제9장. CUDA 커널 효율성 및 산술 집약도 향상

대규모 병렬화와 높은 ILP로 지연 시간을 완전히 숨기더라도, 커널의 성능은 메모리 액세스당 수행하는 유용한 작업량에 의해 여전히 제한될 수 있습니다. *산술 집약도*(*연산 집약도라고도 함*) 는 메모리에서 전송된 데이터 바이트당 수행되는 부동 소수점 연산 횟수, 즉 바이트당 FLOPS를 측정합니다.

신규 GPU 세대는 메모리 대역폭을 훨씬 뛰어넘는 연산 처리량을 달성하고 있습니다. 이 격차 확대로 인해 산술 집약도 증대는 그 어느 때보다 중요해졌습니다. 높은 산술 집약도는 커널이 가져온 바이트당 더 많은 계산을 수행함을 의미하며, 이는 GPU의 연산 능력을 완전히 활용하는 데 필수적입니다.

산술 집약도는 루프라인 성능 모델 에서 핵심 지표입니다. 루프라인 모델은 커널 성능(FLOPs/초)을 산술 집약도(FLOPs/바이트)에 대해 그래프로 표시하는 유용한 시각화 도구입니다. 이 모델은 메모리 대역폭과 연산 처리량에 대한 하드웨어 상한선(루프라인)을 보여주어, 커널이 메모리 전송에 의해 성능이 제한되는 메모리 바운드 상태인지, 아니면 ALU 처리량에 의해 성능이 제한되는 컴퓨트 바운드 상태인지 확인할 수 있게 합니다.

실제 작업에서는 Nsight Compute와 같은 도구로 루프라인 차트를 생성할 수 있습니다. 이 도구에는 루프라인 분석 뷰가 포함되어 있습니다. 이를 통해 커널이 초기 단계에서 메모리 바운드인지 컴퓨트 바운디드인지 확인할 수 있으며, 최적화를 진행하면서 자질과 개선 효과 검증을 계속할 수 있습니다.

목표는 커널을 컴퓨팅 바운드 영역으로 밀어붙여 GPU의 증가하는 연산 성능을 활용하는 것입니다. 루프라인 성능 모델은 이러한 목표를 향해 최적화를 올바르게 안내할 수 있습니다.

이전 장에서 살펴본 바와 같이, 루프라인 차트는 하나의 수평선으로 하드웨어의 최대 연산 처리량(루프)을 나타내고, 원점에서 시작하는 대각선은 메모리 대역폭에 의해 제한되는 달성 가능한 최대 처리량을 나타냅니다. 커널의 산술 집약도는 x축 상의 위치를 결정하며, 그림 9-1과 같이 이러한 상한선과 성능을 비교할 수 있습니다.

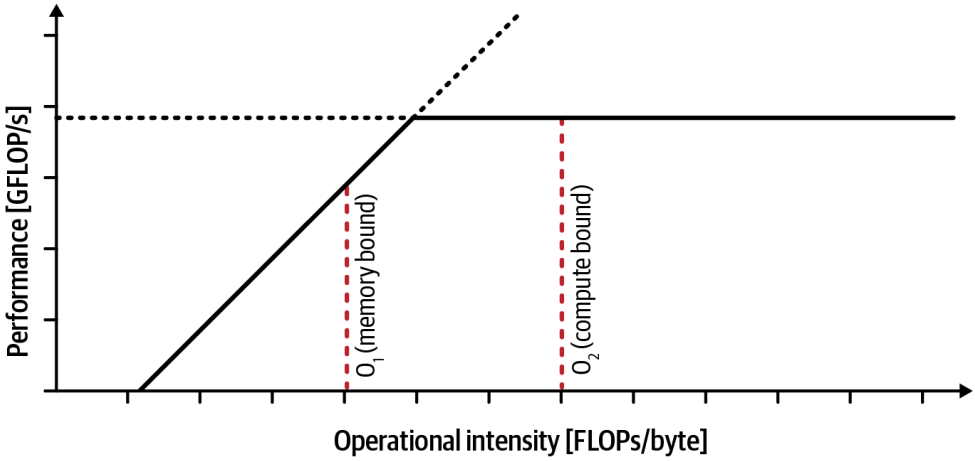

그림 9-1. 예시 루프라인 모델 (GFLOP/s 대 FLOPs/바이트 단위의 산술 집약도)

산술 집약도가 낮은 커널(, 즉 이동된 데이터 바이트당 수학 연산이 적은 경우)은 메모리 속도에 바운디드를 받습니다. 이 경우 GPU가 연산보다 데이터 대기 시간에 더 많은 시간을 소비하므로 커널 속도는 하드웨어의 메모리 대역폭에 의해 바운디드를 받습니다.

반대로, 매우 높은 산술 집약도(이동된 바이트당 많은 FLOPs)를 가진 커널은 ALU와 텐서 코어를 최대 성능에 가깝게 활용하므로 컴퓨팅에 의해 제한됩니다. 이 경우 커널의 메모리 대역폭 사용량은 부차적인 문제입니다.

목표는 항상 글로벌 메모리(GM)로 전송되는 데이터 바이트당 더 많은 연산 작업(바이트당 FLOP)을 수행함으로써 가능한 한 전역 메모리 간 연산( )의 연산 집약도를 높이는 것입니다. 데이터 재사용을 위한 루프 타일링(loop tiling), 재사용을 위한 온칩 L1/공유 메모리 활용, 중간 결과를 글로벌 메모리에 기록하지 않도록 여러 커널을 하나로 융합하는 등의 기법을 사용하여 연산 집약도를 높일 수 있습니다.

PyTorch의 TorchInductor와 같은 현대 컴파일러 프레임워크 는 GPU에서 계산을 유지하고, 오프칩 메모리 트래픽을 줄이며, 효과적인 산술 집약도를 높이기 위해 이러한 최적화 중 일부를 자동으로 수행합니다. 그러나 개발자로서, 예를 들어 데이터가 캐시에서 제거되기 전에 최적의 재사용이 이루어지도록 하기 위해 이러한 기법들을 수동으로 결합하거나 커스텀 CUDA 커널을 작성해야 할 수도 있습니다.

또한 저정밀도 데이터 유형(FP16, FP8, FP4)을 사용하여 메모리 전송량을 줄이고, 텐서 코어를 활용하여 초당 FLOPs를 증가시킬 수 있습니다. 이들을 함께 사용하면 바이트당 FLOPs 비율이 증가하고 산술 집약도가 높아집니다. 이제 이러한 기법 중 일부를 논의해 보겠습니다.

모든 워크로드가 산술 집약도를 쉽게 높일 수 있는 것은 아니라는 점을 명심하십시오. 이는 알고리즘 특성에 의해 제약받습니다. 그러나 알고리즘의 결과(예: 정확도)를 변경하지 않으면서 산술 집약도를 높이기 위해 알고리즘 개선, 데이터 재사용, 연산 융합, 배치 크기 증가 등의 기회를 모색해야 합니다.

다단계 마이크로타일링과 소프트웨어 프리페칭

7장에서 논의한 바와 같이, 타일링(일명 *청킹* 또는 *블로킹*)과 데이터 재사용은 산술 집약도를 높이는 효과적인 방법입니다. 해당 장에서는 A와 B의 작은 부분 행렬(타일)을 공유 메모리에 로드함으로써 전역 메모리에서 가져온 각 바이트를 정적 랜덤 액세스 메모리(SRAM) 속도로 여러 곱셈-누산 연산에 사용할 수 있음을 보여주었습니다.

타일링과 같이 각 요소를 한 번 로드하여 수십 또는 수백 번 사용하는 방식으로 코드를 재구성할 때마다, 바이트당 FLOPs 비율을 재사용 계수로 곱하게 됩니다. 예를 들어 일반적인 행렬 곱셈에서 A와 B의 32×32 타일은 공유 메모리 내 각 요소에 대해 1,024회(1,024 = 32 × 32)의 독립적 곱셈을 생성합니다. 따라서 각 연산마다 DRAM에서 요소를 직접 가져오는 방식에 비해 산술 집약도가 상승합니다.

단순한 공유 메모리 타일링( )을 넘어, 다단계 타일링(multilevel tiling)을 통해 연산 집약도를 더욱 높이고 더 많은 명령어 수준 병렬성(ILP)을 활용할 수 있습니다. 다단계 타일링에서는 타일을 공유 메모리에 스테이징한 후, 각 스레드가 벡터화된 형식(`float4`, `<half2>` 등)을 사용하여 마이크로타일을 레지스터에 로드합니다. 이렇게 하면 반복 작업이 완전히 레지스터 내에서 수행됩니다. 다단계 타일링의 예는 그림 9-2에 나와 있습니다.

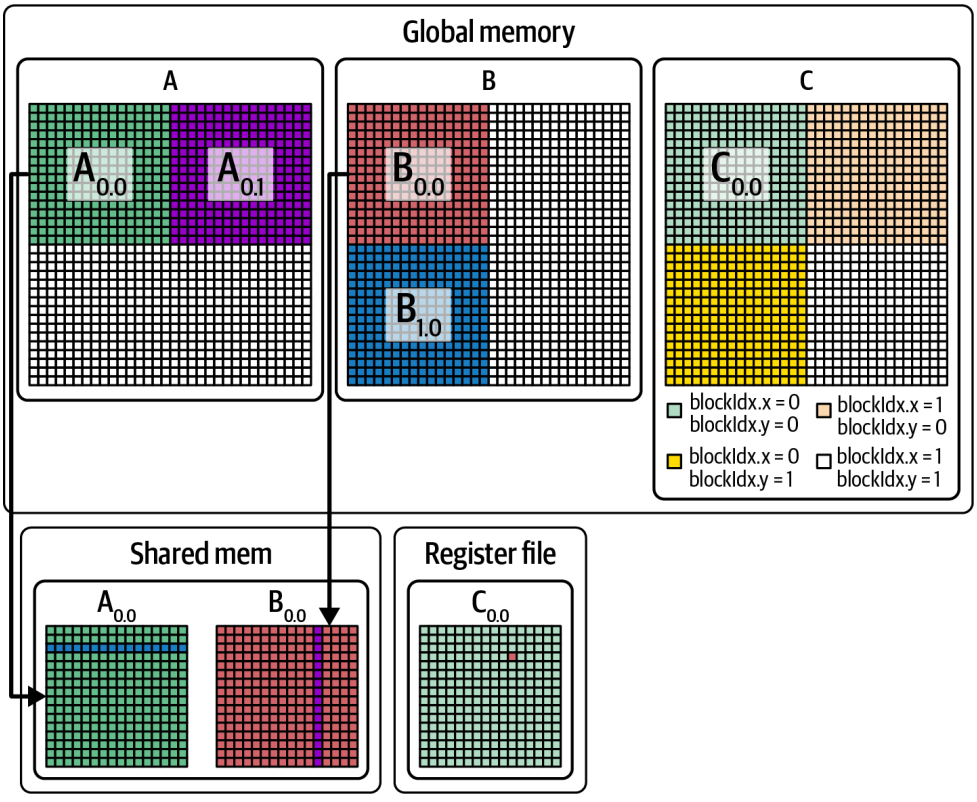

그림 9-2. 글로벌 메모리(DRAM), 공유 메모리(SMEM) 및 레지스터 간의다단계 타일링

이러한 SM 내부 재사용(레지스터 → SMEM → DRAM)은 각 레벨에서 작업 집합을 줄이고 칩 외부 트래픽을 최소화합니다. 항상 그렇듯이, 공유 메모리를 채울

때는 전역 읽기를 통합하고, 7장에서 다룬 바와 같이 메모리 뱅크 충돌을 피하기 위해 공유 데이터를 패딩/스위즐해야 합니다.

현대 GPU에서는 이러한 내부 루프 타일링 단계가 종종 MMA 프래그먼트 API 사용으로 처리됩니다. 하드웨어는 텐서 코어 명령어를 사용하여 공유 메모리와 텐서 메모리(TMEM) 간 데이터를 암시적으로 이동합니다. TMEM 사용은 컴파일러와 라이브러리가 관리합니다. 현대 GPU에서는 텐서 코어 명령어(`tcgen05`)가 공유 메모리와 TMEM 간 데이터를 암시적으로 스테이징합니다. 이들은 별도의 TMEM 주소 공간을 사용합니다. 그러나 특정 알고리즘 구현 시 명시적 제어가 필요한 경우 개발자는 여전히 공유 메모리(`cp.async`)나 TMEM(TMA)로 타일을 수동 이동할 수 있습니다.

이와 밀접하게 관련된 기법은 소프트웨어 프리페칭( )으로, 종종 더블 버퍼링(double buffering)으로 구현됩니다. 예를 들어, 현재 타일의 계산이 완료될 때까지 기다리지 않고 공유 메모리로 다음 타일에 대한 비동기 로드를 발행할 수 있습니다. 이렇게 하면 DRAM → 공유 메모리(SMEM) 전송이 진행 중인 산술 연산과 중첩됩니다. 신중한 프리페칭은 스톨 시간을 크게 줄이고 처리량을 향상시킬 수 있습니다. 핵심은 데이터 전송과 계산을 중첩시켜 ALU가 데이터 대기 상태에 빠지지 않도록 하는 것입니다.

Grace Blackwell과 같은 CPU-GPU 슈퍼칩에서 통합 메모리를 사용할 경우, `cudaMemPrefetchAsync()` 을 통해 타일이 곧 필요할 것임을 암시할 수 있습니다. 이는 런타임에 NVLink-C2C를 통해 페이지를 마이그레이션하도록 암시합니다. 그러나 프리페치는 단지 암시일 뿐 보장 사항은 아닙니다. 페이지 결함 스톨을 피하려면 전송 중첩과 적절한 동기화를 반드시 확보해야 합니다. 이러한 방식으로 데이터 이동과 계산을 중첩하면 새 타일이 필요할 때마다 ALU에 지속적으로 데이터가 공급됩니다. 이는 메모리 지연 시간을 더욱 숨기고 달성 FLOPS를 향상시킵니다.

통합 메모리는 개발을 용이하게 하지만 최상의 성능을 보장하지는 않습니다. 숙련된 사용자들은 페이지 마이그레이션 오버헤드를 완전히 피하기 위해 명시적 커널 간 메모리(`cudaMemcpy`)나 고정 메모리 할당을 선호하는 경우가 많습니다.

요약하면, DRAM의 각 바이트가 온칩 레벨(레지스터 또는 공유 메모리)에서 재사용될수록 산술 집약도가 높아집니다. 그리고 높은 산술 집약도는 커널을 컴퓨팅 병목 영역에 더 가깝게 이동시킵니다.

