data ingestion libraries that batch reads; or ensure metadata caching is enabled on clients. This reduces per-file overhead and speeds up epoch start times.

Shard and cache at the loader

Give each data-loader worker (or distributed rank) its own disjoint shard of the dataset so they aren't chasing the same files and metadata. For epoch-level shuffles, reshuffle shard indices instead of file lists so caches stay warm. Cache hot samples on local NVMe, or stage one epoch ahead on each node, so remote storage only sees sequential, prefetchable access. Pair these tactics with PyTorch's `DistributedSampler` / `prefetch_factor` knobs so every GPU gets unique data without hammering the filesystem.

Bucket variable sequence lengths

For NLP workloads with wildly varying token counts, group samples into buckets of similar lengths before batching. That keeps padding overhead low and ensures CUDA Graphs (or compiled graphs) see mostly static shapes. Rebucket every epoch with a shuffled seed so you still get randomness, and keep a "long sequence" bucket around to exercise worst-case memory paths during profiling.

Use client-side caching when available

Take advantage of any caching layer. If using NFS, increase the client cache size and duration. For distributed filesystems, consider a caching daemon or even manually caching part of the dataset on a local disk. The goal is to avoid repeatedly reading the same data from a slow source. If each node processes the same files at different times, a local cache can drastically cut redundant I/O.

Compress data wisely

Store the dataset compressed if I/O is the bottleneck, but use lightweight compression, such as LZ4 or Zstd fast mode. This trades some CPU to reduce I/O volume. If the CPU becomes the bottleneck due to decompression, consider multithreaded decompression or offloading to accelerators. Also, overlap decompression with reading by using one thread to read compressed data and another thread to decompress the data in parallel. Modern GPUs can perform on-the-fly data decompression using GPU computing resources (or specialized decoders for image/visual data) when paired with GPUDirect Storage and the cuFile I/O stack.

Measure throughput and eliminate bottlenecks

Continuously monitor the data pipeline's throughput. If GPUs aren't near 100% utilization and you suspect input lag, measure how many MB/s you're reading from disk and how busy the data loader cores are. Tools like dstat or NVIDIA's DCGM can reveal if GPUs are waiting on data. Systematically tune each component by bumping up prefetch buffers, increasing network buffer sizes, optimizing disk RAID settings, etc. Do this until the input pipeline can feed data as fast as GPUs consume it. Often, these optimizations raise GPU utilization from ~70% to 〉95% on the same hardware by removing I/O stalls.

Scale I/O for multinode

At cluster scale, ensure the storage system can handle aggregate throughput. For example, 8 GPUs consuming 200 MB/s each is 1.6 GB/s per node. Across 100 nodes, that's 160 GB/s needed. Very few central filesystems can sustain this. Mitigate by sharding data across storage servers, using per-node caches, or preloading data onto each node's local disk. Trading off storage space for throughput (e.g., multiple copies of data) is often worth it to avoid starving expensive GPUs.

Minimize checkpointing and logging overhead

Write checkpoints and logs efficiently. Use asynchronous writes for checkpoints if possible, or write to local disk, then copy to network storage to avoid stalling training. Compress checkpoints or use sparse storage formats to reduce size. Limit logging frequency on each step by aggregating iteration statistics and logging only every Nth iteration rather than every iteration. This will greatly reduce I/O overhead.

Compress and batch data pipeline outputs

If preprocessing expands data (tokenization, augmentation), compress intermediate batches (bf16/FP16, lightweight codecs like LZ4) before copying them over PCIe or the network. Bundle many small samples into contiguous buffers so each DMA saturates bandwidth. Decompress on the GPU or immediately after DMA so the CPU never becomes the choke point. Track end-to-end throughput to make sure compression saves more time than it costs.

Prefetch checkpoints into inference nodes

For high-availability serving, stage the next model version's weights/optimizer state onto each inference node (local NVMe or GPU memory) while the current model runs. When it's time to flip traffic, switch pointers or reuse CUDA Graph captures so the new model goes live instantly, without waiting to copy multi-gigabyte checkpoints over the network.

You can also suspend a running GPU process with cuda-checkpoint and use Checkpoint/Restore in Userspace (CRIU) to persist the process image. When ready to resume, the CUDA driver can restore device memory and CUDA state —even on to other GPUs of the same device type. Treat this as complementary to your model's state-dict or sharded checkpoint files rather than a replacement.

Data Processing Pipelines

The format, layout, and locality of data determine how smoothly the pipeline runs at scale. Binary formats, sharding, caching, and prioritized threads turn I/O from a

bottleneck into a steady stream. Convert datasets to binary or memory-mapped formats, shard across storage and nodes, and raise thread priorities or move simple augments to the GPU to prevent stalls. Cache hot data/KV states, prefetch and buffer aggressively, and size batches to keep the pipeline smooth from disk to device. The following are tips for improving your data processing:

Use binary data formats

Convert datasets to binary formats, such as TFRecords, LMDB, or memory- mapped arrays. This conversion reduces the overhead associated with handling millions of small files and accelerates data ingestion.

Tune the file system

In addition to mounting file systems with noatime and increasing read-ahead, consider sharding data across multiple storage nodes to distribute I/O load and prevent bottlenecks on a single server.

Disable hyperthreading for CPU-bound workloads

For data pipelines that are heavily CPU-bound, disabling hyperthreading can reduce resource contention and lead to more consistent performance. This is especially beneficial on systems where single-thread performance is critical.

Elevate thread priorities

Increase the scheduling priority of data loader and preprocessing CPU threads using tools, such as chrt or pthread_setschedparam. By giving these threads higher priority, you ensure that data is fed to the GPU with minimal latency, reducing the chance of pipeline stalls.

Cache frequently used data

Leverage operating system page caches or a dedicated RAM disk to cache frequently accessed data. This approach is especially beneficial in applications like NLP, where certain tokens or phrases are accessed repeatedly, reducing redundant processing and I/O overhead.

Prefetch and buffer data

Always load data ahead of the iteration that needs it. Use background data loader threads or processes, such as PyTorch DataLoader with prefetch_factor. For distributed training, use DistributedSampler to ensure each process gets unique data to avoid redundant I/O.

Parallelize data transformations

If CPU preprocessing—such as image augmentation and text tokenization—is heavy, distribute it across multiple worker threads/processes. Profile to ensure the CPU isn't the bottleneck while GPUs wait. If it is, either increase workers or move some transforms to GPU, as libraries like NVIDIA's DALI can do image operations on a GPU asynchronously.

Offload lightweight transforms to the GPU

For simple but massive transformations (token casing, normalization, color jitter), launch CUDA/Triton kernels or use DALI immediately after the H2D copy. Run them on dedicated streams so augmentation of batch N+1 overlaps training on batch N. This keeps CPU threads available for heavier parsing work and prevents small Python transforms from throttling the pipeline.

Cache model states and outputs

When inferencing with LLMs, it's beneficial to cache the embeddings and V cache for frequently seen tokens to avoid having to recompute them repeatedly. Similarly, if an LLM training job reuses the same dataset multiple times (called epochs), you should leverage OS page cache or RAM to store the hot data.

Shard data across nodes

In multinode training, give each node a subset of data to avoid every node reading the entire dataset from a single source. This scales out I/O. Use a distributed filesystem or manual shard assignment with each node reading different files. This speeds things up and naturally aligns with data parallelism since each node processes its own data shard. DeepSeek's Fire-Flyer File System (3FS) is one example of a distributed dataset sharding filesystem. DeepSeek's 3FS achieves

multiterabyte-per-second throughput by distributing dataset shards across

NVMe SSDs on each node—while minimizing traditional caching. This design feeds each GPU with local high-speed data, avoiding I/O bottlenecks.

Monitor pipeline and adjust batch size

Sometimes increasing batch size will push more work onto GPUs and less frequent I/O, improving overall utilization—but only up to a point as it affects convergence. Conversely, if GPUs are waiting on data often, and you cannot speed I/O, you might actually decrease batch size to shorten each iteration and thus reduce idle time or do gradient accumulation of smaller batches such that data reads are more continuous. Find a balance where GPUs are nearly always busy.

Apply data augmentation on GPU

If augmentation is simple but applied to massive data, like adding noise or normalization, it might be worth doing on GPU to avoid saturating CPU. GPUs are often underutilized during data loading, so using a small CUDA kernel to augment data after loading can be efficient. But be careful not to serialize the pipeline. Use streams to overlap augmentation of batch N+1 while batch N is training.

Utilize GPU-accelerated libraries like NVIDIA DALI to perform these tasks asynchronously. This helps maintain a smooth and high-throughput data pipeline. Focus on end-to-end throughput (e.g., tokens per second) Remember that speeding up model compute doesn't help if your data pipeline cuts throughput in half. Always profile end-to-end, not just the training loop isolated. Use Nsight Systems and Nsight Compute to measure kernel timelines and stalls, or the PyTorch profiler for framework-level attribution. Then compare iteration time with synthetic versus real data to see how much overhead data loading introduces. Aim for less than 10% overhead from ideal. If it's more than

that, invest time in pipeline optimization; it often yields large "free" speedups in training. Performance Profiling, Debugging, and Monitoring You can't optimize what you don't measure; profiling reveals if you're compute- bound, memory-bound, I/O-bound, or network-bound so you target the right fix. Continuous telemetry and regression tests keep wins from eroding as code, drivers, and data evolve. Specifically, use Nsight Systems/Compute and framework profilers with NVTX to determine whether you're compute-bound, memory-bound, I/O-bound, or communication-bound. Trim Python overhead, watch utilization gaps, balance work across ranks, track memory/network/disk health, and gate changes with performance regression tests and alerts. Use the following guidance to profile, monitor, and debug the performance of your AI workloads:

Profile to find bottlenecks and root cause analysis

Regularly run profilers on your training/inference jobs. Use NVIDIA Nsight Systems to get a timeline of CPU and GPU activity. You can also use Nsight Compute or the PyTorch profiler to drill down into kernel efficiency. Identify whether your job is compute bound, memory bound, or waiting on I/O/communication. Target your optimizations accordingly. For example, if your workload is memory bound, focus on reducing memory traffic rather than implementing compute- bound optimizations. Combine with machine-learning - driven analytics to predict and preempt performance bottlenecks. This can help in automating fine- tuning adjustments in real time. When using GPUDirect Storage, enable GDS tracing to correlate cuFile activity with kernel gaps.

