OpenCL 입문

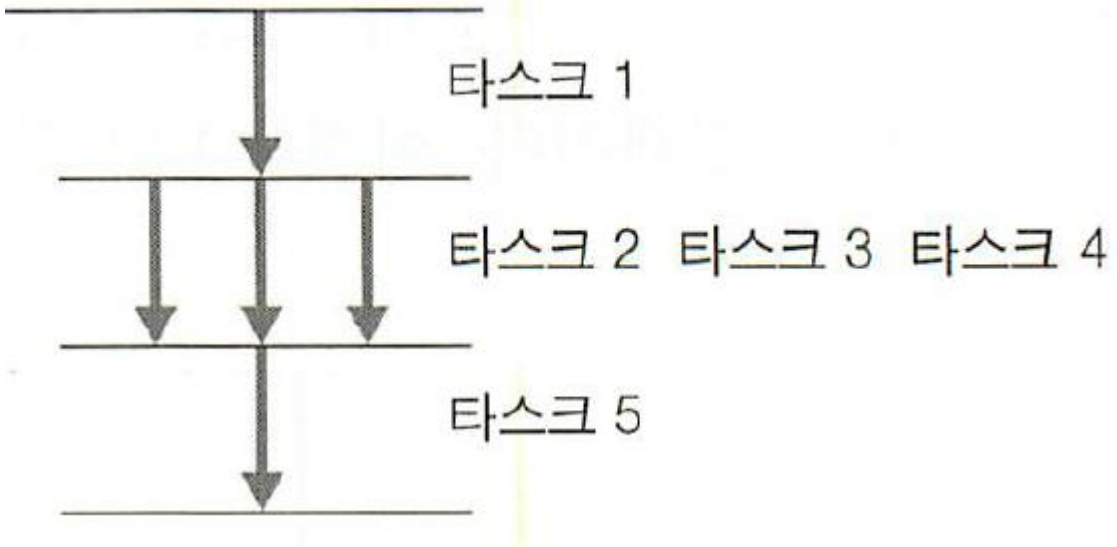
v0.1-beta

#1. OpenCL 개론

병렬 컴퓨팅

- 문제 해결을 위해 필요한 처리를 동시에 실행하는 것이 가능한 작은 타스크로 분할하고, 이 타스크를 복수의 CPU에서 병렬로 실행하는 것으로 전체 처리 시간을 단축, 처리의 효율화를 가질 수 있는 기법
- 각각의 타스크가 다른 타스크의 실행 결과에 의존하지 않는 경우 타스크 그룹으로 분해하는 것이 가능함

타스크1	a = 1
타스크1	b = a * 10
타스크3	c = a * 100
타스크4	d = a * 1000
타스크5	e = a + b + c + d

