제12장. 동적 스케줄링, CUDA 그래프 및 장치 주도 커널 오케스트레이션

지금까지 개별 커널 수준에서 연산 및 메모리 처리량을 극대화했습니다. 이제 GPU가 유휴 상태가 되지 않도록 이러한 커널들을 오케스트레이션할 차례입니다.

이 장에서는 호스트에서의 스케줄링에서 장치 자체에서의 스케줄링으로 전환합니다. 빠른 L2 캐시 원자 연산으로 구동되는 동적 작업 대기열을 탐구하고, 반복되는 커널 실행을 통합하며, 고정된 파이프라인을 배치 처리하고 CPU 핸드셰이크를 최소화하기 위해 CUDA 그래프를 사용할 것입니다.

이후 장치 측 그래프 실행과 동적 병렬화를 통해 오케스트레이션을 한 단계 더 발전시킬 것입니다. 이를 통해 GPU가 CPU로 다시 호출할 필요 없이 다음 실행 작업을 스스로 결정할 수 있습니다.

마지막으로, 피어 투 피어 복사, NCCL 콜렉티브, CUDA 인식 MPI, NVSHMEM 일방적 풋/겟을 중첩하여 다중 GPU 환경을 심층적으로 다룹니다. 이를 통해 GPU 클러스터는 하나의 거대한 공유 메모리 코프로세서처럼 작동합니다. 예를 들어 NVIDIA의 DGX GB200 NVL72 시스템은 36개의 Grace CPU와 72개의 Blackwell GPU를 단일 NVLink 도메인으로 연결합니다. 이 도메인 내에서는 통합 주소 지정과 최대 30TB의 CPU 및 GPU 통합 메모리를 사용할 수 있습니다. 또한 72개 GPU 도메인 내 NVLink 패브릭을 통해 원격 HBM 접근이 가능합니다. 더 큰 NVLink 네트워크 토폴로지는 단일 랙을 넘어 확장될 수 있습니다.

이 과정에서 각 기법을 루프라인 분석과 연계하여 커널의 연산 집약도를 높이기 위한 적절한 도구(스트림, 그래프, 원자 연산, 동적 커널)를 선택하는 방법을 안내합니다. 이는 워크로드의 전반적인 성능 향상에 기여할 것입니다.

이 장을 마치면 다중 GPU 클러스터 전반에 걸쳐 모든 SM에 데이터를 공급하는 동적, 장치 기반, 그래프 기반 커널 오케스트레이션 기법을 이해하게 될 것입니다.

원자적 작업 대기열을 활용한 동적 스케줄

링

스레드 간 작업 할당이 불균형하면 일부 SM은 유휴 상태인 반면 다른 SM은 계속 바쁘게 작동할 수 있습니다. 이는 컴퓨팅 자원을 낭비하고 전체 처리량을 저하시킵니다.

불균형은 입력 의존적 루프나 조건부 워크로드로 인해 서로 다른 스레드나 블록이 변동적인 양의 작업을 처리할 때 자주 발생합니다. 일부 블록은 빠르게 완료되어 해당 SM을 유휴 상태로 남겨두는 반면, 다른 SM들은 더 오래 실행되는 블록을 계속 처리합니다. 수백 개의 SM을 가진 현대 GPU에서는 작업이 균등하게 분배되지 않으면 유휴 기간 동안 많은 SM이 유휴 상태로 남을 수 있습니다. 이는 성능에 심각한 악영향을 미칠 수 있습니다.

가장 오래 걸리는 작업이 완료될 때까지 GPU의 일부가 유휴 상태로 남아 있습니다. 이는 많은 사이클 동안 활성 워프가 실행되지 않았기 때문에 달성된 점유율을 감소시킵니다. Nsight Systems를 사용하여 자질하고 GPU 타임라인에서 이러한 유휴 간격을 명확하게 표시할 수 있다는 점을 기억하십시오.

활성 SM 사이클을 총 경과 SM 사이클과 비교하여 활용도 저하를 측정할 수도 있습니다. Nsight Compute는 이를 단일 지표로 제공하며, 이는 최소 한 개의 워프가 활성 상태였던 시간의 비율을 나타냅니다. 낮은 활성 대 경과 비율은 많은 사이클이 활성 워프 없이 실행되었음을 의미합니다. 즉, GPU가 자주 유휴 상태였다는 뜻입니다.

Nsight Systems 외에도 Nsight Compute를 사용해 달성된 점유율(하드웨어 최대치 대비 SM당 활성 워프의 평균 비율)이나 SM 활성 사이클 비율(최소 하나의 워프가 활성 상태였던 시간 비율)을 확인하여 이러한 활용도 저하를 정량화할 수 있습니다.

타임라인 간격과 특정 코드 섹션을 연관시키려면 중요한 GPU 작업 주변에 NVTX 범위 마커를 삽입하세요.

다음으로 커널 내부에서 작업을 동적으로 할당하기 위한 원자적 큐 구현 방법을 논의하겠습니다. 이는 모든 SM에 걸쳐 임의의 작업 부하를 균형 있게 분배하여 유휴 스레드를 방지하는 데 중요합니다. 이를 위해 먼저 원자적 카운터를 소개해야 합니다.

원자 카운터

원자 카운터는 동적 작업 할당을 가능하게 하는 원자 큐의 기반입니다.

`atomicAdd` 현대 GPU에서는 글로벌 원자 연산이 온디바이스 L2 캐시에서 처리 및 직렬화됩니다. 이는 대상 라인이 상주할 때 DRAM 왕복 대비 지연 시간을 줄여줍니다. 원자 카운터는 여전히 지연 시간을 발생시키며 경합 시 직렬화됩니

다. 그러나 경합이 없는 원자 카운터 연산은 온칩 상태를 유지함으로써 극히 빠르게 수행됩니다. 두 스레드가 원자 카운터를 증가시키는 예시는 그림 12-1에 표시되어 있습니다.

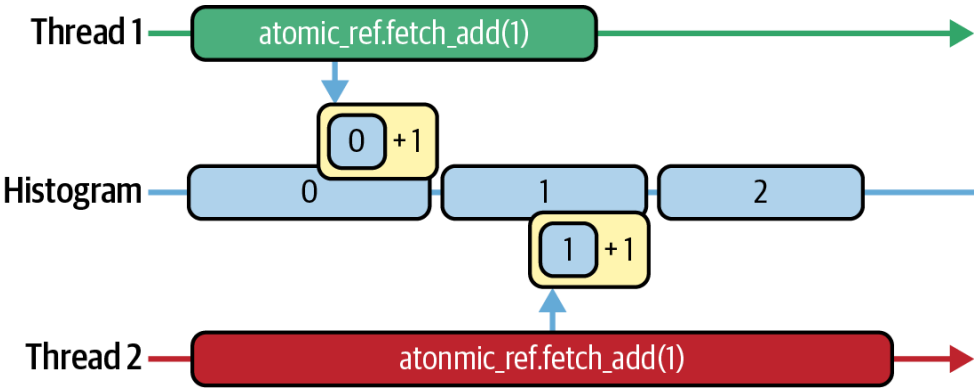

그림 12-1. 히스토그램 계산 컨텍스트에서 다중 스레드 간초고속 온칩 원자 메모리 덧셈 연산

그러나 원-어-타임(`atomicAdd`)은 무료가 아닙니다. 여전히 지연 시간이 발생하며, 경합 시 동일한 메모리 주소를 기다리는 스레드를 직렬화할 수 있습니다. 따라서 L2도 업데이트를 직렬화해야 합니다. 이는 핫스팟을 생성하며, 최적화가 필요합니다. 메모리 작업 부하 분석( )은 Nsight Compute를 통해 비용을 정량화하는 데 도움이 될 수 있습니다.

Nsight Compute의 메모리 워크로드 분석 섹션에서 원자적 캐시 트랜잭션(`atomic_transactions`)과 원자적 캐시 트랜잭션 카운터(`atomic_transactions_per_request`)를 확인할 수 있습니다. 원자적 캐시 트랜잭션 카운터는 경합으로 인한 재실행(replay)을 포함해 L2 캐시 원자적 트랜잭션의 총 횟수를 나타냅니다. 요청당 원자적 캐시 트랜잭션(또는 경합 비율) 지표는 각 원자적 캐시 트랜잭션(`atomicAdd`) 명령어마다 생성되는 평균 L2 트랜잭션 수를 나타냅니다.

`atomicAdd` 가 정확히 하나의 L2 트랜잭션을 발생시킬 때, `atomic_transactions_per_request` 은 1.0 근처에서 유지되며 이는 최소한의 비용만 지불하고 있음을 의미합니다. 이 비율이 1.0을 초과하면 스레드가 유용한 작업을 수행하지 못하고 원자적 업데이트를 재시도하며 지연되고 있음을 나타냅니다. 각 재시도는 경합을 의미합니다.

여기서 최적화는 원자 작업을 배치 단위로 처리하여 비용을 분산시키는 것입니다. 즉, 각 스레드(또는 워프)가 개별 원자 업데이트(`atomicAdd`)를 수행하는 대신, 원자 작업당 여러 작업을 한 번에 처리합니다. 배치 크기 32를 적용한 전후 비교는 다음과 같습니다:

```
// Before batching
int idx = atomicAdd(&queue_head, 1);
if (idx < N) process(data[idx]);

// After batching
const int batchSize = 32;
int start = atomicAdd(&queue_head, batchSize);
for (int i = start; i < start + batchSize && i < N; ++i) {
```

```
        process(data[i]);
    }
```

이제 단일 원자 업데이트로 워프(또는 스레드 블록)에 전체 작업 슬라이스(이 예시에서는 32개 항목)를 할당하고 카운터를 다시 만집니다. 배치당 L2 트랜잭션 비용은 동일하지만, 그 사이에 32배 더 많은 유용한 작업을 수행합니다.

`atomicAdd` 실제 구현에서는 워프당 하나의 스레드만 다음 배치 시작 인덱스를 가져오기 위해 원자적 업데이트를 수행합니다. 해당 스레드는 이를 워프 내 나머지 스레드에 브로드캐스트합니다(예: ` `__shfl_sync` ` 사용). 이후 전체 워프가 해당 32개 항목을 병렬로 처리합니다. 이로 인해 스레드당 원자적 작업이 아닌 워프당 하나의 원자적 작업이 발생하여 경합이 급격히 감소합니다.

Nsight Compute에서 `atomic_transactions` 이 급감하고 요청당 트랜잭션 수가 1.0 수준으로 회복되는 것을 확인할 수 있습니다. 이는 비용이 큰 경합을 지속적 연산으로 대체했음을 증명합니다.

L2 캐시 원자 연산이 매우 빠른 최신 GPU에서는 L2 대역폭이 높아 8이나 16과 같은 적당한 배치 크기만으로도 대부분의 경합을 제거할 수 있습니다. 다만, 단순히 병목 현상이 다른 곳으로 옮겨진 것은 아닌지 항상 확인해야 합니다.

이 최적화가 다른 성능 지표에 부정적 영향을 미치지 않았는지 확인하려면 Nsight Compute의 'Warp Stall Reasons' 및 'Register Pressure' 보고서를 활용하여 융합 루프가 레지스터 스필이나 공유 메모리 뱅크 충돌로 제한받지 않는지 확인하십시오.

이러한 최적화 후에도 원자 연산이 여전히 성능 병목이라면, 블록별 카운터나 작업 분배의 계층적 축소 같은 대체 설계를 고려하십시오.

요약하면, 원자 연산별로 작업을 배치 처리함으로써 GPU의 다수 워프가 실제 계산을 수행하도록 유지합니다. 이는 따라잡지 못하는 단일 카운터에 작업을 대기열로 쌓는 방식과 대비됩니다.

원자적 큐

이제 전역 원자 카운터( )를 사용하여 동적 작업 큐를 조정해 보겠습니다. 목표는 원자 카운터와 `atomicAdd` 를 활용하여 모든 SM에 걸쳐 임의의 작업 부하를 균형 있게 분배함으로써 어떤 스레드나 워프도 유휴 상태가 되지 않도록 하는 것입니다. 이러한 동적 작업 큐의 예시는 그림 12-2에 나와 있습니다.

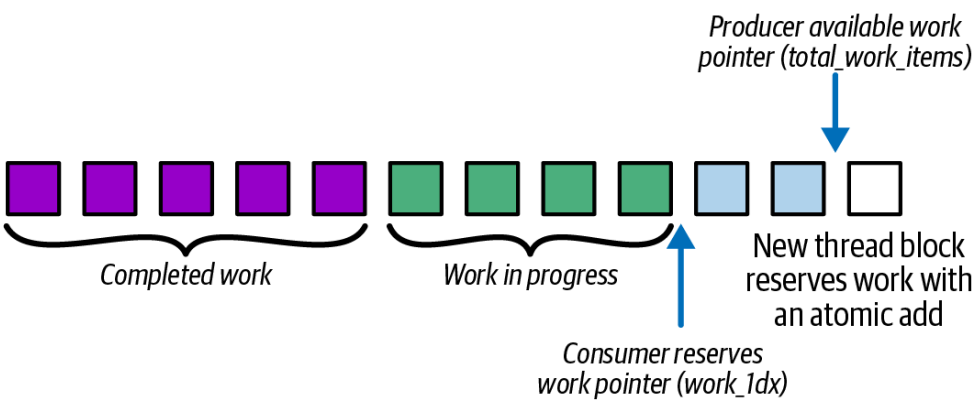

그림 12-2. 원자 카운터와 `atomicAdd` 을 동적 작업 대기열로활용하여 SM 및 워프 간 작업 부하 균형 조정

다음 코드 예시(`computeKernel`)에서 각 스레드는 `idx % 256` 에 따라 서로 다른 반복 횟수를 계산합니다. `idx % 256` 값이 작은 스레드는 거의 작업을 수행하지 않는 반면, `idx % 256` 값이 큰 스레드는 많은 작업을 수행합니다. 이러한 불균형으로 인해 스레드마다 완료 시점이 달라지고, 일부 SM은 가장 오래 걸리는 스레드가 완료되기를 기다리며 유휴 상태가 됩니다. 다음은 스레드당 정적이며 불균등한 작업 부하를 사용하는 코드입니다:

```
// uneven_static.cu
#include <cuda_runtime.h>
#include <cmath>

__global__ void computeKernel(const float* input, float* output, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        // Each thread does a variable amount of work based on idx
        int work = idx % 256;
        float result = 0.0f;
        for (int i = 0; i < work; ++i) {
            result += sinf(input[idx]) * cosf(input[idx]);
        }
        output[idx] = result;
    }
}

int main() {
    const int N = 1<<20;

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

    for (int i = 0; i < N; ++i) h_in[i] = float(i) / N;

    float *d_in, *d_out;
    cudaMalloc(&d_in, N * sizeof(float));
    cudaMalloc(&d_out, N * sizeof(float));
    cudaMemcpy(d_in, h_in, N * sizeof(float), cudaMemcpyHostToDevice);

    dim3 block(256), grid((N + 255) / 256);
```

```
    computeKernel<<<grid, block>>>(d_in, d_out, N);
    cudaDeviceSynchronize();

    cudaMemcpy(h_out, d_out, N * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_in);
    cudaFree(d_out);
    cudaFreeHost(h_in);
    cudaFreeHost(h_out);

    return 0;
}
```

동적 GPU 측 작업 분배를 위한 고급 PyTorch API는 존재하지 않으므로, 커스텀 CUDA 커널로 구현해야 합니다. 간결성을 위해 이 부분은 생략합니다.