스레드 블록 클러스터를 활용한 타일링

현대 GPU에서는 협력 그룹(Cooperative Groups , 제10장에서 논의)의 CUDA 스레드 블록 클러스터를 활용해 타일링 재사용 개념을 확장할 수 있습니다( ). 이를 통해 여러 스레드 블록이 분산 공유 메모리(DSMEM)를 사용해 데이터를 공유할 수 있으며, 이는 그림 9-3에 표시되어 있습니다.

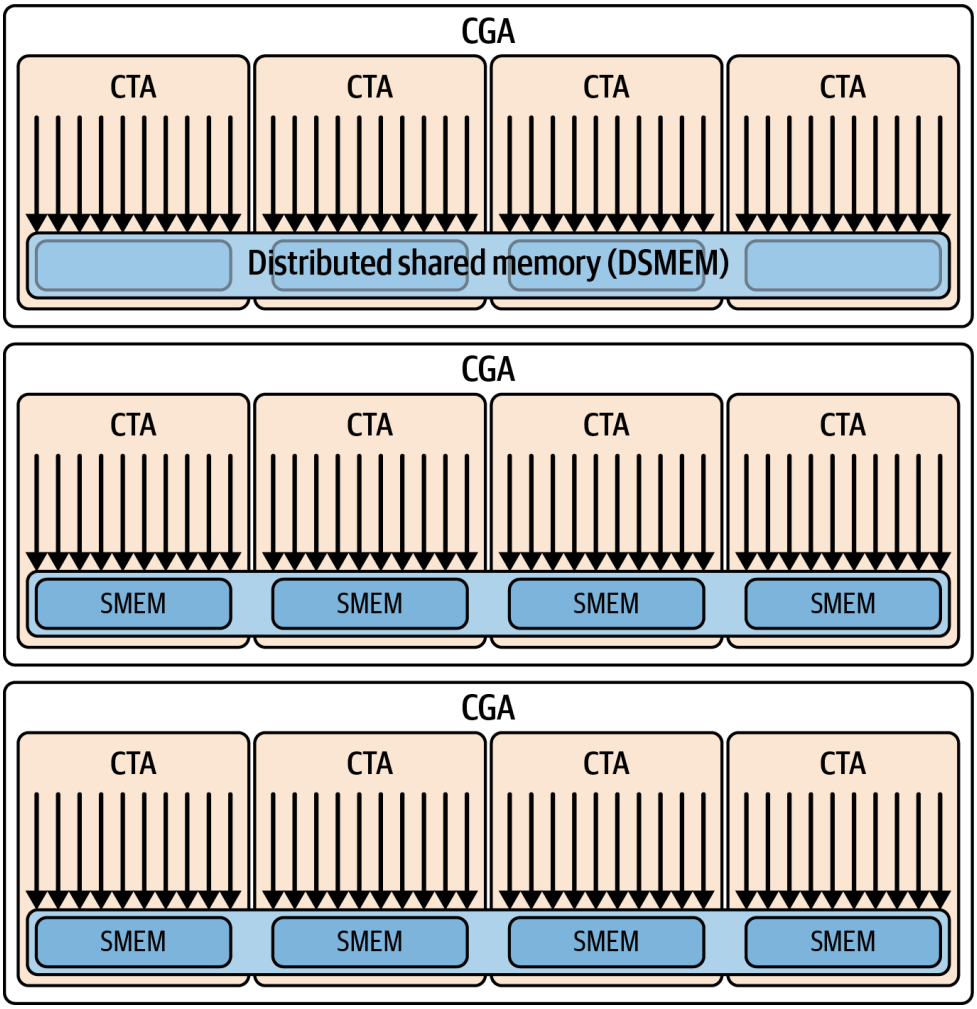

그림 9-3. CGA 내 CTA(스레드 블록) 간 공유되는DSMEM

CGA와 스레드 블록 클러스터는 다음 장에서 자세히 다루지만, 산술 집약도를 직접 높일 수 있으므로 여기서 언급할 가치가 있습니다. 예를 들어, 그림 9-4와 같이 4개의 스레드 블록 클러스터는 텐서 메모리 가속기(TMA) 멀티캐스트 기능을 사용하여 하나의 타일을 협력적으로 로드할 수 있습니다. 이 메커니즘을 보여주기 위해 4개의 CTA를 사용한 예시입니다.

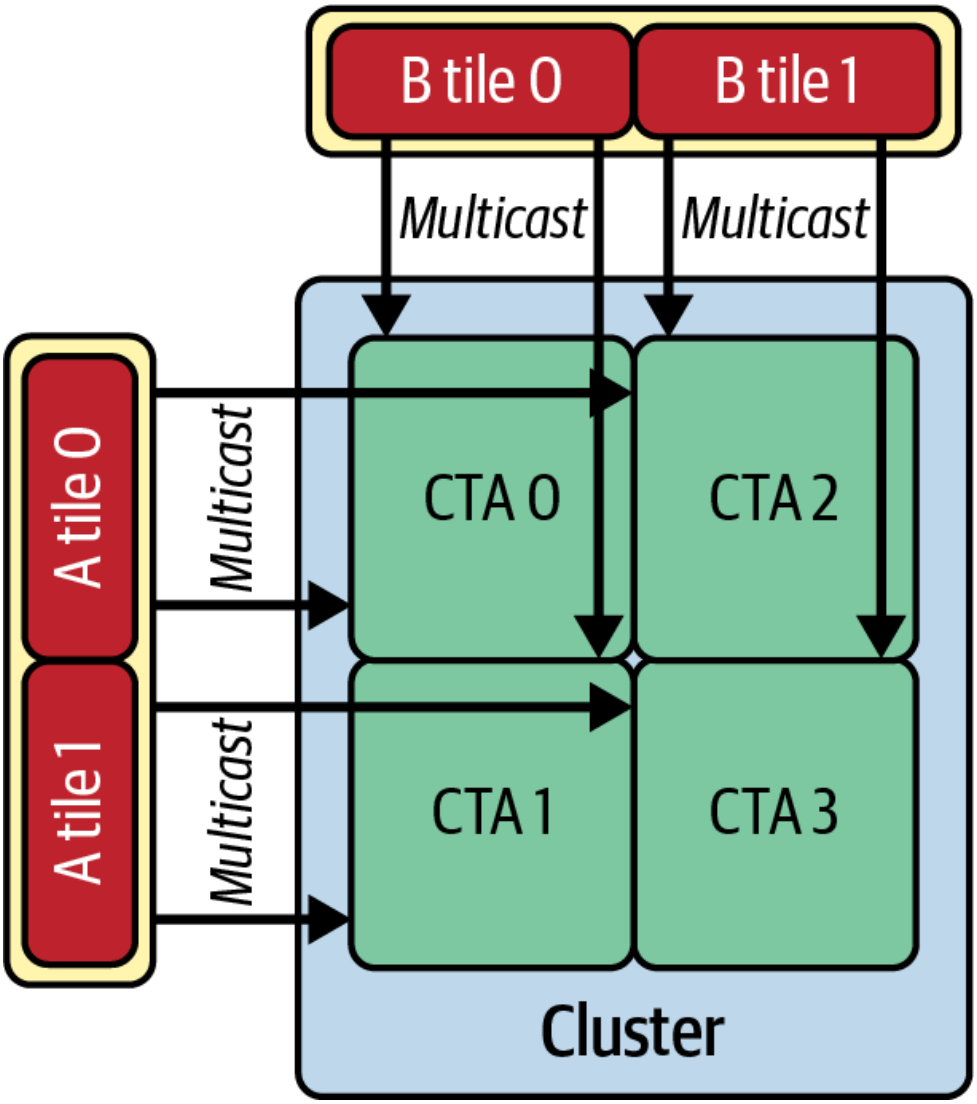

그림 9-4. 이 네 개의 (2 × 2) 스레드 블록 클러스터의경우 , A와 B의 각 타일은 멀티캐스트를 사용하여 네 개의 CTA(스레드 블록)에 동시에 로드됩니다(출처: https://oreil.ly/EEO_O)

각 타일은 네 개의 스레드 블록에 분할되어 해당 타일의 글로벌 메모리 트래픽이 클러스터 전체에 분산됩니다. 타일은 한 번만 가져온 후 네 개의 스레드 블록 모두에서 재사용됩니다.

스레드 블록 클러스터는 2×2 클러스터에서 4개의 CTA가 멀티캐스트를 통해 동일한 데이터를 재사용할 때 글로벌 메모리 트래픽을 최대 4배까지 줄일 수 있습니다. 또한 스레드 블록 클러스터는 분모(글로벌 메모리에서 이동되는 바이트 수)를 줄여 GPU당 연산 집약도를 높입니다. 이들의 특수 형태인 *스레드 블록 페어*는 텐서 코어를 활용한 타일링 맥락에서 잠시 후 논의될 예정입니다.

Blackwell은 기본 이동 가능 한계인 8개 CTA를 초과하는 비이동 가능 클러스터 크기를 선택할 경우 최대 16개 스레드 블록까지 스레드 블록 클러스터 이동( )을 지원합니다. 이를 활성화하려면 커널에 ` cudaFuncAttributeNonPortableClusterSizeAllowed ` 속성을 설정하세요. 더 큰 클러스터는 재사용률을 높일 수 있지만 점유율을 낮출 수 있으므로 16개 활성화 전 자질을 수행하세요. 이는 더 큰 다중 SM 타일(16×)을 지원하여 데이터 재사용을 극대화하고 유사한 배수로 산술 집약도를 증가시킵니다.

커널 융합

산술 집약도 를 높이는 또 다른 방법은 여러 연산 또는 루프 반복을 하나의 연산으로 융합하는 것입니다. 여러 커널을 융합함으로써 메모리에서 로드된 데이터를 여러 계산 및 반복에 사용한 후 다시 쓰기 전에 활용할 수 있습니다.

마찬가지로, 이전 섹션에서 논의한 루프 언롤링 은 로드된 각 데이터 요소에 대해 단일 스레드가 더 많은 계산을 수행할 수 있게 하지만, 더 많은 레지스터 사용을 대가로 합니다. 과도한 융합은 스레드당 레지스터 압박을 증가시키고 점유율을 감소시킬 수 있으므로, 상충 관계가 존재합니다.

융합된 커널은 항상 자질을 확인하십시오. 레지스터 사용량이 과도해져 로컬 메모리로 스필링되기 시작하면, 융합의 이점이 추가 메모리 트래픽으로 상쇄될 수 있습니다. 그러나 적절한 균형을 찾으면 이동된 바이트당 FLOPS를 향상시킬 수 있으며, 이는 메모리 대역폭이 제한 요소일 때 유리합니다.

현대적인 Deep Learning 프레임워크는 저스트인타임 컴파일러와 그래프 최적화기를 통해 자동으로 연산을 융합하고 언롤링할 수 있습니다( ). 예를 들어 PyTorch의 `torch.compile`, 특히 TorchInductor는 요소별 연산 시퀀스를 자동으로 융합합니다. PyTorch 컴파일러는 13장과 14장에서 다룹니다.

요소별 연산(포인트별 연산이라고도 함) 은 텐서의 각 요소에 대해 독립적으로 간단한 계산을 적용합니다.

이러한 요소별 연산을 융합하면 중간 값을 온칩에 유지함으로써 불필요한 메모리 트래픽을 제거합니다. 이는 전역 메모리에서 가져온 바이트당 수행되는 작업량을 증가시켜 산술 집약도를 높입니다.

예를 들어, 단순 구현은 두 개의 커널을 실행합니다. 첫 번째 커널은 `x` 를 읽고 `y` 를 글로벌 메모리에 기록합니다. 두 번째 커널은 `y` 를 읽고 `z` 를 기록합니다:

```
y = sin(x);
z = sqrt(y);
```

여기서 각 요소는 두 번 접근됩니다: `sin(x)` 이후 한 번, `sqrt(y)` 이후 한 번. 따라서 각 커널의 산술 집약도는 매우 낮습니다. 요소당(로드/스토어 작업당) 하나의 비용이 큰 수학 연산(다중 사이클 ALU 명령어)만 수행하기 때문입니다. 반면, 융합 커널은 동일한 작업 집합을 단일 패스로 수행합니다:

```
z[i] = sqrt(sin(x[i]));
```

`x[i]` 는 한 번만 로드된 후, `sin` 과 `sqrt` 가 레지스터에서 동시에 적용되며, 최종 `z[i]` 만 메모리에 기록됩니다. 중간 단계인 `y` 가 글로벌 메모리로 나가지

않기 때문에, 바이트당 유효 FLOPS가 급격히 증가하여 연산 성능이 컴퓨팅 한계에 가까워집니다.

경험적으로, 동일한 스레드 블록 내 스레드들이 데이터를 한 번 이상 읽어야 한다면, 중복된 전역 로드를 제거하기 위해 데이터를 공유 메모리에 스테이징하는 것이 종종 유용합니다. 이는 커널을 메모리 제약 영역에서 벗어나 연산 바운디드 영역으로 끌어올려 풍부한 GPU FLOPS를 더 잘 활용하는 데 도움이 됩니다.

융합은 각 요소가 메모리 읽기/쓰기 작업마다 두 개의 수학 연산을 수행하므로 전역 메모리 트래픽을 줄이고 산술 집약도를 높입니다. 본 예시에서는 중간 쓰기 작업이 없어 메모리 트래픽을 약 절반으로 줄이면서 요소당 FLOPS(`sin` + `sqrt`)를 두 배로 증가시켰습니다. 이는 산술 집약도(FLOPS/바이트)를 현저히 높입니다.

이 점을 명확히 하기 위해 구체적인 예시로 연산 집약도를 설명해 보겠습니다. 2차원 텐서 `x (shape [batch, hidden])`의 길이 숨겨진 각 행을 L2 정규화한다고 가정합니다. 각 행 `b`에 대해 단일 노름 `norm_b = sqrt(Σ_i x[b,i]*x[b,i] + ε)`을 계산한 후, 모든 `i`에 대해 `y[b,i] = x[b,i] / norm_b`를 기록합니다.

초보적인 구현 방식은 하나의 커널에서 제곱 연산을 수행하고, 두 번째 커널에서 각 행을 스칼라로 축소하며, 세 번째 커널에서 나눗셈을 수행할 것입니다. 이는 중간에 HBM에 쓰기 작업을 수행하며 여러 번의 커널 실행이 필요할 것입니다.

4개의 커널 각각이 1 FLOP의 연산을 필요로 한다고 가정합시다. 따라서 4개 커널 각각의 산술 집약도는 12바이트당 1 FLOP(부동소수점 2회 읽기, 1회 쓰기) 또는 0.083 FLOPS/바이트입니다.

