CUDA에서 하드웨어와 논리의 매핑

CUDA는 NVIDIA의 GPU에서 대규모 병렬 처리를 지원하는 플랫폼입니다. 하드웨어와 논리 구조의 매핑은 다음과 같습니다.

1. 하드웨어 구조

Device: GPU 전체
SM(Streaming Multiprocessor): 여러 개의 CUDA Core(연산 유닛)를 포함하는 하드웨어 블록
CUDA Core: 실제 연산을 수행하는 가장 작은 단위

2. 논리 구조

Grid: 하나의 커널 함수 실행 시 생성되는 전체 스레드 집합
Block: Grid를 구성하는 작은 스레드 그룹. 각 Block은 독립적으로 실행됨
Thread: Block을 구성하는 가장 작은 실행 단위

3. 매핑 관계

Grid → Device: Grid 전체가 GPU(Device)에서 실행됨
Block → SM: 각 Block은 하나의 SM에 할당되어 실행됨. 여러 Block이 하나의 SM에서 순차적으로 실행될 수도 있음
Thread → CUDA Core: Block 내의 Thread들은 SM 내의 CUDA Core에서 병렬로 실행됨. Block 내 Thread 수가 SM의 Core 수보다 많으면 여러 번에 나누어 실행됨(스케줄링)

4. 예시

1개의 Grid에 100개의 Block, 각 Block에 256개의 Thread가 있다면:
- 100개의 Block이 여러 SM에 분배됨
- 각 Block의 256개 Thread가 SM 내 CUDA Core에서 병렬로 실행됨

5. 정리

논리적 구조(Grid, Block, Thread)는 하드웨어 구조(Device, SM, Core)에 매핑되어 실행됨
Block/Thread 수가 하드웨어 자원보다 많으면, 스케줄링을 통해 순차적으로 실행됨

CUDA에서 커널 함수 호출 시 블록수와 스레드수를 지정하는 목적

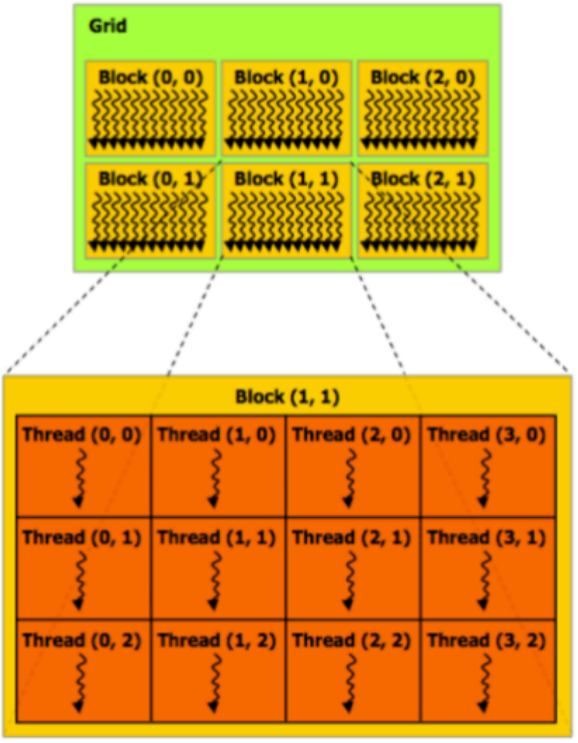
CUDA에서 커널 함수를 호출할 때 `<<<블록수, 스레드수>>>` 와 같이 블록과 스레드의 개수를 지정하는 이유는, 전체 데이터를 처리할 논리적 스레드와 실제 하드웨어 자원(SM, CUDA Core) 간의 **정확한 매핑**을 위해서입니다.
커널 함수가 호출되면, 지정한 블록수 × 스레드수 만큼의 스레드가 생성되어 병렬로 실행됩니다.
각 스레드는 자신의 고유 인덱스(`blockIdx.x`, `threadIdx.x` 등)를 사용해 자신이 처리할 데이터를 결정합니다.
예를 들어, 1000개의 데이터를 처리할 때 `<<<10, 100>>>` 로 호출하면 10개의 Block, 각 Block에 100개의 Thread가 생성되어 총 1000개의 스레드가 각각 하나의 데이터를 담당하게 됩니다.
이렇게 하면 데이터와 스레드가 1:1로 정확히 매핑되어, 병렬 처리가 효율적으로 이루어집니다.
하드웨어 자원(SM, CUDA Core) 상황에 따라 스케줄링되어 실행되지만, 논리적으로는 각 데이터 조각에 정확히 하나의 스레드가 할당되는 구조입니다.
즉, 블록수와 스레드수를 지정하는 목적은 **전체 연산을 병렬로 나누고, 각 스레드가 정확히 하나의 데이터 조각을 처리하도록 논리적-물리적 매핑을 명확히 하기 위함**입니다.