Use a systems-to-kernel profiling flow

Capture the big picture with Nsight Systems first, tagging phases with NVTX so you can see CPU prep, data transfers, and GPU kernels in one timeline. Once you know which kernels or streams dominate the gap, switch to Nsight Compute and profile those kernels in isolation to collect occupancy, warp stall, and throughput metrics. This two-step loop from Chapter 8 keeps you focused on the true limiter instead of guessing from partial data.

Read roofline and automated issue hints

Nsight Compute now ships a Roofline pane and "Issues" recommendations for every captured kernel. Plot achieved FLOPs versus arithmetic intensity to see if you are bandwidth limited or ALU limited, and read the autogenerated hints that flag low load efficiency, branch divergence, or register spilling. Let these reports dictate whether you need tiling, coalescing, or control-flow fixes before you touch the kernel again.

Eliminate Python overhead

Profile your training scripts to identify Python bottlenecks—such as excessive looping or logging—and replace them with vectorized operations or optimized library calls. Minimizing Python overhead helps ensure that the CPU does not become a hidden bottleneck in the overall system performance.

Measure GPU utilization and idle gaps

Continuously monitor GPU utilization, SM efficiency, memory bandwidth usage, etc. If you notice periodic drops in utilization, correlate them with events. For example, a drop in utilization every 5 minutes might coincide with checkpoint saving. Such patterns point to optimization opportunities, such as staggering checkpoints and using asynchronous flushes. Utilize tools like DCGM or nvidia-smi in daemon mode to log these metrics over time.

Use NVTX markers

Instrument your code with NVTX ranges or framework profiling APIs to label different phases, including data loading, forward pass, backward pass, etc. These markers show up in the Nsight Systems or Perfetto timeline and help you attribute GPU idle times or latencies to specific parts of the pipeline. This makes it easier to communicate to developers which part of the code needs attention. For PyTorch, you can use torch.profiler.record_function(). Utilize kernel profiling and analysis tools beyond just the PyTorch profiler For performance-critical kernels, use Nsight Compute to examine kernel-level metrics like occupancy and throughput, or Nsight Systems to analyze GPU/CPU timelines and overlap. Check achieved occupancy, memory throughput, and instruction throughput. Look for signs of memory bottlenecks, such as memory bandwidth near the hardware maximum. This helps to identify memory-bound workloads. The profiler's "Issues" section often directly suggests if a kernel is memory bound or compute bound and why. Use this feedback to guide code changes, such as improving memory coalescing if global load efficiency is low.

Place NVTX ranges strategically

Wrap the exact regions you care about—input staging, forward/backward kernels, communication—rather than blanketing entire functions. Give ranges consistent colors and human-readable names so Nsight timelines tell a story at a glance. Pair NVTX events with CUDA events for precise timings, and propagate range IDs through distributed ranks so you can correlate slow phases across GPUs.

Check for warp divergence

Use the profiler to see if warps are diverging, as it can show branch efficiency and divergent branch metrics. Divergence means some threads in a warp are inactive due to branching, which hurts throughput. If significant, revisit the kernel code to restructure conditionals or data assignments to minimize intrawarp divergence and ensure that each warp handles uniform work.

Verify load balancing

In multi-GPU jobs, profile across ranks. Sometimes one GPU (rank 0) does extra work like aggregating stats and data gathering—and often becomes a bottleneck. Monitor each GPU's timeline. If one GPU is consistently lagging, distribute that extra workload. For example, you can have the nonzero ranks share the I/O and logging responsibilities. Ensuring that all GPUs/ranks have similar workloads avoids the slowest rank dragging the rest.

Monitor memory usage

Track GPU memory allocation and usage over time. Ensure you are not near OOM, which can cause the framework to unexpectedly swap tensors to host, which will cause huge slowdowns. If memory usage climbs iteration by iteration, you have likely identified leaks. In this case, profile with tools like torch.cuda.memory_summary() and Nsight Systems' GPU memory trace to analyze detailed allocations. On the CPU side, monitor for paging, as your process's resident memory (RES) should not exceed physical RAM significantly. If you see paging, reduce dataset preload size or increase RAM.

Monitor network and disk

For distributed jobs, use OS tools to monitor network throughput and disk throughput. Ensure the actual throughput matches expectations. For example, on a 100 Gbps link, you should see 12 GB/s if fully utilized. If not, the network might be a bottleneck or misconfigured. Similarly, monitor disk I/O on training nodes. If you see spikes of 100% disk utilization and GPU idle, you likely need to buffer or cache data better.

Tune distributed collectives

Collective performance depends as much on topology as on math. For intra-node runs, favor NVLink/NVSwitch-aware algorithms such as NCCL's tree or CollNet modes; inter-node jobs often benefit from hierarchical all-reduce (NVLink first, then InfiniBand). Set NCCL environment knobs explicitly (e.g., `NCCL_ALGO=Tree`, `NCCL_COLLNET_ENABLE=1`, `NCCL_NET_GDR_LEVEL=SYS`) and profile link utilization with Nsight Systems or DCGM. Pin communicators to dedicated streams so reductions overlap matmuls, and balance workloads so no single rank is stuck aggregating stats or logging while others sit idle.

Map workloads to NVLink/NVSwitch topology

Large nodes (GB200, H100, etc.) expose multiple NVLink "islands." Use `nvidia-smi topo -m` or vendor topology diagrams to place tensor/model-parallel shards within the same high-bandwidth domain and schedule pipeline/data-parallel groups across islands. When jobs share a node, respect NVLink partitioning: dedicate entire islands to one job or pin ranks with `CUDA_VISIBLE_DEVICES` so cross-island traffic only happens when necessary. This reduces link contention and keeps collectives latency-hiding rather than bottlenecking.

Set up alerts for anomalies

In a production or long-running training context, set up automated alerts or logs for events like GPU errors, such as ECC errors, device overheating, etc. This will help identify abnormally slow iterations. For example, NVIDIA's DCGM can watch health metrics, and you can trigger actions if a GPU starts throttling or encountering errors. This helps catch performance issues—like a cooling failure causing throttling—immediately rather than after the job finishes.

Perform regression testing

Maintain a set of benchmark tasks to run whenever you change software, including CUDA drivers, CUDA versions, AI framework versions, or even your training code. Compare performance to previous runs to catch regressions early. It's not uncommon for a driver update or code change to inadvertently reduce throughput—a quick profiling run on a standard workload will highlight this so you can investigate. For example, maybe a kernel is accidentally not using Tensor Cores anymore. This is something to look into for sure.

GPU Programming and CUDA Tuning Optimizations

Aligning kernels with the memory hierarchy and hardware features is where large, durable gains come from. Fusion, Tensor Cores, CUDA Graphs, and compiler paths (e.g., torch.compile and OpenAI's Triton) convert launch overhead into useful math.

Optimize for the memory hierarchy: coalesce global loads, tile into shared memory, manage registers/occupancy, and overlap transfers (e.g., cp.async/TMA) with compute. Prefer tuned libraries and CUDA Graphs, leverage torch.compile and OpenAI's Triton for fusion, and validate scalability with roofline analysis and PTX/SASS inspection. The following are some GPU and CUDA programming optimization tips and techniques:

Tune Tensor Core math modes

Modern NVIDIA GPUs reserve most of their peak FLOPs for Tensor Cores. Make sure your math settings actually use them: enable TF32 (`cublasSetMathMode(CUBLAS_TF32_TENSOR_OP_MATH)` or `torch.backends.cuda.matmul.allow_tf32 = True`) when FP32 accuracy is sufficient, and default to bf16 (or FP16 with loss scaling) for training loops. Keep accumulators/master weights in FP32 so you retain numerical stability while the hot path runs on Tensor Cores.

Exploit structured sparsity acceleration

Ampere and newer GPUs can double Tensor Core throughput when operands satisfy the 2:4 structured sparsity pattern. If your model can tolerate it, prune weights into 2:4 groups (or train with a sparsity-aware recipe), store metadata alongside the packed values, and use CUTLASS/cuBLASLt or Triton kernels that enable sparse Tensor Core paths. Measure both accuracy and speed—structured sparsity often gives 20%+ extra throughput on transformer blocks with minimal loss when applied carefully.

Automate kernel autotuning pipelines

Wrap CUTLASS/Triton/CUDA kernels in autotuners that sweep `BLOCK_M/N/K`, `num_warps`, prefetch depth, stream counts, and cp.async schedules. Run the sweeps offline on representative shapes and store the winning configs in a runtime registry keyed by `(arch, dtype, shape_bucket)`. Re-run the autotuner whenever you move to new GPUs or encounter out-of-family shapes so you're never stuck with stale launch parameters.

Prefetch and double buffer aggressively

Use `cuda::pipeline`, `cp.async`, or TMA descriptors to stream tiles into shared memory and registers while the current tile computes. Keep at least two staging buffers alive per CTA so global-memory latency is hidden, and size them as `BLOCK_ROWS × VALUES_PER_THREAD`. When the kernel supports it, prefetch one or two tiles ahead per warp so the next dataset is waiting by the time the current tile finishes.

Build asynchronous copy pipelines

Hopper and Blackwell introduce `cp.async` and Tensor Memory Accelerator (TMA) engines that stream tiles into shared memory without stalling warps. Structure kernels with double-buffered stages—stage N computes while stage N+1 issues async copies. Size staging buffers as `BLOCK_ROWS × VALUES_PER_THREAD`, prefetch at least one stage ahead, and use CUDA's `cuda::pipeline` helpers (or CUTLASS/Triton block pointers) to keep the SM fed continuously.

