제6장. GPU 아키텍처, CUDA 프로그래밍 및 점유율 극대화

이 장에서는 먼저 단일 명령 다중 스레드(SIMT) 실행 모델을 검토하고, 워프, 스레드 블록, 그리드가 GPU 기반 알고리즘을 스트리밍 멀티프로세서(SM)에 매핑하는 방식을 살펴보겠습니다.

현대 NVIDIA GPU에서의 SIMT 실행 모델을 검토하며, 워프, 스레드 블록, 그리드가 SM에 매핑되는 방식을 포함합니다. 이어서 CUDA 프로그래밍 패턴을 심층적으로 다루고, 온칩 메모리 계층 구조(레지스터 파일, 공유/L1, L2, HBM3e)를 논의하며, 텐서 메모리 가속기(TMA)와 텐서 코어 연산의 누산기 역할을 하는 텐서 메모리(TMEM)를 포함한 GPU의 비동기 데이터 전송 기능을 시연합니다.

또한 컴퓨팅 바운드 커널과 메모리 바운드 커널을 식별하기 위한 루프라인 분석을 소개합니다. 이는 현대 GPU 시스템을 이론적 최대 처리량 한계까지 끌어올리기 위한 기초를 제공할 것입니다.

GPU 아키텍처 이해

CPU가 낮은 지연 시간의 단일 스레드 성능을 위해 명령어-명령어 간 연산( )을 최적화하는 것과 달리, GPU는 수천 개의 스레드를 병렬로 실행하도록 설계된 처리량 최적화 프로세서입니다. CPU와 GPU 간 간단한 CUDA 프로그래밍 흐름은 그림 6-1에 표시되어 있습니다.

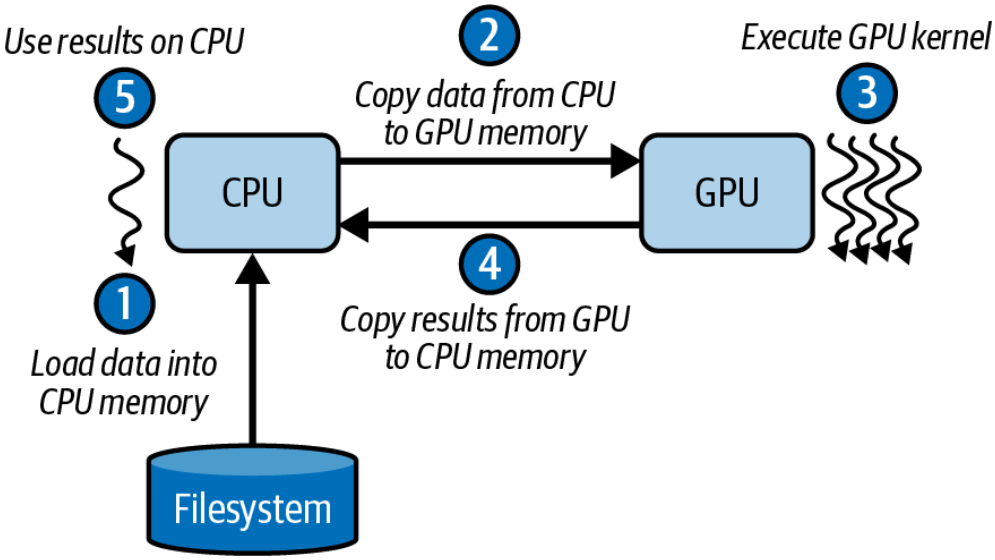

그림 6-1. 간단한 CUDA 프로그래밍 흐름

초기 단계에서 호스트는 데이터를 CPU 메모리에 로드합니다. 이후 CPU에서 GPU 메모리로 데이터를 복사합니다. GPU 메모리의 데이터를 사용하여 GPU 커널을 호출한 후, CPU는 결과를 GPU 메모리에서 CPU 메모리로 다시 복사합니다. 이제 결과는 추가 처리를 위해 CPU에 저장됩니다.

GPU는 그림 6-1에 설명된 CPU-GPU 데이터 전송과 같은 데이터 전송 지연 시간을 숨기기 위해 대규모 병렬 처리에 의존합니다. 각 GPU는 다수의 SM(Streaming Multiprocessor)으로 구성되며, 이는 CPU 코어와 유사하지만 병렬 처리에 최적화되어 있습니다. 각 SM은 Blackwell 아키텍처에서 최대 64개의 워프(32개 스레드 그룹)를 추적할 수 있습니다.

각 GPU에는 다수의 SM이 포함됩니다. 이는 CPU 코어와 유사하지만 처리량에 최적화되어 있습니다. 현대 GPU에서 각 SM은 최대 64개의 워프(2,048개 스레드)를 동시에 추적합니다. Blackwell GPU는 SM당 64K 32비트 레지스터(총 256KB)와 SM당 통합 256KB L1 캐시/공유 메모리를 특징으로 합니다. 해당 SRAM 중 최대 228KB(사용 가능 227KB)를 SM당 사용자 관리 공유 메모리로 구성할 수 있습니다. 단일 스레드 블록은 최대 227KB의 동적 공유 메모리를 요청할 수 있습니다(228KB 중 1KB는 CUDA에 의해 예약됨). 이는 SM이 GPU의 높은 스레드 수준 병렬 처리를 지원하도록 돕습니다.

Blackwell SM 내에서 여러 워프 스케줄러가 사용 가능한 파이프라인에 명령어를 발행합니다. 4개의 독립적인 워프 스케줄러는 매 사이클마다 최대 4개의 워프가 사용 가능한 파이프라인에 명령어를 발행할 수 있게 합니다. 또한 각 스케줄러는 워프당 두 개의 독립적인 명령어(예: 산술 연산 하나와 메모리 작업 하나)를 발행할 수 있는 듀얼 발행 기능을 지원합니다. 단, 듀얼 발행은 동일한 워프 내에서만 가능하며 워프 간에는 불가능합니다.

최상의 경우, 각 스케줄러의 하나의 워프가 매 사이클마다 동시에 명령어를 발행할 수 있어, 사이클당 최대 4개의 워프가 병렬로 실행될 수 있습니다. 이는 그림 6-2에서 보여지듯 명령어 혼합이 활용될 때 처리량을 더욱 향상시킵니다.

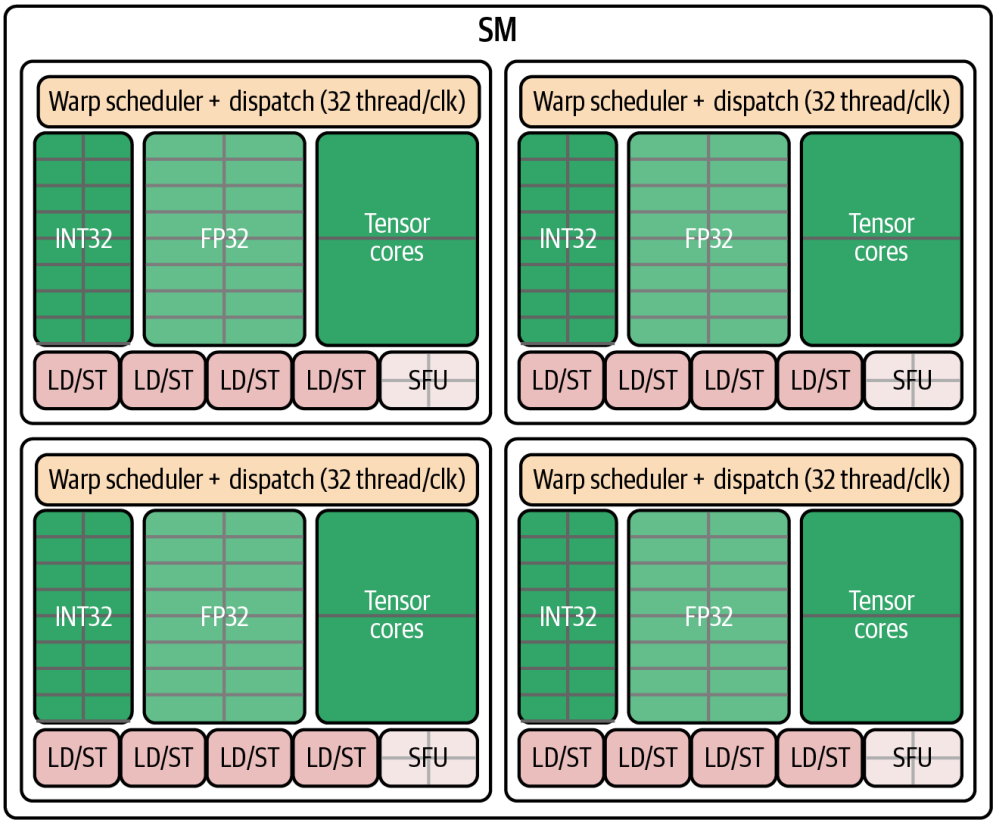

그림 6-2. Blackwell SM에는 네 개의 독립적인 워프 스케줄러가 포함되어 있으며, 각 스케줄러는 사이클당 하나의 워프 명령어를 발행할 수 있고, 스케줄러당 하나의 수학 연산과 하나의 메모리 작업을 이중 발행할 수 있습니다.

여기서 각 SM은 네 개의 독립적인 스케줄링 파티션으로 세분화되며, 각 파티션은 자체 워프 스케줄러와 디스패치 로직을 갖습니다. SM을 온칩 리소스를 공유하는 네 개의 "미니-SM"으로 생각할 수 있습니다. 이를 통해 하드웨어는 준비된 워프를 선택하고 클록 사이클당 최대 네 개의 서로 다른 워프에서 명령어를 발행할 수 있습니다.

네 개의 "미니 SM" 파티션 각각 내에서 스케줄러는 동일한 워프에서 사이클당 두 개의 명령어를 발행할 수 있습니다: 하나는 산술 명령어(예: INT32, FP32 또는 텐서 코어)이고, 다른 하나는 메모리 명령어(로드 또는 스토어)입니다. 이것이 스케줄러가 *듀얼 이슈(dual-issue)*라고 불리는 이유입니다. 표 6-1은 이러한 수치를 요약합니다.

표 6-1. 주요 SM 스케줄러 및 명령어 발행 제한(클록 사이클당)

메트릭	값
스케줄러 수	4
최대 발행 워프 수	네 개 (스케줄러당 하나)
최대 수학 연산	4개 (스케줄러당 산술 명령 1개)
최대 메모리 연산	네 개 (스케줄러당 로드/스토어 발행당 하나)

참고: 모든 메트릭스 표의 수치 값은 개념 설명을 위한 예시입니다. 다양한 GPU 아키텍처의 실제 벤치마크 결과는 GitHub 저장소를 참조하십시오.

따라서 최상의 경우 매 사이클마다 네 개의 워프에 걸쳐 네 개의 연산 명령어와 네 개의 메모리 명령어를 듀얼 발행할 수 있습니다. 이는 연산 처리량과 메모리 처리량을 동시에 극대화합니다. 이러한 수치는 SM의 4방향 분할 구조와 각 분할당 하나의 워프를 선택하여 매 사이클마다 두 개의 직교 명령어를 발행할 수 있는 능력에서 비롯됩니다. 특수 기능 유닛( )

특수 기능 유닛(SFU)은 정수 연산( ), 부동 소수점 연산(FP32), 텐서 코어 파이프라인과 병렬로 배치됩니다. SFU는 초월 함수 연산(예: 사인, 코사인, 역수, 제곱근)을 처리합니다. 그러나 이 유닛들은 이중 발행되는 수학 및 메모리 명령어 쌍에 포함되지 않습니다. SFU는 메인 INT32/FP32 및 로드/스토어(LD/ST) 파이프라인과 독립적으로 작동하는 전용 SFU 파이프라인을 사용합니다.

SFU는 별도의 파이프라인을 차지하며 필요 시 병렬 실행이 가능하므로, SM은 느린 기능이 완료될 때까지 기다리지 않고 수학 및 메모리 명령어를 계속 발행할 수 있습니다. 이러한 분리는 혼합 연산 커널의 명령어 수준 병렬성과 전체 처리량을 더욱 향상시킵니다. SFU는 복잡한 수학 연산이 핵심 연산 및 메모리 파이프라인을 지연시키는 것을 방지합니다.

스케줄러가 4개 존재하며 각 스케줄러는 일반적으로 사이클당 하나의 워프 명령어를 발행할 수 있으므로, 충분한 독립 작업과 발행 페어링이 존재할 경우 사이클당 최대 4개의 워프가 전진 진행할 수 있습니다. 예를 들어, 메모리 작업은 SM의 통합된 16개 로드/스토어(LD/ST) 파이프라인(스케줄러당 4개 LD/ST 파이프라인)을 통해 처리됩니다. 이 파이프라인들은 L1/공유 메모리, L2 캐시 또는 글로벌 메모리(후속 섹션에서 다룸)에 데이터를 읽거나 쓰게 됩니다.

정확한 LD/ST 파이프라인 개수 및 페어링은 보장되지 않습니다. 커널이 메모리 발행 문제인지 계산 문제인지 판단하려면 자질 카운터를 활용하십시오. 아키텍처별 세부 사항은 NVIDIA 문서를 참조하십시오. Blackwell 튜닝 가이드가 좋은 시작점이 될 것입니다.

요약하면, GPU는 대규모 행렬 곱셈, 컨볼루션 및 동일한 명령이 다수 요소에 적용되는 기타 연산을 포함한 데이터 병렬 워크로드에 탁월합니다. 개발자는 CUDA C++로 직접 커널을 작성하거나 PyTorch와 같은 고수준 프레임워크, OpenAI의 Triton과 같은 도메인 특화 Python 기반 GPU 언어를 통해 간접적으로 작성합니다.

커널 개발과 메모리 접근 최적화를 살펴보기 전에, 이러한 모든 작업의 기반이 되는 CUDA 스레드 계층 구조와 핵심 용어를 복습해 보겠습니다.

스레드, 워프, 블록 및 그리드

CUDA는 병렬 작업 을 스레드, 스레드 블록(*협동 스레드 배열* [CTA]이라고도 함), 그리드라는 3단계 계층 구조로 구성하여 프로그래밍 가능성과 대규모 처리량 사이의 균형을 맞춥니다. 최하위 수준에서는 각 스레드가 커널 코드를 실행합니다. 현대 GPU에서는 스레드를 최대 1,024개까지 포함하는 스레드 블록으로 그

룹화합니다. 커널을 실행하면 스레드 블록이 그리드를 형성하며, 이는 그림 6-3에서 확인할 수 있습니다.

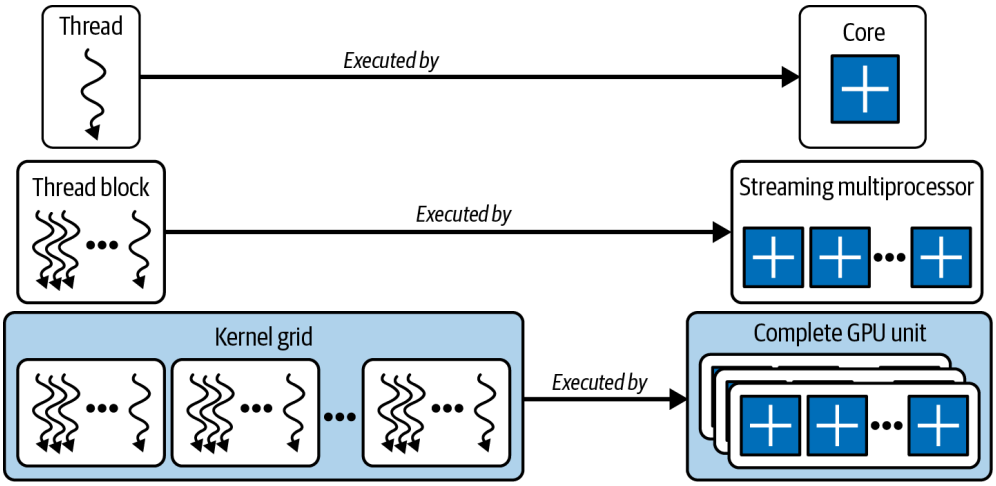

그림 6-3. 스레드 , 스레드 블록(CTA라고도 함), 그리드

그리드 크기를 적절히 조정하면 커널 로직 변경 없이 수백만 스레드로 확장할 수 있습니다. CUDA 런타임(및 PyTorch 같은 프레임워크)이 모든 SM(Streaming Multiprocessor)에 걸친 스케줄링과 분배를 처리합니다. 그림 6-4는 GPU 장치에서 실행되는 CUDA 커널을 호출하는 CPU 기반 호스트를 포함한 스레드 계층 구조의 또 다른 관점을 보여줍니다.

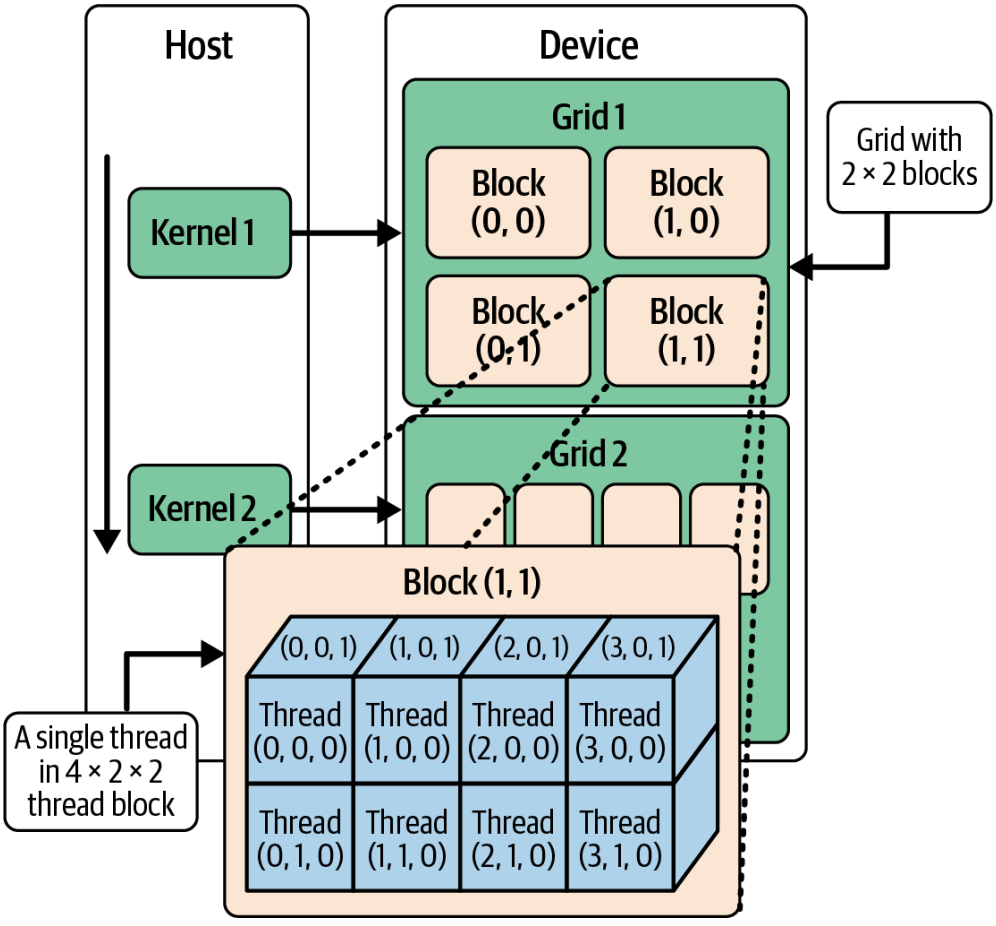

그림 6-4. GPU 장치에서 실행되는 커널을 런칭하는 CPU 기반 호스트를 포함한 스레드 계층 구조보기

기존에는 서로 다른 스레드 블록( )의 스레드들이 직접 상호작용할 수 없었습니다. 그러나 현대 GPU 아키텍처와 CUDA 버전은 스레드 블록 클러스터를 지원합

니다. 스레드 블록 클러스터는 서로 다른 SM 간에 통신할 수 있는 스레드 블록들의 집합입니다.