대신 4개의 커널을 단일 커널로 융합하여 산술 집약도를 높일 수 있습니다. 수동으로 융합한 커널 코드는 다음과 같습니다:

```
__global__ void fusedL2Norm(const float* __restrict__ x,
                            float* __restrict__ y,
                            int hidden) {
  extern __shared__ float sdata[];        // reduction buffer
  const int batch   = blockIdx.x;         // one block per batch row
  const int tid     = threadIdx.x;
  const float* batch_ptr = x + size_t(batch) * hidden;

  // 1) Accumulate sum of squares into shared memory
  float local = 0.f;
  for (int i = tid; i < hidden; i += blockDim.x) {
    float v = batch_ptr[i];
    local   = fmaf(v, v, local);          // v*v + local
  }
  sdata[tid] = local;
  __syncthreads();
```

```
    // 2) Parallel reduction to sdata[0]
    for (int offset = blockDim.x >> 1; offset > 0; offset >>= 1) {
      if (tid < offset) sdata[tid] += sdata[tid + offset];
      __syncthreads();
    }

    // 3) Normalize (guard tiny norms)
    float norm = sqrtf(sdata[0]);
    float inv = rsqrtf(sdata[0]); // prefer inverse

    float* out_batch = y + size_t(batch) * hidden;
    for (int i = tid; i < hidden; i += blockDim.x) {
      // multiply by inverse (rsqrt) vs. divide by sqrt
      out_batch[i] = batch_ptr[i] * inv;
    }
  }
```

이 융합 커널에서 각 스레드는 `x[b,*]` 의 슬라이스를 두 번 순회합니다—한 번은 로컬 제곱합을 누적하고, 한 번은 정규화된 출력을 쓰기 위해—따라서 전역 트래픽은 요소당 약 12바이트(읽기 2회 + 쓰기 1회)입니다. 요소당 커널은 축소 과정에서 약 1회 곱셈 + 1회 덧셈을 수행하고, 정규화 과정에서 1회 곱셈을 수행합니다.

`sqrt` 와 `rsqrt` 는 전체 행에 걸쳐 분산 처리됩니다. 루프라인 배치 시 보수적인 산술 집약도는 ≈3 FLOPs / 12바이트 ≈ 0.25 FLOPs/바이트입니다(여기에 행별 `sqrt` 의 미미한 1/(숨김 연산 * 12) 기여도가 추가됩니다). 이를 통해 각 스레드에 다중 요소를 할당하여 ILP(명령어당 연산량)를 증가시켜 `sqrt` 및 `rsqrt` 의 지연 시간을 숨길 수 있습니다.

또한 이전 코드에서 보듯이, 역행렬 `sqrt` (`rsqrtf`)을 계산할 때 나눗셈 대신 곱셈을 사용합니다. 이는 특히 핫 내부 루프에서 흔히 사용되는 미세 최적화 기법입니다. 느린 나눗셈 명령어 스트림을 처리량이 높은 곱셈 명령어 스트림으로 대체하는 것이 핵심입니다. 또한 정밀도 높은 정수 연산(`sqrtf`)을 비용이 더 낮은 정밀도 낮은 정수 연산(`rsqrtf`) 근사값으로 대체하고 있습니다. 전체적으로 이 파이프라인은 메모리 바인딩(memory-bound)이지 컴퓨팅 바운디드(compute-bound)가 아니기 때문에 이는 마이크로 최적화에 해당하지만, 주목할 만한 점입니다. 여기에 표시되지 않은 또 다른 최적화가 있습니다. 스레드 블록 내 단일 스레드에서 `rsqrtf` / `sqrtf` 연산을 수행하고, 스칼라 결과를 공유 메모리를 통해 다른 스레드에 브로드캐스팅하는 방식입니다. 이 방법은 성능 향상에 더 큰 영향을 미칩니다. 이 최적화에 대한 자세한 내용은 책의 GitHub 저장소를 참조하십시오.

HBM에 중간 쓰기를 수행하는 단순한 3-커널 파이프라인(제곱 → 축소 → 나눗셈)과 비교할 때, 융합 버전은 최소 한 번의 전역 쓰기/읽기 왕복과 한 번의 런치 배리어를 제거합니다. 따라서 요소당 FLOP/바이트가 약 0.25에 불과함에도 불구하고, 지연 시간 절감과 캐시 국소성 개선 덕분에 실제 성능과 실행 시간이 훨씬 우수합니다.

실제 환경에서 이 단일 융합 커널은 더 높은 산술 집약도(FLOPS/바이트), 향상된 캐시 국소성, 그리고 세 개의 별도 커널을 하나로 통합함으로써 감소된 런치 오버헤드 덕분에 일련의 분리된 커널보다 더 빠르게 실행됩니다.

융합은 산술 집약도를 높여 커널을 컴퓨팅 바운디드 측면으로 더 가깝게 이동시킬 뿐만 아니라 메모리 대역폭도 절약합니다. 단순한 다중 커널 버전에서는 중간 결과를 글로벌 메모리에 기록하고 다음 커널에서 다시 읽어야 합니다. 융합 버전에서는 중간 결과(예: `ai*ai`)가 스레드 레지스터를 벗어나지 않아도 됩니다.

코드 예시에서 덧셈 연산은 해당 레지스터를 직접 활용해 합계를 계산할 수 있습니다. 이후 `sqrt` 는 이 합계를 사용하며, 이 모든 과정이 추가적인 전역 메모리 트래픽 없이 이루어집니다. 최종 결과만 전역 메모리에 다시 기록됩니다.

따라서 융합 커널은 전역 메모리와의 12바이트 데이터 이동으로 약 4 FLOPS를 달성하는 반면, 단순한 비융합 방식은 중간 메모리 이동을 고려할 때 36바이트를 로드 및 저장하여 4 FLOPS를 달성합니다. 이는 DRAM 트래픽 감소와 지연 시간 저하를 의미합니다.

이 간단한 예시는 융합이 커널의 연산 집약도와 전체 성능을 어떻게 향상시키는지 보여줍니다. 이제 GPU의 텐서 코어 하드웨어 유닛을 활용하여 연산 집약도를 높이는 또 다른 방법을 살펴보겠습니다.

최첨단 GPU 커널은 동일한 데이터에 대한 순차적 연산을 결합하는 수직 융합과 데이터 간 병렬 연산을 결합하는 수평 융합을 통해 더 높은 연산 집약도를 달성합니다. NVIDIA의 CUTLASS나 OpenAI의 Triton(PyTorch 컴파일러 백엔드인 TorchInductor에 통합됨)과 같은 라이브러리를 사용하면 텐서 코어, TMEM, TMA 등을 활용하여 이러한 다양한 유형의 융합 커널을 매우 효율적으로 구현할 수 있습니다.

구조적 스파시티

현대 GPU에서는 2:4 구조적 스파시티 가 스파스 텐서 코어와 cuSPARSELt에 의해 하드웨어적으로 가속됩니다. 2:4는 연속된 가중치 4개 중 정확히 2개가 0이 아님을 의미합니다. 이러한 유형의 스파시티 생성은 때때로 *'프루닝(pruning)'*이라고 불립니다.

가중치의 절반을 2:4 패턴으로 전정함으로써, 각 메모리 로드 작업은 이제 곱셈에 실제로 참여하는 비영 값을 두 배로 제공합니다. 즉, 결국 영으로 판명되는 가중치를 더 이상 가져오지 않게 됩니다. 따라서 그림 9-5에서 보듯이, 영으로 알려진 값에 대해 행렬 곱셈 연산을 낭비하지 않게 됩니다.

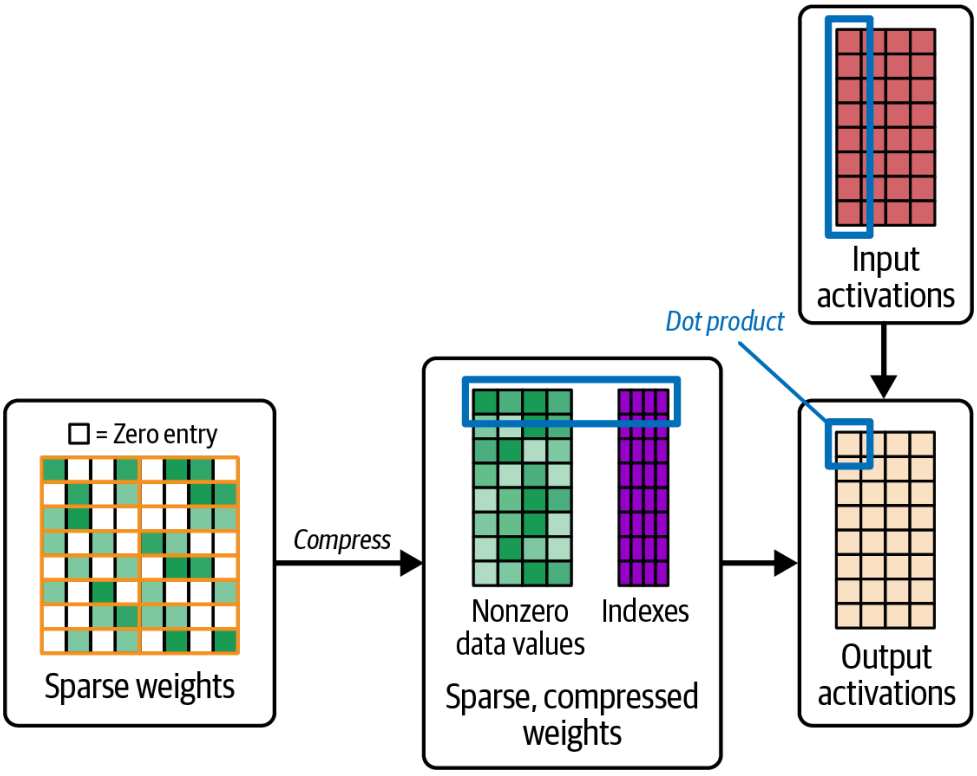

그림 9-5. 2 :4 구조적 스파시티

구조적 스파스성은 모델 훈련 후 적용됩니다. 모델은 추론을 위해 가지치기 및 최적화됩니다. 가지치기와 형식 변환은 cuSPARSELt 및 프레임워크 툴링과 같은 소프트웨어 스택에서 수행됩니다. Transformer Engine은 지원되는 스파스 실행을 가속화하지만 변환 과정에서 스파스성을 강제하지는 않습니다.

정리 및 형식 변환은 소프트웨어(일반적으로 cuSPARSELt 및 프레임워크 도구)를 통해 처리됩니다. PyTorch에서는 스파스 텐서 코어(Sparse Tensor Cores)에 스파스 GEMM을 배포하기 전에 훈련된 덴스 모듈을 2:4 스파스 형식으로 변환하려면 `to_sparse_semi_structured()`를 사용하십시오.

모델이 변환되면 스파스 텐서 코어에서 실행되는 최적화된 스파스 GEMM 커널을 표준 커널 대신 호출합니다. 스파스 텐서 코어는 많은 추론 워크로드에서 덴스 텐서 코어 대비 최대 2배의 속도 향상을 달성할 수 있습니다 . 특히 커널 런치 오버헤드를 상쇄하는 대량 배치 입력을 제출할 때 더욱 그렇습니다.

배칭(Batching)은 연산 집약도를 높이는 매우 일반적이고 실용적인 방법입니다. 메모리 I/O 등과 관련된 모든 작업을 수행하며 한 번에 하나의 항목을 처리하는 대신, 여러 항목을 한 번에 처리하여 메모리 액세스(예: 가중치 로딩 등)가 여러 계산에 걸쳐 분산되도록 합니다.

이를 통해 스파스 가속 행렬 곱셈은 인덱스 처리나 압축 표현으로 인한 오버헤드를 숨길 만큼 충분한 병렬 작업을 확보합니다. 소규모 배치에서는 이 오버헤드가 지배적이어서 관측되는 가속도 향상을 제한할 수 있습니다.

LLMs과 같은 트랜스포머 기반 모델에서 흔히 사용되는 대규모 행렬 곱셈 시 2:4 스파시티가 최대 이점을 제공합니다. 이는 하드웨어가 전용 스파스 텐서 코어를 완전히 활용할 수 있기 때문입니다. 이 스파스 텐서 코어는 하드웨어 내에서 직접

반폭 데이터를 처리합니다. 이는 0을 건너뛰고 동일한 사이클 예산 내에서 비영 요소들에 대해 두 배의 작업을 수행합니다.

Blackwell의 컴퓨팅 성능이 HBM 대역폭보다 빠르게 성장했기 때문에, 구조적 스파시티는 컴퓨팅 바운디드를 유지하는 훌륭한 방법입니다. NVLink-C2C를 통해 GPU가 CPU 메모리에서 매우 높은 처리량으로 데이터를 스트리밍할 수 있는 Grace Blackwell 시스템에서도, 로드된 각 타일당 바이트당 FLOPS를 극대화하는 것이 여전히 중요합니다.

예를 들어 가중치의 50%를 2:4 패턴으로 정돈하면 메모리 트래픽의 절반이 전혀 필요하지 않게 됩니다. 이는 즉시 전역 메모리 읽기를 줄이고 유효 산술 집약도를 거의 2배 가까이 높입니다.

NVIDIA GPU는 하드웨어적으로 이 2:4 구조적 스파스성을 구현하여 16개 요소 덩어리마다 8개 요소를 0으로 설정할 수 있습니다. 이는 스파스 행렬에 대해 텐서 코어 처리량을 두 배로 늘리는 데 사용되는 패턴입니다. 현재 시점에서 다른 임의의 스파스성 패턴은 하드웨어에서 이러한 특별한 가속을 얻지 못합니다.

스파스성으로 인한 속도 향상은 모델의 정확도가 유지된다는 전제 하에 이루어집니다. 실제로는 프루닝 후 미세 조정이나 신중한 보정이 중요합니다. 이를 통해 정확도 손실을 최소화할 수 있습니다.

스파스성을 적용하기 전에, 먼저 앞서 다룬 기본 최적화 기법들을 구현하는 것이 중요합니다: 모든 전역 로드를 통합하고, 타일링을 통해 데이터를 재사용하며, 점별 연산을 융합하여 추가적인 메모리 왕복을 제거하는 것입니다. 이러한 기본 사항들이 구현되고 검증된 후, 구조적 스파스성은 추론에 또 다른 속도 향상을 제공할 수 있습니다.

구조적 스파스성은 일반적으로 추론 작업 부하에 적용됩니다( ). 훈련 중에는 기울기가 2:4 스파스성의 이점을 누리지 못합니다. 또한 기울기 업데이트에서 스파스성을 유지하는 것은 복잡합니다. 따라서 모델을 사전 트리밍하고 보정해 둔 배포 시나리오에 사용하는 것이 권장됩니다. NVIDIA의 2:4 스파스 텐서 코어 기능은 주로 추론에 사용됩니다. 훈련 지원은 제한적이며 모델 및 프레임워크에 따라 다릅니다. 이를 사용하기 전에 소프트웨어 스택에서 지원 여부를 확인하십시오.