Coordinate pipeline stages in shared memory

When you move to CUDA's pipeline API, allocate shared memory as `[stages × tile_span]` and create a `cuda::pipeline_shared_state` per block so producer and consumer phases share progress. In `ch8/threshold_tma_kernel.cuh` (lines 21–113) each CTA keeps two stage buffers, tracks `stage_ready[]`, and alternates between them via `producer_acquire` / `consumer_release`. This structure hides latency while letting you clamp valid elements on the fly.

Clamp pipeline grids to shared-memory budgets

Async copies need enough work per CTA to amortize shared memory and pipeline state. The Threshold TMA launcher (lines 131–150) limits the grid to 2,048 blocks and snaps it to at least one block so the staging area fits in SRAM and the pipeline stays resident. When you add similar kernels, cap the grid when shared memory per block is large, and compute tile spans from `blockDim.x × values_per_thread` so each launch consumes a predictable budget.

Guard async kernels for toolkit and SM support

TMA and warp-specialized features only exist on recent GPUs. Wrap the kernel and its launch path with `#if CUDA_VERSION >= ...` and `#if __CUDA_ARCH__ >= ...` checks as done in `ch8/threshold_tma_kernel.cuh` (lines 14–120). This guarantees older toolkits fall back cleanly while Hopper-and-newer builds keep the optimized path.

Understand GPU memory hierarchy

Keep in mind the tiered memory structure of GPUs—registers per thread, shared memory/L1 cache per block/SM, L2 cache across SM, and global HBM. Maximize data reuse in the higher tiers. For example, use registers and shared memory to reuse values and minimize accesses to slower global memory. A good kernel ensures the vast majority of data is either in registers or gets loaded from HBM efficiently using coalescing and caching.

Coalesce global memory accesses

Ensure that threads in the same warp access contiguous memory addresses so that the hardware can service them in as few transactions as possible. Strided or scattered memory access by warp threads will result in multiple memory transactions per warp, effectively wasting bandwidth. Restructure data layouts or index calculations so that whenever a warp loads data, it's doing so in a single, wide memory transaction.

Use shared memory for data reuse

Shared memory is like a manually managed cache with very high bandwidth. Load frequently used data—such as tiles of matrices—into shared memory. And have threads operate on those tiles multiple times before moving on. This popular tiling technique greatly cuts down global memory traffic. Be cautious of shared-memory bank conflicts. Organize shared-memory access patterns or pad data to ensure threads aren't contending for the same memory bank, which would serialize accesses and reduce performance.

Optimize memory alignment

Align data structures to 128 bytes whenever possible, especially for bulk memory copies or vectorized loads. Misaligned accesses can force multiple transactions even if theoretically coalesced. Using vectorized types like float2 and float4 for global memory I/O can help load/store multiple values per instruction, but ensure your data pointer is properly aligned to the vector size.

Minimize memory transfers

Only transfer data to the GPU when necessary and in large chunks. Consolidate many small transfers into one big transfer if you can. For example, if you have many small arrays to send each iteration, pack them into one buffer and send once. Small, frequent cudaMemcpy can become a bottleneck. If using Unified

Memory, use explicit prefetch (cudaMemPrefetchAsync) to stage data on GPU before it's needed, avoiding on-demand page faults during critical compute sections.

Avoid excessive temporary allocations

Frequent allocation and freeing of GPU memory can hurt performance. For example, frequently using cudaMalloc/cudaFree or device malloc in kernels will cause extra overhead. Instead, reuse memory buffers or use memory pools available within most DL frameworks, like PyTorch, that implement a GPU caching allocator. If writing custom CUDA code, consider using cudaMallocAsync with a memory pool or manage a pool of scratch memory yourself to avoid the overhead of repetitive alloc/free.

Balance threads and resource use

Achieve a good occupancy-resource balance. Using more threads for higher occupancy helps hide memory latency, but if each thread uses too many registers or too much shared memory, occupancy drops. Tune your kernel launch parameters—including threads per block—to ensure you have enough warps in flight to cover latency, but not so many that each thread is starved of registers or shared memory. In kernels with high instruction-level parallelism (ILP), reducing register usage to boost occupancy might actually hurt performance. The optimal point is usually in the middle of the occupancy spectrum, as maximum occupancy is not always ideal. Use the NVIDIA Nsight Compute Occupancy Calculator to experiment with configurations.

Monitor register and shared-memory usage

Continuously monitor per-thread register and shared-memory consumption

using profiling tools like Nsight Compute. If the occupancy is observed to be below 25%, consider increasing the number of threads per block to better utilize available hardware resources. However, verify that this adjustment does not cause excessive register spilling by reviewing detailed occupancy reports and kernel execution metrics. Register spilling can lead to additional memory traffic and degrade overall performance.

Overlap memory transfers with computation

Overlap memory transfers with computation whenever possible. Use cuda Mem cpyAsync in multiple CUDA streams to prefetch while kernels run. Prefer the Tensor Memory Accelerator for bulk movement to shared memory, and use cp.async for fine-grained staged copies and prefetch. These approaches effectively mask global memory latency by overlapping data transfers with computation, making sure the GPU cores remain fully utilized without waiting for memory operations to complete.

Use bulk prefetching when possible

For predictable patterns, use the cp.async bulk prefetch to L2 path to stage lines early. You can also explicitly load data into registers ahead of use. These proactive methods reduce the delay caused by global memory accesses and make sure that critical data is available in faster, lower-latency storage—such as registers or shared memory—right

when it's needed, thus minimizing execution stalls and improving overall kernel efficiency.

Utilize cooperative groups

Utilize CUDA's cooperative groups to achieve efficient, localized synchronization among a subset of threads rather than enforcing a full block-wide barrier. This technique enables finer-grained control over synchronization, reducing unnecessary waiting times and overhead. By grouping threads that share data or perform related computations, you can synchronize only those threads that require coordination, which can lead to a more efficient execution pattern and better overall throughput.

Collect per-kernel telemetry

Add lightweight instrumentation to critical kernels— `atomicMax` ranges, shared-memory utilization counters, NaN/Inf flags—and read them back periodically. These guardrails catch overflow, divergence, or tiling mistakes early, especially when porting kernels to new architectures. Combine telemetry with automated tests so out-of-range metrics fail fast before regressions reach production.

Optimize warp divergence

Structure your code so that threads within a warp follow the same execution path as much as possible. Divergence can double the execution time for that warp— for example, half the warp (16 threads) taking one branch and half the warp (16 threads) taking another branch. If you have branches that some data rarely triggers, consider "sorting" or grouping data so warps handle uniform cases such that all are true or all are false. Use warp-level primitives like ballot and shuffle to create branchless solutions for certain problems. Treat a warp as the unit of work, and aim for all 32 threads to do identical work in lockstep for maximum efficiency.

Quantify ILP vs. divergence

When experimenting with ILP-heavy kernels (like gating or routing steps), profile branch efficiency and instruction throughput in Nsight Compute. Start from a baseline that intentionally thrashes warp-level control flow—random masks, per-iteration syncs—and log how long each phase takes. Then restructure branches (e.g., sort/gather inputs so each warp handles uniform cases, replace divergent updates with warp-level fused math) and compare warp execution efficiency, issued instruction rate, and achieved occupancy. Pair timings with checksum logging so every optimization pass has a correctness guard.

Capture warp-divergent baselines in the harness

Keep a runnable example that highlights the divergence you are fixing so reviewers can run it themselves. The benchmark in `ch6/baseline_warp_divergence_ilp.py` (lines 32–116) uses `BenchmarkConfig` to fix iterations, wraps the hot loop in a conditional NVTX range, and even forces `torch.cuda.synchronize()` each branch to exaggerate the stall. Pairing this baseline with the optimized variant makes ILP wins obvious and keeps regressions reproducible.

Contrast residency strategies

When illustrating an optimization, keep baselines intentionally naive—single-block grids, per-iteration cudaMemcpy, or serial host staging—while the optimized kernel keeps buffers device-resident, strides across large tiles, and lights up the full SM set. Showing both resident and nonresident approaches side-by-side makes the impact of coalescing, async copies, and occupancy tuning unmistakable.

Leverage warp-level operations

Use CUDA's warp intrinsics to let threads communicate without going to shared memory when appropriate. For example, use __shfl_sync to broadcast a value to all threads in a warp or to do warp-level reductions—like summing registers across a warp—instead of each thread writing to shared memory. These intrinsics bypass slower memory and can speed up algorithms like reductions or scans that can be done within warps. By processing these tasks within a warp, you avoid the latency associated with shared memory and full-block synchronizations.

Use CUDA streams for concurrency

Within a single process/GPU, launch independent kernels in different CUDA streams to overlap their execution if they don't use all resources. Overlap computation with computation—e.g., one stream computing one part of the model while another stream launches an independent kernel like data preprocessing on GPU or asynchronous memcpy. Be mindful of dependencies and use CUDA events to synchronize when needed. Proper use of streams can increase GPU

utilization by not leaving any resource idle—especially if you have some kernels that are light.

Schedule inference work across streams

For serving stacks, dedicate streams per request class (high-priority vs. background) or per model stage (embedding lookup, attention, decoder). Use CUDA events to enforce data dependencies but let the scheduler overlap light kernels (tokenization, logits post-processing) with heavy matmuls. When batching, queue incoming requests per stream and consolidate them just before launch so small bursts don't stall larger batches.

Prefer library functions

Wherever possible, use NVIDIA's optimized libraries, such as cuBLAS, cuDNN, Thrust, and NCCL, for core math and collective operations. For point-to-point GPU data movement in distributed inference, use NIXL where available. You can also use NVSHMEM when you need fine-grained GPU-initiated transfers. These are heavily optimized for each GPU architecture and often approach theoretical "speed of light" peaks. This will save you the trouble of reinventing them. For example, use cuBLAS GEMM for matrix multiplies rather than a custom kernel, unless you have a very special pattern. The libraries also handle new hardware features transparently. AI frameworks like PyTorch (and its compiler) use these optimized libraries under the hood.