구체적으로, 스레드 블록 클러스터 내에서 서로 다른 스레드 블록의 CUDA 스레드들은 서로의 공유 메모리에 접근할 수 있으며 하드웨어 지원 클러스터 범위 배리어를 사용할 수 있습니다. 이를 통해 오늘날 대규모 LLM 워크로드에서 매우 흔한 행렬 곱셈을 포함한 훨씬 더 큰 연산 작업을 수행할 수 있습니다. 스레드 블록 클러스터는 그림 6-5에 표시된 바와 같이 클러스터에 참여하는 SM들 간에 분산 공유 메모리(DSMEM) 주소 공간을 공유합니다.

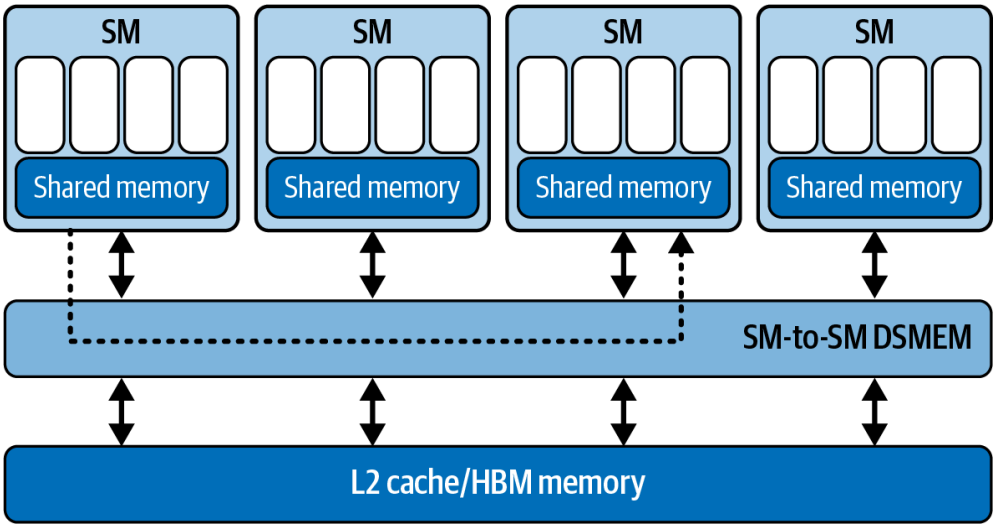

그림 6-5. 다중 스레드 블록을 포함하는 스레드 블록 클러스터에서 사용되는하드웨어 지원 DSMEM

DSMEM은 모든 SM의 공유 메모리 뱅크를 고속 온칩 상호 연결을 통해 스레드 블록 클러스터로 연결하는 하드웨어 기능입니다. DSMEM을 통해 SM들은 결합된 다중 SM 분산 공유 메모리 풀을 공유합니다. 이러한 통합으로 서로 다른 블록의 스레드들은 온칩 속도로, 글로벌 메모리 대역폭을 사용하지 않고도 서로의 공유 버퍼를 읽고, 쓰고, 원자적으로 업데이트할 수 있습니다.

스레드 블록 클러스터와 DSMEM 같은 고급 주제는 10장에서 다룰 예정입니다. 이는 현대 GPU 처리에서 매우 중요한 추가 기능이며, AI 시스템 성능 엔지니어가 이해해야 할 핵심 사항입니다. 본 장에서는 블록 내 공유 메모리 최적화에 집중하겠습니다.

각 스레드 블록 내에서 스레드들은 저지연 온칩 공유 메모리를 사용하여 데이터를 공유하고, 동기화 장벽(`__syncthreads()`)으로 동기화합니다. 각 장벽은 오버헤드를 발생시키므로, 그림 6-6과 같이 동기화 지점을 최소화해야 합니다.

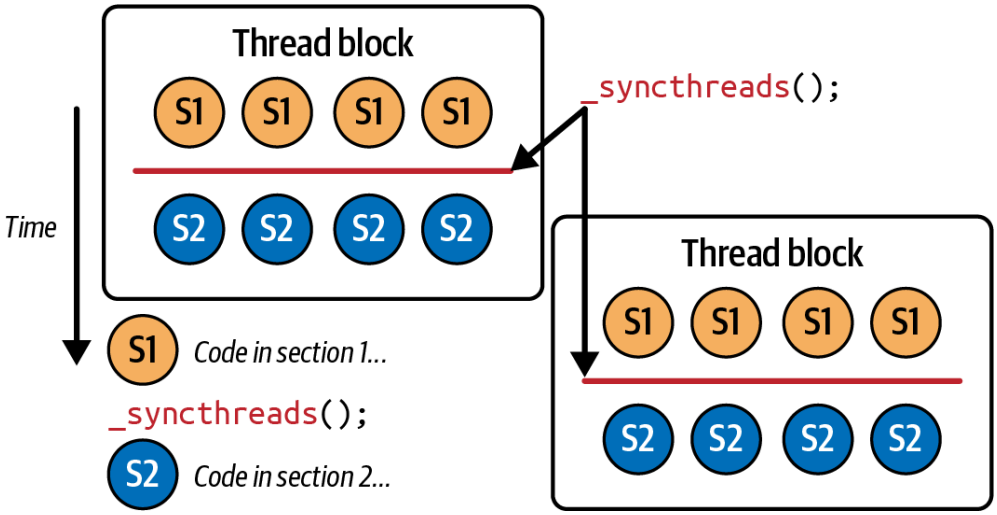

그림 6-6. 두 코드 섹션 간 스레드 블록 내 모든 스레드동기화

목표는 동기화 지점을 최소화하는 것입니다. 그러나 GPU 하드웨어는 글로벌 메모리 로드, 캐시 채우기, 파이프라인 정지 등 장지연 이벤트를 워프 간 빠른 전환으로 숨기려 시도합니다.

스레드 블록은 32개 스레드로 구성된 워프로 세분화되며, 워프 스케줄러를 통해 SIMT 모델 하에서 동기적으로 실행됩니다. 이는 그림 6-7에 표시되어 있습니다.

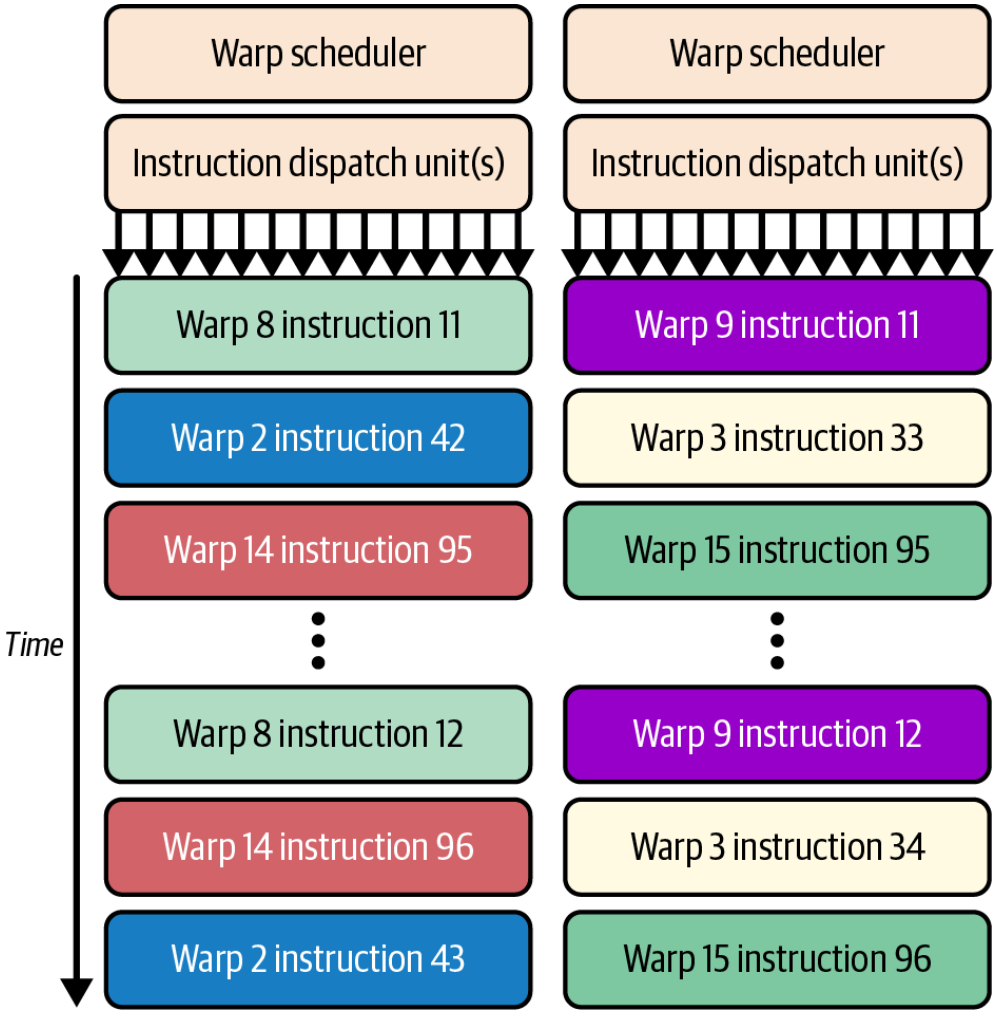

그림 6-7. 워프 스케줄러가 관리하는 명령어로 전체(32개 스레드)가 함께 진행되는워프

더 많은 워프를 동시에 실행 상태 로 유지하는 것을 SM에서의 높은 *점유율(high occupancy*)이라고 합니다. CUDA 코드가 높은 점유율을 허용한다는 것은 한 워

프가 정지할 때 다른 워프가 실행 준비가 되어 있음을 의미합니다. 이는 GPU의 컴퓨트 유닛을 바쁘게 유지합니다.

그러나 높은 점유율은 레지스터나 공유 메모리와 같은 스레드별 리소스 한계와 균형을 맞춰야 합니다. 레지스터를 느린 메모리로 스필링하면 새로운 스톨이 발생할 수 있습니다. 점유율과 레지스터 및 공유 메모리 사용량을 함께 자질하면 리소스 경합을 유발하지 않으면서 처리량을 극대화하는 블록 크기를 선택하는 데 도움이 됩니다.

점유율 튜닝은 8장에서 다루겠지만, SM, 워프, 스레드 등의 맥락에서 이해해야 할 핵심 개념입니다.

스레드 블록은 독립적으로 실행되며 순서는 보장되지 않습니다. 이를 통해 GPU 스케줄러는 모든 SM에 스레드 블록을 분산 배치하여 하드웨어 병렬성을 최대한 활용할 수 있습니다. 이러한 그리드-블록-워프 계층 구조는 향후 더 많은 SM과 스레드를 갖춘 GPU 아키텍처에서도 CUDA 커널이 수정 없이 실행될 것을 보장합니다.

처리량은 워프 실행 효율성에도 좌우됩니다. 워프 내 스레드는 동일한 제어 흐름 경로를 따라야 하며 통합된 메모리 액세스를 수행해야 합니다. 일부 스레드가 분기되어 한 분기는 `if` 경로를, 다른 분기는 `else` 경로를 취하면 워프는 실행을 직렬화하여 각 분기 경로를 순차적으로 처리합니다. 이를 *워프 분기*(warp *divergence*)라고 하며, 그림 6-8에 표시되어 있습니다.

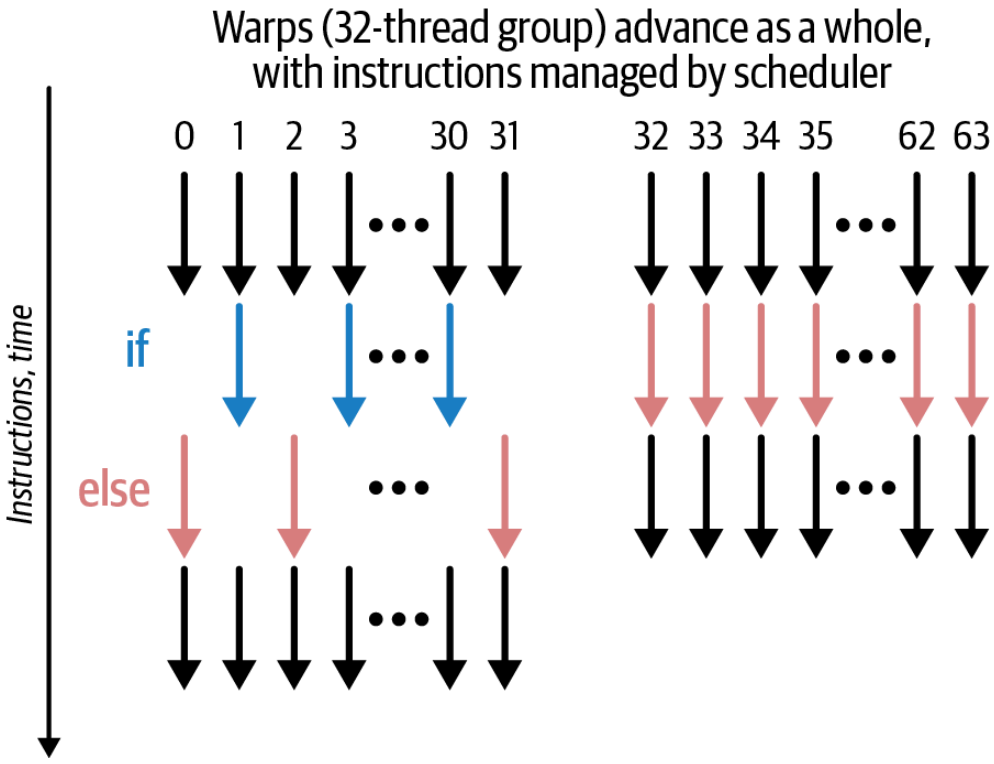

그림 6-8. SIMT 워프 발산도(왼쪽) 대 균일성(오른쪽)

비활성 레인을 마스킹하고 각 분기를 커버하기 위해 추가 패스를 실행함으로써, 워프 분기는 전체 실행 시간을 분기 수만큼 증가시킵니다. 워프 분기에 대한 심층

분석과이를 탐지, 자질, 완화하는 방법은 8장에서다룰 예정입니다.

발산(Divergence)은 단일 워프 내 스레드에만 문제가 됩니다. 서로 다른 워프는 성능 저하 없이 서로 다른 분기 경로를 따를 수 있습니다.

블록당 스레드 수 및 그리드당 블록 크기 선택

GPU 성능 의 핵심 요소 중 하나는 하드웨어의 32스레드 워프 크기와 정렬되는 스레드 블록 크기를 선택하는 것입니다. 따라서 일반적으로 32의 정확한 배수인 스레드 블록 크기를 선택합니다. 예를 들어, 256스레드 블록(8워프 = 256 ÷ 32)은 각 워프를 완전히 점유하지만, 33스레드 블록은 두 개의 워프 슬롯을 차지하며 두 번째 워프 레인의 1/32만 사용합니다. 이는 모든 워프가 스케줄러 슬롯을 차지하기 때문에 병렬 처리 기회를 낭비합니다. 32스레드를 실행 중이든 단 1스레드만 실행 중이든 상관없이 슬롯을 차지하기 때문입니다.

또한 GPU 세대마다 하드웨어 한계가 다릅니다. SM당 최대 스레드 수와 SM당 레지스터 수 등이 이에 해당합니다. 이는 우수한 성능을 유지하려면 블록 크기를 자연스럽게 제한합니다. 예를 들어 블록이 너무 크면 레지스터가 과도하게 필요해져 *레지스터 스필링이* 발생하고 커널 성능이 저하될 수 있습니다.

큰 블록은 GPU 하드웨어에서 유한한 공유 메모리를 지나치게 많이 요구할 수도 있습니다. 구체적으로, Blackwell은 SM당 228KB(사용 가능 227KB)의 공유 메모리만 제공하며, 이는 해당 SM에서 실행 중인 모든 상주 스레드 블록이 주소 지정할 수 있습니다.

이러한 하드웨어 제한은 SM에서 동시에 활성화될 수 있는 블록/워프 수에 영향을 미칩니다. 이는 앞서 소개한 점유율(occupancy)의 측정값입니다. 더 작은 블록은 SM에서 더 많은 동시 워프가 실행될 수 있도록 하여 더 높은 점유율을 가능하게 할 수 있습니다.

사용 중인 GPU 세대의 상대적 규모와 하드웨어 스레드 제한(스레드 수, 스레드 블록, 워프, SM 수 포함)을 이해하는 것이 중요합니다. 그림 6-9는 이러한 리소스의 상대적 규모와 제한 사항을 보여줍니다.

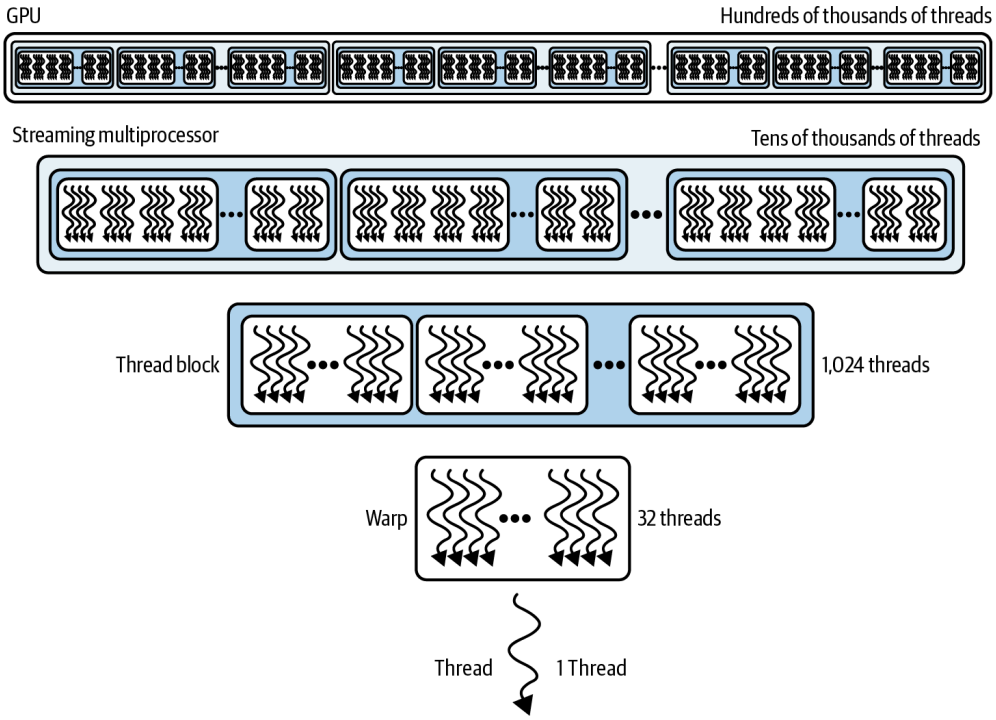

그림 6-9. Blackwell GPU에서 스레드의상대적 규모 및 하드웨어 한계

표 6-2는 Blackwell B200 GPU의 GPU 한계를 요약합니다. 나머지 한계값은 NVIDIA 웹사이트에서 확인할 수 있습니다. (다른 GPU 세대에는 다른 한계값이 적용되므로 시스템의 정확한 사양을 반드시 확인하십시오.)