재계산 대 메모리 트레이드오프

또한 메모리 대역폭이 병목인 경우, 미리 계산된 벡터화 값(예: `x²`)을 저장하거나 로드하는 대신 필요 시 재계산하는 것을 고려하십시오. 예를 들어, 레지스터에서 `x*x` 를 반복적으로 계산하는 것이 글로벌 메모리에서 미리 계산된 `x²` 를 로드하는 것보다 종종 더 빠릅니다. 저렴한 표현식의 연속적인 재계산은 산술 집약도를 높일 수 있으며 메모리가 부족한 경우 유용한 기법입니다.

많은 LLM 추론 엔진은 메모리 절약을 위해 이 기법을 사용합니다. 대규모 활성화 텐서를 HBM에 저장했다가 나중에 다시 읽어오는 대신, 특정 레이어와 활성화를 실시간으로 재계산할 수 있습니다. 이는 모델 훈련 환경에서의 *활성화 체크포인트링과* 유사합니다.

재계산은 효과적인 FLOPS/바이트를 개선하고 대규모 모델을 더 적은 메모리에 수용할 수 있게 합니다. 또한 재계산은 더 큰 배치 크기를 위한 메모리를 확보하며, 약간의 추가 FLOPS를 희생하는 대신 메모리 트래픽을 크게 감소시킵니다.

PyTorch와 산술 집약도

PyTorch에서는 이러한 개념 다수가 자동으로 적용됩니다. 앞서 언급한 바와 같이 PyTorch 컴파일러(13장 및 14장에서 논의)는 요소별 연산 체인, 심지어 일부 축소 연산까지 자동으로 융합할 수 있습니다. 실행 그래프 수준 최적화를 활용하여 데이터를 GPU에 유지하고 최대한 재사용합니다.

내부적으로 cuDNN 및 cuBLAS와 같은 최적화된 라이브러리( )를 사용하기 때문에, PyTorch는 벡터 연산(`torch.matmul`) 시 자동으로 타일링(tiling)을 수행하고 공유 메모리를 활용합니다. 또한 PyTorch의 SPDA(`scaled_dot_product_attention`)는 텐서 형상(tensor shapes)과 연산 유형(`dtypes`)에 따라 FlashAttention, 메모리 효율적 백엔드 또는 cuDNN 백엔드로 분배될 수 있습니다. 백엔드 선택을 제어하려면 예를 들어 ` `torch.nn.attention.sdpa_kernel(SDPBackend.FLASH_ATTENTION)` `를 사용하세요. 성능 중심 개발자라면 이러한 최적화 기법과 적용 여부를 확인하는 방법을 숙지해야 합니다.

PyTorch가 대부분의 연산을 인식하고 컴파일할 수 있지만, 일부 비표준 연산이나 사용자 정의 CUDA 연산은 융합되지 않을 수 있다는 점을 유의해야 합니다. 이러한 경우 융합, 타일링 등과 같은 수동 최적화가 여전히 필요할 수 있습니다.

PyTorch 코드를 작성할 때는 개별 커널을 길게 연속 실행하기보다 여러 계산을 수행하는 융합 연산(fused operations)과 최적화된 라이브러리 호출을 선호하십시오. 실제로 이는 수많은 작은 요소별 커널을 Python에서 작성하는 대신, `torch.nn.functional` 활성화나 `torch.matmul` 같은 고수준 연산을 사용함을 의미합니다. 이러한 라이브러리는 해당 고수준 연산 유형에 대해 효율적인 커널을 호출합니다. 또한 컴파일러는 주변 연산과 이를 효율적으로 융합하는 방법을 알고 있습니다.

PyTorch의 중첩 텐서( ) 또는 불규칙 텐서(ragged tensors)는 패딩 없이 가변 길이 입력 배치를 표현할 수 있게 합니다. 각 중첩 텐서는 가변 길이 시퀀스를 단일하고 효율적인 기본 버퍼에 압축합니다. `NestedTensor` 는 일반 텐서 인터페이스를 노출하지만 불필요한 제로 패딩을 제거합니다. 따라서 가져오는 각 바이트가 계산에 유용하게 사용되므로 글로벌 메모리 로드가 더 효율적으로 됩니다.

중첩 텐서는 길이가 변하는 시퀀스를 가진 LLMs에 유용합니다. 중첩 텐서를 사용할 때, 어텐션이나 배치-행렬 곱셈과 같은 연산은 메모리에서 필수 데이터만 가져옵니다. 이는 커널을 루프라인 모델의 컴퓨팅 바운디드 영역에 가깝게 이동시켜 메모리 제한 스톨을 줄이는 데 도움이 됩니다. 그 결과, 특히 메모리 민감한 워크로드에서 더 높은 지속적 처리량을 얻을 수 있습니다.

실제 적용 시 중첩 텐서는 연산자 커버리지와 성능 특성에 대한 신중한 검증이 필요합니다. 지원 여부는 워크로드에 따라 다르며, 속도 향상은 형상(shape)에 따라 달라집니다. 대표적인 시퀀스 길이 분포와 어텐션 패턴을 통해 종단 간 이점을 검증할 수 있습니다. 메모리 트래픽과 커널 시간을 모두 자질(qualify)하십시오.

요약하면, PyTorch는 커널의 산술 집약도를 높이기 위한 다양한 메커니즘을 제공합니다. 이러한 옵션을 이해하고 워크로드에 가장 적합한 방법을 결정하는 것이 중요합니다. 현대 GPU에서 산술 집약도를 높이는 또 다른 효과적인 방법은 축소 정밀도 텐서 코어를 사용하는 것입니다. 다음으로 이러한 메커니즘을 살펴보겠습니다.

혼합 정밀도 및 텐서 코어 활용

현대 NVIDIA GPU는 텐서 코어에서 TF32, FP16, FP8, FP4, INT8과 같은 텐서 코어 전용( ) 정밀도 감소 연산을 구현합니다. Blackwell 아키텍처의 각 SM(Streaming Multiprocessor)은 텐서 코어 데이터 전용 256KB 온칩 TMEM을 보유합니다. 또한 글로벌 메모리와 공유 메모리 간 타일을 비동기적으로 복사하는 전용 TMA(Tensor Memory Access) 유닛을 갖추고 있습니다. 텐서 코어 명령어(예: `tcgen05.mma`)는 연산자와 누산기를 공유 메모리와 TMEM 간에 암시적으로 이동시킵니다. 이 설계는 텐서 코어에 높은 처리량으로 데이터를 공급하고 스톨을 최소화합니다. Blackwell의 TMEM 기반 누산기는 레지스터에 직접 누산하던 이전 GPU 세대에 비해 레지스터 압박을 줄이는 데 기여합니다.

이러한 기능을 올바르게 활용하면, 한때 메모리 병목 현상이 심하고 텐서 연산이 많은 커널을 완전히 연산 바운디드한 커널로 전환할 수 있습니다. 이는 산술 집약도(바이트당 FLOPS)를 2배, 4배, 심지어 8배까지 높여줍니다. Nsight Compute의 루프라인 차트와 스톨 통계를 모니터링하여 그 효과를 확인할 수 있습니다.

Nsight Compute의 *Speed of Light* 분석 은 "메모리 스로틀" 및 캐시 미스 같은 메모리 제약 스톨 원인을 보여줍니다. 정밀도가 낮은 형식의 텐서 코어를 사용할 경우 이러한 원인이 크게 감소합니다. 또한 Nsight Compute는 연산 강도가 증가하여 커널이 컴퓨팅 한계에 근접했는지 교차 확인하기 위한 루프라인 차트를 통합합니다.

FP32에서 TF32, FP16, FP8 또는 FP4와 같은 저정밀도 텐서 코어 커널로 전환할 때, Nsight Compute의 워프 스톨 메트릭은 일반적으로 메모리 관련 스톨 감소와 함께 종속성 또는 파이프라인 스톨의 상대적 증가를 보여줍니다. 이는 산술 집약

도가 증가하고 메모리 제약에서 컴퓨팅 제약 실행으로 전환되었음을 나타냅니다.

TMEM 및 TMA를 통한 텐서 코어 공급

고처리량 텐서 계산의 핵심은 SM당 256KB SRAM 버퍼인 TMEM입니다. 그러나 프로그래머는 일반적으로 TMEM을 명시적으로 할당하거나 관리하지 않습니다. 텐서 코어 연산을 사용할 때 TMEM은 하드웨어나 라이브러리에 의해 처리됩니다. TMEM은 그림 9-6에 표시되어 있습니다.

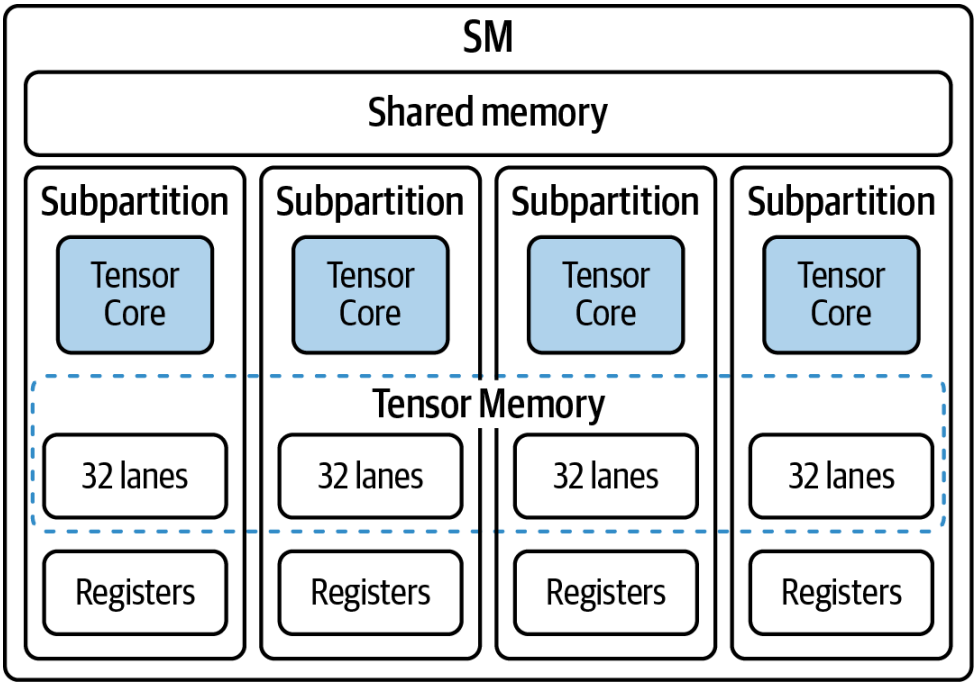

그림 9-6. TMEM은 부분 결과(레지스터 대신)를 누적하여 텐서 코어를 지원합니다

내부적으로 Blackwell은 TMEM을 피연산자 및 누산기 저장소로 사용하는 연산 명령어(`tcgen05.mma`)를 활용합니다. CUTLASS 및 라이브러리 커널은 커널 구성과 병렬 스레드 실행(PTX) 어셈블리를 통해 필요한 할당 및 사용을 관리합니다. 따라서 Transformer Engine은 부분 결과를 저장하기 위해 TMEM을 사용합니다. 이는 MMA의 레지스터 의존성을 줄여줍니다.

CUTLASS와 같은 고수준 API는 이러한 복잡성을 자동으로 처리합니다. CUTLASS는 텐서 코어 행렬 연산 및 메모리 로드/스토어 인터페이스를 구현하는 `tcgen05.*` PTX 명령어를 사용하므로 가능한 경우 CUTLASS 및 기타 고수준 라이브러리를 사용하십시오. CUDA MMA 내장 함수나 CUTLASS GEMM을 사용하여 텐서 코어 MMA 연산을 실행할 때마다, 구현체는 공유 메모리와 TMEM을 통한 연산자 이동을 관리합니다. 의 TMA는 글로벌 메모리와 공유 메모리 간에 타일을 스트리밍하며, 텐서 코어 명령어는 공유 메모리와 TMEM 간에 연산자를 암시적으로 이동시킵니다.

Nsight Compute에는 저정밀도 텐서 코어 경로를 채택한 후 커널이 메모리 바운디드에서 연산 바운디드로 전환되었는지 확인하기 위한 내장형 루프라인 및 속도 분석 기능이 포함되어 있습니다.

텐서 코어 명령어는 MMA 파이프라인의 일환으로 공유 메모리와 TMEM 간에 연산 대상 데이터를 이동합니다. 이는 사용자의 명시적 코드 없이 백그라운드에서 수행됩니다. 이를 통해 데이터는 텐서 코어가 필요로 하는 위치에 배치됩니다. 글로벌 메모리에서 공유 메모리로 이 데이터 전송을 수행하려면, CUDA 파이프라인 API(`<cuda/pipeline>`)의 파이프라인과 함께 TMA(`cuda::memcpy_async`)를 사용하십시오. 코드에서 `cuda::memcpy_async`와 CUDA 파이프라인 API를 사용한 간단한 2단계 파이프라인 구현은 다음과 같습니다:

```
# two_stage_pipeline.cu

#include <cuda/pipeline>
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

extern "C" __global__
void stage_ab_tiles(const float* __restrict__ globalA,
                    const float* __restrict__ globalB,
                    float* __restrict__ outC,
                    int tile_elems,
                    int num_tiles) {
     // Alignment / size guards for vectorized copies (runtime parameter)
    assert((tile_elems % (32 * 4)) == 0 &&
           "tile_elems must be multiple of 128 for float4 vectorization");
    // If you cannot guarantee 16B alignment or sizes, handle
    // the tail/ragged edges with a fallback 4B loop.
  extern __shared__ float smem[];
  auto block = cg::this_thread_block();

  // Shared buffers for double buffering of A and B
  float* A0 = smem + 0 * tile_elems;
  float* A1 = smem + 1 * tile_elems;
  float* B0 = smem + 2 * tile_elems;
  float* B1 = smem + 3 * tile_elems;

  constexpr auto scope = cuda::thread_scope_block;
  constexpr int stages = 2;
  __shared__ cuda::pipeline_shared_state<scope, stages> pstate;
  auto pipe = cuda::make_pipeline(block, &pstate);

  // Prime the pipeline with tile 0
  pipe.producer_acquire();
  cuda::memcpy_async(block, A0, globalA + 0 * tile_elems,
                     cuda::aligned_size_t<32>{tile_elems * sizeof(float)},
  cuda::memcpy_async(block, B0, globalB + 0 * tile_elems,
                     cuda::aligned_size_t<32>{tile_elems * sizeof(float)},
  pipe.producer_commit();

  for (int t = 1; t < num_tiles; ++t) {
    // Stage the next A and B tiles
    pipe.producer_acquire();
    cuda::memcpy_async(block, (t & 1) ? A1 : A0,
```

```
                        globalA + t * tile_elems,
                        cuda::aligned_size_t<32>{tile_elems*sizeof(float)},
      cuda::memcpy_async(block, (t & 1) ? B1 : B0,
                        globalB + t * tile_elems,
                        cuda::aligned_size_t<32>{tile_elems*sizeof(float)},
      pipe.producer_commit();

      // Consume the previously staged tiles
      pipe.consumer_wait();
      float* prevA = (t & 1) ? A0 : A1;
      float* prevB = (t & 1) ? B0 : B1;
      // Perform compute using prevA and prevB
      pipe.consumer_release();
    }

    // Consume the final staged tiles
    pipe.consumer_wait();
    int last = (num_tiles - 1) & 1;
    float* lastA = last ? A1 : A0;
    float* lastB = last ? B1 : B0;
    // Perform compute using lastA and lastB
    pipe.consumer_release();
  }
```

이 커널을 실행할 때 동적 공유 메모리 크기를 `4 x tile_elems x sizeof(float)` 로 설정하면 공유 메모리에 `A0`, `A1`, `B0`, `B1` 를 할당합니다. 이 이중 버퍼링 패턴은 한 타일이 공유 메모리에 상주하는 즉시 텐서 코어가 이를 처리할 수 있도록 보장합니다. 동시에 `cuda::memcpy_async` 는 다음 타일을 공유 메모리로 병렬로 가져옵니다. TMEM이 텐서 코어 명령어를 위한 온칩 데이터 버퍼를 제공하고 공유 메모리가 스테이징 공간을 제공하므로, FP16, FP8 또는 FP4 타일을 온칩에서 완전히 스테이징하고 재사용할 수 있습니다. 결과적으로 파이프라인이 최적화되고 타일 및 복사 크기가 적절하게 설정되면 스톨이 줄어듭니다. `cuda::memcpy_async` HBM에서 공유 메모리로의 전송을 중첩하고 커널을 계속 작동시킬 수 있습니다. 이는 메모리 지연 시간을 연산 뒤에 숨기는 데 도움이 됩니다.