Use CUDA Graphs for repeated launches

If you have a static training loop that is launched thousands of times, consider using CUDA Graphs to capture and launch the sequence of operations as a graph. This can significantly reduce CPU launch overhead for each iteration, especially in multi-GPU scenarios where launching many kernels and memcpy's can put extra pressure on the CPU and incur additional latency.

Check for scalability limits

As you optimize a kernel, periodically check how it scales with problem size and across architectures. A kernel might achieve great occupancy and performance on a small input but not scale well to larger inputs, as it may start thrashing L2 cache or running into memory-cache evictions. Use roofline analysis. Compare achieved FLOPS and bandwidth to hardware limits to ensure you're not leaving performance on the table.

Inspect PTX and SASS for advanced kernel analysis

For performance-critical custom CUDA kernels, use Nsight Compute to examine the generated PTX and SASS. This deep dive can reveal issues like memory bank conflicts or redundant computations, guiding you toward targeted low-level optimizations.

Instrument kernels for repeatable timing

Wrap hot kernels with CUDA events (or emit a `TIME_MS:<value>` log line) so every benchmark captures the same measurement. Record the GPU model, driver, CUDA version, and command line alongside the timing so you can rerun identical conditions later. Consistent instrumentation makes it obvious when a new driver, compiler flag, or kernel change improves—or regresses—performance.

Use the PyTorch compiler

Take advantage of PyTorch's torch.compile to fuse Python-level operations into optimized kernels through TorchInductor. The compiler can also reduce launch overhead by integrating CUDA Graphs. Typical gains of about 10% - 40% are common once the optimizations are warmed up. This eliminates interpreter overhead and unlocks compiler-level optimizations. In practice, enabling torch.compile has produced substantial speedups (e.g., 20% - 50% on many models) by automatically combining kernels and utilizing

NVIDIA GPU hardware (e.g., Tensor Cores) more efficiently. Always test compiled mode on your model. While it can massively boost throughput, you should ensure compatibility and correctness before deploying. When graphs are stable, enable CUDA Graphs to reduce per-iteration CPU overhead. Keep static memory pools to satisfy pointer-stability constraints.

Embrace AMP and bf16 for tensor cores

PyTorch makes it trivial to run the math that Tensor Cores want. Wrap heavy regions in `torch.cuda.amp.autocast(dtype=torch.bfloat16)` (or FP16 with loss scaling), keep master weights and reductions in FP32, and enable TF32 matmuls for inference (`torch.backends.cuda.matmul.allow_tf32 = True`). This lights up Tensor Cores automatically while retaining numerical stability, often yielding 20% - 50% higher throughput with minimal code change.

Share workload knobs across code paths

Express total work as `micro_batch_size × micro_batches` and drive both baseline (serial micro-batches) and optimized (fused megabatches) paths from the same config. That keeps apples-to-apples comparisons honest and makes it easy to dial smoke tests vs. full runs without editing source.

Balance gradient accumulation for throughput

When batch size is capped by memory, trade off `micro_batch_size` vs. `accumulation_steps` so each step keeps tensor cores busy without blowing out activation memory. Profile throughput as you sweep accumulation factors; often, a slightly larger micro batch with fewer accumulation steps delivers better performance even if the effective batch stays constant. Update learning-rate schedules to account for the new effective batch size.

Use precision-aware optimizers

Mixed-precision training benefits from optimizers that keep critical statistics in higher precision. Keep first/second moments and master weights in FP32, but cast intermediate math to bf16/FP16 so tensor cores stay busy. For Adam-like optimizers, fuse bias-correction, weight decay, and update steps into a single CUDA/Triton kernel to avoid repeated casts. Monitor loss scaling or dynamic clipping to catch NaNs early.

Trim Python overhead from training loops

Even after kernel tuning, Python dispatch can bottleneck small ops. Use `torch.profiler` with `with_stack=True` to find high-frequency Python frames, then migrate those hot paths into scripted/compiled modules or fused Triton kernels. Batch tiny tensor ops into larger custom autograd Functions, move bookkeeping onto the GPU, and keep host-side loops minimal so CUDA Graph capture has a clean region to replay.

Run asynchronous validation and guardrails

Spin up lightweight validation jobs (lower-frequency eval datasets, gradient-norm checks, NaN detectors) on spare GPUs or background streams so they don't steal from the main training loop. Wire these checks into dashboards and CI so anomalies trigger alerts or automatic rollbacks before they corrupt long-running jobs. This keeps the fast path safe even as you push aggressive mixed-precision and kernel tweaks.

Plan for dynamic shapes

If your input sizes vary, use torch._dynamo.mark_dynamic() to annotate dynamic dimensions or export shape-polymorphic graphs with torch.export(), and then compile. Control recompilation behavior with torch.com piler.set_stance() using "fail_on_recompile" to surface problematic shape churn in testing and CI. Use static shapes where possible to enable CUDA Graphs to reduce per-iteration CPU overhead. Leverage Triton kernels using torch.compile If PyTorch doesn't fuse an operation well, consider writing a custom GPU kernel in Triton and integrating it. PyTorch makes it easy to register a custom GPU kernel with torch.library.triton_op.

Capture CUDA Graphs with static buffers

Allocate persistent input/output tensors, copy fresh data into them each step, and wrap the steady portion of the training loop in `torch.cuda.make_graphed_callables` . CUDA Graph replay removes CPU launch overhead and keeps streams saturated. Reset Philox seeds or apply per-step RNG offsets so stochastic layers remain correct under graph capture.

Use autotuning when available

Enable library autotuning features to maximize low-level performance. For example, set torch.backends.cudnn.benchmark=True when input sizes are fixed. This lets NVIDIA's cuDNN library try multiple convolution algorithms and pick the fastest one for your hardware. The one-time overhead leads to optimized kernels that can accelerate training and inference. If exact reproducibility isn't required, allow nondeterministic algorithms by disabling cudnn.deterministic to unlock these faster implementations.

Reuse tensors via `out=` buffers

Elementwise chains often end up allocating fresh tensors every iteration. When chasing bandwidth wins, favor PyTorch ops that accept `out` buffers (`torch.addcmul` , `torch.mul` , etc.) or safe in-place updates so you recycle storage. Pair this with the CUDA caching allocator (or `torch.cuda.memory.reserve`) to keep working sets device-resident and minimize pressure on the allocator.

Manage autoregressive caches intelligently

Decoder workloads live and die by KV-cache efficiency. Keep caches in persistent device buffers, update them in-place per token, and prefetch the right shards to each GPU before the next batch arrives. When memory gets tight, compress cold layers (FP8/INT8) or spill the oldest cache blocks to host asynchronously while compute streams advance. Log cache hit rates, DMA volume, and spill latency so you know whether to scale memory or improve eviction heuristics.

Trim Python overhead from training loops

Even after kernel tuning, Python dispatch can bottleneck small ops. Use `torch.profiler` with `with_stack=True` to find high-frequency Python frames, then migrate those hot paths into scripted/compiled modules or fused Triton kernels. Batch tiny tensor ops into larger custom autograd Functions, move bookkeeping onto the GPU, and keep host-side loops minimal so CUDA Graph capture has a clean region to replay.

Leverage the read-only path

Mark frequently used constants or coefficients as read-only so the GPU can cache them in the dedicated L1 read-only cache. In CUDA C++, you can use const **restrict** pointers to hint that data is immutable. On modern GPU architectures, the compiler generates cached global loads for const **restrict** qualified pointers. When using AI frameworks and libraries, make sure that lookup tables or static weights are on the device and treated as constant. This optimization reduces global memory traffic and latency for those values, as each SM can quickly fetch them from cache instead of repeatedly accessing slow DRAM.

Kernel Scheduling and Execution Optimizations

Launch overhead and unnecessary syncs create idle gaps that crush throughput. Fusing small kernels and using persistent/dynamic strategies keeps the device busy and latency hidden.

Keep the device busy by minimizing synchronizations, fusing small kernels, and using persistent kernels when launching the same work repeatedly. For irregular tasks, consider GPU dynamic parallelism—but use it judiciously to avoid adding overhead. The following are tips on improving kernel scheduling and execution:

Minimize GPU synchronization calls

Avoid unnecessary global synchronizations that stall GPU progress. Excessive use of cudaDeviceSynchronize() or blocking GPU operations (like synchronous memory copies) will insert idle gaps where neither the CPU nor GPU can do useful work. Synchronize only when absolutely needed. For instance, synchronize when transferring final results or when debugging. By letting asynchronous operations queue up, you keep the GPU busy and the CPU free to prepare further work. This leads to a more continuous execution pipeline.

Fuse small kernels to amortize launch overhead

If you have many tiny GPU kernels launching back-to-back, consider merging their operations to run in a single kernel where possible. Every kernel launch has a fixed cost on the order of tens of microseconds, so combining operations through manual CUDA kernel fusion, XLA fusion, or tools like NVIDIA CUTLASS/Triton for custom ops can improve throughput. Fused kernels spend more time doing actual work and less time in launch overhead or memory round trips. This is especially helpful in inference or preprocessing pipelines where chains of elementwise ops can be executed in one go. Try torch.compile(mode="reduce- overhead") first. The compiler can fuse operation chains and wrap steady regions in CUDA Graphs. This will reduce CPU launch overhead. For unfused hotspots, consider migrating them to Triton kernels and using asynchronous TMA and automatic warp specialization where applicable.

Adopt persistent kernels for work queues

When a workload launches the same kernel thousands of times (streaming batches, work queues, MoE routing), consider a persistent kernel: launch once, keep warps alive, and feed them new work items via global memory or CUDA graphs. You trade a slightly more complex kernel for dramatically lower launch overhead and better cache reuse. Size the persistent grid so each SM keeps at least one CTA resident, and fall back to traditional launches only when work patterns are highly irregular.

Utilize GPU dynamic parallelism for intra-GPU work scheduling

Utilize CUDA's Dynamic Parallelism to let GPU kernels launch other kernels from the GPU without returning to the CPU. In scenarios with unpredictable or iterative work, such as an algorithm that needs to spawn additional tasks based on intermediate results, dynamic parallelism cuts latency by removing the CPU launch bottleneck. For example, a parent kernel can divide and launch child kernels for further processing directly on the device. This keeps the entire workflow on the GPU, avoiding CPU intervention and enabling better overlap and utilization. Use this judiciously, however, as it can introduce its own overhead if overused.