CUDA에서 인덱싱을 사용하는 이유

CUDA에서 각 스레드는 `blockIdx`, `threadIdx` 와 같은 인덱스를 사용합니다. 그 이유는 **각 스레드가 자신이 맡은 일을 구분하여, 전체 작업을 효율적으로 나누어 처리하기 위해서**입니다.
인덱싱은 항상 0번부터 시작합니다. 즉, 첫 번째 스레드나 블록의 인덱스는 0입니다.
인덱싱을 통해 각 스레드는 자신이 처리해야 할 데이터의 위치(예: 배열의 인덱스)를 계산할 수 있습니다.
이를 통해 중복 없이, 빠짐없이 모든 데이터에 대해 병렬 처리가 가능합니다.
즉, 인덱싱은 일을 구분해서 각 스레드에 정확히 할당해주는 역할을 합니다.
CUDA에서는 인덱싱을 x, y, z 3차원으로 처리할 수 있습니다. 이는 1차원 배열뿐만 아니라 2차원 이미지, 3차원 볼륨 데이터 등 다양한 데이터 구조에 쉽게 매핑할 수 있도록 설계된 것입니다.
왜 3차원으로 나누는가? 이는 실제로 처리해야 하는 데이터 구조(예: 2D 이미지, 3D 볼륨 등)에 논리적으로 매핑하기 위함입니다. 즉, 데이터의 차원에 맞게 스레드와 블록을 배치할 수 있어 코드가 더 직관적이고 효율적으로 작성됩니다.

CUDA의 gridDim, blockIdx, blockDim, threadIdx 시각적 구조

아래 다이어그램은 CUDA에서 gridDim, blockIdx, blockDim, threadIdx의 관계를 시각적으로 표현한 예시입니다.



- **Grid**: 전체 연산 공간 (여기서는 gridDim.x=2, 즉 Block이 2개)
- **Block**: 각 Block은 blockIdx.x로 구분 (0, 1)
- **BlockDim**: 각 Block 안에 포함된 Thread의 개수 (여기서는 blockDim.x=4)
- **Thread**: 각 Block 안에 threadIdx.x로 구분되는 Thread가 존재 (0~3)

이 구조를 통해 각 스레드가 자신의 인덱스를 이용해 일을 나누어 처리하게 됩니다.

CUDA 프로그래밍의 함수 구조와 흐름

```c
#include <stdio.h>
__device__ void device_strcpy(char *dst, const char *src) {
    while (*dst++ = *src++);
}
__global__ void kernel(char *A) {
    device_strcpy(A, "Hello, World!");
}
int main() {
   char *d_hello;
   char hello[32];
   cudaMalloc((void**)&d_hello, 32);
   kernel<<<1,1>>>(d_hello);
   cudaMemcpy(hello, d_hello, 32, cudaMemcpyDeviceToHost);
   cudaFree(d_hello);
   puts(hello);
}
```

- CUDA 프로그래밍은 기본적으로 C처럼 **모듈(함수) 단위**로 작성된다.
- `main()` 에서 변수 선언, 메모리 할당, 커널 함수 호출 등 전체 흐름을 제어한다.
- 커널 함수는 `__global__` 로 선언하며, **호스트(Host, CPU)에서 호출**할 수 있고, GPU에서 병렬로 실행된다.
- 커널 함수가 호출되는 순간, 지정한 블록수와 스레드수만큼 논리적 스레드가 생성되어 각 코어에서 병렬로 실행된다.
- 커널 함수 내부에는 실제 연산(컨텍스트)이 들어가며, 이 안에서 디바이스 함수를 호출할 수 있다.
- 디바이스 함수는 `__device__` 로 선언하며, **커널 함수나 다른 디바이스 함수에서만 호출**할 수 있다. 호스트 코드(main 등)에서는 직접 호출할 수 없다.
- 커널 함수와 디바이스 함수의 구분은 다음과 같다:
  - `__global__` : 커널 함수, Host에서 호출, Device에서 실행
  - `__device__` : 디바이스 함수, Device에서만 호출 및 실행

실행 흐름 요약

1. **메인 함수(main)**
   - GPU 메모리 할당 (`cudaMalloc`)
   - 커널 함수 호출 (`kernel<<<1,1>>>(d_hello)`)
   - 결과 복사 (`cudaMemcpy`)
   - 메모리 해제 (`cudaFree`)
2. **커널 함수**
   - 실제 연산(여기서는 문자열 복사)을 수행
   - 필요시 디바이스 함수 호출

컨텍스트(Context)

커널 함수 내부에서 실행되는 코드(연산 내용)가 바로 **컨텍스트**이다.
각 스레드는 자신만의 컨텍스트(실행 환경)에서 독립적으로 연산을 수행한다.