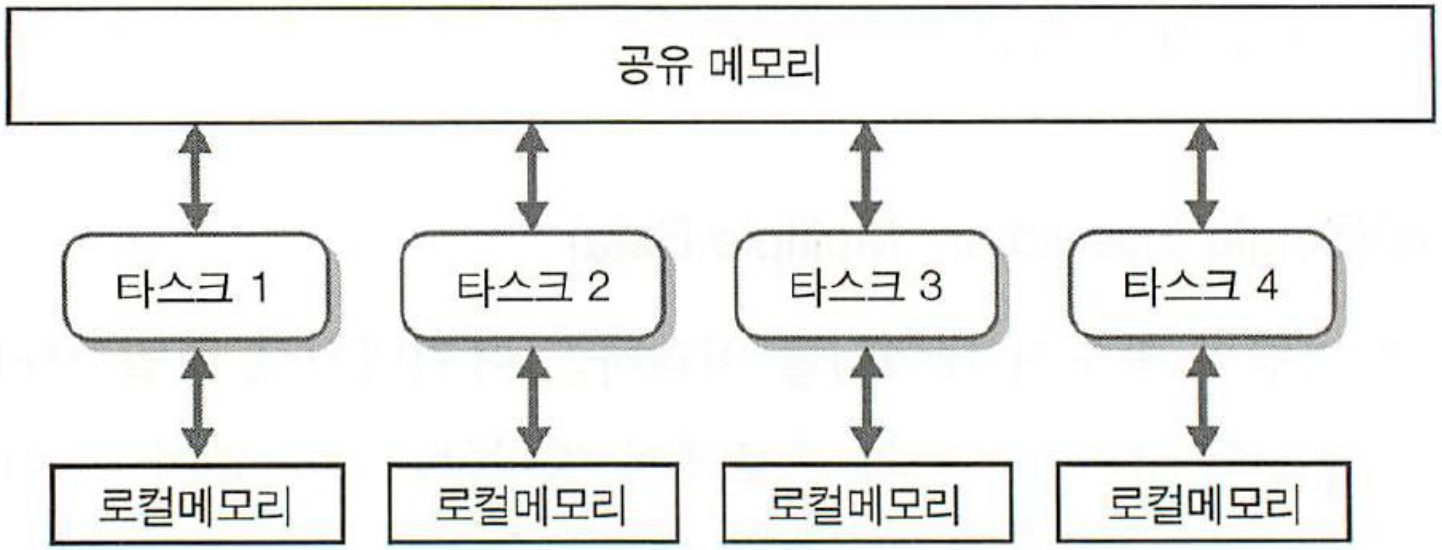


Flynn의 분류

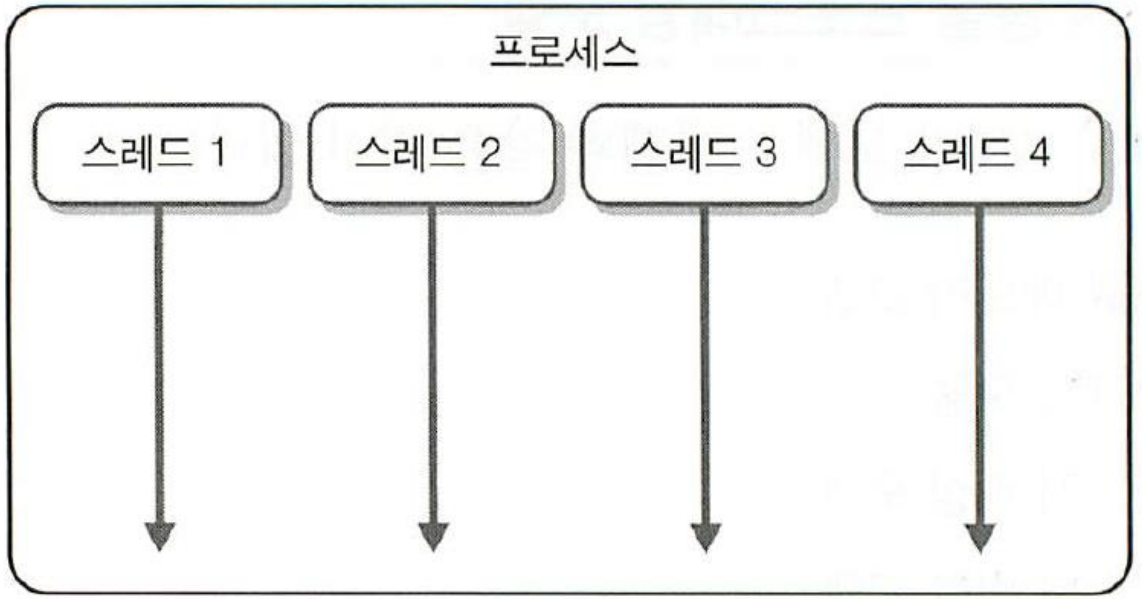
- 컴퓨터 아키텍처를 명령(Instruction)과 데이터(Data)의 2가지 독립된 파라미터를 사용하여 다음과 같이 4가지로 분류
  - ✓SISD(Single Inst., Single Data)
    - 우리의 PC
  - ✓SIMD(Single Inst., Multiple Data)
    - 한 개의 명령이 복수의 데이터를 처리
    - SSE, 3DNow!, OpenCL C언어로 정의된 벡터
  - ✓MISD(Multiple Inst., Single Data)
    - 상식적이지 않음
  - ✓MIMD(Multiple Inst., Multiple Data)
    - 복수의 명령이 복수의 데이터를 처리
    - Sparc 컴퓨터, 네트워크 분산형 컴퓨터, 대칭형 멀티프로세싱 컴퓨터, OpenCL