TF32 및 자동 혼합 정밀도(PyTorch)

텐서 코어는 원래 FP16을 위해 설계되었지만, FP32와 FP16 사이의 위치에 있는 TF32도 지원합니다. TF32는 FP32와 같은 8비트 지수부와 FP16과 같은 10비트 지수부를 사용합니다. TF32는 FP32의 지수 범위 범위를 유지하면서 CUDA 코어에서 실행되는 FP32보다 훨씬 높은 처리량으로 텐서 코어에서 실행됩니다. PyTorch에서 TF32를 활성화하는 방법은 PyTorch 코드에 다음을 설정하는 것만큼 간단합니다:

```
import torch
torch.set_float32_matmul_precision('high') # {'highest'|'high'|'medium'}
```

이 플래그를 설정하면 `torch.matmul` 및 `torch.nn.Linear` 과 같은 고수준 연산이 표준 CUDA 코어에서 FP32로 실행되는 대신 자동으로 TF32 텐서 코어 커널로 실행됩니다.

TF32 외에도 PyTorch의 자동 혼합 정밀도(AMP)는 각 연산에 최적의 정밀도(FP16 또는 BF16)를 선택하고 안정성을 위해 결과를 FP32로 누적할 수 있습니다. BF16은 FP16의 오버플로 문제를 방지하는 데 도움이 됩니다. 기본적으로 CUDA `autocast` 는 float-16을 사용합니다. 이를 지원하는 GPU에서 BF16을 사용하려면 `dtype=torch.bfloat16` 를 전달하기만 하면 됩니다. 예를 들어, 다음과 같이 컨텍스트 매니저로 모델 코드를 감쌀 수 있습니다:

```
with torch.amp.autocast("cuda", dtype=torch.bfloat16):
    output = model(input)
```

내부적으로 TorchInductor(13장 및 14장에서 다루어짐)는 다음과 같은 사항을 보장하기 위해 이러한 정밀도 변환을 자동으로 융합합니다: 대규모 GEMM 연산은 FP32 또는 TF32로 텐서 코어에서 실행되고, 수치적 안정성을 위해 누적은 FP32로 유지되며, 레이어 정규화 및 소프트맥스와 같은 작은 "민감한" 커널은 FP32로 실행되고, `GradScaler` 는 FP32로 훈련할 때 언더플로우를 방지합니다. BF16은 FP32 지수 범위를 갖습니다. 따라서 BF16으로 훈련할 때는 일반적으로 부동소수점 오버플로우 방지(`GradScaler`)가 필요하지 않습니다.

`dtype` PyTorch에서는 이러한 혼합 정밀도 결정이 컴파일러에 통합되어 수동 개입 없이도 최적의 혼합 정밀도 선택(예: 연산에는 FP16/FP8, 누적에는 FP32)을 얻을 수 있습니다. 이는 그림 9-7에서 혼합 정밀도 행렬 곱셈-누적(MMA)으로 보여집니다.

이 자동 혼합 정밀도 파이프라인은 최소한의 코드 변경으로 산술 집약도를 극대화합니다. 융합된 텐서 코어 커널은 공유 메모리(예: 피연산자)와 TMEM(예: 누산기)에서 데이터를 스테이징하고 재사용함으로써 HBM으로의 왕복 횟수를 최소화합니다.

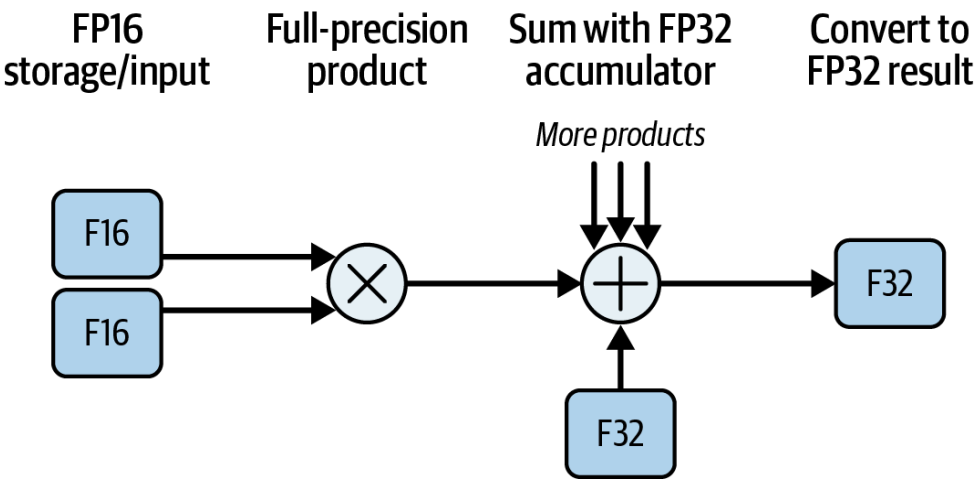

그림 9-7. 혼합 정밀도 및 행렬 곱셈-누적(MMA)

앞서 설명한 구조적 스파스성 또는 극저정밀도(FP8/FP4)를 사용할 때는 TMEM과 텐서 코어가 완전히 활용되도록 충분히 큰 배치 크기 또는 타일 세분성을 유지

해야 합니다. 작은 배치에서는 형식 변환, 스파스 인덱스 처리, 불규칙한 메모리 패턴 등 오버헤드가 발생합니다. 이는 달성된 속도 향상을 저하시킬 수 있습니다.

예를 들어, FP8 또는 2:4 스파시티를 사용할 때 배치 크기 1은 고정 오버헤드가 분산되지 않아 거의 이점을 보지 못할 수 있습니다. 반면 배치 크기 128 또는 256은 TMEM 파이프라인을 완전히 활용하여 거의 최대 처리량을 생성합니다.

BF16/FP16, FP8 및 FP4 정밀도 감소

BF16/FP16(반정밀도)은 여러 GPU 세대에 걸쳐 지원되어 왔습니다. 그러나 최신 GPU의 텐서 코어는 종종 BF16/FP16 피크 처리량의 90% 이상을 유지할 수 있으며, 이는 FP32 피크 처리량의 약 4배에 해당합니다. 이는 하드웨어가 각 사이클마다 다수의 BF16/FP16 FMA 연산을 병렬로 처리하기 때문입니다.

FP16 훈련은 FP32보다 좁은 5비트 지수를 사용하므로, 손실 스케일링을 적용하지 않으면 매우 작은 기울기 값이 언더플로우되어 0이 될 수 있습니다. 손실 스케일링은 역전파 과정에서 수치적 안정성을 유지합니다. 이 스케일링은 정적 또는 동적으로 수행될 수 있습니다.

반면 BF16은 FP32의 8비트 지수 범위를 그대로 사용하므로 언더플로우를 본질적으로 방지합니다. 따라서 손실 스케일링이 거의(혹은 전혀) 필요하지 않습니다. 이는 혼합 정밀도 워크플로를 단순화하고 현대 GPU에서 훈련 정확도를 향상시키는 경우가 많습니다.

BF16은 일반적으로 최신 GPU에서 훈련할 때 선호됩니다. FP16이 요구하는 복잡한 손실 스케일링 없이도 FP32에 필적하는 정확도를 유지할 수 있기 때문입니다.

처리량을 더욱 높이기 위해 FP8을 사용할 수 있습니다. 16비트 가중치를 50% 줄여 8비트로 전환하면 메모리 트래픽을 절반으로 줄일 수 있으며, HBM 트랜잭션당 로드되는 가중치 수를 두 배로 늘릴 수 있습니다. 실제 적용 시 FP8 `matmul`과 FP32 또는 TF32 누적 연산을 결합하면 BF16/FP16 대비 2~3배의 TFLOPS를 달성할 수 있습니다. 단, 양자화 오류로 인한 모델의 미미한 정확도 저하가 허용 가능한 수준이어야 합니다.

매우 낮은 정밀도에서 정확도 문제를 해결하기 위해 Transformer Engine은 FP8과 마이크로 스케일링이 적용된 NVIDIA의 4비트 NVFP4 형식을 지원합니다. NVFP4는 마이크로블록별 스케일링과 상위 레벨 스케일링을 결합한 2단계 스케일링을 적용하여, 가중치에 4비트 저장 공간을 사용하면서도 모델 정확도를 유지할 수 있게 합니다. 또한 Blackwell B200의 NVFP4는 공격적인 마이크로 스케일링 양자화를 통해 10 페타플롭스(dense)를 제공하며, FP32 피크 성능은 약 80 테라플롭스(dense)입니다. 이는 가중치당 이론적 처리량이 약 20배 증가한 속도 향상입니다. 또한 Blackwell의 B300(Ultra)은 B200 대비 50% 향상된 NVFP4 연산 성능(15 페타플롭스, dense)을 자랑합니다.

모델이 보정 후 정밀도 저하를 허용한다면, NVFP4 커널은 지원되는 하드웨어에서 FP32보다 훨씬 높은 처리량을 제공할 수 있지만, 정확도는 모델별로 검증해야 합니다.

정밀도가 매우 낮기 때문에 SM당 256KB TMEM은 큰 FP4 타일(예: 256×256)을 저장할 수 있어 온칩 재사용을 더욱 증가시키고 성능을 향상시킵니다. 모든 저정밀도 → 누적 변환은 자동으로 수행됩니다. 커널은 HBM에서 FP4 입력을 읽고, 텐서 코어는 FP4 × FP4 곱셈을 수행하며, MMA API는 결과를 BF16/FP16 또는 FP32 누적기에 누적합니다.

정밀도가 한 단계 낮아질 때마다 바이트당 연산 횟수가 2배 또는 4배 증가하여 산술 집약도가 높아집니다. TMEM/TMA가 메모리와 연산을 중첩할 때, 이러한 저정밀도 형식은 기존 메모리 바운드 커널을 완전히 연산 바운드 커널로 전환합니다. 이는 최신 GPU의 GPU당 멀티 플로프스급 텐서 코어 엔진을 완전히 활용합니다.

추론을 위한 INT8 정밀도 감소 및 DP4A 명령어

LLM 추론 사용 사례 는 일반적으로 현대 GPU가 지원하는 저정밀도 INT8 양자화를 허용합니다. 이는 일반 CUDA 코어에서 DP4A(SIMD 내적) 명령어와 텐서 코어에서 정수 행렬 곱셈/누적(MMA) 명령어를 사용합니다. 명령어 수준에서 DP4A는 명령어당 4개의 INT8 곱셈-누적(MAC) 연산을 수행하는 반면, FP32는 명령어당 1개의 융합 곱셈-덧셈(FMA) 연산을 수행합니다.

INT8의 가중치 트래픽은 FP32의 4바이트 대신 요소당 1바이트에 불과하므로 가중치 메모리 트래픽이 75% 감소합니다. INT8 추론 워크로드는 더 높은 INT8 텐서 코어 피크 처리량과 감소된 메모리 트래픽 덕분에 FP32보다 훨씬 우수한 성능을 발휘할 수 있습니다. 이는 INT8 가중치를 사용할 때 각 GPU가 메모리에서 초당 약 4배 더 많은 데이터를 처리할 수 있기 때문입니다. 이는 TMEM과 TMA가 데이터와 연산을 완벽하게 중첩시키고 텐서 코어에 최대한 효율적으로 공급함으로써 가능해집니다.

트랜스포머 엔진과 TMEM 심층 분석

최신 NVIDIA GPU에는 저정밀 형식에 대한 텐서 코어 하드웨어 지원과 확장 및 캐스팅을 위한 소프트웨어 런타임을 결합한 트랜스포머 엔진( )이 포함됩니다. cuBLASLt, cuDNN, CUTLASS 또는 OpenAI의 Triton 커널은 공유 메모리(`cp.async`)로 전송하거나 TMA 전송을 수행합니다. 이후 텐서 코어 명령어는 공유 메모리와 TMEM 간에 연산자를 암시적으로 이동시킵니다.

TMEM은 Transformer 엔진과 텐서 코어가 결과 저장(레지스터 대신)에 사용하는 SM당 256KB SRAM 버퍼임을 기억하십시오. 실제로 TMEM을 명시적으로 할당할 필요는 없습니다. 모든 작업은 하드웨어에서 처리됩니다. 예를 들어 텐서 코어의 MMA 연산을 호출할 때 하드웨어가 모든 메모리 할당과 데이터 전송을 처리합니다.

MMA 명령어( )를 사용하면 각 워프가 텐서 코어를 직접 제어하여 고처리량 혼합 정밀도 MMA 연산을 수행합니다. 이 연산들은 프래그먼트 로드, 레지스터 매핑, 혼합 정밀도 MMA 연산을 관리합니다.

현재 시점에서 PyTorch의 INT8 양자화 연산자 지원은 TorchAO 및 벤더 백엔드를 통해 제공됩니다. 양자화 모듈은 전용 INT8 커널로 실행됩니다. 저수준 INT8 GEMM에 cuBLASLt 또는 CUTLASS를 사용하면 텐서 코어 활용도를 보장할 수 있습니다.