다음에 제시된 최적화된 동적 작업 배분 버전에서는 단일 글로벌 카운터(장치 메모리 내)를 워프 수준 작업 대기열로 사용합니다. 우리는 배치 원자 연산을 사용하여 이 카운터를 지속적 워프 수준 작업 대기열로 전환합니다. 이렇게 하면 일찍 완료된 워프는 유휴 상태가 되는 대신 즉시 다른 배치 작업을 가져옵니다:

```
// uneven_dynamic.cu

#include <cuda_runtime.h>

__device__ unsigned int globalIndex = 0;

// Warp-batched dynamic queue: 1 atomic per active warp
__global__ void computeKernelDynamicBatch(const float* input,
                                          float* output,
                                          int N) {
  // lane id in [0,31]
  int lane = threadIdx.x & (warpSize - 1);

  while (true) {
    // Elect an active leader each iteration (handles divergence safely)
    unsigned mask = __activemask();
    int leader = __ffs(mask) - 1;

    // Warp leader atomically grabs a contiguous batch for the whole warp
    unsigned int base = 0;
    if (lane == leader) {
      base = atomicAdd(&globalIndex, warpSize);
    }

    // Broadcast starting index to all active lanes in the warp
    base = __shfl_sync(mask, base, leader);

    unsigned int idx = base + lane;
```

```
    if (idx >= (unsigned)N) break;  // dynamic termination

    // Hoist invariants out of the variable trip-count loop
    // Note: You can also use __sincosf on Blackwell
    float s = sinf(input[idx]);
    float c = cosf(input[idx]);
    int   work = idx % 256;

    float result = 0.0f;
    #pragma unroll 1
    for (int i = 0; i < work; ++i) {
      result += s * c;
    }
    output[idx] = result;
    // loop continues until counter >= N
  }
}

int main() {
    const int N = 1 << 20;
    float *d_in, *d_out;
    cudaMalloc(&d_in,  N * sizeof(float));
    cudaMalloc(&d_out, N * sizeof(float));
    // Host buffers (pinned) for a realistic data path
    float *h_in = nullptr, *h_out = nullptr;
    cudaMallocHost(&h_in,  N * sizeof(float));
    cudaMallocHost(&h_out, N * sizeof(float));
    for (int i = 0; i < N; ++i) {
        h_in[i] = static_cast<float>(i % 1000);
    }

    // Copy inputs to device
    cudaMemcpy(d_in, h_in, N * sizeof(float),
        cudaMemcpyHostToDevice);

    // Reset global counter
    unsigned int zero = 0;
    // If you call this kernel repeatedly (e.g., in a loop),
    // reset 'globalIndex' to 0 before each launch.
    cudaMemcpyToSymbol(globalIndex, &zero,
        sizeof(unsigned int));

    // Launch with 256 threads per block
    dim3 block(256), grid((N + 255) / 256);
    cudaStream_t stream;

    cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

    computeKernelDynamicBatch<<<grid, block, 0, stream>>>(d_in, d_out, N);

    cudaStreamSynchronize(stream);
    cudaStreamDestroy(stream);
    cudaDeviceSynchronize();
```

```
    // Copy results back and clean up
    cudaMemcpy(h_out, d_out, N * sizeof(float),
               cudaMemcpyDeviceToHost);

    cudaFree(d_in);
    cudaFree(d_out);
    cudaFreeHost(h_in);
    cudaFreeHost(h_out);

    return 0;
}
```

각 워프는 전역 큐에서 크기 `warpSize` (32개 스레드)의 다음 작업 배치를 원자적으로 확보한 후 루프에서 처리합니다. 이를 통해 어떤 SM도 유휴 상태가 되지 않도록 보장합니다. 이 코드는 단일 전역 원자 작업 큐로 구현된 동적 작업 분배를 수행합니다.

여기서 각 워프는 해당 글로벌 카운터에서 다음 배치 기본 인덱스를 반복적으로 가져옵니다. 각 워프의 첫 번째 스레드(`if (lane==0)`)는 워프 리더라고 불리며, `base=atomicAdd(&globalIndex, warpSize)` 를 사용하여 이 연속 블록의 시작 인덱스를 원자적 추가 연산으로 획득합니다. 그런 다음 이 장 초반에 설명한 바와 같이 `__shfl_sync(__activemask(), base, 0)` 를 사용하여 이 기본 인덱스를 워프의 나머지 스레드에 브로드캐스트합니다.

다시 말해, 각 스레드가 고정된 요소 인덱스에 묶이는 대신, 이제 모든 워프가 공유 카운터에서 연속적인 작업 블록을 가져옵니다. 그런 다음 `idx = base + lane` 를 사용하여 계산합니다.

해당 워프 내 모든 스레드는 동적으로 가져온 인덱스에 대해 동일한 `sin`/`cos` 루프를 실행합니다. 따라서 작업이 더 이상 스레드별로 사전 할당되지 않습니다. 대신 작업은 런타임에 글로벌 원자적 큐를 사용하여 가져오고 균형을 맞춥니다.

`if (idx >= N)` 의 경계 검사는 작업이 더 이상 없을 때 워프가 종료되도록 합니다. 이는 바운디드 메모리 접근을 방지합니다. 그렇지 않으면 워프 내 각 스레드는 정적 버전과 정확히 동일한 `sin`/`cos` 루프를 실행합니다.

`N = 1 << 20` 와 `work = idx % 256` 을 사용한 간단한 마이크로벤치마크에서 정적 할당 커널은 약 200ms가 소요된 반면, 동적 큐 버전은 약 100ms에 실행되었습니다. 이 2배 속도 향상은 SM 유휴 시간 제거와 원자적 경합 감소의 결과입니다. Nsight Compute는 활성 SM 사이클을 최소 한 개의 활성 워프가 존재하는 경과 사이클의 비율로 정의합니다.

속도 향상은 작업 불균형에 따라 달라지지만, 자질 결과 워프 유휴 정지, 낮은 달성 점유율 또는 작업별 실행 시간 불균형으로 인한 가시적인 타임라인 간격이 나타날 때마다 동적 작업 분배는 탐구할 가치가 있는 최적화 기법입니다. 특히 중간 정도의 불균형이 있는 이러한 시나리오에서는 종종 10%~20%의 속도 향상을 얻을 수 있습니다.

극단적인 불균형 사례에서는 정적 인덱싱을 원자적 작업 큐로 대체하기만 해도 2배의 속도 향상을 얻을 수 있습니다. 경미한 불균형의 경우 원자적 작업과 셔플의 오버헤드가 이득을 상쇄할 수 있습니다.

요약하면, 동적 작업 분배는 카운터가 `N` 를 초과할 때까지 모든 워프가 새로운 작업을 계속 가져오고 처리하므로 거의 균일한 SM 활용도를 보장합니다. 이는 많은 워프가 가장 느린 워프보다 훨씬 일찍 완료되어 하드웨어 리소스를 사용하지 않은 채 남겨두는 것과 대조적입니다.

CUDA 그래프

파이프라인이 커널, 복사 작업, 스트림 이벤트 레코드, 콜백 등 여러 작업( )으로 구성될 때, 호스트에서 매 반복마다 하나씩 런칭하면 여전히 CPU 오버헤드가 발생합니다. CUDA 그래프를 사용하면 전체 워크플로우를 한 번 캡처한 후 CPU 오버헤드 거의 없이 반복적으로 재실행할 수 있습니다. 그림 12-3은 CUDA 그래프를 사용하지 않은 경우(위)와 사용한 경우(아래)의 커널 런칭을 비교합니다.

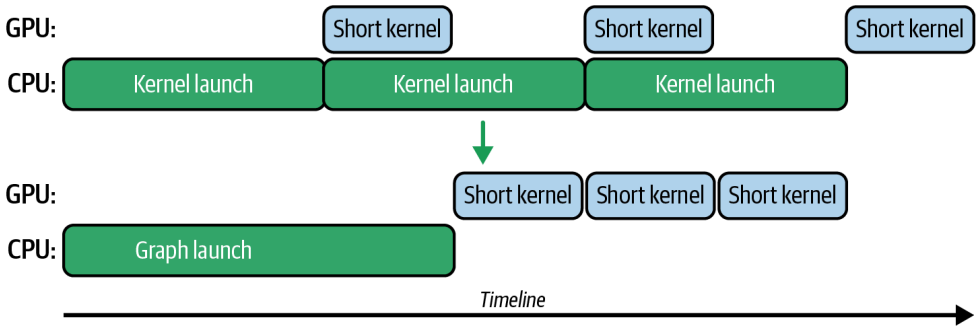

그림 12-3. CUDA 그래프 미적용(위) 및 적용(아래)커널 런치 타임라인

CUDA 그래프를 사용해야 하는 이유는 무엇인가요? 첫째, 런칭 오버헤드를 줄여줍니다. 여러 개의 작은 커널이나 복사 작업을 사실상 하나의 CPU 호출로 런칭할 수 있습니다. 둘째, GPU에서 더 나은 스케줄링을 가능하게 합니다. 작업이 배치로 제출되므로 CUDA 드라이버가 작업 간 내부 지연을 잠재적으로 줄일 수 있습니다.

또한 CUDA 그래프를 사용하면 종속성이 사전에 알려져 있으므로 CPU가 중간에 동기화할 필요가 줄어듭니다. 메모리 전송의 맥락에서 CUDA 그래프는 비동기 복사 작업과 커널 실행이 수동 동기화 없이도 종속성으로 올바르게 연결되도록 보장합니다. 기본적으로 일반 스트림보다 복사 작업과 커널을 더 많이 중첩시키지는 않지만, CUDA 그래프는 이들의 실행을 효율화합니다.

PyTorch, 추론 엔진 및 CUDA 그래프

PyTorch와 같은 AI 프레임워크는 Deep Learning 모델의 정적 부분에 대해 내부적으로 CUDA 그래프( )를 활용합니다. 특히 PyTorch는 연산 시퀀스를 캡처하기 위한 CUDA 그래프 컨텍스트(`torch.cuda.Graph`)를 지원합니다. 또한

PyTorch는 예측 가능한 코드 부분에 CUDA 그래프를 사용하도록 내부 최적화를 지속적으로 진행 중입니다.

vLLM 및 NVIDIA의 TensorRT-LLM과 같은 고성능 추론 엔진 도 모델 실행을 다양한 시퀀스 길이 범위와 입력 배치 크기에 대한 사전 정의된 그래프 집합으로 캡처함으로써 CUDA 그래프를 활용할 수 있습니다. 그래프 캡처가 활성화되면, 이러한 시스템은 지원되는 그래프 배치 크기에 맞추기 위해 입력을 버킷화하거나 패딩하는 경우가 많습니다. 이렇게 하면 캡처된 그래프를 고정된 형태로 재생할 수 있습니다. 이는 대규모 생산 추론 워크로드의 지연 시간을 크게 줄일 수 있습니다.

예를 들어, 시작 또는 모델 로드 시 배치 크기별로 하나의 CUDA 그래프를 캡처합니다. 그런 다음 런타임에 들어오는 요청의 배치 크기와 일치하는 미리 캡처된 그래프를 실행합니다.

PyTorch 컴파일러 `mode='reduce-overhead'` 는 CUDA 그래프 내 적합한 세그먼트를 래핑하여 런치 오버헤드를 줄일 수 있습니다. 단, 정적 텐서 주소나 CUDA 전용 영역 같은 캡처 요구사항이 적용됩니다. 모든 코드 경로의 그래프 생성을 보장하지 않으며, 풀링된 버퍼로 인해 메모리 사용량이 증가할 수 있습니다. 항상 자질을 통해 모델에 대한 이점을 확인하세요.

CUDA 그래프용 메모리 풀

CUDA 그래프와 관련된 중요한 고려 사항 중 하나는 GPU 메모리 관리입니다. CUDA 그래프 내부의 메모리 작업은 CUDA 스트림과 동일한 규칙을 따릅니다. 캡처 내부에서 GPU 메모리를 할당하면 해당 할당은 그래프 실행의 일부가 됩니다.

일반적으로 그래프 내부에서 GPU 메모리를 할당하거나 그래프 외부에서 메모리를 사전 할당하는 것은 피하는 것이 좋습니다. PyTorch와 같은 많은 프레임워크는 그림 12-4에 표시된 것처럼 CUDA 그래프와 함께 정적 메모리 풀을 사용합니다. 정적 메모리 풀을 사용하면 메모리 할당이 캡처된 그래프 시퀀스의 일부가 되는 것을 방지할 수 있습니다.

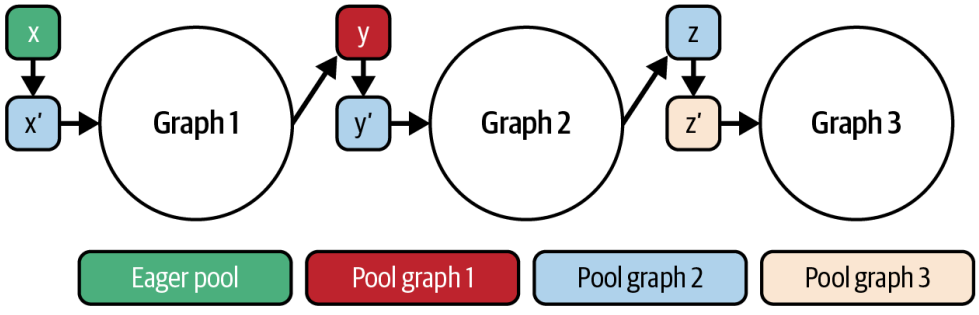

그림 12-4. PyTorch가 CUDA 그래프에 정적 메모리 풀을 사용하는 방식

CUDA 그래프는 개별 메모리 복사나 커널 실행 속도를 직접 향상시키지는 않지만, CUDA 스트림과 유사하게 그래프 내 독립적인 데이터 전송과 연산을 자동으

로 중첩시킬 수 있습니다. 이는 의존성 그래프가 사전에 알려져 있기 때문에 가능하며, 이로 인해 반복당 CPU 스케줄링이 제거됩니다.

CUDA 스트림으로 CUDA 그래프 캡처하기

그래프를 캡처하려면, 스트림에 대해 `cudaStreamBeginCapture()`를 호출하고(), 모든 메모리 전송(`cudaMemcpyAsync()`), 커널 실행, 이벤트(`cudaEventRecord()`), 콜백(`cudaLaunchHostFunc()`)을 큐에 넣은 후 `cudaStreamEndCapture()`를 호출하여 CUDA 그래프 정의를 생성합니다(`cudaGraph_t`).

그런 다음 CUDA 드라이버는 CUDA 그래프를 `cudaGraphLaunch()` 각 반복마다 실행할 수 있습니다. CUDA 드라이버는 전체 종속성 그래프를 사전에 알고 있으므로, 미리 빌드된 스트림 시퀀스를 GPU에서 직접 재생합니다. 이로 인해 실행 오버헤드가 최소화됩니다.

`cudaGraphExecUpdate` 다음 섹션에서 설명하는 '포착 후 변경(capture-after-change)'은 반복자 간 크기, 차원 또는 포인터가 변경되는 시나리오에서 포착된 그래프에 대한 제한적 변경을 허용합니다. 입력 크기가 변동하는 경우 유용하며, 새로운 입력 크기마다 완전히 새로운 그래프를 재포착하는 대신 그래프의 노드 매개변수만 업데이트하면 됩니다.

파이프라인 커널 중 일부만 반복적이라 해도 CUDA 그래프는 해당 부분만 캡처할 수 있습니다. 예를 들어 호스트 → 디바이스 복사 → 두 커널 실행 → 디바이스 → 호스트 복사 순서로 항상 수행한다면, 해당 서브그래프만 캡처하여 단일 함수 호출로 재생할 수 있습니다.

CUDA 그래프를 재생하려면 스트림 핸들을 제공하면 GPU가 추가 CPU 명령어 없이 작업 시퀀스를 실행합니다. 이는 커널 간 동시성과 직접 연결됩니다. CUDA 그래프는 비동기 복사, 세분화된 이벤트 배리어, 커널을 혼합하여 복잡한 중첩 동작을 유지하면서 CPU를 병목 지점으로 완전히 제거하기 때문입니다.

일반적으로 정확성을 보장하기 위해 리플레이 드라이 런을 수행합니다. 실행 가능한 그래프(`cudaGraphExec_t`)를 생성하여 그래프를 인스턴스화한 후, 단일 graph-replay 호출로 실행합니다. 캡처된 그래프를 실행하면 런타임이 GPU에서 모든 작업을 올바른 순서로 실행합니다.

CUDA 그래프 사용법을 보여주기 위해 간단한 커널 시퀀스를 고려해 보겠습니다. 다음은 C++과 PyTorch 모두에서 CUDA 그래프를 캡처하고 실행하는 코드 스니펫입니다:

```
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
cudaGraph_t graph;
cudaGraphExec_t instance;
```

```
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

// Enqueue operations on 'stream' as usual
kernelA<<<grid, block, 0, stream>>>(d_X);
kernelB<<<grid, block, 0, stream>>>(d_Y);
kernelC<<<grid, block, 0, stream>>>(d_Z);

cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&instance, graph, nullptr, nullptr, 0);

// Now 'instance' can be launched in a loop
for (int iter = 0; iter < 100; ++iter) {
    cudaGraphLaunch(instance, stream);
    // No per-kernel sync needed; graph ensures dependencies
}
cudaStreamSynchronize(stream);

// (Destroy graph and instance when done)
cudaGraphExecDestroy(instance);
cudaGraphDestroy(graph);
```

이 의사 코드에서는 CUDA 스트림에서 캡처를 시작하고, 해당 스트림에서 세 개의 커널(A, B, C)을 순차적으로 실행한 후 캡처를 종료하여 그래프를 획득합니다. 그런 다음 그래프를 인스턴스화합니다.

이제 `` `cudaGraphLaunch(instance, stream)` `` 를 원하는 만큼 호출하여 전체 시퀀스 A → B → C를 재현할 수 있습니다. 각 반복마다 세 번의 별도 실행 대신 단일 실행 비용만 지불하면 됩니다. 또한 GPU는 커널 A, B, C를 기록된 순서대로, 즉 이 경우 연속적으로 실행합니다. 모든 반복이 완료되었는지 확인하기 위해 루프 후에 동기화합니다.