병렬 프로그래밍 모델

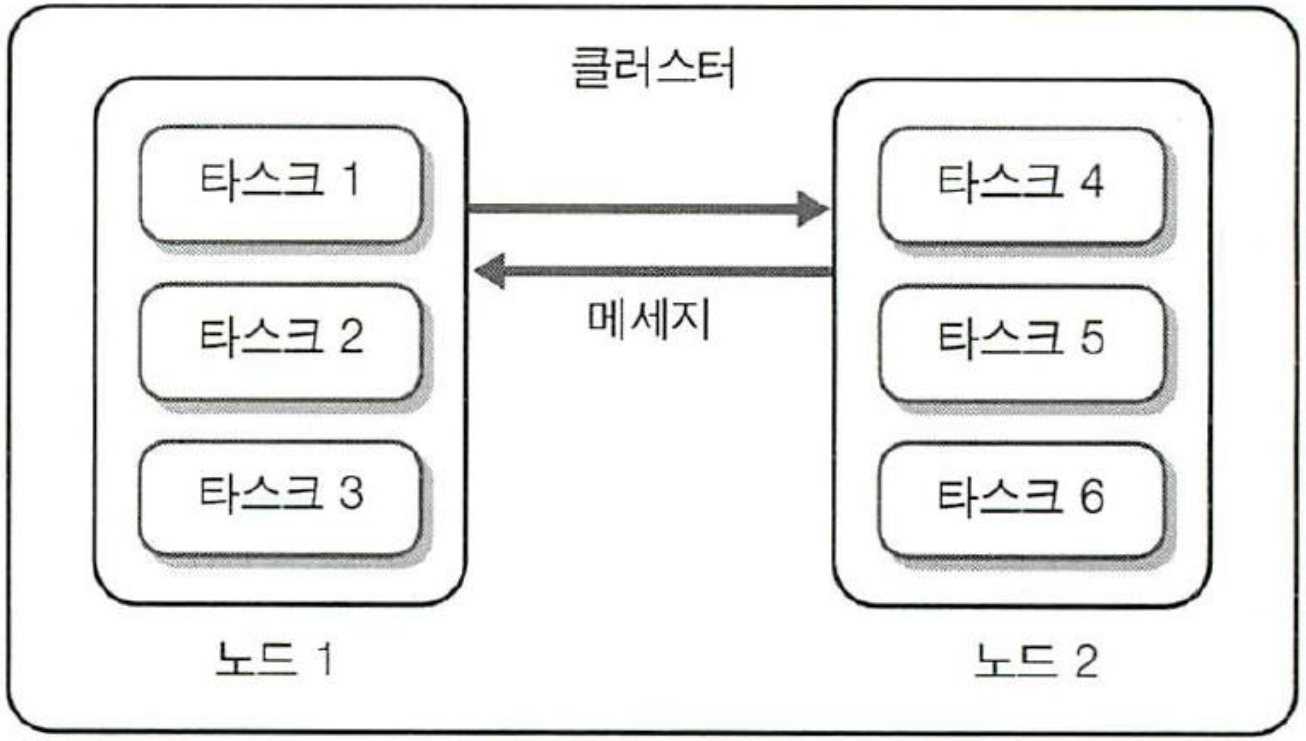
- 공유 메모리
  - ✓동일 어드레스 공간을 공유하는 복수 개의 타스크를 병렬 실행하는 모델, 메모리 엑세스의 경합이 이뤄지기 때문에 lock 및 unlock, 세마포어 등으로 배타적 처리를 수행할 필요가 있음, 또한 로컬 메모리에서 데이터를 캐쉬하는 것과 같은 것이 필요



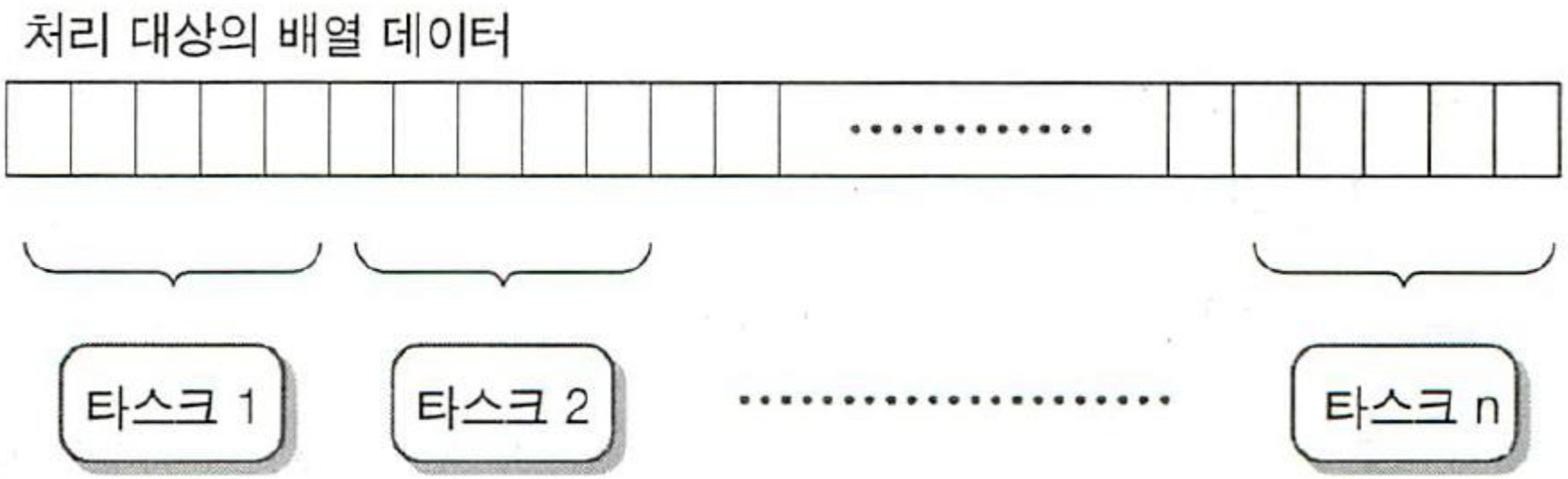
- 쓰레드 모델
  - ✓1개의 프로세스에서 복수 스레드를 이용하여 타스크를 병렬 실행하는 모델, 각 스레드는 프로세스 중의 리소스를 공유, 공유 메모리 모델과 같은 스레드간의 리소스 액세스 경합에 대해 배타적으로 처리할 필요가 있음
  - ✓POSIX의 스레드용 API, OpenMP를 사용하는 방법이 있음