텐서 코어 기반 커널이나 GEMM 라이브러리 함수(예: CUTLASS)를 실행할 때마다 구현체는 공유 메모리와 TMEM을 통해 연산 대상 이동을 자동으로 관리합니다. 이를 통해 텐서 코어는 처리 준비가 된 타일로 가득 차게 유지됩니다. (응용 프로그램 코드가 TMEM을 직접 할당하지 않는다는 점에 유의하십시오.)

Transformer Engine 워크플로는 간단합니다. 먼저 커널이 MMA 호출을 발행하거나 CUTLASS GEMM을 실행합니다. 다음으로 Transformer Engine 펌웨어가 TMA(또는 `cuda::memcpy_async`)를 통해 가중치와 활성화 값을 HBM에서 공유 메모리(SMEM)로 복사하도록 처리합니다. 텐서 코어 명령어(예: `tcgen05.mma`)는 MMA 파이프라인 실행 중 암시적으로 SMEM과 TMEM 간에 연산자를 이동합니다. 이상적으로는 가중치가 FP8 또는 FP4 형식이어야 하며, 활성화 값은 가능한 경우 FP8/FP4로 캐스팅됩니다. 그렇지 않은 경우 활성화 값은 FP16/FP32 형식으로 유지될 수 있습니다.

텐서 코어 MMA 연산은 낮은 정밀도(예: FP8 × FP8에 고정밀도 누적, FP16 × FP16에 FP32 누적)로 실행됩니다. 부분 합계는 TMEM에 고정밀도(예: BF16, FP16, FP32)로 누적되며 커널에 따라 다릅니다. 누적기 상태는 TMEM에 저장됩니다. 이 상태는 `tcgen05` 로드 및 저장 인터페이스를 통해 액세스됩니다. 하드웨어가 이러한 이동을 투명하게 관리합니다.

사용자 정의 타일 루프를 구축하면 텐서 코어 연산과 데이터 이동을 중첩할 수 있습니다. 다음 코드에서 보여주는 것처럼 `cuda::memcpy_async` 및 CUDA 파이프라인 API를 사용하여 이를 수행할 수 있습니다:

```
#include <cuda/pipeline>
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

extern "C" __global__
void double_buffer_a(const float* __restrict__ globalA,
                     int tile_elems,
                     int numTiles) {
  __shared__ float tileA0[TILE][TILE];
  __shared__ float tileA1[TILE][TILE];

  auto block = cg::this_thread_block();

  constexpr auto scope = cuda::thread_scope_block;
  constexpr int stages = 2;
```

```
    __shared__ cuda::pipeline_shared_state<scope, stages> pstate;
    auto pipe = cuda::make_pipeline(block, &pstate);

    // Prime pipeline with tile 0
    pipe.producer_acquire();
    cuda::memcpy_async(block,
                       &tileA0[0][0],
                       globalA + 0 * tile_elems,
                       cuda::aligned_size_t<32>{tile_elems * sizeof(float)}
                       pipe);
    pipe.producer_commit();

    for (int t = 1; t < numTiles; ++t) {
      // Stage next tile into the alternate buffer
      pipe.producer_acquire();
      float* nxtA = (t & 1) ? &tileA1[0][0] : &tileA0[0][0];
      cuda::memcpy_async(block,
                         nxtA,
                         globalA + t * tile_elems,
                         cuda::aligned_size_t<32>{tile_elems * sizeof(float)}
                         pipe);
      pipe.producer_commit();

      // Consume the previously staged tile
      pipe.consumer_wait();
      float* curA = ((t - 1) & 1) ? &tileA1[0][0] : &tileA0[0][0];
      pipe.consumer_release();
    }

    // Consume the final staged tile
    pipe.consumer_wait();
    float* lastA = ((numTiles - 1) & 1) ? &tileA1[0][0] : &tileA0[0][0];
    // Use lastA with your compute
    pipe.consumer_release();
  }
```

TMEM은 텐서 코어 명령어가 사용하는 전용 온칩 버퍼이므로 데이터는 컴퓨트 유닛 근처에 유지됩니다. 텐서 코어가 현재 타일을 처리하는 동안 `cuda::memcpy_async` 는 다음 타일을 HBM에서 공유 메모리로 스트리밍합니다.

이러한 작업 중첩은 메모리 지연 시간을 숨기는 데 도움이 되며, 파이프라인이 최적화되었을 때 텐서 코어를 지속적으로 활용할 수 있게 합니다. 트랜스포머 엔진, TMEM, TMA 간의 이러한 협업은 산술 집약도를 크게 높이고 최적화된 경우 빛의 속도에 가까운 효율성에 근접할 수 있습니다.

로드 및 스토어 작업은 호출 워프에 대해 동기화되지만, 연산과 데이터 이동의 중첩은 CUDA 파이프라인 API를 통해 이루어져야 합니다. wait/release와 같은 파이프라인 프리미티브와 함께 사용되는 `cuda::memcpy_async` 는 Tensor Memory Accelerator(TMA)에 매핑되며, 대량 텐서 전송에는 항상 우선적으로 사용해야 합니다. TMA로 표현할 수 없는 특수한 경우에만 `cp.async` 를 사용하십시오. 그러나 이러한 경우는 드뭅니다. 또한 데이터를 사용하기 전에 복사가 완료되었는지 확인해야 합니다.

최적의 산술 집약도와 텐서 코어 성능을 위한 CUTLASS 활용

의 이러한 최적화를 직접 활용하는 가장 쉬운 방법 중 하나는 NVIDIA의 CUTLASS 라이브러리를 사용하는 것입니다. CUTLASS를 사용하면 단일 템플릿 호출을 작성하기만 하면 다양한 고급 최적화가 자동으로 적용됩니다.

CUTLASS가 적용하는 최적화에는 공유 메모리 타일링, 비동기 메모리 전송, TMEM의 SM당 256KB 버퍼를 활용한 더블 버퍼링 등이 포함됩니다. 이를 통해 텐서 코어는 수동 커널 튜닝 없이도 거의 최대 처리량에 근접한 성능을 발휘합니다.

CUTLASS는 또한 워프 특화( )를 구현합니다. 이는 고성능 GPU 최적화 기법으로, 다음 장에서 논의할 예정입니다.

`C = A * B` 예를 들어, 반정밀도 입력값과 반정밀도 출력값을 사용하여 GEMM 연산을 수행하고, 결과는 상황에 맞게 FP16 또는 FP32로 누적해야 한다고 가정해 보겠습니다. 수동으로 튜닝된 MMA 루프를 작성하는 대신, 다음 코드에서 보여주는 것처럼 CUTLASS를 포함하고 템플릿을 인스턴스화하기만 하면 됩니다:

```
#include <cutlass/numeric_types.h>
#include <cutlass/gemm/device/gemm.h>

using Gemm = cutlass::gemm::device::Gemm<
  cutlass::half_t,  // A (FP16)
  cutlass::layout::RowMajor,
  cutlass::half_t,  // B (FP16)
  cutlass::layout::ColumnMajor,
  cutlass::half_t,  // C / output (FP16)
  cutlass::layout::RowMajor,
  float, // accumulator (FP32 accumulate)
  cutlass::arch::OpClassTensorOp,
  cutlass::arch::Sm100 // e.g., Blackwell B200
>;

// ... (allocate device pointers A_d, B_d, C_d,
// set up dimensions M,N,K, and strides lda, ldb, ldc) ...
```

```
Gemm gemm_op;
cutlass::Status status = gemm_op(
    { M, N, K },              // GEMM shape
    float(1.0f)               // alpha
    A_d, lda,                 // A pointer + leading dimension
    B_d, ldb,                 // B pointer + leading dimension
    float(0.0f),              // beta
    C_d, ldc                  // C pointer + leading dimension
);
```

이 코드를 컴파일하고 실행하면 CUTLASS가 자동으로 몇 가지 핵심 작업을 수행합니다. 먼저 CUTLASS는 레지스터 압력, 공유 메모리 용량, 텐서 코어 활용도를 균형 있게 조정하기 위해 타일을 선택합니다. 최신 GPU에서는 TMEM이 공유 메모리 및 L1과 함께 존재합니다. CUTLASS는 타일을 공유 메모리에 스테이징하고, 누산기 데이터를 저장하기 위해 TMEM과 상호작용하는 텐서 코어 명령어를 사용합니다. 타일 형상은 경험적으로 커널별로 선택됩니다. 예를 들어 128 × 128 또는 256 × 128과 같은 타일 크기를 선택할 수 있습니다. 이러한 크기는 TMEM의 SM당 256KB 버퍼에 맞으며 텐서 코어 계산 전반에 걸쳐 온칩 상태를 유지합니다.

정밀도에 따라 256 × 512 타일은 SM당 256KB TMEM 예산을 최대 활용합니다(256 × 512 요소 × 요소당 2바이트 = 256KiB). 반면 256 × 256 요소 × 요소당 4바이트 = 256KB입니다. 더 큰 타일은 타일당 처리량을 향상시키지만 SM당 동시 처리 타일 수는 감소합니다. 이는 소규모 GEMM에서 활용도 저하로 이어질 수 있습니다. 반면 매우 작은 타일은 병렬성을 위해 연산 집약도를 희생합니다.

CUTLASS는 이후 비동기 메모리 복사(`cp.async`, TMA)를 생성하여 각 타일을 DRAM에서 공유 메모리로 스트리밍합니다. 그림 9-8과 같이, `cp.async` 명령어는 스레드별 레지스터(또는 선택적으로 L1 캐시)를 사용하지 않고 전역 메모리의 데이터를 공유 메모리로 스테이징합니다. 캐싱 동작은 `cp.async` 수정자를 사용하거나 대량 텐서 전송을 위해 TMA를 사용하여 제어됩니다.

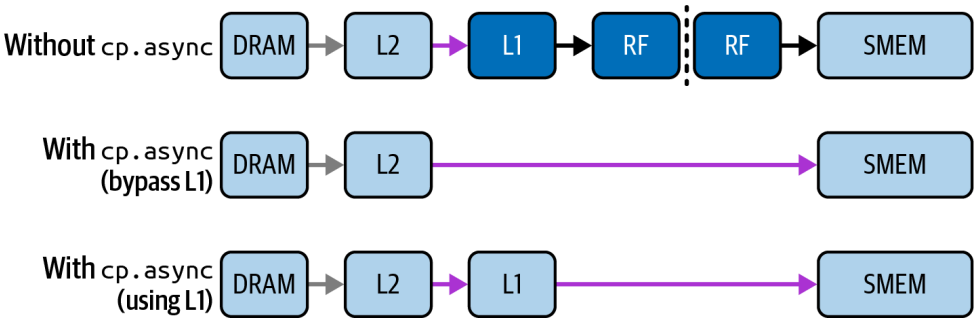

그림 9-8. 레지스터 파일 및 선택적으로 L1 캐시를 거치지 않고 글로벌 메모리에서 공유 메모리로 데이터를 로드하기 위한 비동기 메모리 복사 명령어(`cp.async`)사용

CUTLASS는 글로벌 DRAM의 타일을 TMA(`cp.async`) 또는 TMA(`cp.async.bulk.tensor`)를 사용하여 SMEM으로 스테이징합니다. 이후 텐서 코어 `tcgen05.mma` 명령어는 SMEM에서 연산자를 읽고 결과를 암시적으로 TMEM에 누적합니다. 이는 공유 메모리 내에 소프트웨어로 관리되는 스테이징 영역을 생성하며, 이 영역은 더블 버퍼링에 사용됩니다. 이렇게 하면 텐서

코어가 현재 타일을 처리하는 동안 TMA는 이미 다음 타일을 공유 메모리로 가져옵니다.

CUDA 파이프라인 API와 워프 특화 컴퓨트 스테이지(다음 장에서 설명)를 활용하여 CUTLASS는 모든 텐서 코어 파이프라인을 지속적으로 가동합니다. 수치적 정확성을 보장하기 위해 지정된 정밀도(예: 입력값이 FP16 또는 FP8일 때 FP32)로 부분 합계를 누적한 후, TMEM의 결과를 공유 메모리 또는 전역 메모리에 병합된 형태로 출력합니다.

CUTLASS는 또한 이점이 있을 경우 스레드 블록 클러스터를 활용하여 여러 SM에 걸쳐 타일링함으로써 더 큰 유효 타일을 생성합니다. 스레드 블록 클러스터는 다음 장에서 다룰 예정입니다.

이러한 복잡성이 모두 숨겨져 있기 때문에, CUTLASS는 수동으로 튜닝된 MMA 커널과 동등한 성능을 제공하는 드롭인 방식의 고성능 GEMM 커널을 제공합니다. 표 9-1에서 볼 수 있듯이, 수동 작성 버전 대비 전체 텐서 코어 활용률과 성능이 종종 몇 퍼센트 이내의 차이를 보입니다.

표 9-1. 수동 튜닝된 MMA와 CUTLASS 커널의 성능 및 리소스 사용량 비교

메트릭	수동 튜닝된 MMA 커널	CUTLASS GEMM
텐서 코어 활용도	98%	98%
스레드당 레지스터	~52	~60 (약간 높음)
스레드 블록당 공유 메모리 (CTA)	~2 KB	~4 KB
개발 노력	높음	낮음 (간단한 템플릿 구성)

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

여기서 양쪽 모두 FP16 입력값에 FP32 누적을 사용합니다. 또한 양쪽 모두 텐서 코어 활용도를 극대화하는 것을 목표로 합니다. 표에서 볼 수 있듯이, CUTLASS는 수동으로 튜닝된 MMA 성능을 약 2% 이내로 따라잡거나 초과합니다. 이 경우 CUTLASS가 레지스터를 약간 더 사용하고 공유 메모리를 두 배로 사용했음에도 불구하고, 하드웨어 한계 내에서 충분히 유지되었습니다. 이 미미한 증가는 점유율에 영향을 미치지 않습니다.

레지스터 및 공유 메모리 사용량의 미미한 차이는 CUTLASS가 유연성을 위해 커널을 일반화했기 때문입니다. 수동 튜닝으로 최적화할 수 있지만, 대부분의 경우 추가적인 복잡성을 감수할 만한 가치가 없으며 CUTLASS의 성능은 수동 튜닝 옵션과 거의 동일하게 유지됩니다.

몇 주에 걸친 저수준 튜닝 대신 몇 줄의 템플릿 코드만 필요합니다. 또한 CUTLASS 템플릿은 이미 FP4, FP8, FP16, TF32 연산자 유형을 지원합니다. 그리고 바이어스 추가 및 활성화와 같은 일반적인 후처리 작업을 동일한 커널에 융합할 수 있습니다.