표 6-2. 스레드 수준 및 블록 수준 제한 (Blackwell B200)

리소스	하드웨어 한계	참고 사항
워프 크기	32개 스레드	기본 SIMT 실행 유닛은 32개 스레드(워프)입니다. 낭비를 피하려면 항상 32의 배수를 사용하십시오.
스레드 블록당 최대 스레드 수	1,024개 스레드	`blockDim.x * blockDim.y * blockDim.z ≤ 1024`.
스레드 블록당 최대 워프 수	32개 워프	(1,024 스레드 ÷ 워프당 32 스레드) = 블록당 최대 32 워프.

우리는 이미 32개 스레드로 제한되는 워프 크기에 대해 논의한 바 있습니다. 이는 블록 크기를 32개 스레드의 배수로 선택하여 "완전한 워프"를 생성하고 활용도가 낮은 워프를 피하도록 권장합니다. 각 블록은 최대 1,024개의 스레드를 가질 수 있으며, 이에 따라 블록은 32개의 워프만 포함할 수 있습니다. 이러한 제한은 점유율에 영향을 미칩니다. 블록이 스케줄링되면 각 SM은 동시에 제한된 수의 워프와 블록만 호스팅할 수 있기 때문입니다.

또한 각 GPU 세대별로 SM당 제한( ) 또는 일반적으로 *SM 상주 제한*으로 불리는 *값이* 존재합니다. 이러한 SM 상주 블랙웰 제한은 표 6-3에 요약되어 있습니다.

표 6-3. SM 상주 리소스 제한 (Blackwell B200)

리소스 (SM당)	하드웨어 제한	참고
SM당 최대 상주 워프 수	64 워프	하드웨어는 최대 64개의 워프를 동시에 실행할 수 있습니다(64 × 32 스레드 = 2,048 스레드). 참고: 이 제한은 여러 세대에 걸쳐 유지되어 왔으며 Blackwell에서도 여전히 유효합니다.
SM당 최대 상주 스레드 수	2,048 스레드	64 워프 × 32 스레드/워프와 동일합니다. 각 블록이 1,024개의 스레드를 사용하는 경우, 최대 2개의 블록(64 워프)이 하나의 SM에 동시에 상주할 수 있습니다. 더 작은 블록(예: 256 스레드)을 사용하면 SM에 더 많은 블록(최대 8개 블록 × 256 = 2,048 스레드)이 상주할 수 있어 점유율을 높이고 지연 시간을 숨기는 데 도움이 될 수 있습니다.
SM당 최대 활성 블록 수	32개 블록	하나의 SM에는 최대 32개의 스레드 블록이 동시에 상주할 수 있습니다(블록이 더 작으면 이 한도까지 더 많은 블록을 수용할 수 있음).

여기서 Blackwell의 SM당 동시 워프 최대 수는 64개임을 확인할 수 있습니다. 이는 최근 GPU 세대에서도 변경되지 않았으므로 점유율 고려 사항이 그대로 적용됩니다. SM당 활성 블록 최대 수는 32개이며, 이에 따라 SM당 상주 스레드 최대 수는 2,048개입니다. SM당 CUDA 그리드도 최대 차원을 가지며, 이는 표 6-4에 표시되어 있습니다.

표 6-4. CUDA 그리드 제한

그리드 차원	제한	참고
X, Y 또는 Z 방향 최대 블록 수	X: 2,147,483,647개 블록 X: 65,535개 블록 Z: 65,535개 블록	3D 그리드는 최대 2,147,483,647 × 65,535 × 65,535 블록까지 생성할 수 있습니다.
동시 실행 가능한 그리드(커널) 최대 수	128개 그리드	단일 장치에서 최대 128개의 커널이 동시에 실행될 수 있습니다(즉, 한 번에 128개의 그리드가 상주).

이론적인 그리드 한계를 아는 것도 중요하지만, 일반적으로는 앞서 설명한 스레드/블록/SM당 한도에 의해 제한받게 됩니다. 한 차원에서 65,535개 이상의 블록이 필요한 경우, 2D 또는 3D 그리드를 실행하여 작업을 여러 커널 실행(멀티런치)으로 분할할 수 있습니다. 후속 섹션에서 이에 대한 예시를 보여드립니다. 실제로는 다른 리소스 한계에 도달하기 전에 그리드 크기 한계에 부딪히는 경우는 드뭅니다.

CUDA GPU의 하위 호환성 및 상위 호환성 모델

CUDA의 핵심 강점 중 하나는 전후방 호환성 모델입니다. 오늘 컴파일된 커널은 일반적으로 향후 GPU 세대에서 수정 없이 실행됩니다. 단, 전방 호환성을 위해 바이너리에 PTX를 포함해야 합니다. PTX 없이 단일 아키텍처용 SASS만 배포하는 경우(예: Hopper용 `sm_90` 또는 Blackwell용 `sm_100`), 해당 바이너리는 최신 아키텍처에서 전방 실행되지 않습니다. `sm_100f` 이나 `compute_100f` 과 같은 패밀리별 타겟은 동일한 기능 패밀리의 장치로만 이식성을 제한합니다. 일반적인 cubin/PTX와 필요한 패밀리별 cubin(예: 최적화 등)을 모두 포함하는 fatbin을 배포하는 것이 가장 좋습니다.

로드 시점에 `CUDA_FORCE_PTX_JIT=1` 를 설정하여 PTX JIT 컴파일을 강제하고 결과를 캐시함으로써 호환성을 검증할 수 있습니다. 바이너리에 PTX가 누락된 경우 커널 실행이 실패합니다. 이는 PTX 지원으로 재빌드하도록 강제합니다. 이 호환성 모델은 방대한 CUDA 생태계의 핵심입니다. 단일 코드베이스로 레거시 하드웨어와 최신 하드웨어를 모두 타겟팅할 수 있게 합니다.

현재 및 향후 GPU 세대 전반에 걸쳐 진정한 하위 호환성과 상위 호환성을 유지하려면 일반 타깃으로 컴파일하거나 PTX를 명시적으로 포함해야 합니다. 최신 하드웨어 기능의 특정 최적화가 필요한 경우 세대별 타깃을 사용할 수 있습니다. 이때 다른 아키텍처에 대한 대체 경로를 반드시 제공해야 합니다.

CUDA 프로그래밍 복습

CUDA C++에서는 커널을 작성하여 병렬 작업 을 정의합니다. 커널은 GPU 장치에서 실행되는 ` `__global__` `로 주석 처리된 특수 함수입니다. CPU(호스트) 코드에서 커널을 호출할 때는 ` `<<< >>>` `"쳌본" 구문을 사용하여 두 가지 구성 매개변수(스레드 블록 수를 지정하는 ` `blocksPerGrid` `및 각 블록 내 스레드 수를 지정하는 ` `threadsPerBlock` `)를 통해 실행될 스레드 수와 조직 방식을 지정합니다.

다음은 CUDA 커널과 커널 실행의 핵심 구성 요소를 보여주는 간단한 예시입니다. 이 커널은 입력 배열의 모든 요소를 그 자리에서 단순히 2배로 늘려주므로 추가 메모리가 생성되지 않습니다. 입력 배열만 사용됩니다. 내부적으로 CUDA는 ` `__global__` `함수를 GPU 장치 코드로 컴파일하여 수천 또는 수백만 개의 경량 스레드가 병렬로 실행할 수 있게 합니다:

```
//------------------------------------------------------
// Kernel: myKernel running on the device (GPU)
//   - input : device pointer to float array of length N
//   - N   : total number of elements in the input
//------------------------------------------------------

__global__ void myKernel(float* input, int N) {
```

```
    // Compute a unique global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Only process valid elements
    if (idx < N) {
        input[idx] *= 2.0f;
    }
}

// This code runs on the host (CPU)
int main() {
    // 1) Problem size: one million floats
    const int N = 1'000'000;
    float *h_input = nullptr;
    float *d_input = nullptr;

    // 1) Allocate input float array of size N on host
    cudaMallocHost(&h_input, N * sizeof(float));

    // 2) Initialize host data (for example, all ones)
    for (int i = 0; i < N; ++i) {
        h_input[i] = 1.0f;
    }

    // 3) Allocate device memory for input on the device
    cudaMalloc(&d_input, N * sizeof(float));

    // 4) Copy data from the host to the device
    cudaMemcpy(d_input, h_input, N * sizeof(float),
      cudaMemcpyHostToDevice);

    // 5) Choose kernel launch parameters

    // Number of threads per block (multiple of 32)
    const int threadsPerBlock = 256;

    // Number of blocks per grid (3,907 for N = 1000000)
    const int blocksPerGrid = (N + threadsPerBlock - 1) /
      threadsPerBlock;

    // 6) Launch myKernel across blocksPerGrid blocks
    // Each block has threadsPerBlock number of threads
    // Pass a reference to the d_input device array
    myKernel<<<blocksPerGrid, threadsPerBlock>>>(d_input,
      N);
    // 7) Wait for the kernel to finish running on device
    cudaDeviceSynchronize();

    // 8) When finished, copy the results
    //    (stored in d_input) from the device back to
    //     host (stored in h_input)
    cudaMemcpy(h_input, d_input, N * sizeof(float),
```

```
                cudaMemcpyDeviceToHost);

    // Cleanup: Free memory on the device and host
    cudaFree(d_input);
    cudaFreeHost(h_input);

    // return 0 for success!
    return 0;
```

이 코드는 완전히 최적화되지 않았습니다. 책의 진행에 따라 성능을 최적화할 예정입니다. 하지만 이 코드는 여러분이 직접 CUDA 커널을 구축하기 시작할 수 있는 간단하고 완전한 템플릿을 제공합니다.

여기서 커널 입력 인자 `d_input`와 `N`를 전달하며, 이 값들은 커널 함수 내부에서 처리할 수 있습니다. 처리는 다수의 스레드에 걸쳐 병렬로 공유됩니다. 이는 설계상 의도된 동작입니다.

전체 데이터 흐름은 다음과 같습니다:

1. 호스트에서 메모리 할당 (`h_input`).
2. 호스트(`h_input`)에서 장치(`d_input`)로 데이터를 복사합니다. 이때 `cudaMemcpy` 함수와 `cudaMemcpyHostToDevice` 매개변수를 사용합니다.
3. `d_input` 로 디바이스에서 커널 실행.
4. 커널이 장치에서 실행을 완료했는지 확인하기 위해 동기화합니다.
5. `cudaMemcpyDeviceToHost` 를 사용하여 `cudaMemcpy` 로 장치(`d_input`)에서 호스트(`h_input`)로 결과를 전송합니다.
6. `cudaFree` 및 `cudaFreeHost` 를 사용하여 장치와 호스트의 메모리를 정리합니다.

커널 실행 시 `<<< >>>` 를 통해 공유 메모리 크기(및 기타 여러 매개변수)를 포함한 추가적인 고급 CUDA 전용 매개변수( )를 전달할 수 있지만, 두 가지 핵심 실행 매개변수인 `blocksPerGrid` 와 `threadsPerBlock` 가 모든 CUDA 커널 호출의 기초가 됩니다. 다음 섹션에서는 이러한 실행 매개변수 값을 최적으로 선택하는 방법에 대해 논의하겠습니다.

그리고 입력 배열의 크기인 `N`를 왜 전달해야 하는지 궁금할 수 있습니다. 커널이 배열 크기를 검사할 수 있어야 하므로 이는 중복처럼 보입니다. 그러나 이는 GPU CUDA 커널 함수와 일반적인 CPU 함수의 핵심 차이점입니다: CUDA 커널 함수는 단일 스레드 내에서, 수천 개의 다른 스레드와 함께, 입력 데이터의 파티션(partition)을 처리하도록 설계되었습니다. 따라서 `N` 은 이 특정 커널이 처리할 파티션의 크기를 정의합니다.

내장 커널 변수 `blockDim`(이 경우 1차원 입력 배열을 전달하므로 1), `blockIdx`, `threadIdx`와 결합하여 커널은 입력 배열에 특정 `idx` 를

계산합니다. 이 고유한 `idx` 덕분에 커널은 서로 다른 여러 SM에서 동시에 실행되는 수많은 스레드에 걸쳐 병렬적으로 입력 배열의 모든 요소를 깨끗하고 고유하게 처리할 수 있습니다.

범위 검사(`if (idx < N)`)에 유의하십시오. 이는 `N` 가 블록 크기의 정확한 배수가 아닐 수 있으므로 범위를 벗어난 접근(범위 검사)을 방지하기 위해 필요합니다. 예를 들어, 입력 배열의 크기가 63인 경우를 고려해 보십시오. 따라서 `N = 63` 입니다. 워프 스케줄러는 입력 배열의 63개 요소를 처리하기 위해 두 개의 워프(각각 32개 스레드)를 할당할 가능성이 높습니다.

첫 번째 워프는 커널 인스턴스 32개를 동시에 실행하여 요소 0–31을 처리하며 `N = 63` 을 초과하지 않습니다. 이는 간단합니다. 첫 번째 워프와 병렬로 실행되는 두 번째 워프는 요소 32–64를 처리할 것으로 예상됩니다. 그러나 `N = 63` 에 도달하면 중단됩니다.

`if (idx < N)` 경계 검사가 없다면, 두 번째 워프는 `idx = 64` 를 처리하려 시도할 것이며, 이는 불법 메모리 접근 오류(예: `cudaErrorIllegalAddress`)를 발생시킬 것입니다. 경계 검사는 각 스레드가 유효한 입력 요소를 처리하거나, 해당 요소의 주소(`idx`)가 범위를 벗어날 경우 즉시 종료되도록 보장합니다.

CUDA 커널은 장치에서 비동기적으로 실행되며( ), 스레드별 예외는 발생하지 않습니다. 대신 모든 불법 작업(범위 초과 접근, 정렬 오류 접근 등)은 전체 런치에 대한 글로벌 결함 플래그를 설정합니다. 호스트 드라이버는 동기화 함수나 다른 CUDA API 함수를 다음에 호출할 때만 해당 플래그를 확인하므로, 오류는 지연적으로 표면화됩니다(예: `cudaErrorIllegalAddress` 또는 일반 런치 실패).

이 설계는 GPU의 파이프라인과 상호 연결을 완전히 점유하지만, 호스트에서 명시적으로 동기화하고 결함을 확인해야 합니다. 일반적으로 커널 실행 직후 `cudaGetLastError()` 및 `cudaDeviceSynchronize()` 를 사용합니다. 이렇게 하면 결함이 발생하자마자 포착할 수 있습니다.

많은 CUDA 커널에서 경계 검사를 확인할 수 있습니다. 이를 발견하지 못한다면, 그 이유가 무엇인지 이해해야 합니다. 어떤 형태로든 경계 검사가 존재하거나, CUDA 커널 개발자가 불법 메모리 접근 오류가 절대 발생하지 않음을 보장할 수 있는 경우일 것입니다.

마지막으로 실제 커널 로직을 살펴보겠습니다. 입력 배열 `idx` 에 대한 고유 인덱스를 계산한 후, 이 커널(수많은 SM에 걸쳐 수천 개의 스레드에서 병렬로 별도 실행됨)은 입력 배열 `idx` 의 인덱스 값에 2를 곱합니다. 그런 다음 입력 배열의 값을 (인플레이스 방식으로) 업데이트합니다. 이 특정 커널에서는 `int` 유형의 임시 변수 `idx` 외에 추가 메모리가 필요하지 않습니다.

실행 매개변수 구성: 그리드당 블록 수 및 블록당 스레드

수

앞서 논의한 바와 같이, 에서 워프 크기(32)의 배수인 블록 크기를 사용하는 것이 중요합니다. `threadsPerBlock` 크기를 256(8개 워프)로 설정하는 것은 점유율과 리소스 사용량 사이의 균형을 맞추기 위한 일반적인 시작점입니다. 이는 커널 실행 중 부분적으로 채워진 워프를 피하고, 숨김 지연 시간을 줄이며, SM 및 기타 하드웨어 리소스 간 균형을 맞추는 데 도움이 됩니다:

32 스레드의 배수

32 스레드의 배수인 블록 크기를 선택하면 빈 워프 슬롯을 피하는 데 도움이 됩니다. 그렇지 않으면 채워지지 않은 워프가 유용한 작업을 수행하지 않으면서도 부족한 스케줄러 자원을 차지하게 됩니다.

지연 시간 숨기기

DRAM 및 명령어 지연 스톨을 숨기기 위해서는 SM당 수백 개의 스레드가 필요합니다. 예를 들어, 2,048개 스레드 용량의 SM에서 256개 스레드로 구성된 8개 블록을 실행하면, 과부하 없이 파이프라인을 바쁘게 유지할 수 있습니다.

점유율

예를 들어 256 `threadsPerBlock` 를 사용하면 블록당 8개의 워프만 필요합니다. 이는 블록당 레지스터나 공유 메모리가 부족해지지 않으면서도 우수한 점유율을 제공하는 경향이 있습니다.

Blackwell과 같은 최신 GPU의 경우, 레지스터 및 공유 메모리 제한을 준수하면서 점유율을 극대화하기 위해 블록당 256~512개의 스레드를 고려하십시오.

자원 균형

256은 블록당 1,024스레드 제한을 거의 초과하지 않을 만큼 충분히 작습니다. 동시에 다른 워프의 스레드가 정지할 때 너무 많은 워프가 유휴 상태로 남지 않을 만큼 충분히 큽니다.

`threadsPerBlock=256` 부터 시작하여 커널의 레지스터 및 공유 메모리 요구 사항과 점유율 특성에 따라 상향 또는 하향 조정(128, 512 등)할 수 있습니다.

`blocksPerGrid` 의 경우, `N` 입력 요소 수와 `threadsPerBlock` 값을 기준으로 설정할 수 있습니다. 예를 들어, `blocksPerGrid` 는 일반적으로 `(N + threadsPerBlock - 1) / threadsPerBlock` 로 설정하여 올림 처리합니다. 이는 `N` 가 `threadsPerBlock` 의 정확한 배수가 아닐 때 모든 요소를 커버하기 위함입니다. 이는 모든 입력 요소가 스레드에 의해 커버되도록 보장하는 일반적인 선택입니다. 계산 과정을 보여주는 코드는 다음과 같습니다:

```
//------------------------------------------------------
// Kernel: myKernel running on the device (GPU)
//   - input : device pointer to float array of length N
//   - N   : total number of elements in the input
//------------------------------------------------------
__global__ void myKernel(float* input, int N) {
    // Compute a unique global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Only process valid elements
    if (idx < N) {
        input[idx] *= 2.0f;
    }
}

// This code runs on the host (CPU)
int main() {
    // 1) Problem size: one million floats
    const int N = 1'000'000;

    float* h_input = nullptr;
    cudaMallocHost(&h_input, N * sizeof(float));

    // Initialize host data (for example, all ones)
    for (int i = 0; i < N; ++i) {
        h_input[i] = 1.0f;
    }

    // Allocate device memory for input on the device (d_)
    float* d_input = nullptr;
    cudaMalloc(&d_input, N * sizeof(float));

    // Copy data from the host to the device using cudaMemcpyHostToDevice
    cudaMemcpy(d_input, h_input, N * sizeof(float),
      cudaMemcpyHostToDevice);

    // 2) Tune launch parameters
    const int threadsPerBlock = 256; // multiple of 32
    const int blocksPerGrid = (N + threadsPerBlock - 1) /
      threadsPerBlock; // 3,907, in this case

    // Launch myKernel across blocksPerGrid number of blocks
    // Each block has threadsPerBlock number of threads
    // Pass a reference to the d_input device array
    myKernel<<<blocksPerGrid, threadsPerBlock>>>(d_input, N);
    // Wait for the kernel to finish running on the device
    cudaDeviceSynchronize();

    // When finished, copy results (stored in d_input) from device to host
    // (stored in h_input) using cudaMemcpyDeviceToHost
    cudaMemcpy(h_input, d_input, N * sizeof(float), cudaMemcpyDeviceToHost
```

```
    // Cleanup: Free memory on the device and host
    cudaFree(d_input);
    cudaFreeHost(h_input);

    return 0; // return 0 for success!
```

이 커널은 이전과 동일하지만, `N` 의 크기에 따라 `blocksPerGrid` 와 `threadsPerBlock` 을 동적으로 계산합니다. 익숙한 `if (idx < N)` 경계 검사에 유의하세요. 이는 최종 블록에서 `N` 범위를 벗어난 "추가" 스레드가 단순히 아무 작업도 수행하지 않도록 보장하며, 불법 메모리 주소 오류를 유발하지 않습니다. 다음으로 2D 이미지나 3D 볼륨과 같은 다차원 입력을 살펴보겠습니다.