앞서 언급했듯이, 와 같은 고급 AI 프레임워크는 CUDA 그래프를 지원합니다. PyTorch는 Python 개발자에게 CUDA C++에 대한 깊은 지식 없이도 네이티브에 가까운 CUDA 성능을 제공합니다. 여기서는 PyTorch의 `torch.cuda.Graph` 컨텍스트 매니저를 사용하여 작업을 캡처하고 재현하는 방법을 보여줍니다:

```
import torch, time

X = torch.randn(1<<20, device='cuda')

# Define operations for reference
def opA(x): return x * 1.1
def opB(x): return x + 2.0
def opC(x): return x.sqrt()

# Persistent buffers for pointer stability
static_x = torch.empty_like(X)
static_y = torch.empty_like(X)
static_z = torch.empty_like(X)
```

```
    static_w = torch.empty_like(X)

    # Warm up once to initialize CUDA kernels and caches
    _ = opC(opB(opA(X)))
    torch.cuda.synchronize()

    # Seed the static input before capture
    static_x.copy_(X)

    # Capture the graph
    g = torch.cuda.CUDAGraph()
    stream = torch.cuda.Stream()
    torch.cuda.synchronize()
    with torch.cuda.graph(g, stream=stream):
        # Record A then B then C using out parameters to avoid allocations
        torch.mul(static_x, 1.1, out=static_y)
        torch.add(static_y, 2.0, out=static_z)
        torch.sqrt(static_z, out=static_w)

    # Replay the captured graph 100 times
    for i in range(100):
        # If inputs change, copy new values into static_x before replay
        # static_x.copy_(new_X)
        g.replay()
```

실제 운영 환경에서는 캡처된 커널이 고정된 메모리 주소를 참조하도록 영구적인 입력/출력 버퍼를 할당해야 합니다. 예를 들어, 캡처 전에 `static_y = torch.empty_like(X)`를 생성한 후 그래프 내부에서 `static_y.copy_(opA(X))`를 작성합니다. 이렇게 하면 캡처 중 할당을 피하고 CUDA 그래프 포인터의 안정성 규칙을 충족시킵니다. PyTorch CUDA 그래프는 재실행 시 캡처된 텐서에 동일한 메모리 주소를 사용해야 합니다.

이 PyTorch 예제에서는 `opA`, `opB`, `opC` 연산을 정의합니다. 실제 적용 시 이는 신경망 레이어나 기타 GPU 연산이 될 수 있습니다. 모든 커널, 메모리 할당, 라이브러리 컨텍스트(예: cuBLAS/cuDNN)가 사전에 초기화되도록 1회 워밍업 패스(`opC(opB(opA(X)))`)를 실행합니다. CUDA 그래프 캡처는 이러한 지연 설정 단계를 기록하지 않기 때문에 이 작업이 필요합니다.

워밍업 패스를 생략하면 그래프 캡처가 실패하거나 지연 초기화가 발생할 때 예상치 못한 정체가 발생할 수 있습니다.

먼저 GPU를 워밍업하기 위해 시퀀스를 한 번 실행하여 모든 커널과 라이브러리를 초기화합니다. 그런 다음 전방(`A`), 변환(`B`), 역방향(`C`) 연산을 새 CUDA 스트림에서 `with torch.cuda.graph(g, stream=stream):` Python 컨텍스트 매니저 블록으로 감싸 단일 `torch.cuda.CUDAGraph` 로 캡처합니다. 캡처 후 `g.replay()` 를 100회 호출하면 반복마다 하나의 호스트 호출로 전체 A → B → C 파이프라인이 실행됩니다. 결과는 표 12-1에 요약되어 있습니다.

표 12-1. CUDA 그래프가 반복 오버헤드에미치는 영향

메트릭	CUDA 그래프 이전	CUDA 그래프 적용 후
100회 반복자당 CPU 런치 호출 횟수	300개의 별도 커널 런치	100회 그래프 재생 (반복자당 1회)
호스트 동기화 호출	`cudaDeviceSynchronize` 호출 300회	0
커널 간 평균 GPU 유휴 시간	반복당 ~3 마이크로초 간격	0 마이크로초 (연속적인 백투백 실행)
종단 간 반복 지연 시간	~1.00 ms	~0.75 밀리초 (25% 더 빠름)

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

여기서 CUDA 그래프가 반복당 CPU 스케줄링과 호스트-장치 핸드셰이크를 제거함을 확인할 수 있습니다. 이는 각 반복에 대한 GPU 작업이 세 개의 별도 커널 실행 대신 단일 커널 런치(`g.replay()`) 호출로 배치되기 때문입니다. 결과적으로 CPU가 경량 재생 명령만 발행하고 GPU와 완전히 비동기 상태를 유지하므로 반복자 실행 속도가 25% 빨라집니다.

CUDA 그래프를 사용할 때 커버티드 그래프( )에는 몇 가지 흔한 함정이 있으며, 이를 적절히 처리하는 것이 중요합니다. 예를 들어, 작업 부하 크기가 변경되면 캡처된 그래프가 유효하지 않을 수 있습니다. 이 경우 재캡처하거나 커버티드 그래프 재구성(`cudaGraphExecUpdate`)을 호출해야 하며, 이는 다음 섹션에서 다룹니다.

메모리 할당이나 호스트-장치 동기화 프리미티브와 같은 특정 CUDA API 호출은 일반적으로 그래프 캡처에 포함해서는 안 됩니다. 최신 CUDA 그래프 버전은 캡처된 그래프 내에서 제한된 메모리 관리 작업을 지원하지만, 모든 메모리 할당은 캡처 전에 수행하는 것이 좋습니다. 또한 그래프에서 사용되는 데이터가 동일한 메모리 주소에 유지되도록 해야 합니다.

그래프 실행 중 메모리가 동일한 메모리 주소에 유지되어야 한다는 요구사항은 PyTorch와 같은 프레임워크가 CUDA 그래프와 함께 정적 메모리 풀을 사용하는 주요 이유입니다. 예를 들어, PyTorch는 포인터 안정성 CUDA 그래프 캡처를 위한 전용 메모리 풀을 생성하는 ` `torch.cuda.graph_pool_handle()` ` API를 제공합니다. 별도의 할당기 풀을 사용하면 캡처와 재생 간 텐서 주소가 고정되도록 보장합니다. 이는 포인터 안정성 요구사항을 충족시킵니다. 반복 사이에는 정적 텐서로 복사하여 입력값을 업데이트하십시오. 매 반복마다 텐서를 재할당하지 마십시오.

또한 CUDA 그래프 캡처 내부에 호스트 측 콜백이나 지원되지 않는 작업을 포함하지 않도록 해야 합니다. 여기에는 `print()`, 난수 생성기(RNG) 호출, 중첩된 캡처, 새로운 메모리 할당 등이 포함됩니다. 그래프는 순수하고 결정론적인 GPU 작업 시퀀스를 기록해야 하기 때문입니다.

또한 캡처에 사용되는 모든 텐서는 고정된 형상과 고정된 주소에 미리 할당되어 있어야 합니다. 캡처 중 크기 조정이나 `cudaMalloc` 호출은 그래프를 손상시킵니다.

동적 그래프 업데이트

CUDA 그래프를 기록한 후에는 일부 런치 매개변수가 변경되었다고 해서 해당 그래프를 버릴 필요가 없습니다. 재캡처 대신 그래프 업데이트 API를 호출하여 기존 그래프 내에서 그리드/블록 차원, 포인터 주소 또는 커널 매개변수를 직접 업데이트할 수 있습니다. 그래프 업데이트 API에는 `graph_update_grid_dimensions`(`cudaGraphExecUpdate`) 및 하위 수준의 `graph_update_block_dimensions`(`cudaGraphExecKernelNodeSetParams`)가 포함됩니다.

`cudaGraphExecUpdate` 동일한 형상의 새 커널 노드로 교체할 수 있습니다. 예를 들어, 동일한 형상이라면 다른 융합 커널 구현체로 교체할 수 있습니다. CUDA 런타임이 수정 사항을 검증하고 수정된 그래프를 즉시 재실행할 수 있게 합니다. 이를 통해 전체 캡처 비용을 피할 수 있습니다.

현재 시점에서 노드를 임의로 추가하거나 제거할 수 없습니다. 업데이트가 기존 그래프가 처리할 수 있는 범위를 벗어날 경우 런타임은 오류를 반환합니다. 이 경우 새 그래프를 캡처해야 합니다.

예를 들어, 최대 배치 크기를 사용한 앞서 소개한 3개 커널 A → B → C 그래프를 고려해 보십시오. 각 추론 루프에서 커널 B의 런치 차원을 현재 배치에 맞게 업데이트한 후 동일한 그래프를 재실행하기만 하면 됩니다. 이는 전체 파이프라인은 고정되어 있지만 일부 매개변수가 변동될 수 있는 반정적 워크로드의 오버헤드를 줄여줍니다.

실제 작업 흐름은 일반적으로 세 단계로 구성됩니다. 첫째, 예상 최대 크기(예: 최대 배치 크기)를 사용해 템플릿 그래프를 캡처합니다. 예를 들어 최대 배치 크기 128로 그래프를 캡처할 수 있습니다. 이후 배치 64로 요청이 들어오면 `cudaGraphExecUpdate`를 호출해 런치 매개변수를 64로 조정하고, 필요 시 메모리 포인터를 더 작은 버퍼로 업데이트합니다.

`cudaGraphExecUpdate` 를 사용하면 커널 매개변수, 그리드/블록 크기 또는 메모리 주소가 변경될 때 그래프를 재구축할 필요가 없습니다. 또한 소요 시간은 몇 마이크로초에 불과하므로 CUDA 그래프 재생의 100µs 미만 빠른 런치 오버헤드를 유지할 수 있습니다. 더불어 런타임에 주요 매개변수를 조정할 수 있는 유연

성도 확보됩니다. 호환되지 않는 변경 사항은 오류 상태를 반환하며 재캡처가 필요하다는 점에 유의하십시오.

예를 들어 커널 수를 달리 지정하기 위해 그래프 구조를 변경해야 하는 경우, 재캡처 후 업데이트 워크플로로 전환할 수 있습니다. 이 경우 코드의 한 이터레이션을 `cudaStreamBeginCapture` 로 감싸 `cudaStreamEndCapture` 새로운 그래프를 구축합니다. 이후 가벼운 `cudaGraphExecUpdate` 를 사용하여 후속 실행에서 사소한 조정을 수행할 수 있습니다.

실질적으로 동적 그래프 업데이트를 통해 GPU 상에서 완전히 "매개변수화"되거나 조건부 실행 경로를 생성할 수 있습니다. GPU 작업의 고주파 루프가 있지만 배치 크기처럼 변경되는 매개변수가 몇 개뿐인 경우, 한 번 캡처하고 빠르게 업데이트하여 최소한의 CPU 오버헤드와 사용 사례에 필요한 적응성을 모두 누릴 수 있습니다.

장치에서 시작한 CUDA 그래프 실행

을 실행하고 CPU에서 캡처한 파이프라인을 낮은 오버헤드로 적응시키는 방법을 이해했으니, 다음 단계는 실행 결정에서 CPU를 완전히 제거하는 것입니다. 장치 시작 CUDA 그래프 실행을 사용하면 실행 중인 GPU 커널이 호스트를 완전히 우회하여 장치에서 직접 사전 기록된 그래프를 트리거할 수 있습니다.

장치 측 런치를 활성화하려면 먼저 호스트에서 평소처럼 그래프를 캡처합니다. 그런 다음 ` `cudaGraphInstantiate` `로 인스턴스화하고 ` `cudaGraphInstantiateFlagDeviceLaunch` `를 전달합니다. 인스턴스화 후 장치 측 런치 전에 호스트 스트림에 ` `cudaGraphUpload` `로 실행 파일을 업로드합니다.

그런 다음 `cudaGraphUpload` 를 사용하여 그래프를 GPU 메모리에 업로드합니다. 이는 GPU가 그래프를 실행하기 전에 반드시 수행되어야 합니다. (그래프 업로드 없이 장치 실행을 시도하면 오류가 발생합니다.)

실제 적용 시, 이는 지속적 또는 동적 커널 내에 "그래프 실행" 노드를 내장하거나 장치 측 그래프 API를 호출하는 것과 같습니다. 때가 되면 GPU는 그림 12-5와 같이 자체 소유 스트림에서 전체 그래프를 실행합니다.

장치에서 시작한 그래프 실행은 데이터 주도형 워크플로를 완전히 GPU 내에서 유지합니다. 결정 조건 계산은 CPU가 아닌 커널이 담당합니다. 따라서 커널은 다음 그래프를 직접 생성하여 CPU 왕복을 제거하고 지연 시간을 더욱 줄일 수 있습니다.

그래프가 이미 GPU에 상주하고 CPU-GPU 핸드셰이크가 필요 없기 때문에, 장치 시작 런치는 호스트 스케줄링을 크리티컬 패스에서 제거하고 호스트 바운드 루프의 종단 간 지연 시간을 줄일 수 있습니다. 실제로 장치 시작 CUDA 그래프 런치는 동등한 호스트 측 그래프 런치에 비해 약 2배 낮은 런치 지연 시간을 보였습니다. 또한 그래프의 크기나 복잡성이 증가해도 오버헤드는 일정하게 유지됩니다.

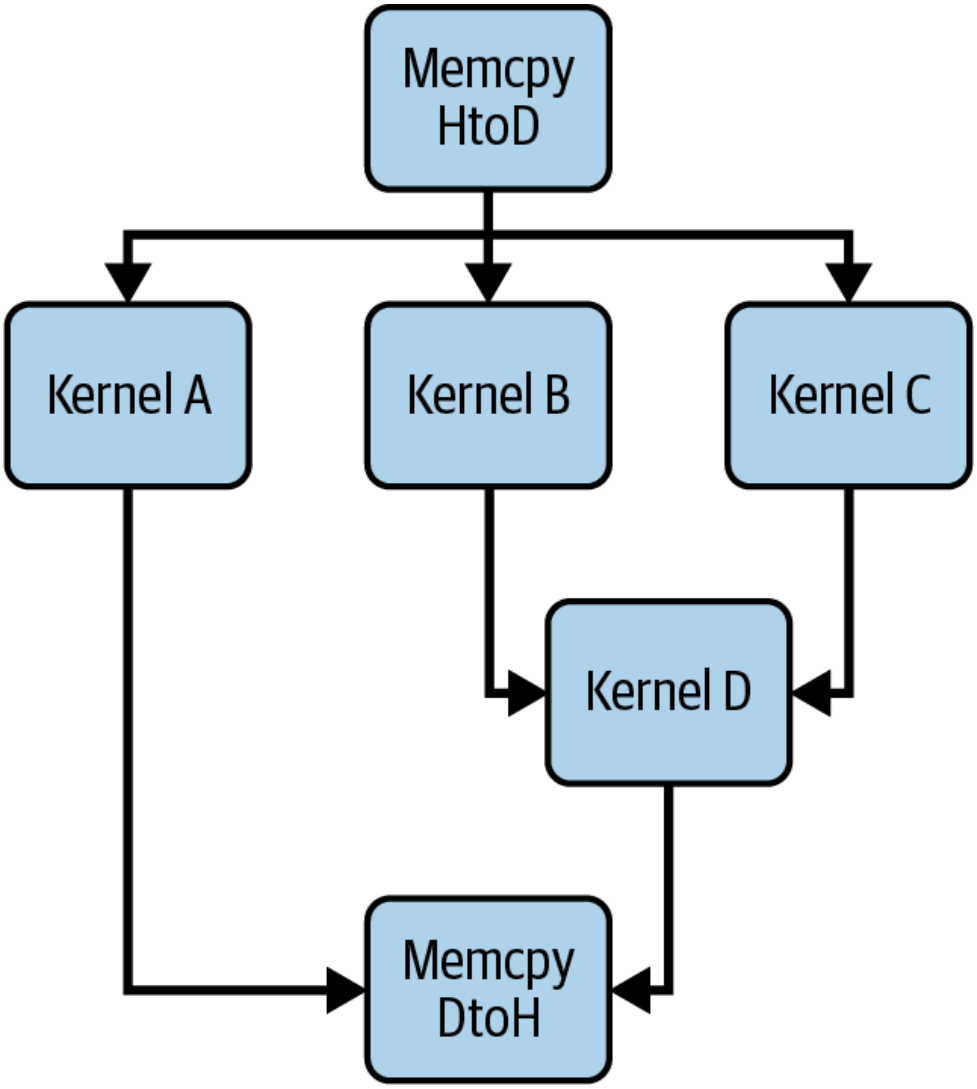

그림 12-5. CUDA 그래프에 의해 실행되는 연산순서 (커널 및 데이터 전송 노드)와 의존성(에지)

장치 런치 그래프 지연 시간은 그래프 내 노드 수나 병렬 분기 수에 영향을 받지 않습니다. 이는 CPU 스케줄링 오버헤드로 인해 그래프가 커질수록 지연 시간이 증가하는 호스트 런치 그래프와 대조적입니다.