Use persistent kernels for repeated workloads

Use a persistent kernel strategy when a workload involves launching identical kernels in rapid succession, such as processing a work queue or streaming batches with the same computation. A persistent kernel is launched once and remains active, reusing threads to handle many units of work in a loop, rather

than launching a fresh kernel for each unit. This approach trades a more complex kernel design for significantly lower scheduling overhead. By keeping the kernel alive, you avoid repeated launch costs and can achieve higher sustained occupancy. High-performance distributed training and inference systems often employ this technique to

maximize throughput and minimize latency for iterative tasks.

Evaluate thread block clusters

Thread block clusters to keep data close and reduce relaunch overheads. Up to 16 thread blocks can form a cluster on Blackwell (after increasing the non-portable limit). Use cluster-aware synchronization and shared-memory residency to improve locality in persistent-style designs. Profile occupancy vs. residency trade-offs with kernel-level profiling tools like Nsight Compute.

Arithmetic Optimizations and Reduced/Mixed Precision

Lower precisions and sparsity let you trade bits for big speed and memory wins— often with negligible accuracy impact. Mixed precision, TF32/FP8/INT8, and fused scaling exploit hardware math paths to raise throughput per dollar. Specifically, use mixed precision (BF16/FP16) and Tensor Cores for big gains, adopt TF32 for easy FP32 speedups, and evaluate FP8/FP4 where quality allows. Exploit structured sparsity, lower-precision gradients/communications, and INT8/INT4 quantization for inference—fusing scales/activations to preserve accuracy. The following optimization techniques apply to improving the performance of arithmetic computations and utilizing reduced/mixed precision:

Use mixed-precision training

Leverage FP16 or BF16 for training to speed up math operations and reduce memory usage. Modern GPUs have Tensor Cores that massively accelerate FP16/ BF16 matrix operations. Keep critical parts like the final accumulation or a copy of weights in FP32 for numerical stability, but run bulk computations in half- precision. This often gives about a 1.5 - 3.5" speedup (depending on the model and kernel mix, with larger gains on matmul-heavy workloads) with minimal accuracy loss and is now standard in most frameworks with automatic mixed precision (AMP).

Embrace gradient accumulation and activation checkpointing

Detail the use of gradient accumulation to effectively increase the batch size without extra memory usage, and consider activation checkpointing to reduce memory footprint in very deep networks. These techniques are crucial when training models that approach or exceed GPU memory limits.

Favor BF16 instead of FP16 on newer hardware

If available, use BF16 instead of FP16, as it has a larger exponent range and doesn't require loss scaling. Modern GPUs support BF16 Tensor Cores at the same speed as FP16. BF16 will simplify training by avoiding overflow/underflow issues while still gaining the performance benefits of half precision. Exploit FP8, novel precisions, and scaling techniques On modern GPUs, FP8 Tensor Cores provide roughly double the math throughput of FP16 or BF16 on compute-bound kernels while, at the same time, reducing activation and weight bandwidth. Additionally, FP4 (NVFP4) Tensor Cores double the throughput of FP8 and are used for inference with micro tensor scaling (an error-correction technique to maintain accuracy) to raise token throughput. For training, use FP8 with the NVIDIA Transformer Engine and maintain FP16 or FP32 accumulators when required. For inference, evaluate FP8 first and adopt NVFP4 only after calibration shows acceptable quality for your task. It's recommended to use hybrid FP8 (E4M3 for forward activations/weights and E5M2 for gradients) for training. Specifically, consider using E4M3 for the forward pass (e.g., activations and weights) and E5M2 for the backward pass (e.g., gradients). It's often beneficial to use a delayed scaling window of 256 - 1024. For inference, consider NVFP4 after calibration. TE integrates with PyTorch and is supported by modern GPU hardware. Prefer framework TE kernels over ad-hoc FP8 custom operations. End-to-end speedup depends on kernel mix, memory bandwidth, and calibration, so validate accuracy and performance on your model and workload.

Leverage Tensor Cores and Tensor Memory (TMEM)

Make sure your custom CUDA kernels utilize Tensor Cores for matrix ops if possible. This might involve using CUTLASS templates for simplicity. By using Tensor Cores and TMEM-based accumulators, you can achieve dramatic speedups for GEMM, convolutions, and other tensor operations—often reaching near- peak FLOPS of the GPU. Ensure your data is in FP16/BF16/TF32 as needed and aligned to Tensor Core tile dimensions, which are multiples of 8 or 16.

Use TF32 for easy speedup

For 32-bit matrix multiplies, set torch.set_float32_matmul_preci sion("high") to enable TF32 (fast FP32) for operations that are numerically safe in PyTorch. Libraries like cuBLAS and cuDNN will automatically pick optimal Tensor Core code paths on modern GPU hardware. If you force full-precision FP32 with "highest" (instead of "high"), make sure to understand the performance impact.

Exploit structured sparsity

Modern NVIDIA GPUs support 2:4 structured sparsity in matrix multiply, which zeros out 50% of weights in a structured pattern. This allows the hardware to double its throughput. Leverage this by pruning your model. If you can prune weights to meet the 2:4 sparsity pattern, your GEMMs can run ~2" faster for those layers. Use NVIDIA's SDK or library support to apply structured sparsity and ensure the sparse Tensor Core paths are used. This can give a free speed boost if your model can tolerate or be trained with that sparsity, which often requires retraining with sparsity regularization.

Reduce precision for gradients and activations when possible

Even if you keep weights at higher precision, consider compressing gradients or activations to lower precision. For instance, use FP16/BF16 or FP8 communication for gradients. Many frameworks support FP16 gradient all-reduce. Similarly, for activation checkpointing, storing activations in 16-bit instead of FP32 saves memory. Research continues on FP8 and FP4 optimizers and quantized gradients. These help maintain model quality while reducing memory and bandwidth costs. In bandwidth-limited environments, gradient compression in particular can be a game changer. DeepSeek demonstrated this by compressing gradients to train on constrained GPUs.

Use custom quantization for inference

For deployment, use INT8 quantization wherever possible. INT8 inference on GPUs is extremely fast and memory-efficient. Use NVIDIA's TensorRT or quantization tools to quantize models to INT8 and calibrate them. Many neural networks like transformers can run in INT8 with a negligible accuracy drop. The speedups can be 2 - 4" over FP16. On the newest GPUs, also explore and evaluate FP8 or INT4 for certain models to further boost throughput for inference.

Fuse scaling and computing operations when possible

When using lower precision, remember to fuse operations to retain accuracy. For example, Blackwell's FP4 "microscaling" suggests keeping a scale per group of values. Incorporate these fused operations by scaling and computing in one pass —rather than using separate passes, which could cause precision loss. Many of these are handled by existing libraries, so just use them rather than implementing them from scratch.

Advanced Tuning Strategies and Algorithmic Tricks

Algorithmic shifts routinely beat hardware upgrades on ROI by reducing work rather than pushing it faster. Autotuning, FlashAttention, overlap of comm/compute, and sharding unlock scale while cutting waste.

Specifically, autotune kernel and layer parameters, swap in fused/FlashAttention kernels, and overlap communication with computation in distributed training. Scale deep models with pipeline/tensor parallelism and ZeRO sharding, and consider asynchronous updates or pruning/sparsity to trade a little accuracy work for big throughput wins. The following are some advanced performance optimizations and algorithmic tricks:

Autotune kernel parameters

Autotune your custom CUDA kernels for the target GPU. Choosing the correct block size, tile size, unroll factors, etc., can affect performance, and the optimal settings often differ between GPUs' generations, such as Ampere, Hopper, Blackwell, and beyond. Use autotuning scripts or frameworks like OpenAI Triton—or even brute-force search in a preprocessing step—to find the best launch config. This can easily yield 20% - 30% improvements that you'd miss with static "reasonable" settings. Use Triton features in your autotuning loop—for instance, set num_warps and num_stages, enable automatic warp specialization, and test asynchronous TMA layouts. Prefer descriptor/block-pointer APIs for shared- memory staging. Re-benchmark tile shapes when migrating to different hardware, as optimal choices will differ across GPU generations.

Align Triton kernels with hardware limits

Pick `BLOCK_M/N/K` as multiples of 64 or 128 so memory transactions stay coalesced, then tune `num_warps` (4 - 8 for bandwidth-bound, 8 - 16 for compute-bound kernels) and `num_stages` (2 - 3) to balance register pressure against latency hiding. Benchmark candidate configs in isolation with `triton.testing.do_bench` or CUDA events before wiring them into PyTorch. When epilogues are lightweight—bias adds, GELU, scale—fuse them directly inside the Triton kernel so data never makes an extra trip through global memory.

Triton kernel tuning quick reference

- Map `pid_m`, `pid_n`, etc., to contiguous tiles, use `tl.arange` broadcasting for index math, and prefer `cache_modifier=".cg"` when streaming.
- Fuse bias/activation/residual epilogues inside the Triton kernel so data stays in registers/shared memory.
- Use descriptor/block-pointer APIs plus automatic warp specialization for async TMA pipelines.
- Keep fallbacks handy: detect unsupported SMs at import time and route to CUDA extensions or `torch.compile` variants so the optimized logic still runs everywhere.

Use kernel fusion in ML workloads

Utilize fused kernels provided by deep learning libraries. For example, enabling fused optimizers will fuse elementwise ops like weight update, momentum, etc. This will also use fused multihead attention implementations and fused normalization kernels. NVIDIA's libraries and some open source projects like Transformer Engine and FasterTransformer provide fused operations for common patterns, such as fused LayerNorm + dropout. These reduce launch overhead and use memory more efficiently.