2D 및 3D 커널 입력

입력 데이터가 자연스럽게 2차원(예: 이미지)으로 존재할 경우( ), 2차원 블록 그리드를 실행할 수 있습니다. 예를 들어, 16×16 차원의 스레드 블록을 사용하여 총 256개의 스레드로 2차원 1,024×1,024 행렬을 처리하는 커널은 다음과 같습니다:

```
// 2d_kernel.cu

#include <cuda_runtime.h>
#include <iostream>

//------------------------------------------------------
// Kernel: my2DKernel running on the device (GPU)
//   - input  : device pointer to float array of size width×height
//   - width  : number of columns
//   - height : number of rows
//------------------------------------------------------
__global__ void my2DKernel(float* input, int width, int height) {
    // Compute 2D thread coordinates
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Only process valid pixels
    if (x < width && y < height) {
        int idx = y * width + x;
        input[idx] *= 2.0f;
    }
}

int main() {
    // Image dimensions
    const int width  = 1024;
    const int height = 1024;
    const int N      = width * height;

    // 1) Allocate and initialize host image
    float* h_image = nullptr;
```

```
    cudaMallocHost(&h_image, N * sizeof(float));

    for (int i = 0; i < N; ++i) {
        h_image[i] = 1.0f;  // e.g., initialize all pixels to 1.0f
    }

    // 2) Allocate device image and copy data to device
    cudaStream_t s; cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    float* d_image = nullptr;
    cudaMallocAsync(&d_image, N * sizeof(float), s);
    cudaMemcpyAsync(d_image, h_image, N * sizeof(float),
                    cudaMemcpyHostToDevice, s);

    my2DKernel<<<blocksPerGrid2D, threadsPerBlock2D,
                 0, s>>>(d_image, width, height);

    cudaMemcpyAsync(h_image, d_image, N * sizeof(float),
                    cudaMemcpyDeviceToHost, s);
    cudaStreamSynchronize(s);

    cudaFreeAsync(d_image, s);
    cudaStreamDestroy(s);
```

여기서 다시 전체 커널(디바이스) 및 호출(호스트) 코드를 보여드립니다. 동일한 패턴은 3D로 일반화될 수 있습니다. `blocksPerGrid` 및 `threadsPerBlock` 모두에 `dim3(x, y, z)` 를 사용하여 체적 데이터를 GPU의 스레드 계층 구조에 직접 매핑할 수 있습니다.

본서( )에서는 대부분 1차원 또는 2차원(타일링) 값을 사용합니다(`blocksPerGrid`, `threadsPerBlock` 참조). 1차원 경우 `blocksPerGrid` 및 `threadsPerBlock` 을 `dim3` 대신 단순 상수로 정의할 수 있습니다.

비동기 메모리 할당 및 메모리 풀

이전 예시( )에서 보았듯이 표준 `cudaMalloc` / `cudaFree` 호출은 동기식이며 상대적으로 비용이 큽니다. 이는 전체 장치 동기화(상대적으로 느림)를 필요로 하며 GPU 메모리 관리를 위해 `mmap` / `ioctl` 같은 OS 수준 호출을 포함합니다.

이러한 OS 수준 상호작용은 커널 공간 컨텍스트 전환과 드라이버 오버헤드를 발생시키므로 순수한 장치 측 작업에 비해 상대적으로 느립니다. 따라서 GPU에서 보다 효율적인 메모리 할당을 위해 비동기 버전인 `cudaMallocAsync` 및 `cudaFreeAsync` 를 사용하는 것이 권장됩니다.

기본적으로 CUDA 런타임은 전역 GPU 메모리 풀을 유지합니다. 비동기적으로 메모리를 해제하면 후속 할당에서 재사용될 수 있도록 풀로 되돌려집니다.

`cudaMallocAsync` 및 `cudaFreeAsync` 은 내부적으로 CUDA 메모리 풀을 사용합니다.

메모리 풀은 해제된 메모리 버퍼를 재활용하여 새 메모리 할당을 위한 반복적인 OS 호출을 방지합니다. 예를 들어 장시간 실행되는 훈련 루프에서 매 반복마다 새 블록을 생성하는 대신 이전에 해제된 블록을 재사용함으로써 시간이 지남에 따라 메모리 조각화를 줄이는 데 도움이 됩니다. PyTorch와 같은 많은 고성능 라이브러리 및 런타임에서는 기본적으로 메모리 풀이 활성화되어 있습니다.

실제로 PyTorch는 `PYTORCH_ALLOC_CONF` (구 `PYTORCH_CUDA_ALLOC_CONF`)로 구성된 맞춤형 메모리 캐싱 할당기를 사용합니다. PyTorch 메모리 캐싱 할당기는 CUDA의 메모리 풀과 유사한 개념으로, 예를 들어 장시간 실행되는 훈련 루프의 각 반복마다 생성되는 새로운 PyTorch 텐서에 대해 동기식 `cudaMalloc` 작업을 호출하는 비용을 피하면서 GPU 메모리를 재사용합니다.

CUDA 애플리케이션에서 빈번하고 세분화된 할당을 수행할 때는, 전체 장치 동기화와 OS 수준 호출까지 발생하는 기존의 동기식 `cudaMalloc`/`cudaFree` 대신 비동기 풀 기반 루틴—`cudaMallocAsync` 및 `cudaFreeAsync`—을 사용하는 것이 훨씬 효율적입니다. 스트림 순서 할당을 사용하려면 비차단 스트림을 생성하세요:

```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
```

명시적인 CUDA 스트림 사용은 전송, 커널, 메모리 연산을 중첩하는 데 권장되는 방법입니다. 각 스트림을 자체 연산 간 순서를 강제하는 독립된 채널로 생각하십시오. 또한 레거시 기본 스트림 배리어를 피하기 위해 `` `cudaStreamCreateWithFlags(..., cudaStreamNonBlocking)` ``로 비차단 스트림을 생성하는 것이 좋습니다. 다중 스트림 중첩 기법과 권장 사항은 11장에서 자세히 살펴보겠습니다.

그런 다음, `N` 개의 부동 소수점 버퍼가 필요할 때마다 다음과 같이 해당 스트림에서 `cudaMallocAsync` 및 `cudaFreeAsync` 를 사용하여 할당하고 해제합니다:

```
float* d_buf = nullptr;
cudaMallocAsync(&d_buf, N * sizeof(float), stream1);

// ... launch kernels into stream1 that use d_buf ...
myKernel<<<blocksPerGrid, threadsPerBlock, 0, stream1>>>(d_buf, N);

// Free is deferred until all work in stream1 completes—
cudaFreeAsync(d_buf, stream1);
```

이 API들은 디바이스별 메모리 풀에서 할당하지만 전달된 스트림의 순서를 존중하므로, 해당 스트림 작업이 완료될 때까지 해제는 연기됩니다. 또한 `cudaFreeAsync`는 `stream1`` 완료만 기다리기 때문에 비용이 큰 전역 `cudaDeviceSynchronize`가 필요하지 않으며 다른 스트림과의 암묵적 동기화도 발생하지 않습니다. 결과적으로 코드가 수천~수백만 번의 할당/해제 사이클을 수행할 때 할당 오버헤드가 크게 감소하여 조각화를 줄이고 지연 시간 급증을 완화합니다. 전반적으로 이 패턴은 기존의 `cudaMalloc` 및 `cudaFree`에 비해 전역 동기화와 조각화를 줄입니다.

스트림 순서 할당의 동작은 장치 메모리 풀에서 추가로 조정할 수 있습니다. 예를 들어, 풀이 해제를 시도하기 전에 유지해야 할 예약 메모리 양을 암시하도록 `cudaMemPoolAttrReleaseThreshold`를 설정하거나, `cudaMemPoolTrimTo`를 사용하여 메모리를 선제적으로 반환할 수 있습니다. 이는 총 GPU 메모리 사용량과 조각화를 균형 있게 관리하는 데 도움이 됩니다.

단순한 일회성 버퍼의 경우, 차단형 `cudaMalloc`` 및 `cudaFree`로 충분할 수 있습니다. 그러나 메모리를 반복적으로 할당하고 해제하는 복잡한 장기 실행 루프에서는 전용 스트림에 `cudaMallocAsync`` 및 `cudaFreeAsync`로 전환하고 해당 풀을 활용하면 더 일관된 성능과 높은 처리량을 얻을 수 있습니다.

전용 스트림에서 `cudaMallocAsync` 및 `cudaFreeAsync`로 전환하고 해당 풀을 활용하면 더 일관된 성능과 높은 처리량을 얻을 수 있습니다. `cudaMemPoolSetAttribute`(예: `cudaMemPoolAttrReleaseThreshold` 조정)을 통해 풀 동작을 추가로 *튜닝하여 해제 임계값을 조정하고 최소 메모리 사용량과 낮은 조각화 사이의 적절한 균형을 맞출* 수 있습니다.

GPU 메모리 계층 구조 이해

지금까지 우리는 일반적으로 글로벌 메모리를 기준으로 메모리 할당을 높은 수준에서 광범위하게 논의해 왔습니다. 이러한 할당은 기본 스트림 0 메모리 풀을 포함한 스트림의 메모리 풀에서 비롯됩니다.

그러나 실제로 GPU는 다단계 메모리 계층 구조를 제공하여 용량과 속도의 균형을 맞춥니다. 이 계층 구조에는 레지스터, 공유 메모리, 캐시, 글로벌 메모리, 그리고 Blackwell GPU 이상에서 사용되는 특수한 TMEM이 포함됩니다. TMEM은 잠시 후 자세히 설명하겠지만, Blackwell의 5세대 텐서 코어 명령어(`tcgen05.*`)가 사용하는 SM당 약 256KB의 전용 온칩 메모리입니다. CUDA C++에서 직접 포인터로 접근할 수 없습니다. 대신 데이터 이동은 TMA 하드웨어(글로벌 메모리 ↔ SMEM)와 tcgen05 텐서 코어 데이터 이동 명령어(텐서 디스크립터를 암시적으로 사용하여 SMEM ↔ TMEM)에 의해 조정됩니다.

글로벌 메모리(HBM 또는 DRAM)는 대용량이며 오프칩에 위치해 상대적으로 느립니다. 레지스터는 소용량이며 온칩에 위치해 극히 빠릅니다. L1 캐시, L2 캐시 및 공유 메모리는 그 중간 정도입니다. 캐싱과 공유 메모리의 장점은 대규모 오프칩 메모리 저장소 접근 시 발생하는 상대적으로 긴 지연 시간을 숨겨준다는 점입니다. GPU 메모리 계층 구조(CPU 포함)의 고수준 개요는 그림 6-10에 표시되어 있습니다.

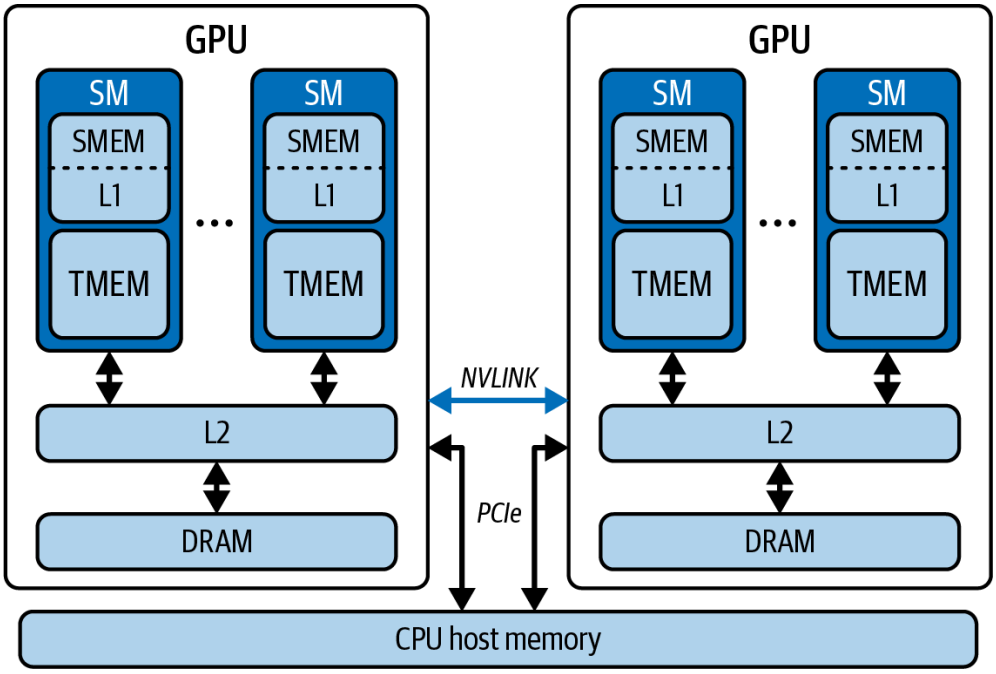

그림 6-10. CPU를 포함한GPU 메모리 계층 구조

TMEM은 SM당 전용 256KB 버퍼로, 초당 수십 테라바이트 대역폭으로 텐서 코어와 투명하게 통신합니다. 이는 텐서 코어의 글로벌 메모리 의존도를 줄입니다. 그림 6-11은 C = A × B 행렬 곱셈 연산을 위해 SMEM과 함께 텐서 코어를 지원하는 TMEM을 보여줍니다.

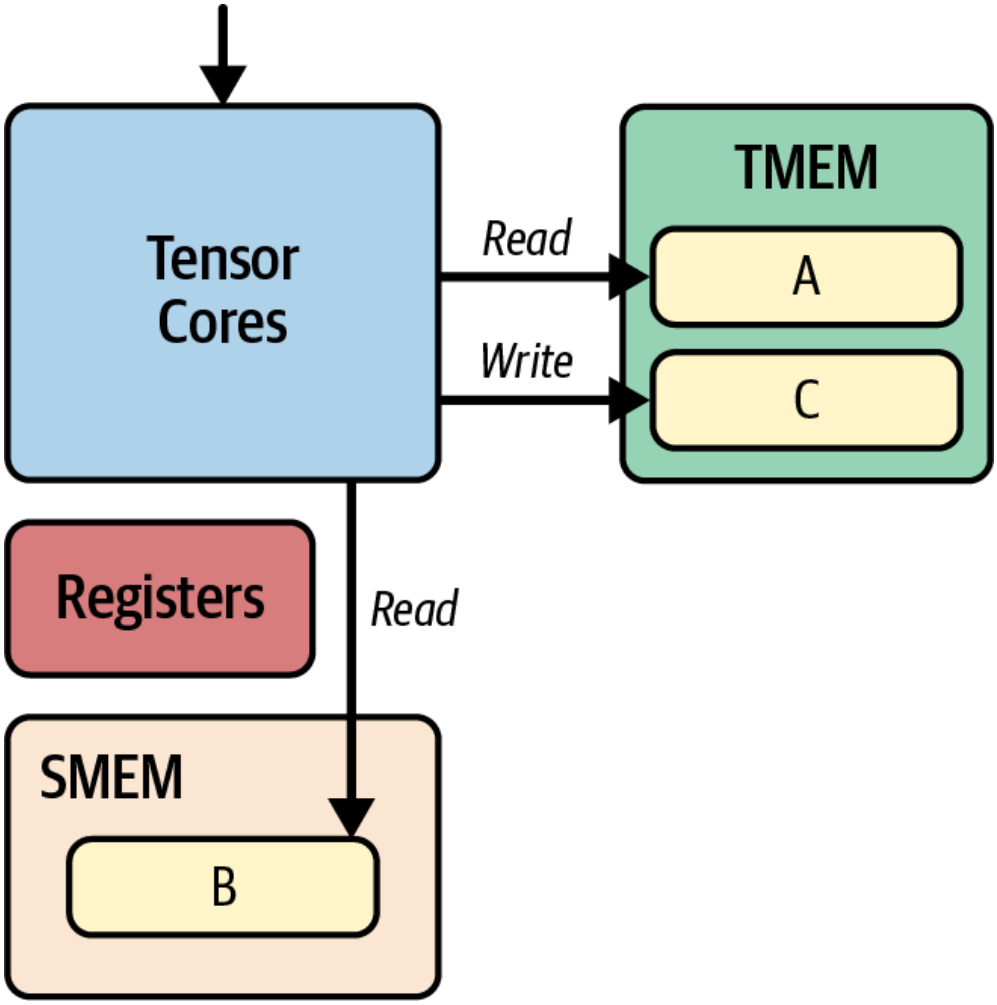

그림 6-11. C = A × B 행렬 곱셈을 위한 텐서 코어에 서비스를 제공하는TMEM 및 SMEM

여기서 피연산자 B는 SMEM에서 가져옵니다. 피연산자 A는 TMEM에 있습니다(SMEM에도 있을 수 있음). 누산기(accumulator) 역시 TMEM에 있습니다. 타일(tiles)은 TMA(`cuda::memcpy_async` 등)를 통해 L2 캐시를 거쳐 글로벌 메모리에서 SMEM으로 이동합니다. 연산 대상은 통합 행렬 곱셈 누산(UMMA) 및 벡터화된 행렬 곱셈 누산(`tcgen05.mma`)과 같은 텐서 코어 명령어를 통해 SMEM과 TMEM 사이에서 암시적으로 이동합니다.

표 6-5는 Blackwell GPU의 다양한 메모리 계층과 그 특성을 보여줍니다. 각 메모리 계층에 대한 설명은 다음과 같습니다.