또한 장치 런치는 그래프 너비에 따라 잘 확장됩니다. 더 많은 병렬 노드가 추가되면 호스트 측 런치는 추가 동기화 비용으로 인해 성능이 저하되지만, 장치 런치 지연 시간은 거의 평평하게 유지됩니다.

디바이스 런치된 그래프 디버깅은 까다로울 수 있으나, Nsight Systems 같은 도구를 사용하면 GPU 타임라인에서 자식 그래프를 별도의 작업 스트림으로 확인할 수 있습니다. 부모 커널에서 `cudaGraphLaunch` 호출 전후에 NVTX 마커를 사용하여 디바이스 런치 위치를 표시하는 것이 권장됩니다. 이는 그래프가 부모 스레드와 관련하여 예상대로 실행되는지 확인하는 데 도움이 됩니다.

장치 코드 내부에서는 간단한 API인 `` `cudaGraphLaunch(graphExec, stream)` ``로 그래프를 런칭합니다. 런타임은 다음 지원 런칭 모드를 구분하기 위해 특별히 예약된 `` `cudaStream_t` `` 값을 사용합니다: "fire-and-forget" (`cudaStreamGraphFireAndForget`), "tail" (`cudaStreamGraphTailLaunch`), "sibling" (`cudaStreamGraphFireAndForgetAsSibling`). 이러한 모드는 호스트 개입 없이도 CUDA 스트림 내에서 올바른 순서를 자동으로 강제 적용합니다.

fire-and-forget 런치에서는 자식 그래프가 런칭하는 부모 커널과 동시에 즉시 실행을 시작합니다. 부모 커널은 자식이 완료될 때까지 기다리지 않으며, 이는 독립적인 작업 스레드를 런칭하는 것과 유사합니다. fire-and-forget 런치는 커널 내부에서 비동기 작업을 생성하는 데 유용합니다.

그래프는 실행 과정에서 최대 120개의 파이어 앤 포겟(fire-and-forget) 그래프를 가질 수 있습니다.

반면, 장치 시작 그래프 테일 런치는 런칭 커널이 동기화 지점에 도달하거나 완료될 때까지 그래프 실행을 연기합니다. 이는 그림 12-6과 같이 현재 커널 이후에 실행될 그래프를 효과적으로 대기열에 추가하여 연속 작업으로 처리합니다.

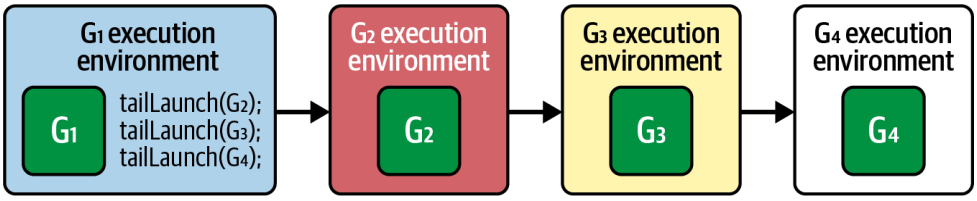

그림 12-6. 특정 그래프에 의해 큐에 등록된테일 런치는 등록된 순서대로 하나씩 실행됨

꼬리 실행은 GPU 상주 작업 스케줄러 구현 시 특히 강력합니다. 지속적 "스케줄러" 커널은 그래프를 꼬리 실행한 후 해당 그래프가 완료되면 스스로를 재실행할 수 있습니다. 이 기법은 호스트 재호출 없이 GPU에서 효과적으로 루프를 생성합니다. 재실행을 위해 커널은 `cudaGetCurrentGraphExec()`를 호출하여 실행 중인 자체 그래프의 핸들을 획득합니다. 이후 `cudaGraphLaunch(..., cudaStreamGraphTailLaunch)`를 사용하여 그래프를 실행함으로써 자신을 다시 대기열에 추가합니다.

또한 테일 그래프는 추가적인 테일 런치를 수행할 수 있습니다. 이 경우 새로운 테일 런치는 그림 12-7과 같이 이전 그래프의 테일 런치 전에 실행됩니다.

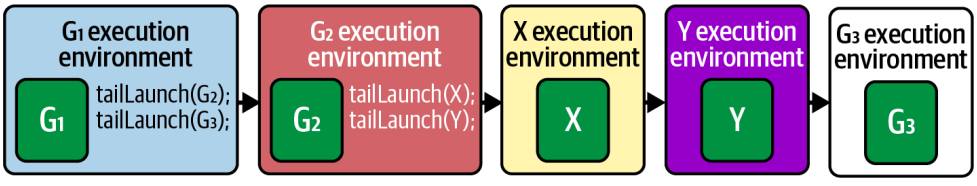

그림 12-7. 여러 그래프에서 대기열에 추가된테일 런치

CUDA 그래프에는 최대 255개의 대기 중인 테일 런치가 큐에 등록될 수 있습니다. 그러나 자체 테일 런치(예: 그래프가 재실행을 위해 자체를 큐에 등록하는 경우)의 경우, 한 번에 하나의 대기 중인 자체 테일 런치만 가질 수 있습니다.

동급자 실행(sibling launch)은 발사 후 방치(fire-and-forget)의 변형으로, 실행된 그래프가 부모 그래프의 자식이 아닌 동급자로 실행됩니다. 또한 동급자 실행은 부모의 스트림 환경에서 실행됩니다. 이는 그림 12-8과 같이 즉시 독립적으로 실행되지만 부모 그래프의 테일 실행을 지연시키지 않음을 의미합니다.

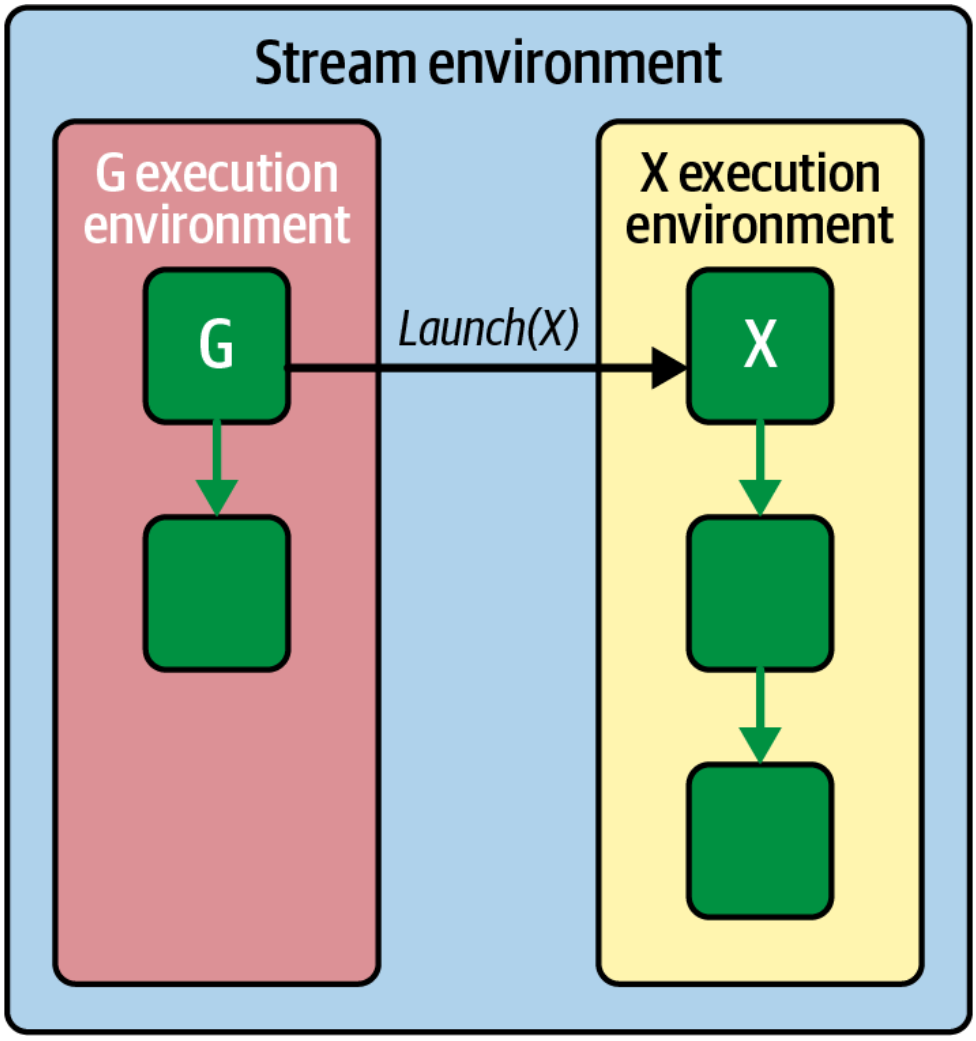

그림 12-8. 부모 스트림 환경에서 실행되는형제 그래프

이 모드에서는 `cudaGraphLaunch(graphExec, cudaStreamGraphFireAndForgetAsSibling)`를 사용하여 "동생 모드"로 런칭할 수 있습니다. 이는 그래프를 현재 그래프 실행 환경의 동생으로 제출합니다.

장치에서 시작한 CUDA 그래프를 사용할 때는 종속성을 신중하게 관리해야 합니다. 예를 들어, 실행된 커널이 그래프의 결과를 소비해야 하는 경우, 부모 커널이 그래프 작업이 완료될 때까지 일시 중지되므로 꼬리 실행이 적합합니다. 반면 실행된 그래프가 부수적인 작업에 가깝다면, 발사 후 잊어버리기 모드를 사용하면 부모 커널이 기다리지 않고 진행할 수 있습니다.

실제 적용 시, 장치 시작 CUDA 그래프는 새로운 패턴을 가능하게 합니다. 예를 들어, 데이터 내용에 따라 서로 다른 압축 알고리즘 중 하나를 선택해야 하는 GPU 압축 파이프라인을 생각해 보십시오. 커널을 종료하고 CPU에 선택된 압축 커널을 실행하도록 지시하는 대신, GPU 커널은 "LZ 압축"이나 "허프만 압축"과 같은 사전 기록된 그래프를 직접 실행할 수 있습니다.

이 압축 예시에서 GPU는 CPU의 결정을 기다리며 유휴 상태가 되지 않습니다. 지속적 커널 내에서 LLM 추론을 위한 스케줄링에 원자적 큐/카운터와 장치 시작형, 테일 런칭 CUDA 그래프를 결합한 또 다른 유용한 패턴을 살펴보겠습니다.

원자적 큐와 디바이스 시작 CUDA 그래프를 활용한 커널 내 지속적 스케줄링

앞서 소개한 원자 카운터 기반 작업 대기열( )을 장치 시작형 그래프 테일 런치와 결합할 수 있습니다. LLM 추론 루프 사용 사례를 고려해 보겠습니다. 이 사례에서는 트랜스포머 블록의 전방 전달(어텐션 + 피드포워드)을 포함하는 그래프를 캡처하여 디코딩을 수행하는 CUDA 그래프를 사용합니다.

경량 지속적 스케줄러 커널은 `atomicAdd(&queueHead,1)` 를 사용하여 다음 작업 항목을 확보할 수 있습니다. 그런 다음 미리 캡처된 디코딩 CUDA 그래프를 테일 런칭하여 출력을 계산하고 즉시 큐의 다음 항목을 위해 루프를 되돌립니다.

각 CUDA 그래프가 완료되면 커널 내 스케줄러 루프는 `atomicAdd(&queueHead,1)` `를 사용하여 다음 인덱스를 확보하고 다른 디코딩 그래프를 테일 런칭합니다. 이는 CPU에 접근하지 않고 작업 결정 및 실행을 모두 수행하는 완전한 GPU 상주 스케줄러를 효과적으로 생성합니다.

이러한 테일 런치를 연결함으로써 각 토큰은 사실상 CPU 오버헤드 없이 장치에서 처음부터 끝까지 처리됩니다. 또한 CPU가 크리티컬 패스에 재진입하지 않으므로 SM이 완전히 활용되고, 토큰당 지연 시간이 감소하며, 다양한 시퀀스 길이와 배치 크기에 실시간으로 적응할 수 있습니다. 이를 위해 그래프 매개변수를 업데이트하거나 사전 기록된 그래프 간에 전환하기만 하면 됩니다.

조건부 그래프 노드

기존 CUDA 그래프에서는 모든 노드와 그 종속성이 캡처 시점에 고정되어 의사 결정 로직을 호스트로 되돌려야 했습니다. 조건부 그래프 노드는 노드에 연결된 작은 "조건 핸들"을 기반으로 분기 결정을 GPU 자체로 연기함으로써 이러한 경직성을 깨뜨립니다.

그래프가 실행되는 동안 GPU는 해당 핸들을 평가하고 CPU로 제어권을 반환하지 않은 채 본체 하위 그래프 중 하나를 선택적으로 실행(또는 반복)합니다. 구체적으로 조건부 그래프 노드를 사용하면 제어 흐름(IF, IF/ELSE, WHILE, SWITCH)을 CUDA 그래프에 직접 내장하여 GPU 장치에서 실행할 수 있습니다. 조건부 그래프 노드는 호스트 왕복 통신을 제거하여 최신 GPU에서 상당한 성능 향상을 제공합니다.

본질적으로 조건부 그래프 노드는 장치 커널에서 계산된 값에 따라 그래프 실행을 제어할 수 있게 해주며, 이 모든 과정이 CPU 개입 없이 이루어집니다. 이 기능을 통해 복잡한 분기 워크플로를 단일 반복 가능한 그래프 런치로 구현할 수 있습니다. CUDA 그래프는 그림 12-9에 표시된 것처럼 여러 유형의 조건부 노드를 지원합니다:

IF

조건이 0이 아닐 때 단일 바디 그래프를 정확히 한 번 실행합니다.

IF/ELSE

두 개의 본체 그래프를 지정함으로써, 조건이 참일 때는 하나가 선택되고, 거짓일 때는 다른 하나가 선택됩니다.

WHILE

조건이 0이 아닌 동안 본체 그래프를 반복적으로 실행하며, 각 반복자 후 다시 조건을 확인합니다.

SWITCH

N개의 본체 그래프를 보유하며 조건이 *i일* 때 i번째 그래프를 실행합니다. 조건이 *N* 이상이면 실행 자체를 건너뜁니다.