Utilize memory-efficient attention like FlashAttention

Integrate advanced algorithms like FlashAttention for transformer models. FlashAttention computes attention in a tiled, streaming fashion to avoid materializing large intermediate matrices, drastically reducing memory usage and increasing speed—especially for long sequences. Replacing the standard attention with FlashAttention can improve both throughput and memory footprint, allowing larger batch sizes or sequence lengths on the same hardware.

Overlap communication and computation

In distributed training, overlap network communication with GPU computation whenever possible. For example, with gradient all-reduce, launch the all-reduce asynchronously as soon as each layer's gradients are ready, while the next layer is still computing the backward pass. This pipelining can hide all-reduce latency entirely if done right. Use asynchronous NCCL calls or framework libraries like

PyTorch's Distributed Data Parallel (DDP), which provide overlapping out of the box. This ensures the GPU isn't idle waiting for the network.

Use pipeline parallelism for deep models

When model size forces you to pipeline across GPUs using tensor parallelism or pipeline parallelism, you can use enough microbatches to keep all pipeline stages busy. Exploit NVLink/NVSwitch to send activations quickly between stages. Overlap and reduce pipeline bubbles by using an interleaved schedule. Some frameworks automate this type of scheduling. The NVL72 fabric is especially helpful here, as even communication-heavy pipeline stages can exchange data at multiterabyte speeds, minimizing pipeline stalls.

Utilize distributed optimizer sharding

Use a memory-saving optimization strategy like Zero Redundancy Optimizer (ZeRO), which shards tensors like optimizer states and gradients across GPUs instead of replicating them. This allows scaling to extreme model sizes by distributing the memory and communication load. It improves throughput by reducing per-GPU memory pressure, avoiding swapping to CPU, and reducing communication volume if done in chunks. Many frameworks like DeepSpeed and Megatron-LM provide this type of sharding. Leverage it for large models to maintain high speed without running OOM or hitting slowdown from swapping.

Train asynchronously when possible

If applicable, consider asynchronous updates. For example, you can use stale stochastic gradient descent (SGD) in which workers don't always wait for one another to share updates. This approach can increase throughput, though it may require careful tuning to not impact convergence. Asynchronous training can provide large performance benefits if done properly.

Incorporate sparsity and pruning

Large models often have redundancy. Use pruning techniques during training to introduce sparsity, which you can exploit at inference—and partially during training if supported. Modern GPU hardware supports accelerated sparse matrix multiply (2:4), and future GPUs will likely extend this feature. Even if you leave training as dense and prune only for inference, a smaller model will run faster and use less memory. This increases cost-efficiency for model deployments. Explore the lottery ticket hypothesis, distillation, or structured pruning to maintain accuracy while trimming model size.

Distributed Training and Network Optimization

At cluster scale, the network becomes the limiter. Untreated, the network can break linear scaling and inflate costs. RDMA/Jumbo frames, hierarchical collectives, affinity, and compression protect bandwidth and tame latency.

Use RDMA (InfiniBand/RoCE) when available; if on Ethernet, tune TCP buffers, enable jumbo frames, and select modern congestion control. Align NIC/CPU affinity, adjust NCCL threads/buffers (and SHARP/CollNet where supported), compress or accumulate gradients, and test the fabric to catch loss or misconfigurations. Follow this guidance to optimize your network for distributed environments such as multi- GPU and multinode model training:

Use RDMA networking when available

Equip your multinode cluster with InfiniBand or RoCE for low latency and high throughput. Ensure NCCL and MPI are using RDMA for training. NCCL will autodetect InfiniBand and use GPUDirect RDMA if available. RDMA bypasses the kernel networking stack and can reduce latency significantly versus traditional TCP. If you only have Ethernet, enable RoCE on RDMA-capable NICs to get RDMA-like performance. On NVLink domain systems (NVL72, GB200/ GB300, etc.), keep collectives on-fabric when possible. Reserve host networking for inter-island links. Align NCCL topology hints with your NVLink/NVSwitch domains.

Tune the TCP/IP stack if using Ethernet

For TCP-based clusters, increase network buffer sizes. Raise /proc/sys/net/ core/ {r,w}mem_max and the autotuning limits (net.ipv4.tcp_{r,w}mem) to allow larger send/receive buffers. This helps saturate 10/40/100 GbE links. Enable jumbo frames (MTU 9000) on all nodes and switches to reduce overhead per packet, which improves throughput and reduces CPU usage. Also consider modern TCP congestion control like BBR for wide-area or congested networks.

Assign CPU affinity for NICs

Pin network interrupts and threads to the CPU core(s) on the same NUMA node as the NIC. This avoids cross-NUMA penalties for network traffic and keeps the networking stack's memory accesses local. Check /proc/interrupts and use irqaffinity settings to ensure, for example, your NIC in NUMA node 0 is handled by a core in NUMA node 0. This can improve network performance and consistency, especially under high packet rates.

Optimize NCCL environment variables for your environment

Experiment with NCCL parameters for large multinode jobs. For example, increase NCCL_NTHREADS, the number of CPU threads per GPU for NCCL, from the default 4 to 8 or 16 to drive higher bandwidth at the cost of more CPU usage. Increase NCCL_BUFFSIZE, the buffer size per GPU, from the default 1 MB to 4 MB or more for better throughput on large messages. If your cluster uses SHARP- capable switches, install the NCCL SHARP plugin and enable CollNet by setting NCCL_COLLNET_ENABLE=1, then use the SHARP plugin variables such as SHARP_COLL_ENABLE_SAT=1 as documented. Expect speedups only when your reductions are large enough and the network fabric supports SHARP offload.

Use gradient accumulation for slow networks

If your network becomes the bottleneck because you are scaling too many nodes linked by a moderate-performance interconnect, use gradient accumulation to perform fewer, larger all-reduce operations. Accumulate gradients over a few minibatches before syncing so that you communicate once for N batches instead of every batch. This trades a bit of extra memory and some model accuracy tuning for significantly reduced network overhead. It's especially helpful when adding more GPUs yields diminishing returns due to communication costs.

Optimize all-reduce topologies

Ensure you're using the optimal all-reduce algorithm for your cluster topology. NCCL will choose ring or tree algorithms automatically, but on mixed interconnects like GPUs connected by NVLink on each node and InfiniBand or Ethernet between nodes, hierarchical all-reduce can be beneficial. Hierarchical all-reduce will first perform the all-reduce operation within the node, then it will proceed across nodes. Most frameworks will perform NCCL-based hierarchical aggregations by default but verify by profiling. In traditional MPI setups, you may consider manually doing this same two-level reduction—first intranode and then internode.

Avoid network oversubscription

On multi-GPU servers, ensure the combined traffic of GPUs doesn't oversubscribe the NIC. For example, eight GPUs can easily generate more than 200 Gbps of traffic during all-reduce, so having only a single 100 Gbps NIC will constrain you. Consider multiple NICs per node and 200/400 Gbps InfiniBand if scaling to many GPUs per node. Likewise, watch out for PCIe bandwidth limits if your NIC and GPUs share the same PCIe root complex.

Compress communication

Just as with single-node memory, consider compressing data for network transfer. Techniques include 16-bit or 8-bit gradient compression, quantizing activations for cross-node pipeline transfers, or even more exotic methods like sketching. If your network is the slowest component, a slightly higher compute cost to compress/decompress data can be worth it. NVIDIA's NCCL doesn't natively compress, but you can integrate compression in frameworks (e.g., gradient compression in Horovod or custom AllReduce hooks in PyTorch). This was one key to DeepSeek's success—compressing gradients to cope with limited internode bandwidth.

Monitor network health

Ensure no silent issues are hampering your distributed training. Check for packet loss (which would show up as retries or timeouts—on InfiniBand, use counters for resend, and on Ethernet, check for TCP retransmits). Even a small packet loss can severely degrade throughput due to congestion control kicking in. Use out-

of-band network tests (like iPerf or NCCL tests) to validate you're getting expected bandwidth and latency. If not, investigate switch configurations, NIC firmware, or CPU affinity.

Efficient Inference and Serving

Serving is a cost-and-latency game—utilization rises through orchestration and batching, not just bigger GPUs. Specialized runtimes, KV cache strategies, and warmups keep throughput high without violating SLOs. Orchestrate for demand with autoscaling, microservices, and dynamic/continuous batching to keep GPUs hot without violating latency SLOs. Use specialized runtimes (vLLM, SGLang, TensorRT-LLM), exploit NIXL and KV cache offloading for disaggregated serving, warm models, and isolate resources to control tail latency. Follow these techniques to improve model inference efficiency and performance:

Orchestrate dynamic resources efficiently

Integrate advanced container orchestration platforms, such as Kubernetes augmented with custom performance metrics. This enables dynamic scaling and balancing workloads based on live usage patterns and throughput targets.

Embrace serverless architectures for inference

Explore serverless architectures and microservice designs for inference workloads, which can handle bursty traffic efficiently and reduce idle resource overhead by scaling down when demand is low.

Optimize batch and concurrency

For inference workloads, find the right batching strategy. For inference workloads, favor dynamic or continuous batching to automatically batch incoming requests. Larger batch sizes improve throughput by keeping the GPU busy, but too large can add latency. Also, run multiple inference streams in parallel if one stream doesn't use all GPU resources—e.g., two concurrent inference batches to use both GPU SMs and Tensor Cores fully.

Leverage NIXL for distributed inference

When serving large models across GPUs or nodes, use the NVIDIA Inference Xfer Library to stream KV cache between prefill and decode workers over RDMA. In the case of NIXL, the large transformer-based KV cache is transferred between nodes. NIXL provides a high-throughput, low-latency API for streaming the KV cache from a prefill GPU to a decode GPU in a disaggregated LLM inference cluster. It does this using GPUDirect RDMA and optimal paths—and without involving the CPU. This reduces tail latency for disaggregated prefill decode serving across nodes.