표 6-5. Blackwell 메모리 계층 구조 및 특성

레벨	범위	용량	지연 시간	대역폭 (대략)
레지스터	스레드당 (SM 내)	SM당 64K 32비트 레지스터 (스레드당 최대 255개)	레지스터 접근 지연 시간 (레지스터 읽기/쓰기는 단일 사이클이며 사실상 무료)	SM당 수십 TB/s (레지스터 파일 포트)
공유 메모리 및 L1 캐시	SM당	228KB (사용 가능 227KB) 공유 + 나머지는 L1/데이터 캐시	~20–30 사이클 (L1/공유 벤치마크)	SM당 TB/s (뱅크 충돌 없음)
TMEM	SM당	SM당 256KB SRAM (텐서 코어 전용)	~10 사이클 (SM 내 전용 SRAM)	텐서 코어와의 TB/s 규모 통신
상수 메모리 캐시	SM당	64KB의 `__constant__` 공간을 위한 ~8KB 캐시	~1 사이클 (워프 브로드캐스트) 상수 캐시 및 브로드캐스트 동작으로 인해 캐시되고 워프 내 모든 스레드가 동일한 주소를 액세스할 경우 레지스터만큼 빠름. 발산 또는 미스된 경우 직렬화되거나 더 높은 지연 발생	TB/s 규모 (브로드캐스트 처리량)
L2 캐시	GPU 전체 (모든 SM)	총 126MB	~200 사이클	멀티 TB/s 집계
로컬 메모리	스레드당 (DRAM으로 스필)	거의 무제한 (글로벌 메모리 지원)	100~1,000 사이클 (DRAM 유사)	~8 TB/s (HBM3e)
글로벌 메모리 (HBM 또는 DRAM)	장치 전체 (오프칩 DRAM)	Blackwell B200 GPU당 최대 180GB (Blackwell B300 GPU당 최대 ~288GB)	100~1,000 사이클 (글로벌 메모리 지연 시간)	총 ~8TB/s

여기서 레지스터, 공유 메모리, L1/L2 캐시에서의 데이터 재사용을 극대화하고 글로벌 메모리 및 글로벌 메모리로 백업되는 로컬 메모리에 대한 의존도를 최소화하는 것이 고처리량 GPU 커널에 필수적인 이유를 알 수 있습니다. 다음은 계층 구조의 각 레벨에 대한 좀 더 자세한 설명입니다:

레지스터

Blackwell 아키텍처에서 각 스레드는 레지스터 파일에서 작업을 시작합니다. 레지스터 파일은 SM 내부에 위치한 소형 SRAM 배열로, 각 스레드의 로컬 변수를 추가 지연 시간 없이 저장합니다. 각 SM은 64K개의 32비트 레지스터(총 256KB)를 보유하지만, 하드웨어는 스레드당 최대 255개의 레지스터만 노출합니다.

읽기 및 쓰기가 단일 사이클 내에 완료되며 거의 다른 작업과 경쟁하지 않기 때문에, 레지스터 대역폭은 SM당 초당 수십 테라바이트에 이를 수 있습니다. 그러나 커널이 더 많은 레지스터를 필요로 하는 경우(많은 스레드 로컬 변수나 컴파일러 임시 변수 때문) 오버플로가 오프칩 DRAM에 매핑된 로컬 메모리로 스필되며, 수백에서 수천 사이클 이상의 지연 시간이 발생합니다. 이 로컬 메모리는 그림 6-12에 표시되어 있습니다.

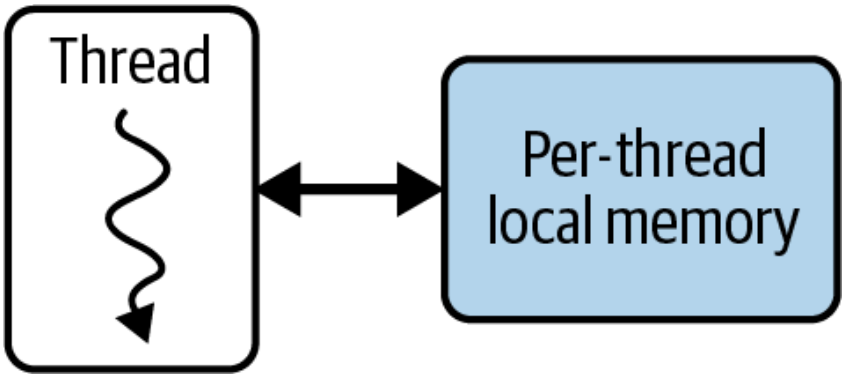

그림 6-12. 스레드당로컬 메모리

공유 메모리 및 L1 데이터 캐시

한 단계 상위에는 통합 L1/데이터 캐시 및 공유 메모리 블록이 있습니다. 이는 SM당 256KB의 온-SM SRAM으로, 통합 L1/텍스처/공유 메모리를 갖춘 Blackwell과 같은 아키텍처에서 쿼타리드 메모리 할당(`cudaFuncSetAttribute()`)과 메모리 카빙 선택(`cudaFuncAttributePreferredSharedMemoryCarveout`)을 사용하여 사용자 관리 공유 메모리(블록당 최대 228KB)로 동적으로 분할할 수 있습니다. 블록당 최대 동적 공유 메모리는 227KB(CUDA가 블록당 1KB 예약)이며, SM당 할당 가능한 총 공유 메모리는 이 한도로 바운디드됩니다.

여기에서 액세스 비용은 대략 20~30사이클이지만, 뱅크 충돌을 피하도록 스레드 블록을 설계하면 초당 테라바이트 단위의 처리량을 달성할 수 있습니다. 스레드 블록 공유 메모리는 그림 6-13에 표시되어 있습니다.

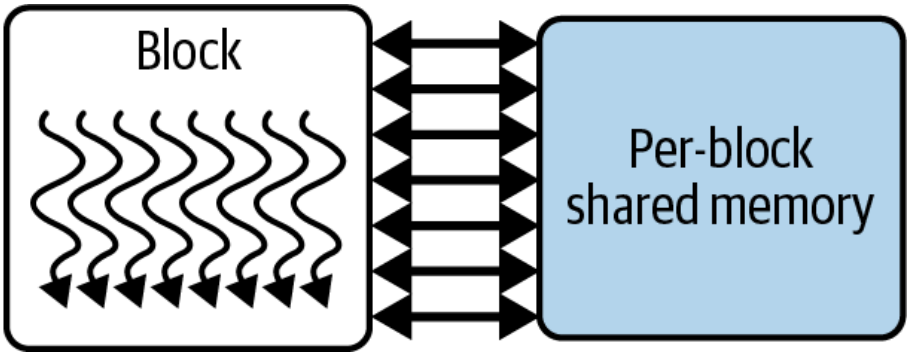

그림 6-13. 스레드 블록 공유 메모리

TMEM

TMEM은 SM당 전용 온칩 메모리(Blackwell 기준 256KB)로, 10장에서 설명하는 통합 행렬 곱셈-누적(unified matrix-multiply-accumulate) 및 벡터화된 행렬-벡터 연산(`tcgen05`)을 포함한 텐서 코어 전용 연산 및 명령어에 사용됩니다. 이는 CUDA C++에서 일반적인 포인터 주소 지정 공간이 아닙니다. 대신, 디스크립터를 사용하여 텐서 메모리 가속기(TMA)를 통해 전송이 조정됩니다. 이를 통해 개발자는 텐서 코어와의 데이터 흐름을 수동으로 관리할 필요가 없습니다. 예를 들어 일부 산술 연산자는 공유 메모리에 상주하는 반면, 누산기는 TMEM에 상주합니다. 이후 TMA는 계산을 수행하기 위해 전역 메모리, 공유 메모리, TMEM 메모리 간 데이터 이동을 담당합니다.

상수 메모리 캐시

아주 작은 읽기 전용 테이블의 경우, Blackwell은 64KB의 테이블 메모리 공간(`__constant__`) 앞에 약 8KB의 상수 메모리 캐시를 SM(Streaming Multiprocessor)별로 제공합니다. 워프(warp) 내 32개 스레드 모두가 동일한 주소를 로드할 때, 이 캐시는 단일 사이클 내에 값을 브로드캐스트합니다.

분산된 읽기 작업은 레인 간에 직렬화됩니다. 이는 회전 위치 인코딩, 선형 바이어스 어텐션(ALiBi) 기울기, 레이어 정규화 γ/β 벡터, 임베딩 양자화 스케일 등의 소형 조회 테이블 공유에 이상적입니다. 이러한 테이블은 전역 메모리 트래픽 없이 모든 스레드 간에 공유됩니다.

L2 캐시

온칩 SRAM을 넘어서는 L2 캐시는 126MB GPU 전체 버퍼로, 모든 SM을 오프칩 HBM3e에 연결합니다. 약 200사이클의 지연 시간과 초당 수십 테라바이트의 집계 대역폭을 가진 L2는 L1의 스필오버를 흡수합니다.

L2를 통해 한 스레드 블록이 데이터를 가져오면 다른 스레드 블록이 DRAM을 재방문하지 않고 재사용할 수 있습니다. L2의 이점을 극대화하려면 글로벌 로드를 128바이트로 결합된 트랜잭션으로 구성하여 캐시 라인과 깔끔하게 매핑되도록 하십시오. 이에 대한 방법은 잠시 후 설명하겠습니다.

글로벌 로드를 128바이트 정렬된, 캐시 라인에 깔끔하게 매핑되는 결합된 세그먼트로 구조화하십시오. 이렇게 하면 분할 트랜잭션을 피하고 L2 및 DRAM 대역폭을 최대한 활용할 수 있습니다.

글로벌 메모리(HBM 또는 DRAM)

글로벌 메모리 계층, 로컬 스필 공간 및 HBM은 오프칩에 위치합니다. 스필된 레지스터나 과도하게 큰 자동 배열은 로컬 메모리에 상주하며, HBM3e의 약 8TB/s 대역폭에도 불구하고 전체 DRAM 지연 시간(수백 사이클에서 1,000 사이클 이상)을 소모합니다.

Blackwell의 경우 HBM3e 계층은 총 약 8TB/s로 최대 180GB의 장치 전체 저장 공간을 제공합니다. 그러나 높은 지연 시간으로 인해 체인에서 가장 느린 링크입니다. 장치별 글로벌 메모리는 그림 6-14에 표시되어 있습니다.

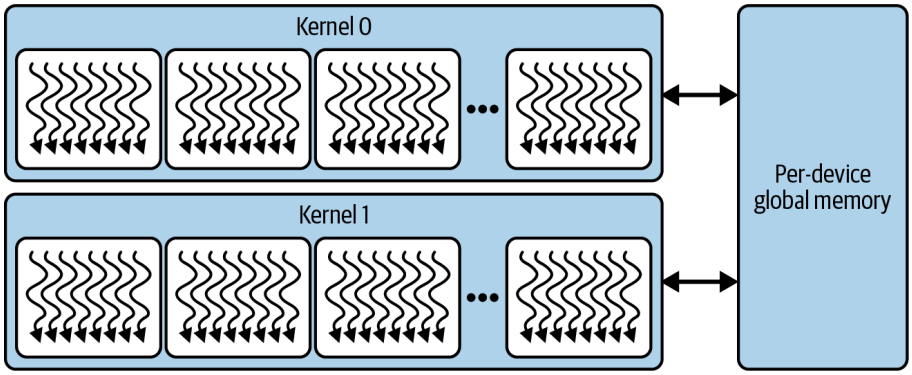

그림 6-14. 장치별 글로벌 메모리(HBM)

Nsight Compute와 같은 도구를 사용하여 스필(spill) 및 캐시 적중률(cache hit rate)을 추적하면 커널이 이 메모리 계층 구조의 온칩(on-chip) 성능 한계에 최대한 근접하여 작동하도록 유지할 수 있습니다. 이러한 도구는 레지스터, 공유/L1, 상수 캐시(constant cache), L2 캐시를 통해 데이터를 효과적으로 조율하는 데 도움이 됩니다. Blackwell과 같은 최신 GPU는 커널 개발자가 L2 캐시와 통합 L1/공유 메모리를 활용하여 HBM 접근을 버퍼링하고 통합할 수 있도록 하여 메모리 계층 구조를 활용할 수 있게 합니다. 곧 살펴보겠지만,

---

Blackwell B200은 통합된 글로벌 주소 공간으로 구축된 단일 GPU로 로 제공됩니다. 그러나 10TB/s 칩 간 상호 연결로 연결된 두 개의 레티클 제한 다이로 구성됩니다. 각 다이는 4개의 HBM3e 스택에 연결되어 총 8개의 HBM3e 스택을 구성합니다. 개발자의 관점에서 보면 HBM 메모리 액세스는 이 결합된 주소 공간 전체에서 균일하지만, 이 아키텍처의 저수준 세부 사항을 이해하는 것이 중요합니다.

의 메모리 계층 구조 내 각 레벨 간 일관성 지점(PoC)은 요구 사항과 스레드 간 통신 레벨에 따라 달라집니다. 일반적으로 스레드, 스레드 블록(*CTA*), 스레드 블록 클러스터(*CTA 클러스터*), 디바이스, 시스템 레벨에서 발생하며, 이는 그림 6-15에 표시되어 있습니다.

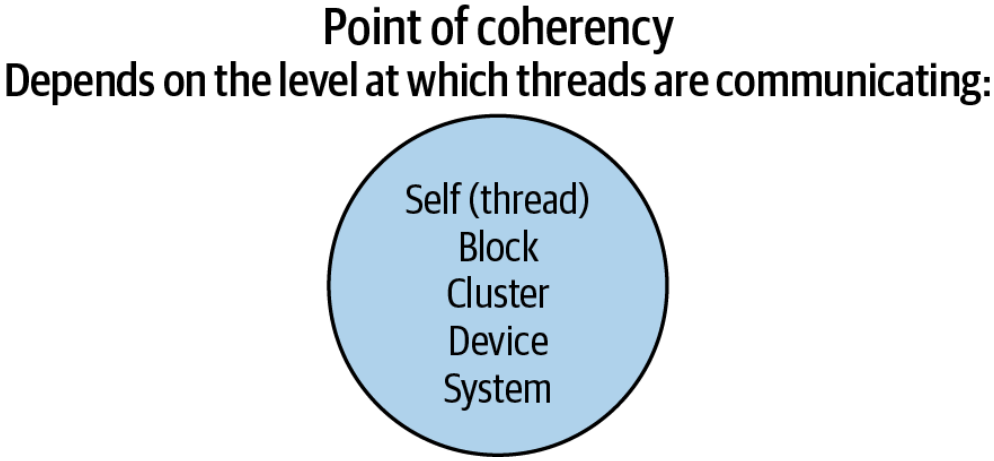

그림 6-15. GPU 스레드에 대한 메모리지점 일관성

요약하자면, GPU의 메모리 계층 구조를 이해하고 각 레벨을 적절히 타겟팅하는 것이 중요합니다. 이를 통해 CUDA 커널을 구조화하여 데이터 로컬리티를 극대화하고, 메모리 접근 지연 시간을 숨기며, 점유율을 높이고, 블랙웰의 대규모 병렬 컴퓨팅 능력을 완전히 활용할 수 있습니다. 이에 대해서는 잠시 후 살펴보겠습니다. 먼저, Grace Hopper와 Grace Blackwell의 통합 CPU-GPU 슈퍼칩 설계를 고려할 때 이해해야 할 NVIDIA의 통합 메모리에 대해 논의해 보겠습니다.

통합 메모리

통합 메모리(CUDA 관리 메모리라고도 함)는 CPU와 GPU를 아우르는 단일 한 일관성 주소 공간을 제공합니다. 따라서 더 이상 별도의 호스트 및 디바이스 버퍼를 관리하거나 명시적인 메모리 복사(`cudaMemcpy`) 호출을 수행할 필요가 없습니다. 내부적으로 CUDA 런타임은 모든 메모리 할당(`cudaMallocManaged()`)을 CPU와 GPU를 연결하는 인터커넥트 링크를 통해 필요 시 이동할 수 있는 페이지로 지원합니다(그림 6-16 참조).

통합 메모리 접근은 개발자에게 매우 편리하지만, CPU와 GPU 간 원치 않는 페이지 이동을 유발할 수 있습니다. 이는 숨겨진 지연 시간과 실행 정지를 초래합니다. 예를 들어 GPU 스레드가 현재 CPU 메모리에 있는 데이터에 접근하면, GPU는 페이지 결함이 발생하고 해당 데이터가 NVLink-C2C 인터커넥트를 통해 전송되는 동안 대기합니다. 통합 메모리 성능은 기본 하드웨어에 크게 좌우됩니다.

기존 PCIe 또는 초기 NVLink 시스템에서는 이러한 마이그레이션이 상대적으로 낮은 대역폭으로 전송되어, 결함 발생 시 전송 속도가 수동 HBM 간 전송(`cudaMemcpy`)보다 느린 경우가 많습니다. 그러나 Grace Hopper 및 Grace Blackwell 슈퍼칩에서는 NVLink-C2C 패브릭이 CPU의 HBM과 GPU의 HBM3e 간에 최대 ~900GB/s의 대역폭을 제공합니다. 따라서 페이지 결함으로 인한 마이그레이션은 비록 여전히 제로가 아닌 지연 시간을 가지지만, 장치 네이티브 속도에 훨씬 근접합니다.

그러나 커널 실행 중 예기치 않은 페이지 결함이 발생하면 런타임이 필요한 페이지를 제 위치로 이동시키는 동안 GPU가 정지됩니다. 이러한 "예상치 못한" 정지를 피하려면 그림 6-17과 같이 ` `cudaMemPrefetchAsync()` `로 메모리를 미리 가져올 수 있습니다.

이는 커널 실행 전에 지정된 범위를 대상 GPU(또는 CPU)로 미리 이동하도록 드라이버에 암시하여, 비용이 많이 드는 첫 터치 마이그레이션을 중첩 가능한 비동기 전송으로 전환합니다. 다음 코드에서 볼 수 있듯이 메모리 조언을 제공할 수도 있습니다:

```
cudaMemAdvise(ptr, size, cudaMemAdviseSetPreferredLocation, gpuId);
cudaMemAdvise(ptr, size, cudaMemAdviseSetReadMostly, gpuId);
```

여기서 그림 6-18과 같이 `PreferredLocation` 로 데이터 사용 위치를, `ReadMostly` 로 주로 읽기 전용인 경우를 드라이버에 알릴 수 있습니다.