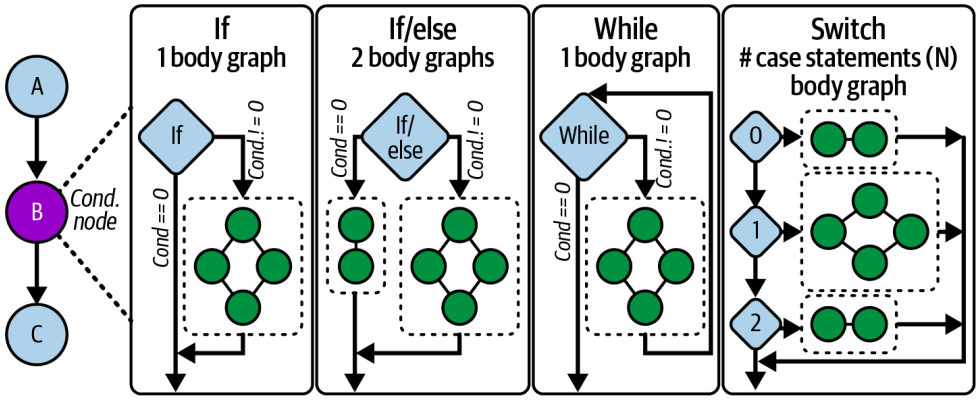

그림 12-9. 조건부 그래프 노드유형

다음은 IF 조건부 노드를 생성하고 채우는 방법을 보여주는 예시입니다. IF 노드를 제어하는 플래그를 작성하기 위해 `cudaGraphSetConditional`를 사용하는 점에 유의하십시오. 이 경우 조건은 합계가 주어진 임계값보다 큰지 확인합니다. 이렇게 하면 데이터가 주어진 기준(`flag = 1u`)을 충족하면 다음 서브그래프를 실행합니다. 반대로 조건이 충족되지 않으면 조건부 노드는 서브그래프를 실행하지 않습니다:

```
#include <cuda_runtime.h>
#include <cstdio>

// Device kernel that computes and sets the condition handle
__global__ void setHandle(cudaGraphConditionalHandle handle,
                          int *data, size_t N) {
    // Example threshold
    constexpr int threshold   = 123456;

    // Test whether the sum of data exceeds a threshold
    //     using a custom reduce_sum() function
    //     (recommended to implement this with
    //     CUB's DeviceReduce::Sum routine.)
    unsigned int flag =
        (reduce_sum(data, N) > threshold) ? 1u : 0u;

    cudaGraphSetConditional(handle, flag);
}
```

```
// A simple body kernel that runs only if flag != 0
__global__ void bodyKernel() {
    printf("Conditional body executed on GPU!\n");
}

int main() {
    cudaStream_t stream;
    cudaStreamCreateWithFlags(&stream,
        cudaStreamNonBlocking);

    // 1) Create the graph
    cudaGraph_t graph;
    cudaGraphCreate(&graph, 0);

    // 2) Create a condition handle associated with graph
    cudaGraphConditionalHandle condHandle;
    cudaGraphConditionalHandleCreate(&condHandle, graph);

    // 3) Add the upstream kernel node to set the handle
    cudaGraphNode_t setNode;
    cudaKernelNodeParams setParams = {};
    setParams.func          = (void*)setHandle;
    setParams.gridDim       = dim3(1);
    setParams.blockDim      = dim3(32);

    // 4) Allocate input data
    constexpr size_t N              = 1 << 20;
    int*             d_data         = nullptr;
    cudaMalloc(&d_data, N * sizeof(int));

    void* setArgs[] = { &condHandle, &d_data, &N };
    setParams.kernelParams  = setArgs;
    cudaGraphAddKernelNode(&setNode, graph, nullptr, 0,
        &setParams);

    // 5) Add the IF conditional node
    cudaGraphNode_t condNode;
    cudaConditionalNodeParams ifParams = {};
    ifParams.handle = condHandle;
    ifParams.type   = cudaGraphCondTypeIf;
    // One-node body graph, in this case
    ifParams.size   = 1;
    cudaGraphAddConditionalNode(&condNode,
                                graph,
                                &setNode,
                                1,
                                &ifParams);

    // 6) Populate the body graph: one kernel that prints a message
    cudaGraph_t bodyGraph = ifParams.phGraphOut[0];
    cudaGraphNode_t bodyNode;
    cudaKernelNodeParams bodyParams = {};
```

```
    bodyParams.func          = (void*)bodyKernel;
    bodyParams.gridDim       = dim3(1);
    bodyParams.blockDim      = dim3(32);
    cudaGraphAddKernelNode(&bodyNode, bodyGraph, nullptr,
        0, &bodyParams);

    // 7) Instantiate, upload, and launch the graph on the device
    cudaGraphExec_t graphExec;
    cudaGraphInstantiate(&graphExec, graph, nullptr, nullptr,
    cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(graphExec, stream);
    cudaGraphLaunch(graphExec, stream);


    // 8) Wait for completion and clean up
    cudaStreamSynchronize(stream);
    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
    cudaStreamDestroy(stream);

    cudaFree(d_data);

    return 0;
}
```

여기서 `cudaGraphCreate` 로 새 CUDA 그래프를 생성합니다. 이 그래프는 이후 모든 노드를 포함할 것입니다. 그런 다음 `cudaGraphConditionalHandleCreate` 를 사용하여 조건 핸들을 생성합니다. (이는 장치에서 설정할 수 있는 작은 정수 값을 그래프에 연결합니다.)

그런 다음 상류 커널인 `setHandle`를 추가합니다. 이 커널은 경합 상태를 피하기 위해 하나의 스레드에서 실행됩니다. 이후 `cudaGraphSetConditional`를 호출하여 IF 노드를 제어하는 플래그를 기록합니다.

`cudaGraphAddConditionalNode`로 IF 조건부 노드를 추가합니다. 이때 `cudaGraphCondTypeIf` 와 `size=1` 을 지정합니다. 이는 지원할 조건부 분기 또는 반복 횟수를 정의합니다.

여기서 조건부 플래그가 0이 아닌 값을 반환할 경우 실행될 빈 서브그래프(`body`) 하나를 할당합니다. 본체 그래프는 `ifParams.phGraphOut[0]` 에서 가져온 후 `bodyKernel` 를 추가하여 채웁니다. 이 노드는 실행 시 단순히 메시지를 출력합니다.

그래프 생성 후 `cudaGraphInstantiate` 를 호출하여 실행 가능한 그래프 객체를 생성합니다. 디바이스 코드에서 그래프를 실행하려면 디바이스 측 실행 전에 `cudaGraphInstantiateFlagDeviceLaunch` 플래그로 인스턴스화하고 `cudaGraphUpload` 로 업로드해야 합니다.

CUDA 스트림에서 `cudaGraphLaunch` 로 실행하면 상류 set-handle 커널이 실행되고, 조건부 검사가 수행된 후 플래그가 설정된 경우 본체 커널이 실행됩니다. 이 모든 과정은 그림 12-10과 같이 GPU에서 직접 수행됩니다.

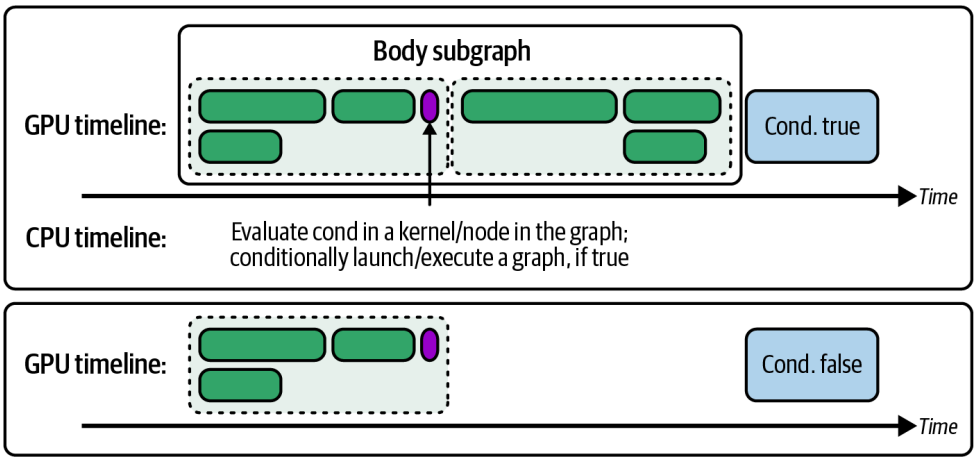

그림 12-10. 조건 충족 시추가 처리(`body` 서브그래프)

그런 다음 `cudaStreamSynchronize` 로 스트림을 동기화하여 완료를 기다립니다. 마지막으로 인스턴스화된 그래프, 그래프 자체, 스트림을 파괴하여 정리합니다.

경합 상태를 최소화하려면 조건을 항상 단일 스레드(예: `` `if (threadIdx.x == 0)` ``)에서 설정하는 것이 중요합니다. 또한 조건부 노드가 실행되기 전에 값이 가시화되도록 선행 커널이 메모리를 플러시하도록 해야 합니다.

조건부 노드는 중첩될 수도 있습니다. 예를 들어, WHILE 노드의 본문에 IF 노드를 포함시킬 수 있으며, 이는 그림 12-11에 표시되어 있습니다. 이를 통해 CPU 호프 없이도 다단계 결정 논리를 구현할 수 있습니다.

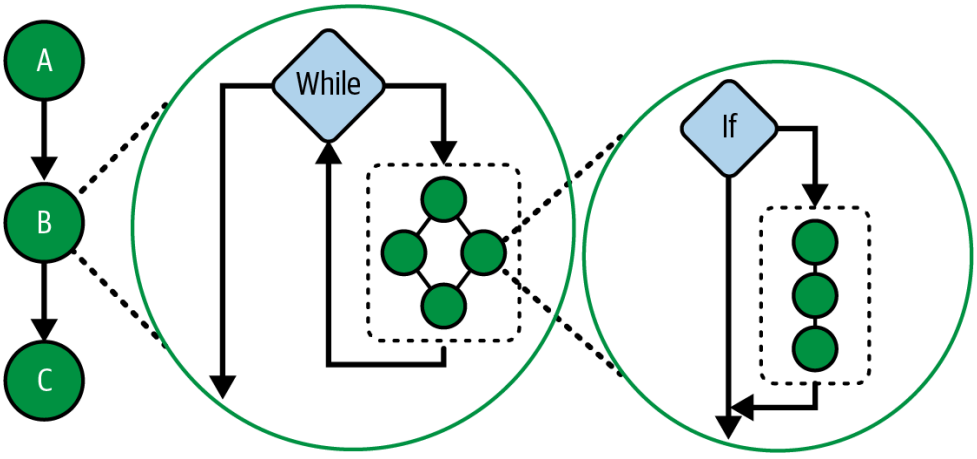

그림 12-11. 중첩된 조건부 그래프 노드

요약하면, 조건부 그래프 노드를 사용하여 GPU에서 결정을 유지하고 CPU 오버헤드를 줄이며 복잡한 제어 흐름을 CUDA 그래프에 직접 표현해야 합니다. 그래프 생성 비용은 여러 반복에 걸쳐 분산될 수 있으므로 동적 워크플로를 완전히 온디바이스로 표현하면 상당한 성능 향상을 얻을 수 있습니다.

현재 시점에서 PyTorch의 CUDA 그래프 API는 Python으로 조건부 CUDA 그래프 노드를 직접 생성하는 방법을 제공하지 않습니다. 프레임워크 및 도구에서의 조건부 그래프 실행 지원은 발전 중입니다. 현재 시점에서 PyTorch로 if/while/switch 노드를 구현하려면 사용자 정의 C++ 통합이 필요합니다.

동적 병렬 처리

이전에는 장치에서 시작된 CUDA 그래프 런치( )가 최소한의 CPU 개입으로 고정된 작업 시퀀스를 캡처하고 재현하는 방식을 살펴보았습니다. 그러나 이 모델은 전체 실행 흐름이 사전에 알려져 있어야 한다는 전제를 갖는데, 이는 항상 가능한 것은 아닙니다. 많은 워크로드는 입력 데이터, 중간 결과 또는 문제의 복잡성에 따라 런타임에 형태가 변경됩니다. 바로 여기에 동적 병렬 처리(DP)가 필요합니다.

DP는 GPU 커널이 CPU를 기다리지 않고 스스로 새로운 작업을 생성할 수 있는 능력을 부여합니다. CUDA 그래프가 전체 파이프라인을 사전에 파악해야 하는 반면, DP는 실행 중인 "부모" 커널이 자신의 출력을 검사하고 다음에 실행할 "자식" 커널의 수를 즉석에서 결정할 수 있게 합니다. 이는 계층적 축소, 적응형 메쉬 정밀화, 그래프 탐색과 같이 후속 작업 수가 데이터 처리 과정에서야 명확해지는 진정한 비정형 문제에 있어 판도를 바꾸는 기술입니다.

특정 입력에 대해 가끔 특수한 "플러그인" 모델 평가가 필요한 추론 파이프라인을 상상해 보십시오. CPU 주도 흐름에서는 커널 A를 실행하고, 그 결과를 호스트로 복사한 후, 커널 B를 실행할지 여부를 결정하고, 그 실행 명령을 내리게 됩니다. 이 과정에서 GPU는 왕복 시간 동안 유휴 상태로 남게 됩니다. 반면 DP에서는 커널 A가 자신의 출력을 검사하고, 조건이 충족되면 커널 B를 장치에서 직접 실행합니다. 전체 결정 및 디스패치 과정이 하나의 GPU 상주 워크로드 내에서 이루어지므로 유휴 간격이 사라지고 SM이 지속적으로 작동합니다.

LLM 환경에서 대부분의 토큰은 표준 트랜스포머 경로를 따르지만, 일부는 보조 어텐션 블록이 필요합니다. DP 지원 트랜스포머 커널은 런타임에 이러한 특수 토큰을 감지하고 해당 위치에만 추가 어텐션 커널을 테일 런칭합니다. 호스트 개입 없이 사이클 낭비 없이 처리됩니다. NVIDIA 라이브러리는 이미 적응형 알고리즘의 유사 패턴에 DP를 활용합니다: 데이터가 계산을 통과할 때 새로운 하위 작업이 동적으로 생성됩니다.

자질 타임라인에 `Kernel A → GPU idle gap → Kernel B` 와 같은 연속 패턴이 나타나면 DP가 적합하다는 것을 알 수 있습니다. 여기서 유휴 시간은 CPU가 다음 런치를 준비하는 시간에 해당합니다. 이 간격을 디바이스 측 런치로 대체하면 모든 SM을 지속적으로 활용하고 종속 단계 간 지연 시간을 대폭 줄일 수 있습니다.

물론 DP의 성능 향상은 공짜가 아닙니다. 각 자식 런치는 GPU 스케줄링 리소스를 사용하고 추가 스택 공간이 필요합니다. "스택 오버플로" 오류를 피하려면

`cudaDeviceSetLimit(cudaLimitStackSize, newSize)` 로 런타임 스택 크기를 늘려야 할 수 있습니다. CUDA는 기본 한도에 도달하면 경고합니다.

관련하여, CUDA는 대기 중인 자식 커널 실행 수에 대해 최대 제한을 두고 있습니다. 기본적으로 CUDA는 한 번에 2,048개의 미처리 장치 실행을 허용합니다. 그러나 이는 구성 가능합니다.

대형 루프를 통해 수천 개의 소형 커널을 실행하는 등 부모 커널이 2,048개 이상의 자식 커널을 생성해야 하는 경우, `` `cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, newLimit)` `` API를 사용하여 이 한도를 높일 수 있습니다. 그렇지 않으면 기본값 2,048을 초과하면 런타임 오류가 발생합니다. 실제로 대부분의 용도에서는 기본값으로도 충분합니다. 하지만 극단적인 경우에는 중요한 고려 사항입니다.

GPU에서 다수의 자식 커널을 생성할 때는 각 대기 중인 자식 커널 실행이 리소스를 예약하므로 장치 메모리 사용량을 반드시 모니터링하세요. 이는 GPU 하드웨어의 하드 리미트를 초과할 수 있습니다.

DP는 명령어 수준 오버헤드를 추가하므로, 지속적 커널, 스트림 또는 CUDA 그래프와 같은 정적 오케스트레이션이 CPU가 다음 단계를 결정하는 동안 GPU를 유휴 상태로 방치하는 경우에 가장 적합합니다. 즉, 워크로드가 진정한 고정 작업 순서인 경우 CUDA 그래프로 캡처하여 장치 측에서 재실행하는 것이 더 나은 경우가 많습니다.

작업이 데이터 자체에서 동적으로 생성되는 경우, DP를 사용하면 스케줄링과 실행을 모두 GPU에서 완전히 처리할 수 있습니다. 이는 중첩된, 데이터 의존적인, 또는 예측 불가능한 병렬 처리에서 더 나은 확장성과 낮은 종단 간 지연 시간을 제공합니다.

DP는 런치당 오버헤드가 발생하고 대기 중인 런치 슬롯을 소모하므로, 런타임에 데이터로부터 새로운 병렬성이 발생하고 사전 기록된 CUDA 그래프로 표현할 수 없을 때 사용하십시오. 반면, 제어 흐름이 표현 수준이고 반복되는 경우에는 CUDA 그래프가 비용을 분산하므로 장치 런치 CUDA 그래프를 선호하십시오.

간단한 부모-자식 워크플로의 두 가지 구현 방식을 비교해 보겠습니다. 다음에 소개하는 호스트 주도 버전은 부모 커널을 실행한 후 완료를 기다린 다음 CPU에서 두 개의 자식 커널을 발행합니다. 이 방식은 결정 간격 동안 GPU를 유휴 상태로 남겨둡니다:

```
// dp_host_launched.cu
#include <cuda_runtime.h>

__global__ void childKernel(float* data, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (idx < N) {
        data[idx] = data[idx] * data[idx];
    }
}

__global__ void parentKernel(float* data, int N) {
    // Parent does setup work. Here, CPU decides on child launches.
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        // maybe mark regions or compute flags here
    }
}

int main() {
    const int N = 1 << 20;
    float* d_data;
    cudaMalloc(&d_data, N * sizeof(float));
    // ... initialize d_data ...

    // 1) Launch parent and wait
    cudaStream_t s; cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);

    parentKernel<<<1,1,0,s>>>(d_data, N);

    cudaStreamSynchronize(s);

    // 2) CPU splits work in half and launches children
    int half = N / 2;
    childKernel<<<(half+255)/256,256>>>(d_data,      half);
    childKernel<<<(half+255)/256,256>>>(d_data+half, half);
    cudaStreamSynchronize(s);
    cudaStreamDestroy(s);

    cudaFree(d_data);
    return 0;
}
```

호스트 주도 버전에서는 GPU가 다중 작업(`parentKernel`)을 실행한 후 대기 상태가 되며, CPU가 각 다중 작업(`childKernel`)을 차례로 준비하고 실행합니다. 부모 프로세스 이후와 자식 프로세스 사이에 명시적인 ` `cudaDeviceSynchronize()` ` 호출이 있음을 유의하십시오. 이러한 호출은 그림 12-12에 표시된 대로 제거해야 할 대기 간격을 초래합니다.