또한 앞서 언급했듯이(다음 장에서 더 자세히 다룸), CUTLASS 템플릿은 데이터 재사용을 극대화하기 위해 스레드 블록 쌍, 다중 SM 타일링, 분산 공유 메모리(DSMEM)를 활용한 TMA 멀티캐스트를 투명하게 사용한다는 점을 기억하십시오.

이는 타일 크기 수동 선택, 비동기 복사 루프 작성, 더블 버퍼 관리, 워프 전문화 파이프라인 구현, 스레드 블록 클러스터 타일 처리 등 맞춤형 MMA 커널 작성과 대조됩니다. CUTLASS는 이 모든 작업을 자동으로 처리합니다.

cuBLAS와 같은 최적화 라이브러리는 CUTLASS를 기반으로 구축됩니다. PyTorch와 같은 고수준 라이브러리는 많은 커널에 대해 이러한 최적화 라이브러리를 호출합니다. 앞서 소개한 융합 어텐션 예시에서 TorchInductor가 정확히 동일한 더블 버퍼링 TMEM 파이프라인을 사용하는 CUTLASS 융합 어텐션 커널을 디스패치하는 것을 보여주었습니다. 이로 인해 텐서 코어 활용도가 98%에 달하고 메모리 스톨이 거의 발생하지 않습니다.

PyTorch 및 기타 상위 레벨 라이브러리의 더 많은 연산자가 내부적으로 CUTLASS를 채택함에 따라, 직접 CUDA C++ 코드를 작성하지 않고도 동일한 최적화를 활용할 수 있습니다.

CUTLASS가 아직 지원하지 않는 고도로 특화된 데이터 레이아웃이나 독특한 융합 패턴이 필요한 경우처럼 수동으로 MMA 커널을 작성해야 하는 시나리오가 여전히 존재할 수 있습니다.

이러한 경우 복잡성을 직접 구현해야 합니다. 먼저 TMEM에 맞는 타일 크기(예: 128×128 FP16)를 선택한 후, 각 타일에 대해 비동기 메모리 복사 명령어(`<cuda/pipeline>`, `cp.async` 참조)를 수행해야 합니다.

그런 다음 워프 전용 MMA 루프를 구현하고 TMEM을 이중 버퍼링하여 DRAM 지연 시간을 숨겨야 합니다. 마지막으로 소프트맥스나 요소별 비선형 연산 같은 사용자 정의 후처리 단계를 가능한 한 동일한 루프 내에서 인터리빙해야 합니다 .

그러나 거의 모든 표준 GEMM 또는 융합 어텐션 사용 사례에서는 CUTLASS 및 이를 기반으로 구축된 라이브러리가 권장되는 접근 방식입니다.

템플릿 기반 설계, GPU별 튜닝, TMEM 및 TMA 파이프라인에 대한 내장 지원 덕분에 지원되는 형상에서 일반적으로 높은 텐서 코어 활용도를 달성합니다. 이를 통해 개발자의 최소한의 노력으로 경우에 따라 96%~98%의 텐서 코어 활용도를 달성할 수 있습니다.

CUTLASS의 자동 최적화에 의존함으로써, 모델 아키텍처, 수치 정밀도 전략 및 종단 간 성능에 시간을 할애할 수 있습니다. CUTLASS는 특정 GPU 하드웨어에 최적화된 저수준 텐서 연산이 거의 최대 연산 강도로 실행될 것이라는 확신을 제공합니다.

NVIDIA는 FP8, FP4, 스레드 블록 클러스터 쌍, TMEM 등 최신 하드웨어 기능을 활용하기 위해 CUTLASS 및 cuBLAS 와 같은 라이브러리를 지속적으로 업데이트합니다. 이러한 라이브러리를 사용하면 새로운 GPU 세대가 나올 때마다 커널을 다시 작성할 필요가 없습니다. 새로운 GPU 아키텍처로 전환할 때는 항상 CUTLASS의 새 버전을 확인하세요.

마이크로최적화를 위한 인라인 PTX 및 SASS 튜닝

C++을 넘어 저수준 마이크로최적화 로 나아가려는 개발자를 위해, CUDA는 인라인 병렬 스레드 실행(PTX) 코드와 SASS(NVIDIA 어셈블리 언어)를 허용하여 성능 향상의 마지막 가능성을 끌어냅니다.

이는 CUDA 컴파일러가 이미 최적화 성능이 매우 뛰어나기 때문에 진정한 고급 영역에 해당합니다. 그러나 극단적인 경우, 어셈블리 명령어를 직접 스케줄링하거나 특수 목적 명령어를 사용하여 매우 특정 상황에서 소폭의 성능 향상을 얻을 수 있습니다.

PTX와 스트리밍 어셈블러(SASS)를 통해 상위 수준의 CUDA 언어에서는 아직 노출되지 않은 기능들도 활성화할 수 있습니다. 현대 GPU는 일반적으로 급진적인 새로운 어셈블리 명령어를 도입하지는 않지만, 맞춤형 튜닝의 기회를 제공합니다. 예를 들어 GPU 캐싱 전략을 조정하거나, CPU-GPU 통합 메모리 접근의 조정을 수정하거나, 기타 세밀한 마이크로 최적화를 구현할 수 있습니다.

PTX("피텍스")는 저수준 병렬 스레드 실행 가상 머신으로, GPU를 병렬 컴퓨팅 장치로 노출합니다. NVIDIA GPU용 프로그래밍 모델과 명령어를 제공합니다. 고수준 컴파일러(예: CUDA C++)는 PTX 명령어를 생성하며, 이는 대상 아키텍처의 네이티브 명령어로 변환됩니다. SASS는 NVIDIA GPU 하드웨어에서 실제로 네이티브로 실행되는 저수준 어셈블리 언어입니다.

예를 들어, 특정 명령어 시퀀스가 최적일 것임을 알고 있지만 컴파일러가 해당 시퀀스를 생성하지 않는 코드 조각을 생각해 보십시오. 일반적인 시나리오에는 직접적인 CUDA 내장 함수가 없는 GPU 명령어 사용, 특정 액세스에 메모리 로드 수

정자(캐시 힌트) 적용, 정확한 지점에 메모리 펜스 또는 배리어 삽입, 파이프라인 정체를 피하기 위한 명령어 수동 재정렬 등이 포함됩니다.

또 다른 용도는 SM ID, 워프 레인 ID 등과 같은 특수 레지스터나 상태를 읽는 것으로, 이에 대한 상위 레벨 API가 없을 수 있습니다. 인라인 PTX를 사용하면 `asm()` 문을 통해 CUDA C++ 코드에 어셈블리를 직접 삽입할 수 있습니다. 어셈블리 코드에 대한 입력과 출력을 지정하여 C++과 PTX를 혼합할 수 있습니다. 그러면 컴파일러가 최종 SASS에 PTX 명령어를 통합합니다.

L2 캐시에 글로벌 메모리 주소를 프리페치하기 위해 인라인 PTX 지시어를 사용하는 간단한 예시를 살펴보겠습니다. 여기서는 PTX 명령어 `cp.async.bulk.prefetch.global`를 사용한 커널 측 프리페칭을 수행합니다:

```
__global__ void PrefetchExample(const float *in, float *out) {
    // ... assume idx is our thread's data index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Manually prefetch the next cache line (128B) of in[] into L2:
    // Prefetch 128B from global to L2.
    // Address must be 16B-aligned
    // and size is a 16B multiple.
    asm volatile("cp.async.bulk.prefetch.L2.global [%0], %1;"
                 :: "l"(in + idx + 32), "n"(128));
    float x = in[idx];
    // (do some work here before using in[idx+32] to give time for prefetch
    out[idx] = x;
}
```

이 코드 조각에서 인라인 PTX 명령어 `cp.async.bulk.prefetch.L2.global [%0]`는 제공된 주소 연산자(`in + idx + 32` 바이트, 즉 32개의 부동소수점 연산자 앞)를 사용하여 L2로 프리페치를 발행합니다. 컴파일러가 이를 최적화하여 제거하지 않도록 `volatile`로 표시합니다.

이러한 PTX 명령어는 기계 코드에 주입됩니다. 이와 같은 인라인 어셈블리를 사용하면 매우 세밀한 제어가 가능합니다. 예를 들어, `.L1` 을 사용하여 L2 또는 L1로 프리페치하거나 거리(이 경우 32개의 부동소수점 연산자 앞)를 선택할 수 있습니다.

이는 기본적으로 `__prefetch_async`가 컴파일될 때의 형태입니다. 보다 일반적으로, 인라인 PTX를 사용해 일반 로드 명령어의 캐싱 동작을 제어할 수 있습니다. 예를 들어, `asm("ld.global.cg.f32 %0, [%1];" : "=f"(val) : "l"(ptr))`를 작성해 `.cg`("캐시 글로벌") 수정자를 적용한 부동소수점 로드를 수행할 수 있습니다.

일부 아키텍처에서는 이로 인해 데이터를 L2에 캐싱하되 L1 캐시를 우회하도록 지시하는 의미가 됩니다. 특정 액세스가 L1을 스래싱하고 있으며 L2만 사용하기

를 선호하는 경우, 이 방법이 도움이 될 수 있습니다. 일반적으로 컴파일러의 선택은 L1 캐싱(`.ca`)으로 기본 설정될 수 있지만, PTX를 사용하여 컴파일러의 결정을 재정의할 수 있습니다.

최신 아키텍처에서 L2 프리페치를 수행하려면 가능한 경우 `cp.async.bulk.prefetch.tensor.L2` 를 사용하십시오. 문서화되지 않은 내장 함수 사용보다 이 방법이 선호됩니다. 어쨌든 이 기능이 존재한다는 점을 아는 것이 유용합니다.

인라인 PTX가 유용한 또 다른 영역은 명령어 스케줄링입니다. 기본적으로 컴파일러는 최적이라고 판단하는 순서로 명령어를 발행합니다. 그러나 연산을 더 효과적으로 혼합하고 싶은 경우가 있을 수 있습니다.

예를 들어, 두 개의 독립적인 메모리 로드와 그 결과값을 사용하는 두 개의 명령이 있다고 가정해 보겠습니다. 컴파일러는 load1, use1, load2, use2 순으로 명령을 발행할 수 있습니다. 그러나 더 나은 명령 스케줄은 load1과 load2를 연속적으로 수행한 후 두 결과값을 모두 사용하는 것일 수 있습니다. 이렇게 하면 메모리 지연 시간을 중첩시킬 수 있습니다.

로드 명령어에 대해 인라인 PTX를 작성하면 이를 조기에 강제 실행한 후 계산을 수행할 수 있습니다. 이는 앞서 논의한 수동적 ILP(명령어 간 병렬성) 증대 기법입니다. 실제로 현대 컴파일러는 로드 지연 시간을 다른 독립적 명령어로 채우려 하므로 이미 효과적으로 처리합니다. 하지만 인라인 PTX와 SASS 어셈블리를 사용하면 확실성을 확보할 수 있습니다.

CPU와 GPU 메모리를 공유하는 현대적인 CPU-GPU 슈퍼칩에서는, GPU가 CPU가 업데이트한 메모리 위치를 폴링하는 워크로드를 관리할 때 이러한 세밀한 제어가 유용할 수 있습니다. 여기서는 로드 및 스토어에 원하는 캐시 연산자와 함께 `membar.sys` 또는 `__threadfence_system()` 와 같은 적절한 메모리 펜스를 사용하여 의도된 범위에서 일관성을 보장할 수 있습니다. 이는 고수준 CUDA가 직접 노출하지 않을 수 있는 부분입니다.

PTX는 일반적으로 전방 호환성을 유지하지만, SASS 어셈블리는 GPU 아키텍처 세대마다 변경됩니다.

인라인 PTX를 활용해 특수 레지스터를 사용할 수도 있습니다. 예를 들어 스레드 블록이 실행 중인 SM ID를 위한 C++ 내장 함수는 없지만, ` asm("mov.u32 %0, %smid;" : "=r"(smid)) `를 통해 SM ID를 얻을 수 있습니다. 이러한 유연성은 디버깅 및 작업 분할에 유용합니다.

일부 개발자는 지속적 커널에서 ` %smid `을 활용해 특정 작업을 SM당 하나의 블록만 수행하도록 구성하기도 합니다. 이는 CUDA C++ API가 제공하는 범위를 넘어서는 수동 SM 분할을 효과적으로 구현하는 방법입니다.

알고리즘 수준에서 코드가 이미 잘 최적화되어 있다면, 인라인 PTX/SASS로 얻는 성능 향상은 대부분 경우 몇 퍼센트 정도의 점진적 개선에 그치는 경향이 있습니다. 예를 들어 메모리 바운드 커널에서는 로드-사용 지연 버블을 줄이기 위해 명령어를 신중하게 언롤링하고 스케줄링하여 PTX로 두 개의 독립적인 로드 스트림을 사용함으로써 5~10% 정도의 속도 향상을 볼 수 있습니다. 이 경우 컴파일러는 더 보수적으로 동작할 수 있습니다.

연산 바운디드 시나리오에서는 정밀도가 높은 명령어 대신 더 빠른 수학 명령어를 사용하기 위해 인라인 어셈블리를 활용할 수 있습니다. CUDA는 이를 위해 빠른 수학 내장 함수(`__sinf()` 등)를 제공합니다.

C++ 내장 함수가 제공되기 전에는 행렬 곱셈 커널을 작성하는 개발자들이 텐서 코어를 사용하기 위해 PTX 명령어를 임베드하는 경우가 있었습니다. 현재는 이를 위한 고수준 내장 함수가 존재합니다. 하지만 간단히 말해, 어셈블리는 하드웨어 기능을 알게 되는 즉시 CUDA 지원 없이도 활용할 수 있게 해줍니다.

Nsight Compute는 어셈블리 튜닝에 유용한 정보를 제공합니다. "SASS 처리량" 메트릭과 "워프 스톨 원인"을 분석하면 메모리 종속성으로 인한 다수의 스톨을 식별할 수 있습니다. 앞서 언급한 대로 로드 명령어의 순서를 재조정해 볼 수 있습니다.

변경 후에는 "메모리 종속성 스톨"이 줄어들고, 사이클당 더 많은 명령어가 실행되어 명령어 발행 속도가 높아지길 기대할 수 있습니다. 어셈블리 조정은 노동 집약적이며 코드 이식성을 저하시킬 수 있다는 점을 유의하세요.

이는 일반적으로 상위 수준의 최적화를 모두 적용한 매우 핫한 내부 루프에서만 가치가 있습니다. 또한 GPU 세대 변경 시 메모리 지연 시간, 캐시 동작 등을 포함한 가정이 여전히 유효한지 재조정 및 검증해야 할 수 있습니다.

잠재적인 미세 최적화를 설명하기 위해, 커널이 이미 상당히 최적화되었지만 하나의 병목 현상이 남아 있는 시나리오를 고려해 보자: 정수 인덱스 계산과 메모리 로드를 수행하는 타이트한 루프가 그것이다.