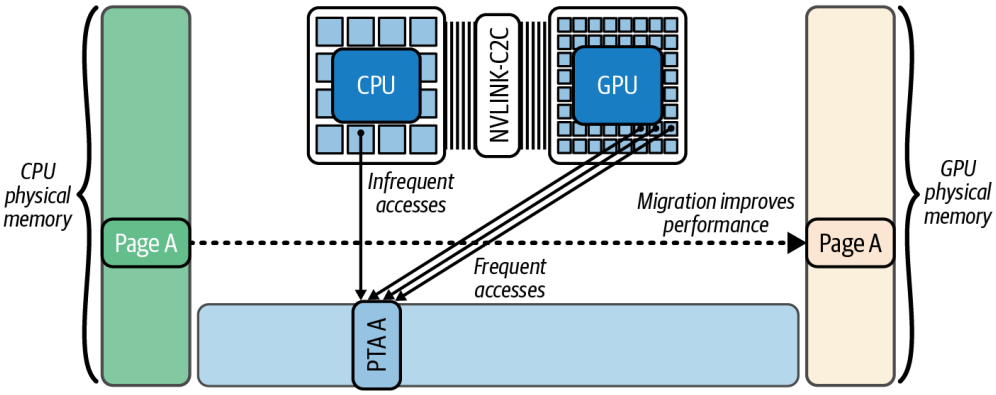

그림 6-16. CPU-GPU 통합 메모리를 통한자동 페이지 마이그레이션

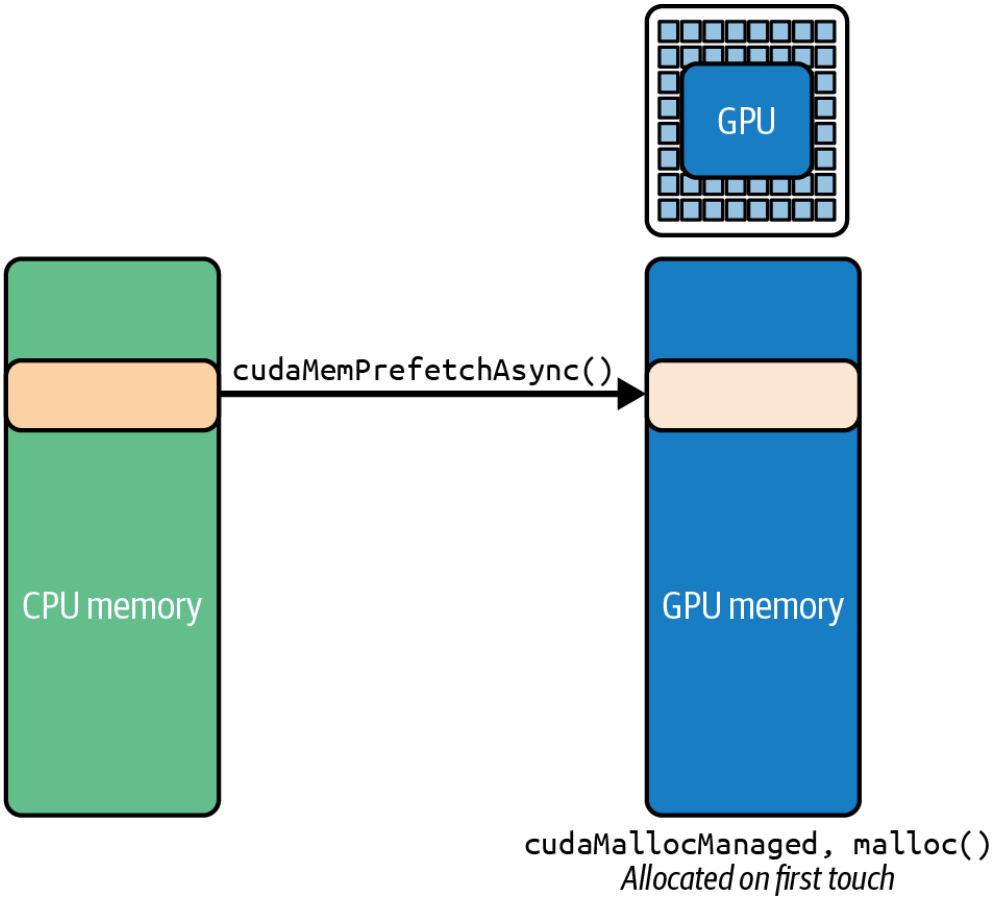

그림 6-17. NVLink-C2C를 통한 CPU에서 GPU로의 스트리밍 데이터 전송 (`cudaMemPrefetchAsync()`

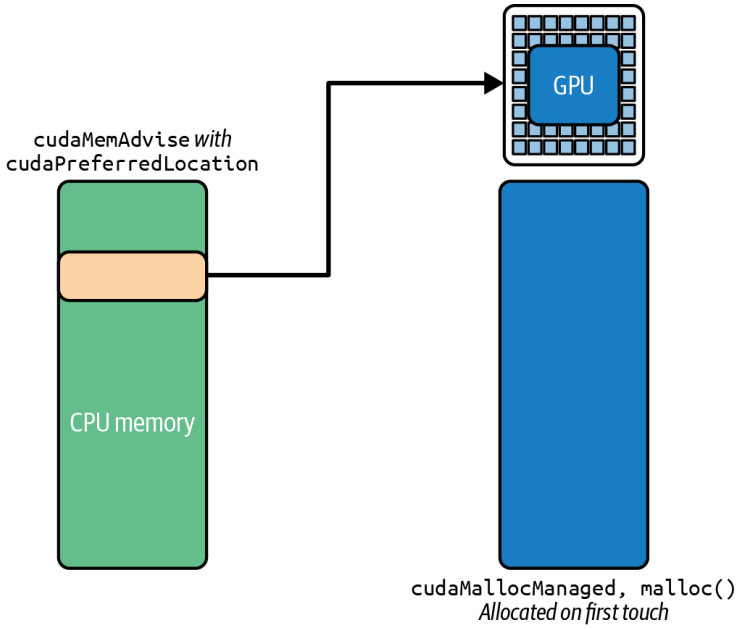

그림 6-18. CUDA 드라이버에 데이터 사용 패턴을 알려주는 "선호 위치"지정 (예: 주로 읽기 전용 워크로드인 경우 `ReadMostly`)

다음 호출을 통해 두 번째 GPU가 해당 페이지를 매핑하도록 허용할 수 있으며, 이 경우 런치 시 마이그레이션이 발생하지 않습니다:

```
cudaMemAdvise(ptr, size, cudaMemAdviseSetAccessedBy,
   otherGpuId);
```

기본적으로 모든 CUDA 스트림 또는 디바이스 커널은 관리형 할당에서 페이지 결함을 유발할 수 있습니다. 이는 예상치 못한 마이그레이션과 암시적 동기화를 초래할 수 있습니다. 특정 버퍼가 한 번에 하나의 스트림/GPU에서만 사용될 것임을 알고 있다면, 해당 스트림에 버퍼를 연결하면 마이그레이션이 다른 스트림의 작업과 중첩될 수 있습니다. 다음 호출은 해당 메모리 범위를 지정된 스트림에 바인딩합니다:

```
cudaStreamAttachMemAsync(stream, ptr, 0,
    cudaMemAttachSingle);
```

이 경우 해당 스트림의 작업만 해당 페이지에 대한 결함 및 마이그레이션을 발생시킵니다. 이는 다른 스트림이 우연히 해당 작업으로 인해 지연되는 것을 방지합니다. 따라서 특정 스트림에 범위를 연결하면 마이그레이션이 해당 스트림 작업과만 중첩되도록 지연됩니다. 이는 스트림 간 동기화를 피합니다.

NVLink-C2C가 없는 다중 GPU 시스템에서는 데이터 로컬화(`cudaMemcpyPeerAsync()`)나 특정 장치로의 프리페치(prefetch)를 사용하여 데이터를 가장 가까운 NUMA 로컬 GPU 메모리에 고정할 수 있습니다. 이렇게 하면 느린 원격 액세스를 방지할 수 있습니다.

요약하면, 관리형 메모리를 명시적으로 프리페칭하고 메모리 어드바이스를 제공하면 통합 메모리에서 발생하는 대부분의 "예상치 못한" 스톨을 제거할 수 있습니다. GPU가 데이터 요청 시 일시 정지하는 대신, 커널 실행 시 필요한 위치에 데이터가 미리 배치됩니다.

선제적 프리페칭, 타깃형 메모리 어드바이스, 스트림 어태치먼트 같은 기법을 통해 통합 메모리는 통합 주소 공간의 단순성을 유지하면서도 수동 메모리 관리(`cudaMemcpy`)에 매우 근접한 성능을 제공할 수 있습니다.

높은 점유율 및 GPU 활용도 유지

GPU는 다수의 워프를 동시에 실행하는 병렬 처리( )를 통해 성능을 유지합니다. 한 워프가 데이터 대기 중 정지하더라도 다른 워프가 실행될 수 있기 때문입니다. 워프 간 신속한 전환 능력은 GPU가 메모리 지연 시간을 숨길 수 있게 합니다. 앞서 설명한 바와 같이, SM 용량 중 활성 워프가 실제로 차지하는 비율을 *점유율 (occupancy)*이라 합니다.

점유율이 낮을 경우(활성 워프가 소수일 때), 하나의 워프가 메모리를 기다리는 동안 SM 전체가 유휴 상태가 될 수 있습니다. 이는 SM 활용도 저하로 이어집니다. Blackwell 아키텍처에서는 대규모 레지스터 파일(SM당 64K 레지스터) 덕분에 스필링 없이 다수의 워프를 지원할 수 있어 높은 점유율 달성이 다소 용이합니다.

앞서 살펴본 바와 같이, 워프 내 각 스레드는 최대 255개의 레지스터를 사용할 수 있습니다. 자질 도구를 활용하여 달성된 점유율을 확인하고, 커널의 블록 크기와 레지스터 사용량을 적절히 조정하세요.

반대로 높은 점유율(SM당 많은 활성 워프)은 메모리 접근을 기다리는 한 워프가 있는 동안 다른 워프들이 SM으로 교체되어 실행되므로 GPU 연산 유닛을 바쁘게 유지합니다. 이는 긴 메모리 접근 지연을 가려줍니다. 이를 흔히 *'지연 숨기기'*라고 합니다.

점유율과 궁극적으로 GPU 활용도, 처리량, 전체 커널 성능을 향상시키는 예시를 살펴보겠습니다. 이는 CUDA 성능 최적화의 가장 기본적인 규칙 중 하나입니다: GPU를 완전히 점유할 만큼 충분한 병렬 작업을 실행하십시오.

달성된 점유율 (사용 중인 하드웨어 스레드 슬롯의 비율)이 GPU 한계치보다 훨씬 낮고 성능이 저조하다면, 첫 번째 해결책은 병렬성을 높이는 것입니다 . 더 많은 블록이나 스레드를 사용하여 점유율이 최신 GPU 기준 80%~100% 범위에 근접하도록 하십시오.

반대로 점유율이 이미 중간에서 높은 수준이지만 커널이 메모리 처리량에 의해 병목 현상을 보인다면, 점유율을 100%까지 끌어올리는 것은 도움이 되지 않을 수 있습니다. 일반적으로 지연 시간을 숨기기 위해 필요한 만큼의 워프만 있으면

되며, 그 이상으로 늘리면 병목 현상이 다른 곳(예: 메모리 대역폭)에 있을 수 있습니다.

점유율의 영향을 설명하기 위해 매우 간단한 연산(길이 N인 두 벡터의 덧셈, C = A + B 계산)을 고려해 보겠습니다. 두 가지 커널 구현(`addSequential` 및 `addParallel`)을 살펴보겠습니다. `addSequential` 는 단일 스레드(또는 단일 워프)를 사용하여 루프 내에서 모든 N개 요소를 더합니다. `addParallel` 는 다수의 스레드를 사용하여 배열 전체에 걸쳐 덧셈이 동시에 수행되도록 합니다.

순차적 버전에서는 하나의 GPU 스레드가 전체 작업 부하를 순차적으로 처리합니다. 다음과 같습니다:

```
#include <cuda_runtime.h>

const int N = 1'000'000;

// Single thread does all N additions
__global__ void addSequential(const float* A,
                              const float* B,
                                    float* C,
                              int N)
{
    if (blockIdx.x == 0 && threadIdx.x == 0) {
        for (int i = 0; i < N; ++i) {
            C[i] = A[i] + B[i];
        }
    }
}

int main()
{
    // Allocate and initialize host
    float* h_A = nullptr;
    float* h_B = nullptr;
    float* h_C = nullptr;
    cudaMallocHost(&h_A, N * sizeof(float));
    cudaMallocHost(&h_B, N * sizeof(float));
    cudaMallocHost(&h_C, N * sizeof(float));

    for (int i = 0; i < N; ++i) {
        h_A[i] = float(i);
        h_B[i] = float(i * 2);
    }

    // Allocate device
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, N * sizeof(float));
    cudaMalloc(&d_B, N * sizeof(float));
    cudaMalloc(&d_C, N * sizeof(float));
```

```
    // Copy inputs to device
    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

    // Launch: one thread
    // Note: This kernel assumes <<<1,1>>>
    // (one block, one thread).
    // Do not change the launch config when running this example.
    addSequential<<<1,1>>>(d_A, d_B, d_C, N);

    // Ensure completion before exit
    cudaDeviceSynchronize();

    // Copy d_C => h_C (back to host)
    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFreeHost(h_A);
    cudaFreeHost(h_B);
    cudaFreeHost(h_C);

    return 0;
}
```

이 단일 스레드 버전에서는 GPU의 방대한 자원이 대부분 유휴 상태입니다. 단 하나의 워프, 심지어 워프 내 하나의 스레드만이 작업을 수행하는 동안 나머지는 모두 유휴 상태로 남아 있습니다. 그 결과 점유율이 매우 낮아지고 궁극적으로 성능이 저하됩니다.

또한 PyTorch와 같은 고수준 라이브러리 및 프레임워크에서 비효율적인 GPU 코드가 간접적으로 실행되지 않도록 주의해야 합니다. 예를 들어, 다음과 같은 단순한 PyTorch 코드는 Python for-루프를 사용하여 GPU에서 *N개의* 별도 덧셈 연산을 연속적으로 수행하는 요소별 연산을 잘못 수행합니다:

```
import torch

N = 1_000_000
A = torch.arange(N, dtype=torch.float32, device='cuda')
B = 2 * A
C = torch.empty_like(A)

# Ensure all previous work is done
torch.cuda.synchronize()

# Naive, Sequential GPU operations - DO NOT DO THIS
with torch.inference_mode(): # avoids unnecessary autograd graph construct
    # This launches N tiny GPU operations serially
```

```
    for i in range(N):
        C[i] = A[i] + B[i]

torch.cuda.synchronize()
```

이 코드는 GPU를 스칼라 방식의 비병렬 프로세서처럼 사용합니다. 이는 앞서 소개한 네이티브 `addSequential` CUDA C++ 코드와 유사하게 매우 낮은 점유율을 보입니다.

벡터 덧셈 연산의 병렬 버전을 구현하기 위해 CUDA 커널과 PyTorch 코드를 최적화해 보겠습니다. 그림 6-19는 벡터화된 덧셈 연산이 어떻게 작동하는지 보여줍니다.

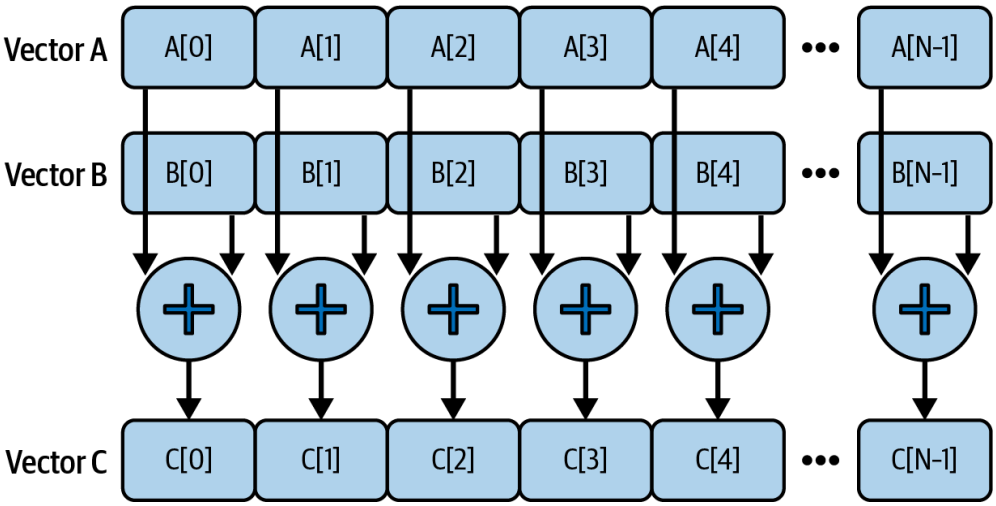

그림 6-19. 벡터 요소 간 병렬로 수행되는벡터화 덧셈 연산

다음 CUDA C++ 코드에서는 모든 요소를 처리할 수 있도록 충분한 스레드(`<<< (N+255)/256, 256 >>>`)를 런치합니다. 이를 통해 블록당 256개의 스레드가 필요한 블록 수에 걸쳐 *N개의* 요소를 병렬로 처리합니다:

```
#include <cuda_runtime.h>

const int N = 1'000'000;

// One thread per element
__global__ void addParallel(const float* __restrict__ A,
                            const float* __restrict__ B,
                                  float* __restrict__ C,
                            int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}

int main()
{
```

```
    // Allocate and initialize host (pinned for faster DMA)
    float* h_A = nullptr;
    float* h_B = nullptr;
    float* h_C = nullptr;
    cudaMallocHost(&h_A, N * sizeof(float));
    cudaMallocHost(&h_B, N * sizeof(float));
    cudaMallocHost(&h_C, N * sizeof(float));
    for (int i = 0; i *lt; N; ++i) { h_A[i] = float(i); h_B[i] = float(2*i)

    // Create a non-blocking stream and allocate device buffers
    cudaStream_t s; cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    float *d_A = nullptr, *d_B = nullptr, *d_C = nullptr;
    cudaMallocAsync(&d_A, N * sizeof(float), s);
    cudaMallocAsync(&d_B, N * sizeof(float), s);
    cudaMallocAsync(&d_C, N * sizeof(float), s);

    // Async HtoD copies on the same stream
    cudaMemcpyAsync(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice, s);
    cudaMemcpyAsync(d_B, h_B, N*sizeof(float), cudaMemcpyHostToDevice, s);

    // Launch (same stream)
    int threads = 256;
    int blocks  = (N + threads - 1) / threads;
    addParallel<<<blocks, threads, 0, s>>>(d_A, d_B, d_C, N);

    // Async DtoH copy and stream sync
    cudaMemcpyAsync(h_C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost, s);
    cudaStreamSynchronize(s);

    // Cleanup (stream-ordered free)
    cudaFreeAsync(d_A, s); cudaFreeAsync(d_B, s); cudaFreeAsync(d_C, s);
    cudaStreamDestroy(s);
    cudaFreeHost(h_A); cudaFreeHost(h_B); cudaFreeHost(h_C);
    return 0;
}
```

*N*이 충분히 클 경우 GPU 활용도 차이는 상당합니다. 이제 PyTorch 코드를 최적화해 보겠습니다. 이 코드는 단일 벡터화 커널(`A + B`)을 실행하여 이전에 최적화된 `addParallel` CUDA C++ 예제처럼 GPU에서 다수의 스레드를 동시에 활용합니다. 다음은 PyTorch 코드의 병렬 버전입니다:

```
# add_parallel.py
import torch

N = 1_000_000
A = torch.arange(N, dtype=torch.float32, device='cuda')
B = 2 * A

torch.cuda.synchronize()
```

```
# Proper parallel approach using vectorized operation
# Launches a single GPU kernel that adds all elements in parallel
C = A + B

torch.cuda.synchronize()
```

실제 사용 시 PyTorch와 같은 고수준 프레임워크는 벡터화된 텐서 연산을 사용할 때 최적화된 처리를 수행합니다. 다만 GPU 연산 주변에 Python 수준의 루프를 도입하면 작업이 직렬화되어 성능에 부정적인 영향을 미칠 수 있음을 유의하십시오. 가능하면 이를 피하세요. 독창적인 코드를 작성하는 경우가 아니라면, PyTorch 컴파일러가 생성하는 코드를 포함해 거의 항상 최적화된 PyTorch 네이티브 구현체가 존재합니다.

병렬 구현과 순차적 구현의 성능 영향 을 정량화하기 위해 Nsight Systems 및 Nsight Compute를 사용하여 두 접근법의 총 커널 실행 시간, GPU 활용도, 점유율 및 워프 실행 효율성 지표를 측정할 수 있습니다. 다음은 Nsight Systems(`nsys`) 및 Nsight Compute(`ncu`) 명령어입니다:

```
# Sequential add
nsys profile \
  --stats=true \
  -t cuda,nvtx \
  -o sequential_nsys_report \
  ./add_sequential.py

ncu \
 --section SpeedOfLight \
 --metrics
     sm__warps_active.avg.pct_of_peak_sustained_active,gpu__time_duration
 --target-processes all \
 --print-summary per-gpu \
 -o sequential_ncu_report \
 ./add_sequential.py

# Parallel add
nsys profile \
  --stats=true \
  -t cuda,nvtx \
  -o parallel_nsys_report \
  ./add_parallel.py

ncu \
 --section SpeedOfLight \
 --metrics sm__warps_active.avg.pct_of_peak_sustained_active \
 --target-processes all \
 --print-summary per-gpu \
 -o parallel_ncu_report \
 ./add_parallel.py
```

`nsys` 를 사용해 시간이 어디에 소모되는지, GPU가 리소스 부족 상태인지 차단되었는지 파악합니다. 그런 다음 `ncu` 를 사용해 커널 성능이 저하되는 원인 (예: 낮은 점유율 등)을 분석합니다.