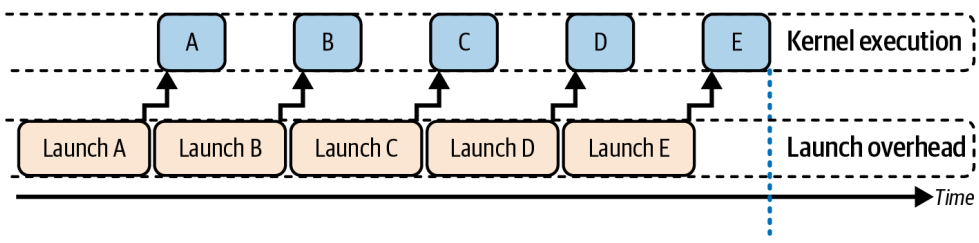

그림 12-12. 자식 커널 런치로 인한유휴 간격

반면, 장치 실행형 DP 버전에서는 부모 커널이 자식 커널을 장치에서 직접 생성합니다. 이 방식은 부모와 자식 커널 실행 간 호스트 동기화가 필요하지 않습니다. 이렇게 하면 부모의 자식 커널 실행이 암시적으로 자식들을 큐에 넣고 마지막에만 동기화합니다. 코드 예시는 다음과 같습니다:

```
// dp_device_launched.cu
// Dynamic parallelism requires relocatable device code enabled with -rdc=t

#include <cuda_runtime.h>

__global__ void childKernel(float* data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        data[idx] = data[idx] * data[idx];
    }
}

__global__ void parentKernel(float* data, int n) {
    // Launch children from a single thread to avoid duplicate launches.
    if (blockIdx.x == 0 && threadIdx.x == 0) {
        const int threadsPerBlock = 256;
        const int firstHalfCount  = n / 2;
        const int secondHalfCount = n - firstHalfCount;

        const int blocksFirst  = (firstHalfCount  + threadsPerBlock - 1)
                                 / threadsPerBlock;
        const int blocksSecond = (secondHalfCount + threadsPerBlock - 1)
                                 / threadsPerBlock;

        // Device launched child kernels.
        // No device side cudaDeviceSynchronize is needed.
        childKernel<<<blocksFirst,  threadsPerBlock>>>(data,
                                                       firstHalfCount);
        childKernel<<<blocksSecond, threadsPerBlock>>>(data + firstHalfCou
                                                       secondHalfCount);
        // Parent kernel will not finish until both children finish.
    }
}

int main() {
    const int N = 1024 * 1024;    // 1M elements, avoids bit shifting for c
    float* d_data = nullptr;

    cudaMalloc(&d_data, N * sizeof(float));
    // Initialize to zero as a concrete, valid initialization.
    cudaMemset(d_data, 0, N * sizeof(float));

    // Launch parent on the default stream.
    parentKernel<<<1, 1>>>(d_data, N);

    // Wait for completion without cudaDeviceSynchronize. Sync stream inste
    cudaStreamSynchronize(0);
```

```
    cudaFree(d_data);
    return 0;
}
```

여기서 `parentKernel`는 두 자식 실행을 GPU에서 직접 발행합니다. 호스트는 커널 하나만 제출한 후 한 번만 대기합니다. 부모 커널이 완료되면 장치 런타임은 진행하기 전에 발행된 모든 자식 커널이 완료되었는지 확인합니다.

이 동적 병렬 처리 버전은 호스트 주도 병렬 처리(`cudaDeviceSynchronize()`)를 전혀 사용하지 않습니다. 이는 부모 커널이 모든 장치 시작 자식 커널이 완료될 때까지 완료되지 않는다는 암묵적 규칙에 의존하며, 호스트는 단순히 스트림을 기다립니다.

장치 측 DP 접근법에서는 CPU 의사결정을 위한 유휴 간격이 없습니다. 따라서 종속 단계 간 지연 시간을 단축하고 SM을 종단 간 바쁘게 유지합니다. 이는 장치에서 발생하는 소량의 런치당 오버헤드라는 약간의 비용을 치르면서 GPU 활용도를 높입니다(그림 12-13의 타임라인 참조).

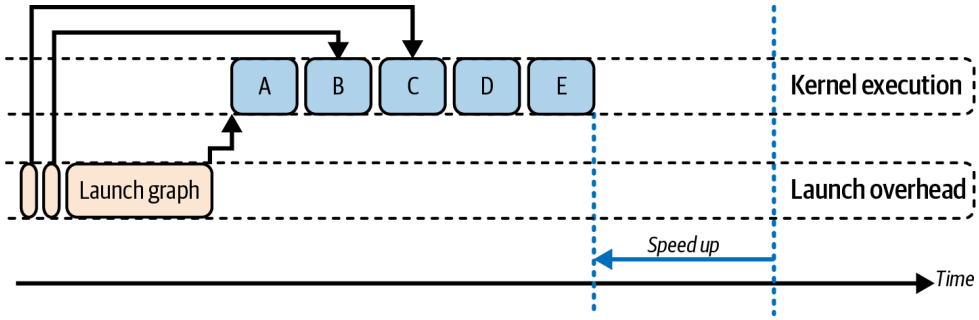

그림 12-13. 장치 측 런치와 동적 병렬화를 통한 간격제거

간단한 두 자식 커널 예시에서 호스트 주도 버전은 세 번의 별도 런치(부모 커널 1개 + 자식 커널 2개)를 수행합니다. 이 경우 CPU가 각 자식 커널의 런치 시점을 결정하는 동안 GPU는 유휴 상태로 대기합니다.

이는 부모 커널에 대해 단 한 번의 호스트 측 런치만 수행하는 장치 주도형 DP 버전과 대조됩니다. 부모 커널은 이후 추가 호스트 개입 없이 GPU에서 두 자식 커널을 동시에 생성합니다. 표 12-2는 호스트 주도형과 장치 주도형 DP 자식 런치의 성능을 비교합니다.

표 12-2. 호스트 주도형 대 GPU 주도형 중첩 자식 커널 실행 성능 비교 (자식 2개)

메트릭	이전 (호스트 실행)	이후 (디바이스 런치)
총 호스트 실행 횟수	3	1
호출당 평균 실행 오버헤드	~20 µs	~25 마이크로초
시퀀스 중 GPU 유휴 사이클	~40%	~5%
전체 실행 시간	1.00 ms	0.75 ms

여기서 자식 커널 디스패치를 GPU로 이동함으로써 DP가 약 35%의 유휴 시간을 제거하고 총 실행 시간을 약 25% 단축하는 것을 확인할 수 있습니다. 런치당 비용의 소폭 증가(20 µs → 25 µs)는 GPU의 온디바이스 스케줄링 오버헤드를 반영합니다. 그러나 이 오버헤드는 다중 CPU-GPU 핸드셰이크 제거로 인한 절감 효과에 비하면 무시할 수 있는 수준입니다.

GPU 주도 런치의 추가 이점은 중간 결과가 단계 간에 CPU로 복사될 필요가 없어 데이터 로컬리티가 개선된다는 점입니다. 본 예시에서 부모 커널이 계산한 데이터는 GPU 메모리를 벗어나지 않고 자식 커널이 즉시 사용할 수 있습니다. 이는 추가 메모리 전송을 피하고 캐시 데이터를 보존합니다. CPU 개입이 없으므로 GPU에서 재사용될 데이터의 캐시 이젝션이나 DRAM 재페치 가능성도 줄어듭니다.

요약하면, DP는 중단-시작 방식의 호스트 주도 워크플로우를 GPU 상주형 원활한 파이프라인으로 전환하여 높은 SM 활용도를 유지하고 호스트-GPU 간 조정을 최소화합니다. 또한 동적 병렬 처리(DP)는 다른 고급 기법과 마찬가지로 영향력 테스트가 필요하다는 점을 기억하십시오.

DP는 CPU 상호작용을 제거하지만, 장치에서 시작한 커널 실행은 호스트 실행과 거의 동일한 수준의 오버헤드를 가집니다. 따라서 모든 알고리즘이 이점을 얻는 것은 아닙니다. 사실 일부 알고리즘은 DP로 실행 시 오히려 느려질 수 있습니다—특히 실행된 커널의 작업량이 오버헤드를 상쇄하기에 너무 작을 경우 더욱 그렇습니다. 즉, 장치 측 실행 오버헤드가 일부 소규모 커널에 대해 DP의 이점을 상쇄할 수 있습니다.

변경 전후의 GPU-resident pipeline 성능을 항상 자질하십시오. Nsight Compute와 같은 도구를 사용하면 DP로 실행된 자식 커널을 자질하여 비용을 정량화하고, GPU-resident pipeline의 이점이 추가 오버헤드를 상쇄하며 처리량을 실제로 향상시키는지 확인할 수 있습니다.

단일 GPU 오케스트레이션에 대해 살펴본 후, 이제 다중 GPU 및 다중 노드 시나리오로 넘어갑니다. 여기서 상호 연결 대역폭과 집합 연산은 우리의 루프라인 고려 사항을 클러스터 수준으로 확장합니다.

다중 GPU 및 클러스터 노드 간 오케스트레이션 (NVSHMEM)

하나의 GPU에서 다수의 GPU로 확장할 때 핵심 목표는 동일하게 유지됩니다: 유용한 작업 뒤에 데이터 이동을 숨겨 모든 장치를 바쁘게 유지하는 것입니다. 호스트가 각 GPU에 작업을 분배한 후(별도의 CPU 스레드, 비동기 런치, 또는 다중 GPU 그래프를 통해), GPU가 작업을 인수합니다. 각 장치의 하나의 스트림이 계산을 주도하는 동안, 두 번째 스트림은 호스트 메모리를 전혀 거치지 않고

NVLink 또는 PCIe를 통해 피어 투 피어(peer-to-peer)로 데이터를 전송할 수 있습니다.

이는 대규모 환경에서 피어 투 피어 전송과 계산을 중첩시켜야 함을 의미합니다. NVLink를 사용하더라도 대역폭과 지연 시간이 장치 내 HBM과 동일하지 않다는 점을 유의해야 합니다. 따라서 중첩을 통해 이 통신을 숨겨야 합니다.

실제 환경에서 클러스터 규모가 커질수록 작업과 데이터 전송의 중첩은 환경을 선형적으로 확장하는 데 절대적으로 필수적입니다. 간단한 작업 인계를 위해 GPUDirect Peer Access를 사용하면 다음과 같이 백그라운드에서 대용량 메모리 블록을 이동할 수 있습니다:

```
cudaMemcpyPeerAsync(dest_ptr, dest_gpu, src_ptr, src_gpu, size, comm_strea
```

PyTorch 분산 데이터 병렬(DDP)에서 기울기 전체 축소(all-reduce)와 같은 집합적 통신이 필요할 때는 NCCL의 비동기 집합 호출을 사용해 별도의 스트림에서 NCCL의 비차단 루틴을 실행합니다. 그러면 NCCL이 텐서를 링 또는 트리 구조로 배열하여 모든 NVLink 및 NVSwitch 경로를 포화 상태로 만듭니다. 이 모든 작업이 수행되는 동안 컴퓨팅 커널은 자체 스트림에서 계속 실행됩니다.

사용 중인 MPI 라이브러리가 CUDA를 인식하고 GPU 장치 포인터를 감지할 경우, 다음과 같은 호출을 통해 InfiniBand를 통한 데이터 전송에 GPUDirect RDMA를 자동으로 사용합니다:

```
MPI_Send(device_buf, count, MPI_FLOAT, peer_rank, ...);
```

MPI(및 NCCL)의 이러한 CUDA 인식 기능은 GPU 데이터가 호스트 메모리를 경유하지 않고 InfiniBand를 통한 GPUDirect RDMA로 네트워크를 직접 이동함을 의미합니다. (노드 내에서는 GPUDirect Peer-to-Peer를 사용하여 NVLink 또는 PCIe를 통해 피어 복사가 실행됨을 참고하십시오.)

이러한 기본 기능을 사용하고 CPU를 회피함으로써 데이터 전송 지연 시간을 줄이고 GPU 간 전송에서 거의 유선 속도에 가까운 성능을 달성할 수 있습니다. 결과적으로 노드 간 데이터 전송 및 통신은 GPU 계산과 적절히 중첩될 수 있습니다.

NVSHMEM을 통한 세밀한 GPU-to-GPU 메모리 공유

동적 작업 큐 및 세분화된 이벤트 알림과 같이 초밀착형, 실시간( ) 이벤트 기반 조정이 필요한 워크로드에는 NVIDIA의 NVSHMEM(NVIDIA SHMEM) 라이브러리가 탁월한 선택입니다. 이 라이브러리는 각 GPU를 분할된 전역 주소 공간(PGAS) 내의 처리 요소(PE)로 취급합니다.

PGAS를 통해 GPU는 CPU를 거치지 않고 장치 코드에서 다른 GPU의 메모리에 직접 쓰기 작업을 수행할 수 있습니다. 지연 시간은 인터커넥트에 따라 달라지며,

일반적으로 NVLink가 PCIe 또는 네트워크 전송 방식보다 낮습니다. 다음은 NVSHMEM을 사용한 전형적인 '전송 후 신호 전송(send-and-signal)' 패턴입니다:

```
#include <cstdio>
#include <cuda_runtime.h>
#include <nvshmem.h>
#include <nvshmemx.h>

// Device symbols for the symmetric buffers
__device__ int   *remote_flag;
__device__ float *remote_data;

//---------------------------------------------------------------------------
// GPU 0: send data then signal GPU 1
//---------------------------------------------------------------------------
__global__ void sender_kernel(float *local_data, int dest_pe) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float value = local_data[idx];

    // 1) Put the payload into remote_data[1] on dest_pe
    nvshmem_float_p(remote_data + 1, value, dest_pe);
    // 2) Wait for the RMA to complete before setting the flag
    nvshmem_quiet();
    // 3) Signal completion by setting remote_flag[0] = 1 on dest_pe
    nvshmem_int_p(remote_flag + 0, 1, dest_pe);
}

//---------------------------------------------------------------------------
// GPU 1: wait for flag then consume payload
//---------------------------------------------------------------------------
__global__ void receiver_kernel(float *recv_buffer) {
    // 1) Spin until remote_flag[0] == 1
    nvshmem_int_wait_until(remote_flag + 0,
        NVSHMEM_CMP_EQ, 1);
    // 2) Once flag is set, the payload at remote_data[1] is valid
    float val = remote_data[1];
    recv_buffer[0] = val * 2.0f;
}

//---------------------------------------------------------------------------
// Host-side setup and teardown
//---------------------------------------------------------------------------
int main(int argc, char **argv) {
    // 1) Initialize the NVSHMEM runtime
    nvshmem_init();

    // 2) Determine this PE's rank and bind to the matching GPU
    int mype = nvshmem_my_pe();
    cudaSetDevice(mype);

    // 3) Allocate symmetric buffers on each PE
```

```
    //    - Two ints for the flag
    //    - Two floats for the data payload
    int   *flag_buf = (int*)   nvshmem_malloc(2 * sizeof(int));
    float *data_buf = (float*) nvshmem_malloc(2 * sizeof(float));

    // 4) Zero out flags on PE 0 and synchronize
    nvshmem_barrier_all();
    if (mype == 0) {
        int zeros[2] = {0, 0};
        cudaMemcpy(flag_buf, zeros, 2 * sizeof(int),
                   cudaMemcpyHostToDevice);
    }
    nvshmem_barrier_all();

    // 5) Register the device pointers for use in kernels
    cudaMemcpyToSymbol(remote_flag, &flag_buf, sizeof(int*));
    cudaMemcpyToSymbol(remote_data, &data_buf, sizeof(float*));

    // 6) Launch either the sender or receiver kernel
    dim3 grid(1), block(128);
    if (mype == 0) {
        // Example input buffer for the sender
        float *local_data;
        cudaMalloc(&local_data, 128 * sizeof(float));
        // ... initialize local_data as needed ...
        sender_kernel<<<grid, block>>>(local_data, 1);
        cudaFree(local_data);
    } else {
        float *recv_buffer;
        cudaMalloc(&recv_buffer, sizeof(float));
        receiver_kernel<<<grid, block>>>(recv_buffer);
        cudaFree(recv_buffer);
    }

    // 7) Wait for all GPU work to finish
    cudaDeviceSynchronize();

    // 8) Clean up NVSHMEM resources
    nvshmem_free(flag_buf);
    nvshmem_free(data_buf);
    nvshmem_finalize();

    return 0;
}
```

여기서 일방적 원격 메모리 작업은 완전히 장치 내에서 수행됩니다. GPU/PE 0은 결과를 GPU/PE 1의 메모리에 직접 기록하고 해당 위치에서 플래그를 설정합니다. 구체적으로 GPU/PE 0은 `nvshmem_float_p`를 발행하여 페이로드 데이터를 GPU/PE 1의 메모리에 직접 기록하고, `nvshmem_quiet()`를 호출하여 완료를 확인한 후 `nvshmem_int_p`를 사용하여 플래그를 설정합니다.