인라인 PTX를 사용하면 ` `mad.wide.u32 (base + index * stride)` ` 명령 하나로 바이트 주소를 계산할 수 있습니다. 다음으로, 스트리밍 액세스 패턴에서 L1 캐시를 우회하기 위해 로드 명령을 ` `ld.global.cg` ` 로 발행합니다. 그 결과 루프에서 사용하는 명령어가 줄어들고 L1 캐시 이젝션이 방지됩니다. 이 경우 해당 커널에서 약 7%의 속도 향상을 달성하여 1.07ms에서 1.0ms로 단축했습니다. 표 9-2는 최적화된 CUDA C++과 수동 스케줄링 및 캐시 힌트를 적용한 수작업 PTX의 가상의 전후 비교를 요약합니다.

표 9-2. 최적화된 CUDA C++ 대 수동 스케줄링 및 캐시 힌트가 포함된 수기 PTX

버전	워프 스톨 (메모리)	IPC 문제	커널 시간	속도 향상
최적화된 C++ (컴파일러 스케줄)	사이클의 35%	1.5	1.07 밀리초	1.0× 기본값
수동 조정된 PTX (수동 스케줄링 및 힌트)	사이클의 20%	1.6	1.00 ms	1.07배

여기서 볼 수 있듯이, 튜닝 후 로드 명령어 중첩으로 커널의 메모리 스톨이 감소했고, 캐시 힌트가 일부 지연 시간을 줄였습니다. 또한 ILP(명령어당 연산량)와 IPC(사이클당 명령어 수)가 증가했습니다. 이로 인해 커널 전체 실행 시간이 7% 순수하게 개선되었습니다.

이러한 수치는 이미 최적화된 커널에서 수동 어셈블리에서 기대할 수 있는 결과와 일치합니다. 예를 들어 컴파일러가 부적절한 선택을 한 경우, 수정 사항을 구현하면 더 큰 이득을 얻을 수도 있습니다. 반면 컴파일러가 이미 최적의 구현을 선택했다면 이득은 사실상 제로에 가깝습니다. 또한 잘못된 어셈블리 순서로 성능이 저하될 수 있으므로, 각 변경 사항에 대해 실험과 자질을 수행해야 합니다.

인라인 PTX 및 SASS 어셈블리는 최후의 수단으로 활용할 수 있는 최적화 도구로 볼 수 있습니다. 복잡성을 대가로 궁극적인 제어권을 제공합니다. CUDA C++에서 접근할 수 없는 하드웨어 기능이나 명령어가 필요할 때, 또는 컴파일러 스케줄링을 개선할 수 있는 코드 조각을 정확히 찾아냈을 때 이 마지막 수단을 사용하는 것이 좋습니다. 예로는 맞춤형 메모리 접근 패턴(캐시 힌트, 프리페치), 세밀한 동기화 또는 펜스, 공식 지원 전 신규 명령어 활용 등이 있습니다.

인라인 어셈블리를 적용할 때는 자질을 통해 영향을 반드시 검증해야 합니다. 의도한 대로 스톨 원인이나 명령어 수가 감소하는지 확인해야 합니다. 또한 이러한 코드는 분리하여 보관하고 철저히 문서화해야 합니다. 특히 항상 전방 호환되지 않는 SASS 어셈블리를 사용하는 경우, 새로운 GPU 아키텍처에 맞춰 업데이트가 필요할 가능성이 높습니다.

PTX가 SASS보다 안정적이지만, 일부 하드웨어 변경 사항으로 인해 성능 향상을 위해 인라인 PTX를 업데이트해야 할 수도 있습니다.

인라인 PTX/SASS 튜닝은 추가 지연 시간을 줄이거나 특정 스케줄링을 강제하기 위한 전문가 수준의 미세 조정입니다. 소폭의 속도 향상과 특정 사용자 정의 동작을 가능하게 하지만, 다른 모든 상위 수준 최적화를 모두 시도한 후에야 고려해야 합니다. 예를 들어, 수백만 번 실행되는 핵심 루프에 대해 어셈블리를 수작업으로 작성할 수 있습니다. 최소한 하드웨어가 코드를 정확히 어떻게 실행하는지 이해하는 효과적인 방법입니다.

요약하자면, 인라인 PTX/SASS는 신중하게 사용하고 자질을 철저히 수행하세요. 성능 향상은 현실적이지만 일반적으로 점진적입니다. 유지 관리 비용은 훨씬 더

높습니다. 대부분의 사용 사례에서는 CUDA의 내장 최적화 기능이나 CUB, Thrust와 같은 고도로 튜닝된 라이브러리에 의존하는 것이 충분할 것입니다. 하지만 필요할 경우 어셈블리 수준으로 내려가 GPU에 대한 완전한 제어권을 얻을 수 있다는 점을 알아두는 것이 좋습니다.

DeepSeek의 메모리 할당 최적화를 위한 인라인 PTX 활용

DeepSeek의 DeepEP 전문가 병렬 처리 라이브러리에서 맞춤형 PTX 의 잘 알려진 사례를 확인할 수 있습니다. 이 라이브러리는 L1 캐시 할당을 우회하고, 핵심 데이터를 보존하며, 256바이트 L2 캐시 청크를 활용하여 글로벌 메모리 접근을 최적화하기 위해 맞춤형 PTX 명령어 `ld.global.nc.l1::no_allocate.l2::256b` 를 사용했습니다. 이는 L1 캐시의 빈번한 접근 메모리 작업을 방해하지 않으면서 대규모 데이터셋을 L2 캐시로 직접 스트리밍하는 데 이상적입니다.

이 명령어는 NVIDIA의 공식 PTX ISA 사양에 포함되지 않으나, DeepSeek 엔지니어들이 미국 수출 제한이 적용된 Hopper GPU의 H800 변종에서 캐시 동작을 미세 조정하기 위해 '문서 외'로 발견했습니다.

`ld.global.nc.l1::no_allocate.l2::256b` PTX 명령어를 분석해 보겠습니다. `ld.global.nc` 접두사는 비일관성(`nc`) 글로벌 메모리 로드(`ld.global`)를 발행하며, 수정자 `l1::no_allocate` 와 `l2::256b` 는 하드웨어가 데이터를 L1 캐시에 할당하지 않도록 지시합니다(`l1::no_allocate`). 대신 256바이트 단위로 L2에 직접 가져옵니다(`l2::256b`).

L1을 우회함으로써, 로드 작업은 자주 사용되는 L1 상주 데이터를 제거하지 않고도 대용량 데이터 블록을 L2로 직접 스트리밍할 수 있습니다. 이는 저지연 메모리 액세스를 위해 L1에 상주해야 하는 핫 작업 세트가 존재할 때 매우 중요합니다.

실제 환경에서 전문가 병렬 전수 대 전수 통신 커널과 같은 스트리밍 워크로드는 이 접근 방식의 혜택을 받습니다. 이러한 워크로드는 대개 디스패치 당 정확히 한 번에 큰 연속 버퍼를 읽기 때문입니다. 만약 이러한 로드가 L1을 통과했다면, SM에서 여전히 활발히 사용 중인 이전 캐시 라인을 이젝트할 수 있습니다.

256바이트 정렬된 청크를 L2로 직접 가져옴으로써, 이 명령어는 불필요한 L1 트래픽을 줄입니다. 이는 메모리 제약 통신 및 연산 작업 모두에서 높은 처리량을 유지하는 데 도움이 됩니다.

그러나 이러한 방식으로 PTX를 사용하는 것은 위험을 수반합니다. 왜냐하면 ' `ld.global.nc.l1::no_allocate.l2::256b` ' 명령어는 GPU 세대 간 안정성이 보장되지 않기 때문입니다. 따라서 이 명령어에 의존하는 코드는 향후 아키텍처에서 작동하지 않거나 잘못된 결과를 생성할 수 있습니다.

DeepSeek의 DeepEP 설정에는 호환성 문제가 발생할 경우 이러한 공격적인 명령어를 비활성화하는 빌드 타임 플래그 `DISABLE_AGGRESSIVE_PTX_INSTRS=1`가 포함되어 있습니다. DeepEP의 맞춤형 PTX 해킹은 상당한 속도 향상을 가져올 수 있지만, 인라인 PTX/SASS는 새로운 GPU 아키텍처로 업데이트할 때마다 신중하게 사용하고 철저히 테스트해야 합니다.

핵심 요약

느린 글로벌 메모리에서 더 빠른 온칩 리소스와 컴퓨트 유닛으로 작업을 이동하여 GPU 커널 병목 현상을 발견하고 제거하는 방법을 살펴보았습니다. 자질, 진단, 최적화, 재프로파일링의 주기를 따르면 커널을 활용도가 낮거나 메모리 바운디드 상태에서 컴퓨트 포화 상태의 고처리량 루틴으로 전환할 수 있습니다. 이러한 기법은 GPU의 전체 성능을 활용하는 데 도움이 됩니다:

타일링 및 퓨전으로 산술 집약도 향상

전송된 바이트당 FLOPS를 높이기 위해 다단계 타일링을 사용하여 데이터를 공유 메모리와 레지스터에 단계적으로 배치하십시오. 예를 들어, 32×32 부분 행렬을 SMEM에 로드하여 각 요소가 여러 FMA에 걸쳐 재사용되도록 합니다. 연속적인 커널을 결합하여 요소별 연산을 융합함으로써 중간 결과가 온칩에 유지되도록 합니다. CUDA 파이프라인 API의 `cuda::memcpy_async`를 통한 비동기 메모리 로딩으로 소프트웨어 프리페칭을 활용하여 데이터 이동과 계산을 중첩시킵니다. 이는 DRAM 지연 시간을 숨깁니다.

혼합 정밀도, 텐서 코어 및 트랜스포머 엔진 활용

FP32에서 TF32/BF16/FP16/FP8/FP4로 전환하면 데이터 전송량을 2배에서 8배까지 줄일 수 있습니다. 이에 따라 연산 집약도가 향상됩니다. PyTorch에서는 `torch.set_float32_matmul_precision('high')` 및 `torch.cuda.amp`를 사용하여 혼합 정밀도를 활성화할 수 있습니다. 최신 GPU에서는 TMEM 및 TMA 엔진이 타일을 텐서 코어로 스트리밍합니다. 또한 최신 GPU는 AI 워크로드에 특화된 저정밀도 연산을 위한 트랜스포머 엔진을 제공합니다. 이를 통해 처리량을 더욱 향상시킬 수 있습니다.

고성능 GEMM 및 융합 커널을 위한 CUTLASS 활용

MMA 루프를 수동으로 코딩하기보다는 CUTLASS GEMM 템플릿을 인스턴스화하여 더블 버퍼링, TMEM 스테이징, 텐서 코어 파이프라인을 자동으로 관리하세요. 이렇게 하면 수동으로 튜닝한 커널과 몇 퍼센트 차이로 성능을 발휘하는 커널이 생성됩니다. cuBLAS 및 TorchInductor와 같은 고급 프레임워크는 이미 CUTLASS에 의존하고 있습니다. 비표준 레이아웃이나 독특한 퓨전 패턴이 필요한 경우에만 커스터마이징이 필요합니다.

PyTorch 전용 모범 사례

> PyTorch는 시간이 지남에 따라 CUDA의 텐서 코어 및 기타 하드웨어 최적화를 상속하는 경향이 있으므로, `torch.matmul`, 융합 어텐션, 중첩 텐서와 같은 내장 텐서 연산을 우선적으로 사용하십시오. PyTorch용 커스텀 커널 확장을 작성할 경우에도 동일한 전략을 적용하십시오: 레지스터 사용량 최소화, 병합 메모리 로드를 위한 데이터 정렬, 텐서 코어 타겟팅. 이를 통해 커널이 네이티브 커널과 동등한 성능을 보장할 수 있습니다.

자질, 진단, 그리고 점유율 조정과 워프 수준 개선부터 ILP 및 타일링, 퓨전, 텐서 코어를 활용한 고산술 집약도 적용에 이르는 체계적인 최적화 워크플로를 따르면, 메모리 제약 GPU 커널을 컴퓨팅 제약 커널로 전환할 수 있습니다. 이는 상당한 속도 향상을 제공하며, 특히 메모리 제약이 심한 워크로드에서는 때로 수십 배에 달하는 성능 향상을 가져올 수 있습니다.

결론

이 장에서는 TMA, TMEM, 트랜스포머 엔진, 텐서 코어와 같은 고급 메모리 및 컴퓨팅 하드웨어 기능과 관련된 최적화 기법에 대해 알아보았습니다. 동일한 원칙은 단일 GPU에서 다중 GPU 클러스터까지 확장 적용됩니다. 먼저 시스템 수준 프로파일러(예: NVIDIA Nsight Systems)를 사용하여 여러 GPU 간 CPU 활동, GPU 커널, NVLink/NVSwitch 트래픽 간의 상관관계를 분석하십시오. 그런 다음 Nsight Compute를 사용하여 커널별 세부 사항을 심층 분석하십시오.

체계적인 자질 분석, 주요 스톨 제거, 고급 하드웨어 기능 숙달을 통해 메모리 제약 워크로드를 컴퓨팅 바운디드 워크로드로 전환할 수 있습니다. 이는 강력한 메모리 제약 경로에서 수십 배에 달하는 성능 향상을 포함해 상당한 가속화를 가져옵니다.

고대역폭 NVLink/NVSwitch를 갖춘 NVIDIA GB200/GB300 NVL72와 같은 초대형 다중 GPU 시스템에서도 GPU별 연산 집약도 최적화는 대부분의 병목 현상을 제거하는 핵심입니다. 바이트당 작업량이 너무 적은 커널은 과도한 메모리 이동을 유발하여 상호 연결을 포화시키고 메모리 대역폭을 한계까지 끌어올립니다.

이러한 상호 연결 및 메모리 대역폭 포화는 커널이 다수의 상호 연결된 GPU(예: NVL72의 경우 72개)의 컴퓨팅 성능을 완전히 활용하기 훨씬 전에 발생합니다. 따라서 커널 산술 집약도를 높이는 것이 다중 GPU 훈련 및 추론 워크로드의 효율적인 확장을 위한 핵심입니다.

다음 장에서는 지속적 커널(persistent kernels), 메가커널(megakernels), 워프 전문화(warp specialization), 협력 그룹(cooperative groups), 스레드 블록 클러스터(thread block clusters)와 같은 기법들을 심층적으로 계속 살펴보겠습니다. 이러한 아이디어들은 대부분의 현대적 LLM 런타임 최적화의 기반이 되므로, 그 구현 방식과 여러분의 저수준 시스템 최적화 작업에 이를 적용하는 방법을 이해하는 것이 중요합니다.