`nsys` 만 실행하면 커널의 미세한 비효율성을 놓칠 수 있습니다. 반대로 `ncu` 만 실행하면 커널에 데이터가 충분히 빠르게 공급되는지 등을 알 수 없습니다. 표 6-6은 통합된 결과를 보여줍니다.

표 6-6. 순차적 CUDA 커널과 병렬 CUDA 커널비교

메트릭	add_sequential	add_parallel
커널 실행 시간 (ms)	48.21	2.17
GPU 사용률	1.5%	95
달성 점유율	1.3%	38.7%
워프 실행 효율성	3.1%	100%

다른 자질 도구는 이러한 지표를 다르게 표시할 수 있습니다. 예를 들어, Nsight Systems는 전체 "GPU 활용도"를 보고하는 반면, Nsight Compute는 커널별 "SM 활성 %" 지표를 제공합니다. 그러나 둘 다 활성 워프에 의해 GPU의 SM이 얼마나 완전히 점유되었는지를 반영합니다.

예상대로, 단일 스레드, 단일 워프 구현에서 완전 병렬 다중 워프 구현으로 전환하면 평균 점유율이 1.3%에서 약 38.7%로 향상됩니다. 이로 인해 실행 시간이 48.21ms에서 2.17ms로 약 22배 단축됩니다.

순차적 경우 단일 SM에서 단 하나의 워프, 즉 단일 스레드만 작업을 수행합니다. 이 때문에 GPU 활용률이 1.5%로 낮게 나타납니다. 반면 병렬 처리에서는 다수의 SM이 여러 활성 워프를 동시에 실행합니다. 워프 내 32개 스레드 모두가 각 명령어 실행 시 유용한 작업을 수행하므로 워프 실행 효율이 3.1%에서 100%로 증가합니다. 이로 인해 GPU 활용률은 1.5%에서 95%로 향상됩니다.

이 예시는 GPU에서 충분한 병렬화가 중요한 이유를 보여줍니다. 각 스레드의 속도가 아무리 빠르더라도 GPU의 처리량 잠재력을 활용하려면 많은 수의 스레드가 필요합니다.

GPU는 처리량 최적화 프로세서로, CUDA 커널을 실행하기 위해 CPU와 상호작용하며 캐시, 공유 메모리, 글로벌 메모리에서 데이터를 로드하기 위해 메모리 서브시스템과도 연동합니다. 따라서 이러한 지연 시간을 숨기는 것이 GPU 성능 향상에 크게 기여합니다.

커널이 적절히 작성되면 GPU는 서로 다른 워프의 메모리 로드와 연산(예: 덧셈)을 병렬로 교차 실행하도록 지시합니다. 이는 워프 간 메모리 지연 시간을 숨기는 데 도움이 됩니다.

특히 여러 워프 내에서 실행되는 병렬 커널은 워프 수준 지연 시간 숨김의 혜택을 받습니다. 한 워프가 메모리 로드를 기다리는 동안 다른 워프는 덧셈 연산을 실행하고, 또 다른 워프는 다음 데이터를 가져오는 등의 작업을 수행할 수 있습니다. 향후 장에서 메모리 지연 시간을 숨기는 다양한 기법을 살펴보겠습니다.

순차적 커널에서는 한 워프가 대기 중일 때 실행할 다른 워프가 없으므로 하드웨어 파이프라인이 종종 유휴 상태로 남아 있습니다. 타임라인은 메모리 대기 중 유휴 간격이 있는 하나의 긴 연산 시퀀스입니다. 병렬 버전에서는 이러한 간격이 다른 워프의 작업으로 채워지므로 GPU는 지속적으로 바쁘게 작동합니다. 비교는 그림 6-20에 표시되어 있습니다.

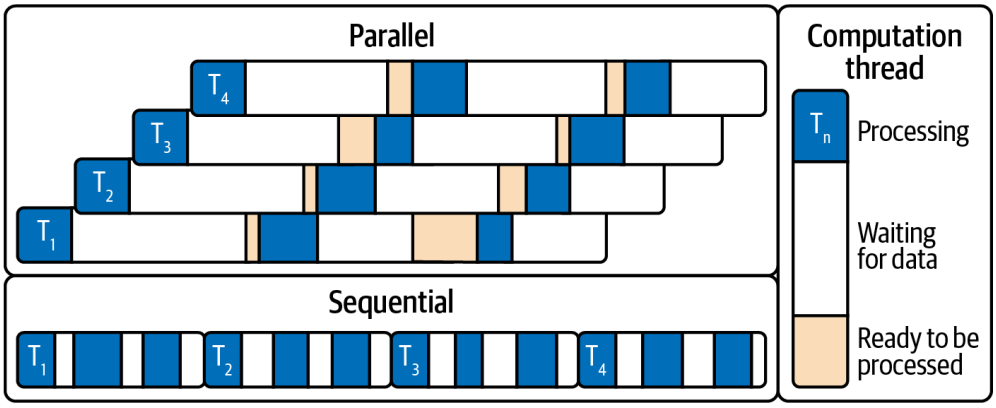

그림 6-20. 병렬 대 순차적 타임라인 비교

여기서 순차적 타임라인은 메모리 대기 중 유휴 간격이 있는 하나의 긴 작업 연속체입니다. 병렬 버전에서는 이러한 간격이 다른 워프의 작업으로 채워지므로 GPU가 지속적으로 바쁘게 작동합니다.

핵심 교훈은 먼저 GPU를 완전히 점유할 만큼 충분한 병렬 작업을 확보하는 것입니다. 높은 점유율(지연 시간을 커버할 만큼 충분한 워프)은 처리량을 극대화하고 유휴 정지를 최소화합니다. 본 예시에서는 GPU 활용도를 약 95%까지 끌어올리는 병렬화를 통해 이를 달성했습니다.

충분한 스레드가 실행되면 다음 단계는 명령어 수준 병렬 처리 및 기타 스레드별 개선을 통해 각 워프의 실행 효율을 최적화하는 것입니다. 그러나 주의할 점은: 100% 점유율에서도 작업 부하가 메모리 바운디드(즉, 연산이 아닌 느린 메모리 액세스로 제한됨)인 경우 성능이 저하될 수 있다는 것입니다.

의 메모리 바운디드 워크로드로 잘 알려진 예는 LLM의 "디코딩" 단계입니다. 디코딩 과정에서 LLM은 대량의 데이터(모델 가중치 또는 매개변수)를 글로벌 HBM 메모리에서 GPU 레지스터 및 공유 메모리로 이동시켜야 합니다.

현대 LLMs은 수천억 개의 매개변수(매개변수당 8비트, 즉 1바이트로 가정)를 포함하므로 모델 크기가 수백 기가바이트에 달할 수 있습니다. 이 정도의 데이터를 GPU로 이동하거나 GPU에서 꺼내는 작업은 메모리 대역폭을 쉽게 포화시킬 수 있습니다.

GPU FLOPS가 메모리 대역폭을 앞지르고 있습니다. 예를 들어, Blackwell의 HBM3e는 약 8TB/s를 제공하지만, 컴퓨팅 성능과 모델 크기는 더 빠르게 증가하고 있습니다. 따라서 현대 AI 워크로드에서 메모리 바운디드 병목 현상을 피하기 위해 메모리 이동 최적화는 절대적으로 중요합니다.

런치 바운디드로 점유율 조정

경우에 따라 단순히 더 많은 스레드( )를 사용하는 것만으로는 충분하지 않습니다. 특히 각 스레드가 레지스터나 공유 메모리 같은 자원을 많이 사용할 때 그렇습니다. CUDA의 커널 어노테이션(`__launch_bounds__`)을 사용하면 컴파일러가 점유율 최적화를 수행하도록 유도할 수 있습니다.

이 어노테이션을 통해 컴파일 시점에 커널에 대해 두 가지 매개변수를 지정할 수 있습니다: 블록당 런칭할 최대 스레드 수와 각 SM에 상주하도록 유지할 최소 스레드 블록 수입니다. 이러한 힌트는 컴파일러의 레지스터 할당 및 인라인 결정에 영향을 미칩니다. 예시는 다음과 같습니다:

```
__global__ __launch_bounds__(256, 16)
void myKernel(...) { /* ... */ }
```

여기서 ` `__launch_bounds__(256, 16)` `는 CUDA 커널이 블록당 256개 이상의 스레드로 실행되지 않도록 보장합니다. 또한 컴파일러가 충분한 레지스터를 할당하고 함수를 인라인 처리하여 최소 16개의 블록(블록당 256개 스레드) 또는 4,096개 스레드(16블록 × 블록당 256스레드)가 SM에 동시에 상주할 수 있도록 요청합니다.

현대 GPU(예: Blackwell)에서는 블록당 최대 1,024개 스레드, SM당 최대 2,048개 상주 스레드만 가질 수 있다는 점을 기억하십시오.

실제 환경에서는 현재 NVIDIA GPU가 각 SM당 총 스레드 수를 2,048개로, 각 블록당 스레드 수를 1,024개로 제한하므로 컴파일러는 요청을 하드웨어 최대치(이 경우 SM당 2,048개 스레드, 즉 블록당 256개 스레드 × 8블록)로 축소합니다. 또한 4,096개 스레드 요청(블록당 256개 스레드 × 16개 블록)이 SM 용량을 초과하므로 경고가 발생합니다.

경고 메시지는 다음과 유사할 것입니다: "ptxas 경고: SM당 스레드 수 값이 범위 초과입니다. `.minnctapersm` 은 무시됩니다."

실제 사용 시, ' `__launch_bounds__` ' 사용은 스필링을 방지하고 더 높은 점유율을 허용하기 위해 컴파일러가 스레드당 레지스터 사용량을 제한(때로는 언

롤링이나 인라이닝을 제한)하도록 하는 경우가 많습니다. 우리는 본질적으로 스레드당 성능을 약간 희생하고 모든 레지스터를 사용하지 않거나 언롤링을 최대한 활용하지 않는 대신, 더 많은 워프를 동시에 실행함으로써 더 일관된 워프 처리량을 얻습니다.

점유율 증가는 스레드별 리소스와 균형을 맞춰야 합니다. 레지스터 스필링(레지스터가 부족해져 로컬 메모리로 스필링되어 느린 메모리 액세스를 유발하는 현상)을 피하려면 스레드당 레지스터 사용량을 제한해야 합니다.

CUDA 점유율 API를 사용해 런타임에 최적의 런치 구성을 결정할 수도 있습니다. 예를 들어, `cudaOccupancyMaxPotentialBlockSize()` 는 주어진 커널의 레지스터 및 공유 메모리 사용량을 고려하여 가장 높은 점유율을 생성하는 블록 크기를 계산합니다. 기본적으로 `cudaOccupancyMaxPotentialBlockSize` 는 최적의 점유율을 위해 블록 크기를 자동 조정할 수 있습니다.

```
int minGridSize = 0, bestBlockSize = 0;

// If your kernel uses dynamic shared memory (extern __shared__),
// set this correctly:
size_t dynSmemBytes = /* bytes per block (e.g., tiles * sizeof(T)) */ 0;

cudaOccupancyMaxPotentialBlockSize(
  &minGridSize, &bestBlockSize,
  myKernel,
  dynSmemBytes,      // must match your kernel's dynamic shared memory use
  /* blockSizeLimit = */ 0);

// Compute a grid that covers N, but don't go below the min grid
// that saturates occupancy
int gridSize = std::max(minGridSize, (N + bestBlockSize - 1) / bestBlockSi

myKernel<<<gridSize, bestBlockSize, dynSmemBytes>>>(...);
```

이 API는 커널의 리소스 사용량을 고려하여 블록당 스레드 수가 점유율을 최적화할 가능성이 얼마나 되는지 계산합니다. 그런 다음 커널 실행에 `bestBlockSize` (및 제안된 그리드 크기)를 사용할 수 있습니다. `minGridSize` 는 이 장치에서 이 커널의 점유율을 포화시키는 최소 그리드 크기라는 점을 유의해야 합니다. 이는 길이 N의 입력을 커버하기 위한 올바른 그리드 크기가 아닐 수 있습니다. `gridSize = max(minGridSize, ceil_div(N, bestBlockSize))` 를 계산하고, 커널이 extern `__shared__` 를 사용하는 경우 커널의 실제 동적 공유 메모리 바이트를 전달하십시오.

유효한 점유율 API 제안은 ±1–2 후보 블록 크기로 커널 실행 시간을 측정하여 검증하십시오. 최신 GPU에서 레지스터 압박과 L2 동작은 실제로 약간 미흡한 점유율 구성이 실제 성능에서 더 빠를 수 있습니다.

컴파일러의 휴리스틱은 일반적으로 효과적이지만, 필요 시 `__launch_bounds__` 및 점유율 계산기를 통해 명시적으로 제어할 수 있습니다. 커널이 스레드당 리소스 사용량을 일부 희생하여 더 많은 활성 워프를 확보할 수 있다고 *판단될* 때 이를 활용하세요. 이는 스레드 부하로 인한 SM의 점유율 부족을 방지하는 데 도움이 됩니다.

레지스터당 스레드 수( )와 점유율 간의 균형은 중요합니다. 스레드당 레지스터 수를 줄이거나 런치 바운디드로 제한하면 더 많은 워프가 상주할 수 있어 지연 시간 숨김이 개선됩니다. 그러나 레지스터를 너무 적게 사용하면 컴파일러가 데이터를 로컬 메모리에 스필하게 되어 성능이 저하될 수 있습니다. 최적점을 찾는 데는 종종 실험이 필요합니다. Nsight Compute의 "Registers Per Thread" 및 "Occupancy" 메트릭이 이를 안내할 수 있습니다.

NVIDIA Compute Sanitizer를 통한 기능적 정확성 디버깅

CUDA 애플리케이션은 커널당 수천 개의 스레드( )를 생성할 수 있으므로, 기존 디버깅 방식으로는 미묘한 메모리 오류나 경합 조건(race condition)을 포착하지 못할 수 있습니다. CUDA 툴킷에 포함된 기능적 정확성 검증 도구인 NVIDIA Compute Sanitizer는 런타임에 코드를 계측하여 개발 초기 단계에서 오류를 찾아내는 방식으로 이러한 문제를 해결합니다. 이를 통해 디버깅 작업이 줄어들고 전체적인 코드 신뢰성이 향상됩니다.

Sanitizer는 `compute-sanitizer` CLI를 사용하여 호출되며, 더 정밀한 분석을 위해 NVIDIA Tools Extension(NVTX) 어노테이션을 지원합니다. NVTX는 정확성 및 성능 분석 모두에 광범위하게 사용되어야 합니다. CLI를 사용하려면 `--option value` 로 옵션을 지정하고 `--error-exitcode` 와 같은 플래그를 포함시켜 오류 발생 시 실패하도록 설정할 수 있습니다. 또한 `--kernel-name` `--kernel-name-exclude` 와 같은 플래그를 포함할 수 있습니다. 예를 들어, `--nvtx yes` 로 NVTX를 활성화하면 분석 범위를 좁히고 메모리 누수 보고서에서 오탐을 최소화하는 데 도움이 됩니다:

```
compute-sanitizer [--tool toolname] [options] <application> [app_args]
```

커널 필터와 NVTX 영역 주석을 사용하여 정확도 저하를 포착하기 위해 `--error-exitcode` 를 통해 컴퓨트 샌티나이저를 지속적 통합(CI) 파이프라인에 통합하는 것이 권장됩니다.

컴퓨트 샌티라이저는 memcheck, racecheck, initcheck, synccheck 네 가지 주요 도구로 구성됩니다. 이 도구들은 CUDA 코드에서 범위를 벗어난 메모리 접근, 데이터 경합, 초기화되지 않은 메모리 읽기, 동기화 문제를 탐지하는 데 도움을 줍니다:

메모리 검사

memcheck 도구는 전역, 로컬, 공유 메모리에서 범위를 벗어난 접근이나 정렬 오류 접근을 정확히 탐지하고 원인을 규명합니다. GPU 하드웨어 예외를 보고하며, 장치 측 메모리 누수를 식별할 수 있습니다. 명령줄 스위치를 통해 힙 할당에 대한 동적 메모리 할당(`--check-device-heap`)과 같은 추가 검사를 지원합니다.

Racecheck

Racecheck는 비결정적 동작을 유발할 수 있는 공유 메모리 데이터 위험(Write-After-Write, Write-After-Read, Read-After-Write 포함)을 보고합니다. Racecheck는 개발자가 워프 및 스레드 블록 내 올바른 스레드 간 통신을 검증하는 데 도움을 줍니다.

Initcheck

Initcheck는 초기화되지 않은 장치 전역 메모리에 대한 초기화 후 액세스( )를 표시합니다. 이는 호스트-장치 간 복사 누락 또는 장치 측 쓰기 생략으로 발생할 수 있습니다. 이 도구는 오래된 데이터나 쓰레기 데이터로 인해 발생하는 미묘한 버그를 방지하는 데 도움이 됩니다.

Synccheck

Synccheck는 불일치하는 배리어(barrier)와 같은 동기화 프리미티브의 잘못된 사용을 감지합니다. 이는 스레드 간 교착 상태(deadlock) 및 상태 불일치를 유발할 수 있는 스레드 순서 위험(thread-ordering hazard)을 식별합니다.

요약하면, NVIDIA Compute Sanitizer는 CUDA 애플리케이션의 메모리, 경합, 초기화 및 동기화 버그를 발견하고 해결하기 위한 도구 세트를 제공합니다. 이러한 도구를 CI 시스템과 통합하면 개발자가 정확성 문제를 조기에 발견하는 데 도움이 됩니다. 이를 통해 신뢰할 수 있고 고성능의 코드를 자신 있게 배포할 수 있습니다.

루프라인 모델: 컴퓨트 바운디드 또는 메모

리 바운디드 워크로드

루프라인 모델은 두 가지 하드웨어 성능 한계선을 시각화한 유용한 개념도( )입니다. 하나는 프로세서의 최대 부동소수점 연산 속도에 해당하는 수평선이고, 다른 하나는 최대 메모리 대역폭에 의해 설정된 대각선입니다. 이 두 선이 함께 '루프라인'이라는 외곽선을 형성하며, 특정 커널이 연산(컴퓨트 바운드)에 의해 제한되는지 데이터 이동(메모리 바운드)에 의해 제한되는지를 보여줍니다.

이 두 선이 교차하는 지점을 *능선점(ridge point)*이라 합니다. 이는 커널이 메모리 바운드(능선 좌측)에서 컴퓨트 바운드(능선 우측)로 전환되는 "산술 집도(arithmetic intensity)" 임계값에 해당합니다. 산술 집도는 오프칩 글로벌 메모리와 GPU 간 전송된 바이트당 수행된 FLOPS 수로 측정됩니다.