한편 GPU/PE 1의 커널은 `nvshmem_int_wait_until()` 에서 대기하다가 플래그가 설정되는 즉시 페이로드를 읽습니다. 이 과정에는 CPU 개입이나 추가 복사 작업이 필요하지 않으며, NVLink를 통한 하드웨어 가속 GPU-to-GPU 전송만으로 이루어집니다.

NVSHMEM 통신은 NVLink 또는 PCIe를 통한 GPU 시작형 단방향 작업을 사용하므로 호스트 스테이징이 필요 없습니다. 따라서 NVSHMEM 통신은 거의 최대 유선 속도에 근접할 수 있습니다. 이는 NVSHMEM 단방향 작업이 CPU와 커널 실행의 소프트웨어 오버헤드를 우회하기 때문입니다. 본질적으로 NVSHMEM은 기존 다단계 통신을 단일 하드웨어 트랜잭션으로 전환합니다.

물론 큰 힘에는 큰 책임이 따릅니다. NVSHMEM은 본질적으로 GPU 수준의 공유 메모리 프로그래밍이므로 동기화를 신중하게 관리하고 경합을 피해야 합니다. 또한 글로벌 배리어를 과도하게 사용하면 가장 느린 피어의 모든 GPU가 정지될 수 있습니다.

실제 적용 시 과도한 동기화는 피하십시오. 가능하면 NVSHMEM의 세밀한 신호(fine-grained signals)나 지점 간 동기화(point-to-point synchronization)를 사용하십시오. 이는 항상 ` `nvshmem_barrier_all()` `를 호출하는 방식과 대비됩니다.

현대적인 NVSHMEM 구현체는 이러한 동기화 루틴에 대한 효율성 개선을 제공합니다. 그러나 여전히 동기화 지점이며, 잘못 사용하면 병목 현상이 될 수 있습니다. NVSHMEM은 장치 변수를 대기하기 위한 `nvshmem_wait_until` 와 같은 세밀한 원시 함수와, 일부 장치만 협조해야 할 때의 점대점 동기화를 위한 `nvshmem_signal_fetch`, `nvshmem_signal_wait_until` 또는 `nvshmemx_signal_op` 변형과 같은 신호 연산을 제공합니다. NVSHMEM이 송신자 GPU와 수신자 GPU 간에 데이터를 공유하고 신호로 동기화하는 저수준 세부 사항은 그림 12-14에 표시되어 있습니다.

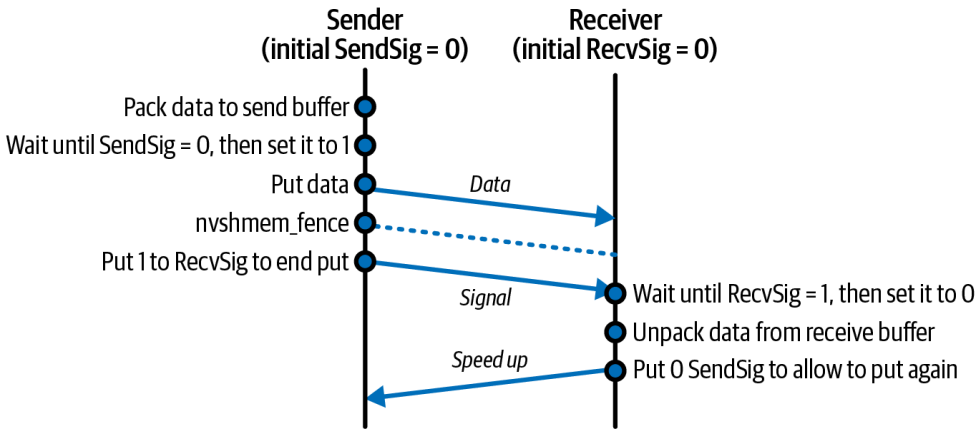

그림 12-14. NVSHMEM 단방향 통신 예시

의 작업 부하가 불규칙하거나 데이터에 의존적인 경우(예: 그래프 알고리즘, 동적 부하 분산, 이산 사건 시뮬레이션)에 NVSHMEM이 빛을 발합니다. 이러한 경우 정적 그래프와 집단 연산만으로는 충분하지 않습니다.

NVSHMEM을 사용하는 커널은 다른 커널과 마찬가지로 CUDA 그래프 내에서 캡처될 수 있습니다( ). NVSHMEM 장치 작업은 해당 커널 내부에서 수행되며 별도의 그래프 노드가 아닙니다. 그러나 NVSHMEM의 진정한 강점은 그래프에서 사용하는 고정된 통신 스크립트 없이 커널이 실시간으로 적응하고 조정할 수 있게 하는 데 있습니다.

간단히 말해, NVSHMEM은 GPU 클러스터를 공유 메모리 영역으로 변환하여 CPU 매개 접근 방식보다 훨씬 뛰어난 지연 시간과 처리량으로 장치 전용 커널 실행, 데이터 전송 및 동기화를 가능하게 합니다.

다음과 같은 2단계 변환기 추론 파이프라인(어텐션 + 다층 퍼셉트론)을 상상해 보십시오: GPU 0이 어텐션을 계산한 후, NVSHMEM이 그 활성화 값을 GPU 1로 전송하고 신호를 보냅니다. 지속적 커널을 실행 중인 GPU 1은 신호를 감지하고 MLP 단계를 시작합니다.

GPU 1이 아직 배치 1을 처리하는 동안 GPU 0은 즉시 배치 2로 넘어가므로, 몇 번의 반복 후 양쪽 장치가 완벽한 협동 상태로 작동합니다. 각 작업 인계는 활성 컴퓨트 워프 뒤에 숨겨져 있습니다. 이는 호스트 스톨 없이 거의 100%에 가까운 활용도를 실현합니다.

유연한 부하 분산이 필요할 때, NVSHMEM의 원자적 연산은 각 PE가 동적으로 작업을 할당받도록 합니다. PE는 병렬 NVSHMEM 애플리케이션의 일부인 OS 프로세스입니다.

여기 코드는 각 GPU가 글로벌 카운터에서 다음 인덱스를 가져와 청크를 처리한 후 루프를 반복하는 방식을 보여줍니다. 이는 호스트 조정 없이 장치 내에서 완전히 진정한 작업 스틸링을 가능하게 합니다:

```
__global__ void work_steal_kernel(/*...*/, int *queue_head, Task *tasks) {
    while (true) {
        // Atomically claim the next task index
        int idx = nvshmem_int_atomic_inc(queue_head);
        if (idx >= N_tasks) break;
        // Process tasks[idx]...
    }
}
```

최소 지터가 요구되는 시나리오(예: 타이트한 다중 GPU 콜렉티브 또는 동기식 모델 병렬 단계)에서는 NVSHMEM을 사용해 모든 GPU에 걸쳐 하나의 협력적 커널을 실행할 수 있습니다. GPU 0과 GPU 1에서 송신자 및 수신자 커널을 동시에 시작하기 위해 `nvshmemx_collective_launch()`를 사용하면 호스트 개입 없이 NVSHMEM을 통해 협조할 수 있습니다. 이후 다음과 같이 NVSHMEM의 장치 측 배리어를 활용할 수 있습니다:

```
__global__ void synchronized_step_kernel(/*...*/) {
    nvshmem_barrier_all();
    // All GPUs proceed in lockstep here
```

```
    // ...
}
```

여기서 모든 PE(프로세싱 엔터티)가 동시에 `nvshmem_barrier_all()` 에 진입한 후 동시에 진행됩니다. 이는 클러스터 전반에 걸쳐 완벽하게 정렬된 실행을 보장합니다.

NVSHMEM의 장치 수준 동기화 또는 콜렉티브를 사용하는 모든 커널은 `nvshmemx_collective_launch()`로 실행해야 합니다. 이는 커널이 작업 내 모든 PE(GPU)에서 동시 실행되도록 보장합니다.

NCCL 및 CUDA 그래프를 사용한 다중 GPU 집단 연산 캡처

브로드캐스트, 리덕션, 전수 전송(all-to-all transfer)과 같은 대량 콜렉티브 연산이 필요한 경우, NVIDIA의 NCCL 라이브러리는 멀티 GPU 시스템에서 가장 적합한 선택입니다. 기존 방식에서는 각 GPU가 호스트에서 콜렉티브 연산(`ncclAllReduce`) 또는 유사한 연산을 시작합니다. 이후 다음 컴퓨팅 단계로 진행하기 전에 대기(동기화)합니다. 이러한 순차적 호스트 오케스트레이션은 전진 패스와 후진 패스 사이에 오버헤드와 유휴 시간을 추가합니다.

그러나 NCCL 호출도 커널과 마찬가지로 CUDA 그래프에 기록될 수 있습니다. 이를 통해 전방 커널, 올-리듀스, 후방 커널을 단일 그래프에 '내장'하여 각 반복마다 재생할 수 있습니다:

```
cudaStreamBeginCapture(captureStream,
    cudaStreamCaptureModeGlobal);

forwardKernel<<<...>>>(...);

ncclAllReduce(sendBuf, recvBuf, count, ncclFloat,
    ncclSum, comm, captureStream);

backwardKernel<<<...>>>(...);

cudaStreamEndCapture(captureStream, &graph);

// Instantiate and upload before launching
cudaGraphExec_t graphExec;
cudaGraphInstantiate(&graphExec, graph, ...);
cudaGraphUpload(graphExec, captureStream);

// Each training step:
cudaGraphLaunch(graphExec, captureStream);
```

모든 작업(NCCL 포함)에 동일한 캡처 스트림이 사용됨을 주목하십시오. 그래프를 사용하면 NCCL 호출도 커널과 마찬가지로 그래프 노드가 됩니다. 각 프로세스가 동일한 그래프를 재현하므로 NCCL의 내부 로직이 피어를 찾아 추가적인 호스트 조정 없이 올-리듀스를 실행합니다. 이 그래프 캡처 방식의 올-리듀스는 대규모 클러스터에서 특히 강력합니다. 반복당 런치 지터를 제거하고 모든 GPU가 계산과 네트워크 작업을 중첩하여 바쁘게 유지하기 때문입니다.

각 GPU가 콜렉티브 노드를 포함한 동일한 그래프를 런치하므로 NCCL은 추가 호스트 개입 없이 내부적으로 랭크 간 랜데뷰를 수행합니다. 호스트의 반복당 작업은 단일 `cudaGraphLaunch`로 감소하여 CPU 오버헤드와 런치 지터를 줄입니다.

CPU 부하 감소 외에도, 그래프 내에서 올-리듀스를 캡처하면 여러 GPU 및 컴퓨팅 노드 간 통신과 연산을 진정한 의미로 중첩할 수 있습니다. 예를 들어, 기울기 계산을 두 단계(레이어 1부터 *L/2까지*, 레이어 (*L/2*+ 1)부터 *L까지)* 로 나누고 별도의 스트림에 *매핑한다고 가정해* 보겠습니다:

```
// Pseudocode in capture:
computeGradientsLayer1<<<...>>>(..., streamA);
ncclAllReduce(..., comm, streamB);       // in streamB, overlaps with strea
computeGradientsLayer2<<<...>>>(..., streamA);
ncclAllReduce(..., comm, streamB);
```

이러한 노드들은 고유한 스트림 할당과 종속성을 함께 캡처되므로, CUDA 드라이버는 `streamB` 에서의 NCCL 네트워크 전송과 `streamA` 에서의 독립 작업 간에 오버랩을 수행할 수 있습니다. 이 버킷형 올-리듀스 패턴은 통신 지연 시간을 계산 뒤에 지속적으로 숨김으로써 다중 GPU 확장성을 향상시킵니다.

NCCL 콜렉티브는 모든 랭크가 동일한 커뮤니케이터로 동일한 시퀀스를 캡처하고 재생할 때 그래프 캡처와 호환됩니다. 그러나 모든 랭크는 동일한 커뮤니케이터로 동일한 NCCL 시퀀스를 캡처하고 재생해야 합니다. 그래프 재생 간 커뮤니케이터 불일치는 교착 상태(최선의 경우, 디버깅이 비교적 용이함) 또는 잘못된 결과(최악의 경우, 무음 실패 및 디버깅이 어려움)를 초래할 위험이 있습니다. 또한, 초기 워밍업 콜렉티브를 캡처하고 미리 실행하여 커뮤니케이터를 초기화한 후, 인스턴스화된 그래프를 재사용하여 안정 상태 지연 시간을 최소화하는 것이 권장됩니다.

실제 대규모 모델 훈련에서는 이 버킷형 올-리듀스 접근법이 표준입니다. 차등 감소 작업 블록을 다음 레이어 계산과 중첩함으로써 네트워크 시간을 거의 모두 숨길 수 있습니다. 별도 프로세스(프로세스 1과 프로세스 2)에서 실행되는 DDP를 활용한 버킷형 올-리듀스 예시는 그림 12-15에 제시되어 있습니다.

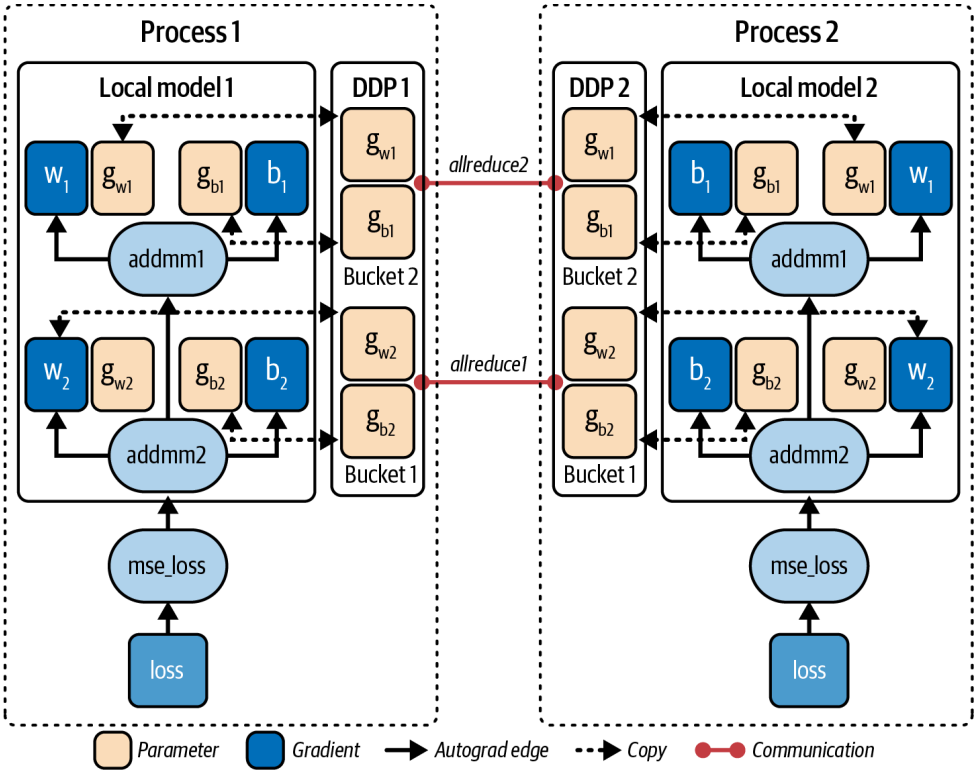

그림 12-15. 계산과중첩된 올-리듀스

PyTorch DDP와 같은 현대적 라이브러리는 이 접근법의 변형을 자동으로 구현합니다. 그러나 CUDA 그래프를 캡처하면 CPU 오버헤드를 추가로 줄이고 더 결정론적인 성능을 제공할 수 있습니다.

다중 GPU 환경에서 CUDA 그래프를 사용할 때 몇 가지 고려 사항을 명심하세요. 첫째, 참여하는 모든 GPU는 교착 상태를 피하기 위해 동일한 순서로 콜렉티브를 기록하고 재생해야 합니다. 이는 모든 랭크가 동일한 순서로 콜렉티브에 진입해야 하는 MPI의 콜렉티브 규칙과 유사합니다.