CUDA에서 호스트 코드와 디바이스 코드의 분리 및 동기화

1. 호스트 코드와 디바이스 코드의 분리

호스트 코드(.c)
- CPU에서 실행되는 코드
- 프로그램의 전체 흐름 제어, 메모리 할당, 커널 함수 호출 등 담당

디바이스 코드(.cu, .cuh)
- GPU에서 실행되는 코드
- 대규모 병렬 연산, 데이터 처리 등 고속 연산 담당

컴파일 과정
- 호스트 코드와 디바이스 코드는 각각 분리되어 컴파일
- 최종적으로 하나의 실행 파일로 결합되어, CPU와 GPU가 협력하여 동작

2. 실행 환경의 차이

호스트 코드는 CPU에서, 디바이스 코드는 GPU에서 실행
서로 다른 하드웨어에서 각각의 역할을 수행

3. 동기화(Synchronization)의 필요성

CPU와 GPU는 독립적으로 동작
데이터 일관성 및 실행 순서 보장을 위해 동기화 필요
예시: GPU 연산이 끝나기 전에 CPU가 결과를 사용하면 오류 발생
대표적 동기화 함수: `cudaDeviceSynchronize()`

4. 예시 코드

```
#include <stdio.h>
__global__ void kernel(int *A) {
    int idx = threadIdx.x;
    A[idx] = idx * idx;
}
int main() {
    int *d_A, A[10];
    cudaMalloc((void**)&d_A, sizeof(int)*10);
    kernel<<<1,10>>>(d_A);
    cudaDeviceSynchronize(); // 동기화: GPU 연산이 끝날 때까지 대기
    cudaMemcpy(A, d_A, sizeof(int)*10, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    for(int i=0; i<10; i++) printf("%d ", A[i]);
}
```

`cudaDeviceSynchronize()` 를 통해 CPU와 GPU의 실행 순서를 맞춤

5. 요약

호스트 코드와 디바이스 코드는 분리되어 작성 및 컴파일
각각 CPU, GPU에서 실행
동기화로 데이터 일관성과 실행 순서 보장

CUDA와 그래픽 파이프라인

그래픽 파이프라인이란?

- 그래픽 파이프라인(Graphics Pipeline)은 3D 그래픽스에서 3차원 장면을 2차원 화면에 그리기까지의 일련의 처리 단계입니다.
- GPU에서 매우 중요한 역할을 하며, 각 단계는 병렬적으로 처리되어 고속 렌더링이 가능합니다.

주요 단계

1. **Vertex Processing (정점 처리)**: 3D 모델의 정점 데이터를 변환(이동, 회전, 투영 등)하며, 정점 셰이더(Vertex Shader)가 실행됨
2. **Primitive Assembly & Rasterization (프리미티브 조립 및 래스터화)**: 정점들을 삼각형, 선 등 기본 도형으로 조립하고, 3D 도형을 2D 픽셀로 변환
3. **Fragment Processing (프래그먼트 처리)**: 각 픽셀에 대해 색상, 텍스처, 조명 효과 등을 계산하며, 프래그먼트 셰이더(Fragment Shader)가 실행됨
4. **Output Merging (출력 병합)**: 최종적으로 계산된 픽셀 데이터를 프레임버퍼에 저장

CUDA와 그래픽 파이프라인의 관계

- CUDA는 본래 범용(GPGPU) 연산을 위해 설계된 NVIDIA의 병렬 컴퓨팅 플랫폼입니다.
- 그래픽 파이프라인은 OpenGL, DirectX, Vulkan 등 그래픽 API에서 GPU를 통해 자동으로 처리되는 전통적인 렌더링 방식입니다.
- CUDA는 그래픽 파이프라인의 각 단계(특히 래스터화 이후 픽셀 처리 등)를 직접 제어하지 않습니다.
- 대신, 그래픽 파이프라인과 별도로 GPU의 연산 자원을 직접 활용하여 대규모 데이터 병렬 처리를 수행할 수 있습니다.

요약

- 그래픽 파이프라인: 3D 그래픽스 렌더링을 위한 GPU의 전통적 처리 흐름(정점 → 래스터화 → 프래그먼트 → 출력)
- CUDA: 그래픽 파이프라인과 별개로, GPU의 병렬 연산 능력을 범용 계산(GPGPU)에 활용하는 기술

구분	그래픽 파이프라인 (OpenGL/DirectX)	CUDA (GPGPU)
목적	3D 그래픽 렌더링	범용 병렬 연산
프로그래밍 모델	셰이더(Shader) 기반	커널(Kernel) 기반
주요 처리 단계	정점, 래스터화, 프래그먼트 등	데이터 병렬 처리
결과	화면에 이미지 출력	메모리/파일 등 데이터