산술 집약도가 중요한 이유를 설명하기 위해 간단한 예를 들어 보겠습니다. 커널이 두 개의 32비트 부동 소수점(총 8바이트)을 로드하고, 이를 더한 후(1 FLOP), 하나의 32비트 부동 소수점 결과(4바이트)를 다시 쓰는 경우를 가정해 보겠습니다. 이 경우 알고리즘은 12바이트의 메모리 트래픽에 대해 1 FLOP을 수행하여 산술 집약도가 0.083 FLOPs/바이트(1 FLOP/12바이트 ≈ 바이트당 0.083 FLOPs)가 됩니다.

이를 GPU의 리지드 포인트인 바이트당 10 FLOPs(10 FLOPs = ~80 TFLOPs ÷ 8 TB/s)와 비교해 보십시오. 이 부동소수점 덧셈 커널의 리지 포인트 0.083은 루프라인의 왼쪽(메모리 제약 측면)으로 수십 배나 떨어져 있습니다. 이는 해당 임계값보다 100배 이상 낮아 산술 논리 장치(ALU)를 지속적으로 가동시킬 수 없습니다. 이 커널은 메모리 제약 영역에 속하며, 성능이 연산이 아닌 메모리 대기 시간에 의해 좌우됩니다. 그림 6-21은 블랙웰의 대표적 루프라인 모델을 보여줍니다. 여기에는 최대 연산 성능(수평선, 약 80 FLOPs/초)과 최대 메모리 대역폭(대각선, 8 TB/s에 해당)이 포함됩니다.

여기서 블랙웰 GPU의 능선점은 지속 FLOPs/초를 지속 HBM 대역폭으로 나눈 값임을 알 수 있습니다. 여기서 교차점은 10 FLOPs/바이트로 표시됩니다. 예시 커널의 산술 집약도는 0.083 FLOPs/바이트에서 메모리 대역폭 대각선(경사선)을 따라 왼쪽에 위치합니다. 따라서 이 커널은 지붕선(roofline)의 경사진 메모리 대역폭 상한선에 놓여 있습니다. 이는 메모리 바운디드가 발생함을 확인시켜 줍니다.

이 커널의 메모리 바운디드를 완화하고(따라서 컴퓨팅 바운디드로 전환하려면) 데이터 바이트당 더 많은 작업을 수행하여 산술 집약도를 높일 수 있습니다. 이렇게 하면 커널이 오른쪽으로 이동하여 성능이 컴퓨팅 루프라인 쪽으로 상승합니다.

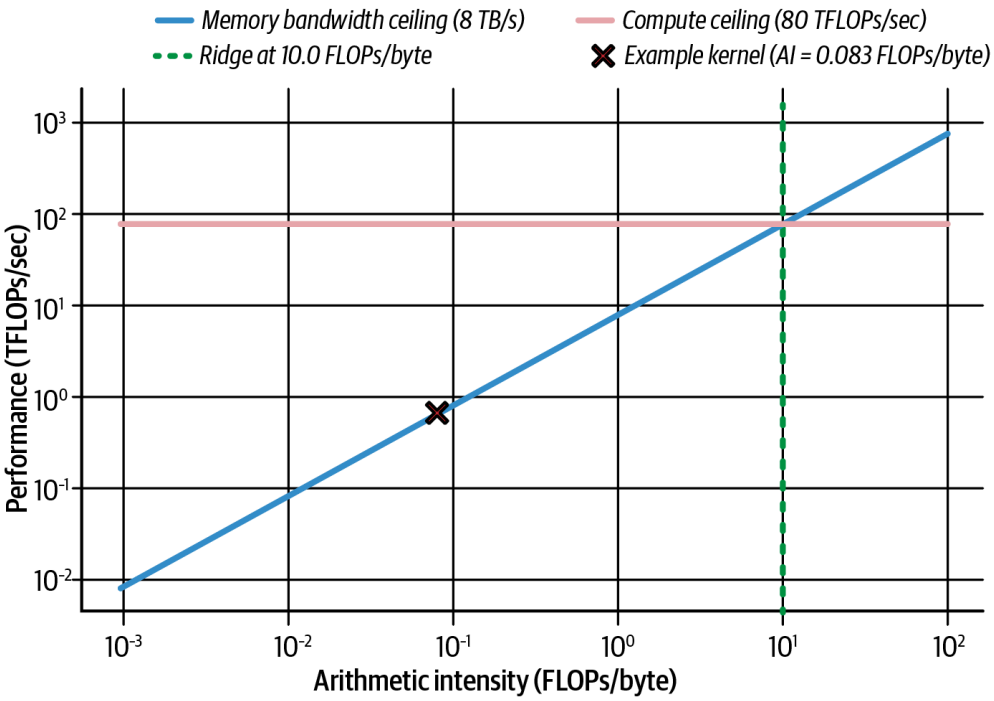

그림 6-21. Blackwell급 GPU(약 80 TFLOPs/초 FP32, 약 8 TB/초 HBM3e)의루프라인 모델. 커널의 위치와 약 10 FLOPs/바이트의 산술 집약도 능선을 보여줍니다.

커널의 메모리 바운디드를 완화하는 간단한 방법 중 하나는 낮은 정밀도 데이터를 사용하는 것입니다. 예를 들어, 32비트(FP32) 대신 16비트 부동소수점(FP16)을 사용하면 작업당 전송되는 바이트 수가 절반으로 줄어들어 즉시 FLOPs/바이트 집약도가 두 배로 증가합니다.

현대 GPU는 전용 8비트 부동소수점(FP8) 텐서 코어도 지원합니다. 블랙웰은 특정 AI 워크로드용 4비트 부동소수점(FP4) 텐서 코어에 대한 네이티브 지원도 도입했습니다. 이는 연산당 바이트 수를 더욱 줄이고 FLOPs/바이트 집약도를 더욱 높입니다.

예를 들어, Blackwell은 FP8 텐서 코어(값당 1바이트)를 지원하여 FP16 대비 처리량을 두 배로 늘리고 메모리 사용량을 절반으로 줄입니다. 또한 모델 추론과 같은 일부 워크로드에 FP4(값당 0.5바이트)도 지원합니다.

단일 128바이트 메모리 트랜잭션으로 32개의 FP32, 64개의 FP16, 128개의 FP8 또는 256개의 FP4 값을 전송할 수 있습니다. 블랙웰은 압축된 모델 가중치를 가속화하기 위해 하드웨어 디컴프레션을 도입했습니다. 예를 들어, 모델은 FP4 압축을 넘어 HBM에 압축된 상태로 저장될 수 있으며, 하드웨어가 가중치를 실시간으로 디컴프레션합니다. 이는 해당 가중치를 읽을 때 사용 가능한 메모리 대역폭을 효과적으로 더욱 증가시킵니다.

따라서 Blackwell은 트랜스포머 기반 토큰 생성 같은 메모리 제약 워크로드에 아키텍처적 이점을 가집니다. 가중치는 압축된 4비트 또는 2비트 방식으로 저장되며, 로드 시점에 하드웨어가 압축을 해제하고 FP16/FP32로 캐스팅하여 고정밀 집계 및 연산을 수행합니다. 이는 낮은 정밀도가 데이터 전송량을 줄이고 커널의 산술 집약도를 높이며 워크로드의 전체 메모리 처리량을 개선하는 방식을 보여줍니다.

메모리 바인딩 워크로드의 목표는 커널의 작동 지점을 루프라인 상에서 우측으로 이동시켜 연산 집약도를 높이고 컴퓨팅 바운디드에 가까워지는 것입니다. 컴퓨팅 바운디드 영역에 가까워질수록 커널은 GPU의 전체 부동소수점 성능을 더 효과적으로 활용할 수 있습니다.

트랜스포머 기반 모델(예: LLMs)은 서로 다른 단계에서 연산 바운디드(compute bound)이거나 메모리 바운디드(memory bound)일 수 있습니다. 예를 들어, 어텐션 레이어(프리필 단계)는 일반적으로 연산 바운디드한 반면, 행렬 곱셈(디코드 단계)은 종종 메모리 바운디드합니다. 추론에 대해 깊이 있게 다루는15~18장에서 이에 대해 더 논의하겠습니다.

커널이 메모리 바운디드 상태일 때, Nsight Compute는 매우 높은 DRAM 대역폭 사용률과 함께 낮은 ALU 사용률과 같은 낮은 컴퓨팅 메트릭을 보고합니다. 이는 워프가 대부분의 시간을 메모리 액세스 대기 상태에서 소비함을 나타냅니다.

발생 중인 현상을 심층 분석하려면 Nsight Compute를 활용하여 지연 시간, 캐시 적중률, 워프 발행 스톨 등 커널별 카운터를 확인하는 것이 가장 효과적입니다. 또한 최신 버전의 Nsight Compute는 범위 재생(명령어 수준 소스 메트릭 포함), 개선된 소스 상관관계 탐색, 런치 스택 크기 메트릭을 제공합니다. 이러한 기능들은 종속성 스톨, 레지스터 압박, 런치 구성 효과를 더 신속하게 진단하는 데 도움이 됩니다.

그런 다음 Nsight Systems를 사용하여 GPU 유휴 간격, CPU 작업과의 중첩, PCIe/NVLink 전송을 보여주는 종합적인 타임라인 뷰를 확인할 수 있습니다. 이 두 도구를 함께 사용하면 *'왜'* (어떤 스톨과 어떤 리소스)와 *'언제'* (해당 스톨이 애플리케이션의 전체 실행에 어떻게 부합하는지)를 모두 파악할 수 있습니다.

핵심은 Nsight Compute와 Nsight Systems의 메트릭을 활용해 반복적으로 자질을 파악하고 메모리 핫스팟을 식별하는 것입니다. 의심스러운 코드 주변에 NVTX 범위를 추가하고, 타임라인 동작을 확대하여 피드백을 활용해 최적화해야 합니다.

예를 들어 NVTX를 사용해 영역을 "메모리 복사" 또는 "커널 실행"으로 표시하고 Nsight Systems 타임라인에서 확인할 수 있습니다. 이는 앞서 논의한 호스트-장치 전송과 컴퓨팅 작업의 중첩을 확인하는 데 매우 유용합니다.

예를 들어 중첩을 확인하려면 NVTX 마커로 데이터 전송과 커널 호출의 시작/종료를 표시하세요. Nsight Systems는 타임라인에 이 NVTX 범위를 표시하여 중첩을 쉽게 확인할 수 있게 합니다. 비동기 메모리 복사(`cudaMemcpyAsync`)의 경우, 데이터 전송이 GPU에서 커널 실행과 중첩됩니다(그림 6-22 참조). 이는 동기식 메모리 전송과 비동기식 메모리 전송을 비교한 것입니다.

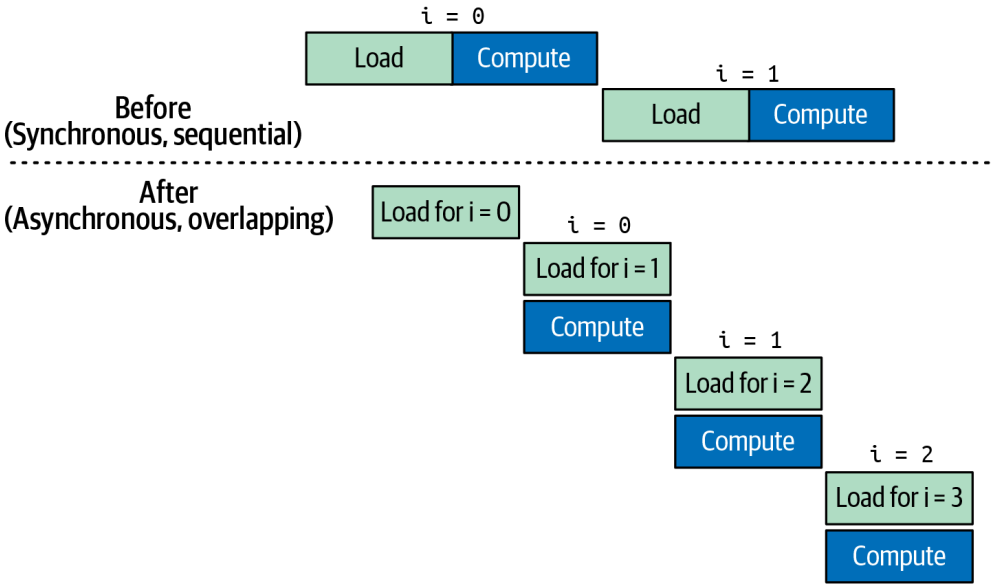

그림 6-22. 커널 연산과동기식 (순차적) 및 비동기식(중첩) 데이터 전송

중첩이 예상되지만 복사 작업과 커널이 병렬이 아닌 순차적으로 실행되는 경우, 원치 않는 기본 스트림 동기화 같은 문제가 있을 수 있습니다. 그렇지 않다면 고정 메모리 버퍼가 누락되어 진정한 중첩이 방해받고 있을 가능성이 높습니다.

고정된(페이지 잠금) 메모리를 사용하지 않으면 커널 실행과 메모리 전송(`cudaMemcpyAsync`)이 중첩될 수 없습니다. 이는 흔한 성능 문제입니다.

커널이 데이터 부족 상태라고 의심될 때는 Nsight Compute와 Nsight Systems에서 실행해 보세요. Nsight Compute에서는 글로벌 로드 효율 지표가 떨어지는 것을 확인할 수 있습니다. 이는 DRAM 요청이 충분히 빠르게 처리되지 않음을 의미합니다. 동시에 Nsight Systems 타임라인에서는 GPU가 데이터 전송을 기다리는 동안 커널 실행 사이에 발생하는 유휴 구간이 드러납니다.

본 장에서 설명한 메모리 계층 최적화를 적용하면 이러한 유휴 간격이 거의 사라지고, Nsight Compute에서 메모리 파이프 활용률 백분율이 최대치로 상승하는 모습을 확인할 수 있습니다. 또한 엔드투엔드 커널 처리량도 이에 상응하는 증가를 보일 것입니다.

모든 변경 후에는 반드시 측정하십시오. 자질 도구를 통해 최적화가 실제로 메모리 스톨을 줄였는지 확인할 수 있습니다.

핵심 요약

이 장에서는 점유율 최적화를 위한 런치 매개변수 선택 방법, GPU 메모리의 비동기적 관리 방법, 그리고 컴퓨팅 바운드 커널과 메모리 바운드 커널을 구분하기 위한 루프라인 분석 적용 방법을 배웠습니다. 다음은 복습할 가치가 있는 주요 내용입니다:

SIMT 실행 모델

GPU는 단일 명령 다중 스레드(SIMT) 모델 하에서 워프(32개 스레드) 단위로 스레드를 실행하며, 각 워프는 동기화된 명령 실행을 수행합니다. 높은 점유율(다수의 워프를 동시에 실행)은 메모리 및 파이프라인 지연 시간을 숨깁니다.

스레드 계층 구조: 스레드 → 잠금 → 그리드

스레드는 스레드 블록(최대 1,024개 스레드)으로 그룹화되며, 스레드 블록은 그리드를 형성하여 코드 변경 없이 수백만 개의 스레드로 확장됩니다. 동기화(`__syncthreads()` 또는 협력적 그룹)는 공유 메모리에서 데이터 재사용을 가능하게 하지만 오버헤드를 발생시키므로 배리어를 최소화하십시오.

점유율 대 리소스 제한

블록 크기는 32의 배수로 선택하여 워프 미충전 현상을 방지하고 스케줄러 활용도를 극대화하십시오. SM당 제한 사항을 유의하십시오.
Blackwell의 경우 스레드당 최대 레지스터는 255개, SM당 공유 메모리는 228KB, 상주 워프는 64개, 상주 스레드 블록은 32개입니다.

CUDA 커널 런치 매개변수

점유율과 자원 사용의 균형을 위해 8개 워프(`threadsPerBlock = 256`)로 시작하십시오. 모든 요소를 커버하려면 16개 워프(`blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock`)를 계산하십시오. 프로파일링 피드백(레지스터/레지스터 스필링, 공유 메모리 사용량, 달성된 점유율)을 기반으로 이러한 값을 조정하십시오.

비동기 메모리 관리

전용 스트림에서는 `cudaMallocAsync` / `cudaFreeAsync` 방식을 우선 적용하고, CUDA 메모리 풀을 활용하여 전역 동기화와 OS 수준 오버헤드를 방지하십시오. PyTorch의 캐싱 할당기는 효율적인 텐서 할당을 위해 유사한 패턴을 따르며, 비용이 많이 드는 ` `cudaMalloc()` ` 및 ` `cudaFree()` ` 호출을 피합니다.

GPU 메모리 계층 구조

레지스터 → L1/공유 → L2 → 글로벌(HBM3e) → 호스트: 각 레벨은 용량과 지연 시간/대역폭을 교환합니다. 레지스터와 공유/L1 캐시에서 데이터 재사용을 극대화하세요.

통합 메모리 고려 사항

CUDA 관리 메모리(통합 메모리)는 프로그래밍을 단순화하지만 암묵적 페이지 마이그레이션이 발생할 수 있습니다. 예상치 못한 스톨을 피하려면 `cudaMemPrefetchAsync` 및 메모리 조언을 활용하세요.

루프라인 모델 분석

산술 집약도(바이트당 FLOPS)는 커널이 메모리 바운드인지 컴퓨트 바운드인지 결정합니다. 낮은 정밀도(FP16/FP8/FP4 및 하드웨어 압축 해제)를 사용하여 FLOPS/바이트 비율을 높이고 커널을 컴퓨트 루프라인 쪽으로 밀어냅니다. Nsight Compute(커널별 메트릭) 및 Nsight Systems(타임라인)으로 자질을 분석하여 메모리 스톨을 식별하고 제거하십시오. TMEM을 Blackwell 통합 행렬 곱셈-누적(UMMA)과 함께 사용하면 FP8 및 FP4와 결합 시 커널을 메모리 바운디드에서 컴퓨팅 바운디드으로 전환할 수 있습니다. UMMA에 대해서는 10장에서 자세히 다룹니다.

결론

이 장에서는 GPU의 SIMT 모델, 스레드 계층 구조, 다단계 메모리 시스템을 명확히 설명함으로써 고성능 CUDA 개발의 기초를 마련했습니다. 점유율(활성 워프 대 이론적 GPU 최대 용량의 비율)이 지연 시간 숨기기에 중요하다는 점을 기억하십시오.

그러나 점유율을 극대화한다고 해서 모든 경우에 최상의 성능이 보장되는 것은 아닙니다. 스레드가 충분한 명령어 수준 병렬성(ILP)을 갖거나 다른 리소스가 병목 현상을 일으키는 경우, GPU는 중간 또는 낮은 점유율에서도 매우 높은 처리량을 달성할 수 있습니다.

점유율이 높을수록 지연 시간 숨김에 도움이 되지만, 활성 스레드 수를 줄여 다른 스레드에 레지스터를 확보할 수 있는 시나리오도 존재합니다. 이는 스레드당 더 많은 계산을 가능하게 하여 궁극적으로 처리량을 향상시킵니다. 항상 다양한 점유율 수준을 벤치마킹하여 작업 부하와 하드웨어에 최적의 설정을 찾으십시오.

이러한 기본 원리와 자질 기법을 숙지했다면, 이제 워프 발산 방지, GPU 메모리 계층 구조 활용, 비동기 메모리 프리페칭과 같은 목표 지향적 최적화에 착수할 준비가 되었습니다. 대량 메모리 전송을 처리하고 GPU가 유용한 작업에 집중하여 계산 유효 처리량을 높일 수 있도록 해주는 TMA(Transaction Memory Accelerator)에 대해서도 살펴보겠습니다.