Offload KV cache if necessary

If an LLM's attention KV cache grows beyond GPU memory, use hierarchical offloading. NVIDIA Dynamo's Distributed KV Cache Manager offloads less frequently accessed KV pages to CPU memory, SSD, or networked storage, while inference engines like TensorRT-LLM and vLLM support paged and quantized KV caches. Reuse caches to lower memory pressure and first-token latency. Validate end-to-end impact because offloaded misses introduce extra I/O latency. This allows inference on sequences that would otherwise exceed GPU memory— and with minimal performance hit thanks to fast NVMe and compute-I/O overlapping. Ensure your inference server is configured to use this if you expect very long prompts or chats. Offloading to disk is better than failing completely.

Serve models efficiently

Use optimized model inference systems, such as vLLM, SGLang, NVIDIA Dynamo, and NVIDIA TensorRT-LLM for serving large models with low latency and high throughput. They should implement quantization, low-precision formats, fusion, highly optimized attention kernels, and other tricks to maximize GPU utilization during inference. These libraries should also handle tensor parallelism, pipeline parallelism, expert parallelism, context parallelism, speculative decoding, chunked prefill, disaggregated prefill/decode, and dynamic request batching—among many other high-performance features.

Monitor and tune for tail latency

In real-time services, both average latency and (long-)tail latency (99th percentile) matter. Profile the distribution of inference latencies. If the tail is high, identify outlier causes, such as unexpected CPU involvement, garbage-collection (GC) pauses, or excessive context switches. Pin your inference server process to specific cores, isolate it from noisy neighbors, and use real-time scheduling if necessary to get more consistent latency.

Warm up to avoid cold-start latency

Warm up the GPUs by loading the model into the GPU and running a few dummy inferences. This will avoid one-time, cold-start latency hits when the first real request comes into the inference server.

Partition resources efficiently for quality of service (QoS)

If running mixed, heterogeneous workloads, such as training and inference—or models with different architectures—on the same infrastructure, consider partitioning resources to ensure the latency-sensitive inference gets priority. This could mean dedicating some GPUs entirely to inference or using MIG to give an inference service a guaranteed slice of a GPU if it doesn't need a full GPU but requires predictable latency. Separate inference from training on different nodes if possible, as training can introduce jitter with heavy I/O or sudden bursts of communication.

Utilize Grace CPU for inference preprocessing

In Grace Blackwell systems, the server-class CPU can handle preprocessing— such as tokenization and batch collation—extremely fast in the same memory space as the GPU. Offload such tasks to the CPU and have it prepare data in the shared memory that the GPU can directly use. This reduces duplication of buffers and leverages the powerful CPU to handle parts of the inference pipeline, freeing the GPU to focus on more compute-intensive neural-network computations.

Tune carefully for edge AI and latency-critical deployments

Extend performance tuning to the edge by leveraging specialized edge accelerators and optimizing data transfer protocols between central servers and edge devices. This will help achieve ultralow latency for time-sensitive applications.

Multinode Inference and Serving

Disaggregating prefill/decode and sharding models lets you handle bigger contexts and more users with higher occupancy. Continuous batching and hierarchical memory/offload maintain flow even under long prompts and heavy concurrency. Specifically, disaggregate prefill and decode across devices, continuously pool tokens across requests, and shard oversized models via tensor/pipeline parallelism. Add hierarchical memory/offload for very long contexts so you serve more without OOMs, trading small latency for much higher capacity. The following performance tips apply to multinode inference and serving:

Disaggregate inference pipelines

Separate the inference workflow into distinct phases, including the "prefill" phase that processes the input prompt through all model layers, and the iterative "decode" phase that generates outputs token by token. Allocate these phases to different resources to allow for independent scaling. This two-stage approach prevents faster tasks from being bottlenecked by slower ones. For large language models, one strategy is to run the full model to encode the prompt, then handle autoregressive decoding on a stage-wise basis, possibly with specialized workers for each phase. By disaggregating the pipeline, you ensure that GPUs continuously work on the portion of the task they're most efficient at, avoiding head-of- line blocking, where one long generation stalls others behind it.

Use continuous batch processing for LLMs

Move beyond simple request batching and use continuous batching strategies to maximize throughput under heavy loads. Traditional dynamic batching groups incoming requests and processes them as a batch to improve GPU utilization. Continuous batching takes this further by dynamically merging and splitting sequences of tokens across requests in real time. Systems like vLLM implement

token pooling, where as soon as any thread is ready to generate the next token, it gets grouped with other ready threads to form a new batch. This approach keeps the GPU at high occupancy at all times and drastically reduces idle periods. The result is significantly higher token throughput and better latency consistency, especially when serving many concurrent users with varying sequence lengths.

Shard models efficiently across GPUs and nodes

For models that are too large to fit into a single GPU's memory, employ model- parallel inference techniques by partitioning the model across multiple GPUs or even multiple servers. This can be done with tensor parallelism, in which it splits each layer's weights and computation across devices, or pipeline parallelism, which splits the model's layers into segments hosted on different GPUs and streams the data through them sequentially. While model sharding introduces communication overhead and some added latency as data must flow between shards, it enables deployment of trillion-parameter models that would otherwise be impossible to serve. Ensure high-speed interconnects, such as NVLink or InfiniBand, between GPUs to make this feasible, and overlap communication with computation where possible. The key is to balance the load so all devices work in parallel and no single stage becomes a bottleneck.

Offload memory for extended contexts

Use hierarchical memory strategies to support inference workloads that demand more memory than GPUs have available. Incorporate memory offloading when serving very large models or long sequence contexts, such as long multiturn conversations and large documents. Less frequently used data, such as old attention KV cache entries or infrequently accessed model weights, can be moved to CPU RAM or even NVMe storage when GPU memory gets tight. Modern inference frameworks can automatically swap out these tensors and bring them back on the fly when needed. While this introduces additional latency for cache misses, it prevents out-of-memory errors and allows you to handle extreme cases. By thoughtfully offloading and prefetching data, you trade a bit of speed for the ability to serve requests with large working sets, achieving a better overall throughput under memory constraints.

Power and Thermal Management

Performance per watt is a first-class metric—thermal or power throttling erases tuning gains and shortens hardware life. Power caps, efficient packing, and proactive cooling stabilize clocks while cutting energy spend. Track perf/watt and thermals alongside speed: cap power or underclock memory- bound workloads for better efficiency with minimal throughput loss. Proactively manage cooling, consolidate jobs to run GPUs near full, monitor per-GPU power

draw, and schedule around energy price/renewables when it reduces cost. Here are some tips on managing your power and thermal characteristics of your AI systems:

Utilize efficient and environmentally friendly energy when possible

Track and optimize energy consumption alongside performance. In addition to managing power and thermal limits, monitor energy usage metrics and consider techniques that improve both performance and sustainability. For example, by implementing dynamic power capping or workload shifting based on renewable energy availability, you can reduce operational costs and carbon footprint. This dual focus reduces operational costs and supports responsible, environmentally friendly AI deployments.

Monitor thermals and clocks

Keep an eye on GPU temperature and clock frequencies during runs. If GPUs approach thermal limits (85°C in some cases), they may start throttling clocks, which reduces performance. Use nvidia-smi dmon or telemetry to see if clocks drop from their max. If you detect throttling, improve cooling, increase fan speeds, improve airflow, or slightly reduce the power limit to keep within a stable thermal envelope. The goal is consistent performance without thermal-induced dips.

Use energy-aware dynamic power management

Modern data centers are increasingly using energy-aware scheduling to adjust workloads based on real-time energy costs and renewable energy availability. Incorporating adaptive power capping and dynamic clock scaling can help optimize throughput per watt while reducing operational costs and carbon footprint.

Optimize for perf/watt

In multi-GPU deployments where power budget is constrained (or energy cost is high), consider tuning for efficiency. Many workloads, especially memory-bound ones, can run at slightly reduced GPU clocks with negligible performance loss but noticeably lower power draw. For example, if a kernel is memory bound, locking the GPU at a lower clock can save power while not hurting runtime. This increases throughput per watt. Test a few power limits using nvidia-smi -pl to see if your throughput/watt improves. For some models, going from a 100% to 80% power limit yields nearly the same speed at 20% less power usage.

Tune GPU clocks deliberately

Use `nvidia-smi -lgc` or vendor APIs to sweep SM/memory clocks and find the knee of the curve. Compute-bound kernels benefit from higher SM clocks; memory-bound workloads often plateau well below max and can run at 80 - 90% without losing throughput. Record power, thermals, and performance for each setting so you can pick the best perf/watt point (or enforce a lower cap when rack power is constrained).

Enforce QoS in multi-tenant clusters

When GPUs are shared across services, throttle noisy neighbors before they tank latency for high-priority jobs. Use MIG partitions or gang-schedule whole GPU slices per tenant, enforce per-job SM and memory caps, and pin critical services to untouched partitions. Monitor queueing delay and preempt background work when latency budgets slip.

Isolate workloads with MIG and scheduler policies

On A100/H100/H200-class parts, carve GPUs into MIG slices aligned with application SLAs (e.g., dedicate 1g.5gb to low-latency inference, 4g.40gb to training). Teach the cluster scheduler to place compatible jobs on the same physical GPU and avoid mixing latency-sensitive slices with bandwidth-heavy ones. Track per-slice utilization and thermals so you can rebalance partitions before contention builds.

Use adaptive cooling strategies

If running in environments with variable cooling or energy availability, integrate with cluster management to adjust workloads. For instance, schedule heavy jobs during cooler times of the day or when renewable energy supply is high—if that's a factor for cost. Some sites implement policies to queue nonurgent jobs to run at night when electricity is cheaper. This doesn't change single-job performance but significantly cuts costs.

Consolidate workloads

Run GPUs at high utilization rather than running many GPUs at low utilization. A busy GPU is more energy efficient in terms of work done per watt than an idle or lightly used GPU. This is because the baseline power is better amortized when the GPU is busy. It may be better to run one job after another on one GPU at 90% utilization than two GPUs at 45% each in parallel—unless you need to optimize for the smallest wall-clock time. Plan scheduling to turn off or idle whole nodes when not in use, rather than leaving lots of hardware running at low utilization.