CUDA에서 인덱싱과 일 분할, 메모리 관리의 핵심

1. 인덱싱을 통한 병렬 작업 분할

- CUDA 프로그램은 대부분 인덱싱(`blockIdx` , `threadIdx` 등)을 통해 각 스레드가 맡을 일을 나눔
- 각 스레드는 자신만의 인덱스를 이용해 전체 데이터를 나누어 처리
- 병렬 처리를 극대화하기 위해 일을 잘게 쪼개는 것이 일반적

2. 일을 너무 많이/적게 나눌 때의 문제

- **너무 잘게 나누면**
  - 스레드 간 통신/동기화 비용이 커지고, 오버헤드가 증가해 오히려 성능 저하
  - 각 스레드가 처리하는 데이터가 너무 작으면, 연산보다 통신/관리 비용이 더 커짐
- **너무 적게 나누면**
  - GPU의 많은 코어를 충분히 활용하지 못해 자원이 낭비되고, 전체 성능이 떨어짐
- → 적절한 분할이 중요

3. CUDA에서 메모리 관리의 중요성

- CUDA 연산의 속도와 효율을 결정하는 가장 핵심은 메모리 관리
- 연산을 위해서는 반드시 데이터를 GPU로 가져와야 하고, 연산 결과도 다시 저장해야 함
- 최종적으로 결과를 호스트(메인 메모리)로 복사해 네트워크 전송 등 후처리

4. CUDA의 메모리 구조

- **글로벌 메모리(Global Memory)**: GPU 전체에서 접근 가능한 메모리, 대용량이지만 접근 속도가 느림, 함수 안/밖에서 선언 가능
- **공유 메모리(Shared Memory)**: 블록 내 모든 스레드가 공유, 접근 속도가 빠름, `__shared__` 키워드로 선언
- **로컬 메모리(Local Memory)**: 각 스레드만 접근 가능, 실제로는 글로벌 메모리에 할당될 수 있음
- **레지스터(Register)**: 가장 빠른 임시 저장소, 스레드별로 할당, 넘치면 로컬 메모리로 spill

5. 메모리 관리와 연산 최적화

- 공유 메모리를 잘 활용하면 연산 속도를 크게 높일 수 있음
- 글로벌 메모리 접근을 최소화하고, 필요한 데이터만 공유 메모리로 옮겨 연산
- 동기화(synchronization)를 통해 데이터 일관성 유지 (`__syncthreads()` 등)

6. CUDA API의 동기화

호스트에서 동기화 명령(`cudaDeviceSynchronize()`)을 내리면, 모든 스레드가 연산을 마칠 때까지 대기
비동기(Async) API는 함수명에 `Async` 가 붙음

7. 커널 함수와 인자 전달

커널 함수의 인자는 글로벌 메모리의 포인터로 전달
인풋 데이터를 공유 메모리로 복사해 연산 후, 결과를 다시 글로벌 메모리에 저장
연산이 끝나면 결과를 호스트로 복사

CUDA의 워프(Warp)와 워프 스케줄러

1. 워프(Warp)란?

워프(warp)는 CUDA에서 32개의 스레드를 하나의 그룹으로 묶은 논리적 실행 단위
GPU는 커널 실행 시, 스레드들을 32개씩 묶어서 워프 단위로 스케줄링하고 실행

2. 워프 스케줄러란?

워프 스케줄러(warp scheduler)는 여러 워프 중에서 어떤 워프를 실행할지 결정하는 하드웨어 유닛
각 SM(Streaming Multiprocessor)에는 여러 워프 스케줄러가 존재할 수 있음

3. 왜 32개씩 나열되어 있을까?

- 논리적으로 32개씩 처리하는 이유는, NVIDIA GPU의 아키텍처가 워프 단위(32개)로 명령을 발행하고, 효율적으로 병렬 처리를 하기 위해서임
- 워프 내의 32개 스레드는 동일한 명령어를 동시에 실행(SIMT, Single Instruction Multiple Thread)

4. 물리적 실행과의 차이

- 실제 하드웨어(예: CUDA Core)는 한 번에 32개 스레드를 모두 처리하지 못할 수도 있음
- 예를 들어, 물리적으로 16개의 CUDA Core만 있다면, 32개 워프 스레드를 2번에 나누어 (16+16) 실행
- 즉, 논리적으로는 32개씩 묶어서 처리하지만, 물리적으로는 16개씩 계산이 이루어짐

5. 요약

- 워프: 32개 스레드의 논리적 묶음, 명령어 발행 단위
- 워프 스케줄러: 여러 워프 중 실행할 워프를 선택
- 왜 32개? 아키텍처의 효율성과 병렬성 극대화, SIMT 모델 때문
- 물리적 실행: 실제 코어 수에 따라 32개 워프를 여러 번에 나누어 실행

예시 그림

```
[논리적 워프]   [실제 실행]
스레드 0~31  →  0~15 (1차 실행)
               16~31 (2차 실행)
```