둘째, CUDA 그래프가 GPU 버퍼를 고정하고 재사용하므로, 캡처 전에 기울기 및 통신 버퍼 할당이 완료되었는지 확인해야 합니다. 또한 앞서 논의한 바와 같이, 배치 크기 같은 그래프 내 매개변수를 수정해야 할 경우, 전체 재캡처 없이 `cudaGraphExecUpdate`를 사용하여 해당 매개변수를 패치할 수 있습니다.

실제 적용 시 그래프 내 NCCL plus 컴퓨트를 캡처하면 단계별 CPU 시간을 단축하고 다중 GPU를 활용한 대규모 모델 훈련 속도를 높일 수 있습니다. 수십만 개의 GPU를 사용하는 대규모 환경에서는 이러한 효율성이 누적되어 클러스터 전체에 걸쳐 동기화가 강화되고 활용도가 높아집니다.

NCCL과 CUDA 그래프는 계산과 병행하여 집단 통신을 스케줄링하는 효율적인 방법을 제공합니다. 그러나 모든 다중 GPU 통신이 집단적인 것은 아닙니다. 때로는 GPU 간에 더 세분화된 또는 비동기적인 데이터 공유가 필요합니다. 바로 여기서 앞서 설명한 NVSHMEM이 도움이 될 수 있습니다.

N-GPU 확장 패턴

단순한 피어 복사, NCCL 링 기반, NVSHMEM 단방향 원자 연산을 사용하든, 다중 GPU 확장의 패턴은 항상 동일합니다. 시스템은 커널을 한 번만 디스패치하고, 데이터 전송을 컴퓨팅과 병렬화 및 중첩하며, 특히 호스트 CPU를 크리티컬 패스에서 제외하여 선형적으로 확장해야 합니다.

예를 들어, 4개의 GPU는 모든 GPU에 데이터를 병렬로 공급할 수 있고(그림 12-16 참조) 호스트가 루프에서 제외된다는 가정 하에, 4배 빠른 단일 GPU처럼 동작해야 합니다. 이를 달성할 수 있다면 통신과 계산을 적절히 중첩시켜 거의 선형적인 속도 향상을 얻을 수 있습니다. 중첩이 없으면 통신 시간이 계산 시간과 같아지는 시점에서 확장성이 정체됩니다.

GPU를 늘릴수록 데이터 전송과 연산의 파이프라인화를 더욱 공격적으로 적용해야 하며, CPU 측의 오케스트레이션과 동기화는 줄여야 합니다. 따라서 비동기 복사, 그래프 내 NCCL 콜렉티브, NVSHMEM의 PGAS 프리미티브를 활용해 더 많은 오케스트레이션을 장치로 오프로드해야 합니다. 이는 소프트웨어에 더 많은 책임을 전가하는 효과를 가져옵니다.

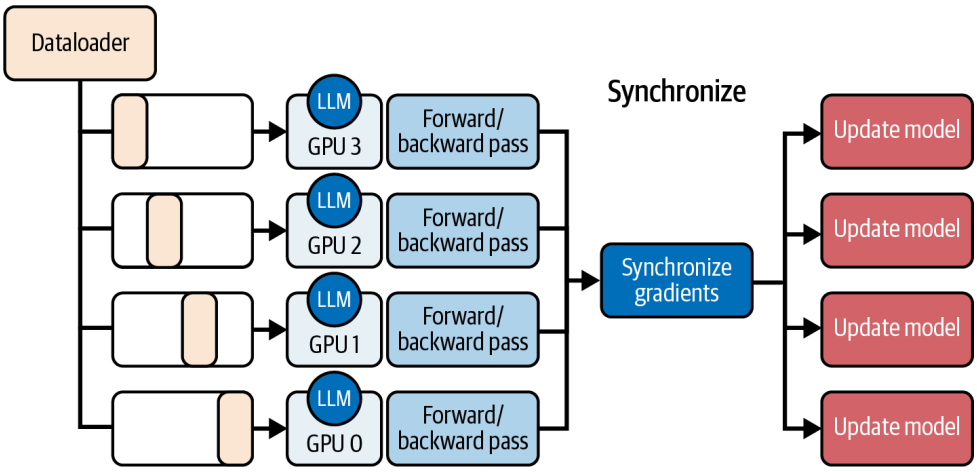

그림 12-16. 각 GPU는 데이터를 동시에 교환하면서 병렬로 계산함—유휴 GPU 및 지연된 데이터 전송 없음

이러한 기법을 적용하면 CPU 병목 현상을 제거하고, 고속 상호 연결을 포화 상태로 끌어올리며, 연산 FLOPS를 극대화하고, 진정한 저지연 다중 GPU 파이프라인을 구축할 수 있습니다. 다음으로 동적 및 장치 측 스케줄링과 오케스트레이션의 맥락에서 루프라인 모델을 재검토해 보겠습니다.

루프라인 기반 스케줄링 및 오케스트레이션 결정

지난 몇 장( )에 걸쳐 CUDA 스트림, 커널 융합, 지속적 커널, CUDA 그래프, 동적 병렬 처리 등 탄탄한 오케스트레이션 기법들을 살펴보았습니다. 루프라인 모델은 특정 상황에서 가장 큰 이점을 제공할 도구를 결정하는 데 도움을 줍니다.

근본적으로 루프라인은 연산 집약도, 즉 이동된 바이트 대비 수행된 FLOPS의 비율로 귀결됩니다. 이는 두 가지 하드웨어 "상한선"으로 구성됩니다: 대역폭에 의

해 제한될 때의 최대 처리량을 나타내는 메모리 루프라인(경사진)과 ALU에 의해 제한될 때의 최대 연산 속도를 표시하는 컴퓨트 루프라인(평평한)으로, 그림 12-17에 표시된 바와 같습니다.

커널이 메모리 지붕 근처(예: 낮은 FLOPS/바이트)에 위치하여 메모리 제약 상태라면, 최적의 개선책은 메모리 전송을 계산 작업으로 숨기거나 중첩시키는 것입니다. 즉, CUDA 스트림을 활용한 비동기 복사나 심지어 여러 메모리 바운디드 커널을 동시에 실행해야 합니다. 이렇게 하면 메모리 시스템의 서로 다른 부분을 더 효과적으로 포화시킬 수 있습니다.

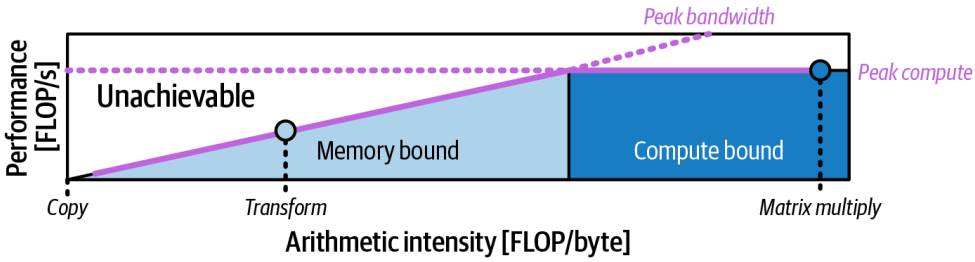

그림 12-17. 두 가지 하드웨어 한계(메모리 바운디드(예: 데이터 변환 연산) 및 컴퓨팅 바운디드(예: 행렬 곱셈))를 가진산술 집약도

커널 융합은 메모리 바운디드 워크로드에서 성능 향상에 다소 도움이 됩니다. 중간 글로벌 메모리 왕복 횟수를 줄일 수 있습니다. 그러나 진정한 이점은 지연 시간 가림과 더 많은 로드/스토어를 동시에 처리하는 데서 나옵니다.

반대로 컴퓨트 루프라인 하단에 위치한 고강도 커널은 ALU를 지속적으로 활용해야 합니다. 이때 커널 융합이 효과적입니다. 별도의 add+scale 연산(앞서 소개한 융합 예시)을 단일 패스로 통합하여 바이트당 FLOPS를 증가시키고, 루프라인 플롯에서 지점을 우측으로 이동시켜 커널이 피크 FLOPS 대비 더 높은 비율에 도달하도록 합니다.

마찬가지로, 지속적 커널, 스레드 블록 클러스터, 장치 시작 CUDA 그래프는 연산 강도 수치를 변경하지 않지만 반복 실행으로 인한 유휴 간격을 줄입니다. 이는 커널 성능을 평탄한 컴퓨팅 한계에 가깝게 끌어올립니다.

많은 실제 워크로드는 중간에 위치합니다. 메모리 바인딩이나 컴퓨팅 바인딩이 강하지 않은 경우죠. 이때는 동시성이 해결책입니다. 스트림, 동시 그래프 또는 여러 지속적 커널을 사용해 여러 개의 중간 강도 커널을 병렬로 실행하면, 두 축 모두에서 집계 처리량 지점이 더 높은 위치에 도달하도록 결합됩니다. 이는 장치 자원을 더 효율적으로 활용하는 방법입니다.

철저한 루프라인 분석에는 체계적인 측정이 필요합니다. Nsight Compute를 사용하여 FLOPS와 전송된 바이트 수를 계산하고, 커널의 위치를 플롯하여 각 루프라인 아래 얼마나 떨어져 있는지 확인하세요.

워크로드가 메모리 바운디드를 보인다면, 스트림을 활용하고 작업 중첩을 적용하며 정밀도 감소(FP16, FP8, FP4)를 고려하여 산술 집약도 방정식에서 분모(예: 전송된 바이트 수)를 줄이십시오.

커널이 컴퓨팅에 바운디드하지만 런치 오버헤드나 유휴 시간으로 인해 최대 FLOPS에 도달하지 못하는 경우, 런치 오버헤드 감소에 집중하세요. 앞서 배운 바와 같이, 연산을 하나의 커널로 융합하거나, 지속적 커널을 사용하거나, CUDA 그래프를 캡처하거나, 디바이스 측 런치를 수행함으로써 이를 달성할 수 있습니다. 이렇게 하면 ALU에 지속적으로 작업이 공급됩니다.

커널이 두 가지 제약 조건 모두에서 여유가 있고 메모리나 컴퓨팅 중 어느 쪽에도 완전히 바운디드되지 않았다면, 동시성을 높여 보십시오. 여러 커널과 스트림을 병렬로 실행하면 시스템 자원을 더 효율적으로 활용할 수 있습니다.

이러한 정량적 지침을 통해 모든 기법을 한꺼번에 시도하기보다 커널에 적합한 조정 전략을 선택할 수 있습니다. 다만 각 최적화가 하드웨어의 진정한 잠재력에 더 가까워지는지 검증하는 것을 잊지 마십시오. 최적화 적용 후에는 항상 측정하십시오.

루프라인 모델은 기대치를 안내하지만, 컴퓨팅 처리량, 달성된 점유율, 메모리 처리량 등을 포함한 실제 성능 측정이 전체적인 상황을 알려줍니다. 루프라인 분석과 반복적·지속적인 자질을 결합하면 선택한 최적화 전략이 실제로 효과적인지 확인할 수 있습니다.

핵심 요약

GPU 성능 극대화는 최소한의 오버헤드로 계산과 데이터 이동을 유기적으로 결합하는 데 달려 있습니다. 효율적인 오케스트레이션은 CPU와 GPU 간 복잡한 워크로드를 원활하게 처리하여 어느 한 쪽도 다른 쪽을 지연시키지 않도록 합니다. 본 장의 핵심 내용은 다음과 같습니다:

L2 캐시 원자 큐를 활용한 동적 스케줄링

현대 GPU의 L2 캐시 원자 연산은 매우 빠릅니다. GPU 상에서 불규칙한 워크로드를 균형 있게 처리하려면 배치 증분 방식의 고속 L2 캐시 원자 연산을 활용하세요. 이 배치 작업 할당은 경합을 줄이고 워프 유휴 간격을 제거하여 워프를 지속적으로 작동 상태로 유지합니다. 극단적인 불균형 사례에서는 처리량을 최대 약 2배까지 크게 향상시킬 수 있지만, 일반적으로 10%에서 30% 사이의 성능 향상을 보입니다. 높은 L2 대역폭 덕분에 8이나 16과 같은 적당한 배치 크기만으로도 대부분의 경합을 제거할 수 있습니다.

고정 파이프라인을 위한 CUDA 그래프

GPU 작업 시퀀스를 한 번 기록한 후, 각 반복마다 단일 호스트 호출로 재생하세요. 이는 반복당 CPU 스케줄링 오버헤드를 줄여, 종종 20~30%의 지연 시간 감소(대규모에서는 더 큼)를 가져옵니다. GPU에서 종속 작업의 중첩을 최대한 확보하고 있는지 확인하세요.

CUDA 그래프를 활용한 저오버헤드 런치

비동기 복사, 커널 실행, 이벤트 기록, 할당 등의 시퀀스를 CUDA 그래프(`cudaStreamBeginCapture/cudaStreamEndCapture`)로 캡처하세요. 동적 병렬 처리(`cudaGraphLaunch`)로 그래프를 재생하면 호출별 CPU 대기열 오버헤드를 제거하면서 모든 스트림 간 종속성을 유지하여 런타임 병목 현상을 추가로 줄일 수 있습니다.

디바이스 측 오케스트레이션

사전 기록된 CUDA 그래프를 테일 런칭하거나 동적 병렬화를 사용하여 자식 커널을 생성함으로써 GPU 자체에서 작업을 런칭하세요. 이는 CPU 스케줄링 간극을 완전히 제거하고 호스트 개입 없이 GPU가 종단 간 지속적으로 작업할 수 있게 합니다.

다중 GPU 오버랩

통신과 계산을 항상 중첩하십시오. 별도의 스트림을 사용하여 GPU 피어 투 피어 전송(`cudaMemcpyPeerAsync`), NCCL 콜렉티브, CUDA 인식 MPI(RDMA) 또는 NVSHMEM 단방향 작업을 파이프라인화하십시오. 이는 유용한 작업 뒤에 통신 지연을 숨기며, 유리한 계산 대 통신 비율과 충분한 중첩 하에서 다수의 GPU에 걸쳐 선형 스케일링에 근접할 수 있습니다. 중첩이 충분할 경우 클러스터 노드 간에도 가능합니다.

루프라인 기반 선택

루프라인 차트가 전략을 주도하도록 하십시오. 커널이 메모리 바운디드라면, 비동기 메모리 복사 및 FP8/FP4와 같은 혼합 정밀도를 활용하여 중첩과 데이터 이동 감축에 집중하십시오. 컴퓨팅 바운디드이지만 오버헤드로 인해 성능이 저하된다면, 커널 융합, 지속적 커널, CUDA 그래프와 같은 런치 감소 기법을 사용하여 컴퓨팅 상한선에 접근하십시오. 중간 수준의 커널은 모든 하드웨어 유닛을 활용하기 위해 여러 작업을 병렬로 실행하여 동시성을 높이십시오. 선택한 최적화가 실질적인 효과를 내는지 항상 자질로 검증하십시오.

동적 디스패치, 협력적 커널, 그래프 캡처/리플레이, GPU 네이티브 메모리 공유 등 이러한 기법들을 결합함으로써, 초스케일 AI 워크로드를 위해 GPU 클러스터의 모든 부분을 포화시키는 파이프라인을 구축할 수 있습니다.

결론

이 장에서는 단일 커널 최적화를 넘어 엔드투엔드 오케스트레이션 기법을 탐구했습니다. 동적 병렬화를 통해 작업을 완전히 디바이스에서 실행하는 방법, CUDA 그래프에 복잡한 워크플로를 캡처하는 방법, NCCL 및 NVSHMEM을 사용하여 다수의 GPU를 조정하는 방법을 다루었습니다. 각 기법은 동일한 목표를 공유합니다: 모든 엔진에 작업을 지속적으로 공급하고, 지연 시간을 숨기며, 호스트-디바이스 간 격차를 해소하여 하드웨어가 최대 성능으로 작동하도록 하는 것입니다.

NVIDIA의 최신 GPU 플랫폼은 CPU와 GPU의 경계를 그 어느 때보다 모호하게 만듭니다. 예를 들어, Grace Blackwell 및 Vera Rubin 슈퍼칩은 방대한 대역폭을 가진 일관성 있는 NVLink를 사용하여 CPU를 여러 GPU와 연결합니다.

하지만 하드웨어가 CPU-GPU 장벽을 낮추더라도, 이 고성능 하드웨어를 완전히 활용하는 책임은 여전히 소프트웨어에 있습니다. 이 장에서 다루는 접근법들—C++의 CUDA든 고수준 라이브러리 API든—이 바로 이러한 발전을 활용하는 방법입니다.

다음 장에서는 PyTorch가 스트림, 그래프, 비동기 작업, 최적화된 커널 등 이러한 아이디어들을 어떻게 통합하여 단 몇 줄의 Python 코드로도 이러한 성능을 달성할 수 있는지 살펴보겠습니다. PyTorch 생태계에 깊이 들어가 고성능 AI 워크로드 구현에 왜 이토록 인기 있는지 이해해 보겠습니다.