Configure cooling efficiently

For air-cooled systems, consider setting GPU fans to a higher fixed speed during heavy runs to preemptively cool the GPUs. Some data centers always run fans at the maximum to improve consistency. Ensure inlet temps in the data center are within specifications. Check periodically for dust or obstructions in server GPUs. Clogged fins can greatly reduce cooling efficiency. For water-cooled, ensure flow rates are optimal and water temperature is controlled.

Monitor power carefully

Use tools to monitor per-GPU power draw. nvidia-smi reports instantaneous draw, which helps in understanding the power profile of your workload. Spikes in power might correlate with certain phases. For example, the all-reduce phase might measure less compute load and less power, while dense layers will spike the load and power measurements. Knowing this, you can potentially sequence workloads to smooth power draw. This is important if operating the cluster on a constrained power circuit. In the power-constrained scenario, you may need to avoid running multiple power-spikey jobs simultaneously on the same node to avoid tripping power limits.

Improve job resilience for long-running jobs

If you are running a months-long training job or 24-7 inference job, consider the impact of thermals on hardware longevity. Running at 100% power and thermal limit constantly can marginally increase failure risk over time. In practice, data center GPUs are built for this type of resiliency, but if you want to be extra safe, running at 90% power target can reduce component stress with minimal slowdown. It's a trade-off of longer training runs versus less wear on the hardware— especially if that hardware will be reused for multiple projects over a long period of time.

Effective Benchmarking Playbook

Reliable speedups only matter when measurements are disciplined and reproducible. Use this playbook to keep baselines honest, optimized variants consistent, and every data point auditable.

Align workloads before timing

Feed baseline and optimized paths the same tensor shapes, precisions, RNG seeds, and data staging. Clamp randomness and preprocessing so any timing delta comes from the optimization rather than a lucky input distribution.

Warm up for steady state

Compile custom extensions up front, run enough warmup iterations to clear allocators, and pin CUDA events to the measured stream. Separate setup from the timed region so first-iteration noise never pollutes reported numbers.

Report throughput and latency together

Log tokens per second or gigabytes per second next to wall-clock milliseconds. Improvements in one dimension can mask regressions in the other. Recording both makes tradeoffs obvious during reviews.

Cache tuning artifacts

Store autotuned tile sizes, compiled TorchDynamo or Triton graphs, TMA descriptors, and NCCL communicators. Reload them between runs and only retune when batch shapes or sequence lengths cross a new regime.

Model realistic baselines

Let baseline kernels exhibit the real inefficiencies you plan to fix, including host staging, tiny grids, redundant synchronizations, or single-stream launches. Avoid fake delays. Authentic baselines make speedups believable.

Validate outputs aggressively

Guard every fast path with CPU or FP32 reference checks, checksums, or tolerance asserts. Run them on smoke tests and full benchmarks so optimizations never trade correctness for speed.

Stage, overlap, and keep data resident

Chunk activations or tiles, use double buffering, and overlap transfers with compute through streams or CUDA Graph replay. Keep data device resident whenever possible so tuned kernels are not bottlenecked by needless PCIe round trips.

Log the environment every run

Capture GPU model, driver version, CUDA toolkit, compiler flags, command line, git SHA, and benchmarking mode labels such as smoke versus full. Store RNG seeds and input descriptions alongside the metrics so others can rerun the experiment exactly.

Archive benchmark artifacts

Write raw JSON or CSV measurements to timestamped directories, snapshot the console log, and refresh summary tables whenever a kernel or graph changes. Having the raw files makes regressions traceable months later.

Parameterize benchmark configs

Keep every harness knob (iterations, warmup, deterministic flags, seeds, timeout multipliers, profiler toggles) in a single config object so baselines and optimizations run under identical rules. The shared BenchmarkConfig in `common/python/benchmark_harness.py` (lines 157–253) pulls defaults from one source and lets you flip deterministic or NVTX settings per run without touching the benchmark code.

Gate NVTX instrumentation automatically

Wrap NVTX ranges behind a helper that checks whether profiling is enabled before touching torch.cuda.nvtx. The `nvtx_range` context manager in `common/python/nvtx_helper.py` (lines 1–118) filters the known threading warning and turns into a no-op when tracing is off. Benchmarks like `ch6/baseline_warp_divergence_ilp.py` (lines 57–96) query `get_nvtx_enabled` once per run so the hot loop stays clean when you are not collecting traces.

Autotune once and cache the winner

When your kernel supports multiple tile shapes, measure a small candidate set on first use, pick the fastest, and cache that choice for the current device and problem size. The Threshold TMA pipeline launcher in `ch8/threshold_tma_kernel.cuh` (lines 168–217) records CUDA event timings for `ValuesPerThread` options, caches the best variant per `(device, count)`, and reuses it on future calls. This keeps benchmarks fast while still adapting to new shapes.

Follow a repeatable workflow

1. Define matched baseline and optimized pairs.
2. Warm up into steady state.
3. Apply the targeted CUDA, Triton, or PyTorch optimizations.
4. Measure smoke and full runs, capturing throughput and latency.
5. Validate outputs against trusted references.
6. Summarize findings with speedup and applied knobs.
7. Trigger automation that rebuilds, reruns, and guards against regressions.

Maintain golden benchmark suites

Keep a small set of representative workloads (single-node, multi-node, inference, training) and rerun them whenever you change drivers, CUDA, frameworks, or kernel code. Store baseline metrics and automatically diff new runs so regressions surface immediately. A few disciplined golden tests catch most performance slips long before they reach production.

Wire performance gates into CI/CD

Automate the golden-suite runs as part of CI/CD so no pull request merges without meeting latency/throughput budgets. Store historical metrics, visualize trends, and fail the build when a regression exceeds tolerance. This keeps the checklist alive—optimizations stay sticky because every change must prove it doesn't undo previous gains.

Keep a postmortem log for regressions

Whenever a performance regression slips through, write a short postmortem describing the root cause, detection gap, and new guardrails (tests, telemetry). A living log helps the team spot patterns—e.g., driver upgrades or shape polymorphism repeatedly causing trouble—and proactively strengthens future optimization work.

Chapter Pattern Playbook

- **Chapter 6 – ILP & launch bounds:** Baselines serialize micro-batches and relaunch tiny grids; optimized kernels keep data resident, stride across large tiles, and overlap multiple streams to prove occupancy gains.
- **Chapter 7 – Copy/transpose/Triton:** Start with redundant gathers or undersized tiles, then move to coalesced vector loads, padded shared-memory tiles, and Triton

kernels tuned per SM target for decisive bandwidth wins.

- **Chapter 8 - HBM & TMA pipelines:** Contrast host→device staging loops with cp.async/TMA double buffers that reuse resident tiles and overlap producer/consumer stages.
- **Chapter 9 - Bank conflicts & GEMM:** Let baselines hammer the same shared-memory banks or rely on naive matmuls; optimized paths pad shared memory, reuse `out` buffers, and call cuBLAS/cuBLASLt tensor-core routines for order-of-magnitude throughput.
- **Chapters 10 - 12 - Cooperative launches & graph capture:** Baselines relaunch per iteration; optimized flows lean on cooperative grids, CUDA Graph replay, or persistent kernels so launch overhead disappears once work is resident.
- **Chapters 15 - 18 - Expert/pipeline parallelism:** Baselines sync every stage and recompute hidden states; optimized versions reuse caches, overlap streams, and report both tokens/s and latency improvements.
- **Chapter 19 - Memory-aware pipelines:** Baselines shuttle activations through host memory; optimized runs keep contexts resident, push work through Tensor Core matmuls, and capture steady-state loops with CUDA Graphs.

Conclusion

Treat the checklist as a repeatable playbook: profile, tune the right bottleneck at the right layer, and verify gains before scaling out. By methodically applying these practices—from OS and kernels to distributed comms and serving—you'll achieve fast, cost-efficient, and reliable AI systems at any size. This list, while comprehensive, is not exhaustive. The field of AI systems performance engineering will continue to grow as hardware, software, and algorithms evolve. And not every best practice listed here applies to every situation. But, collectively, they cover the breadth of performance engineering scenarios for AI systems.

These tips encapsulate much of the practical wisdom accumulated over years of optimizing AI system performance. When tuning your AI system, you should systematically go through each of the relevant categories listed in this chapter and run through each of the items in the checklist. For example, you should ensure the OS is tuned, confirm GPU kernels are efficient, check that you're using libraries properly, monitor the data pipeline, optimize the training loop, tune the inference strategies, and scale out gracefully.

By following these best practices, you can diagnose and resolve most performance issues and extract the maximum performance from your AI system. And remember that before you scale up your cluster drastically, you should profile on a smaller number of nodes and identify potential scale bottlenecks. For example, if you see an all-reduce collective operation already taking 20% of an iteration on 8 GPUs, it will only get worse at a larger scale—especially as you exceed the capacity of a single compute node or data center rack system, such as the Grace Blackwell GB200 and GB300 NVL72 and Vera Rubin VR200 and VR300 NVL systems.

Keep this checklist handy and add to it as you discover new tricks. Combine these tips and best practices with the in-depth understanding from the earlier chapters, and you will design and run AI systems that are efficient, scalable, maintainable, cost- effective, and reliable. Now go forth and make your most ambitious ideas a reality.

Happy optimizing!!

-Chris Fregly, San Francisco - **Chapters 15 - 18 - Expert/pipeline parallelism:** Baselines sync every stage and recompute hidden states; optimized versions reuse caches, overlap streams, and report both tokens/s and latency improvements. Tune micro-batch counts so each stage has enough work to hide latency but not so many that activation memory explodes—profile stage utilization with Nsight Systems before and after adjustments. - Instrument NCCL timelines with `NCCL_DEBUG=INFO` / `TRACE` or Nsight Systems to correlate slow collectives with SM idle gaps. Watch for stragglers—one rank stuck on disk or CPU can stall the whole all-reduce. - When links saturate, try gradient compression (FP16/bf16 all-reduce, sparsification) or pipeline the reduce so the next micro batch computes while the current gradients drain. Apply compression selectively to the largest tensors to balance accuracy and bandwidth.