- 메시지 패싱 모델
  - ✓네트워크에 접속된 복수 컴퓨터(Node) 사이에서 메시지를 교환하여 복수 타스크를 실행, 노드 사이에서 교환하는 메시지는 MPI로 표준화되고 있음



- 데이터 병렬 모델
  - ✓각 타스크가 데이터를 주고 받는 범위를 결정하여 데이터의 각 부분을 처리하는 모델, 각 타스크가 처리하는 인덱스 범위를 결정하여 처리를 실행, OpenCL과 CUDA를 사용한 프로그래밍의 경우, 데이터 병렬 모델에서 적절하게 표현하는 것이 가능



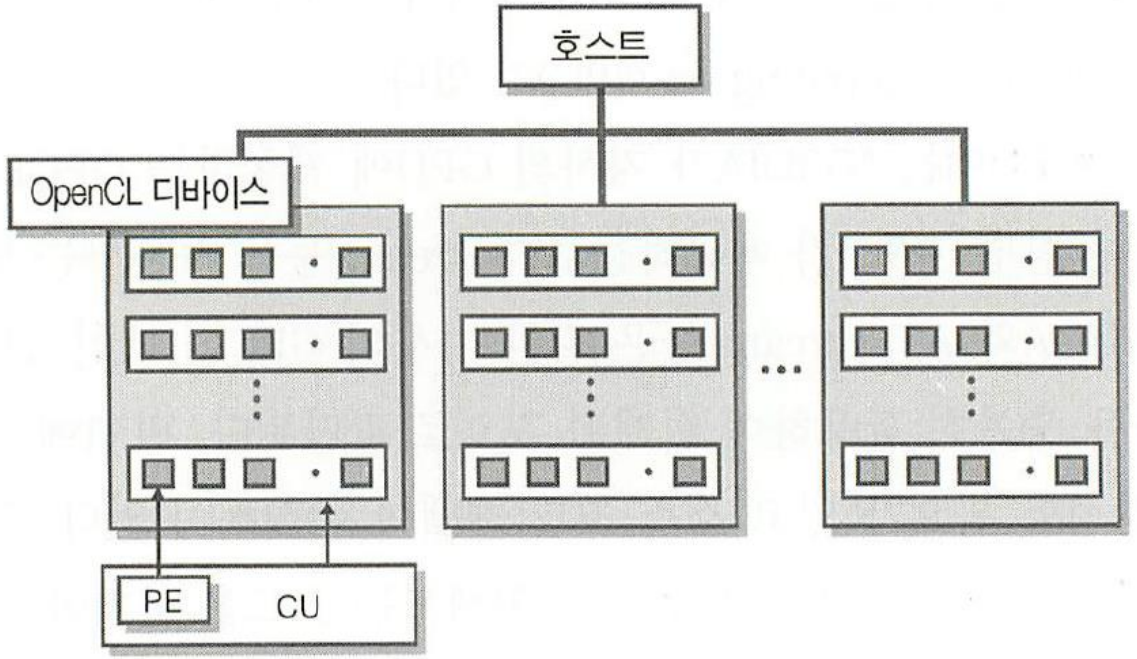
- 파이프라인 모델
  - ✓앞에서 설명한 모델에서 2개 이상의 모델을 조합한 모델
  - ✓MPI와 POSIX 스레드, MPI와 OpenMP 조합 등이 고려됨

OpenCL

- 병렬 컴퓨팅을 실행하는 기술
  - ✓OpenMP
    - 컴파일러가 소스 코드에 기술된 정보를 사용하여 멀티 코어용으로 병렬화된 코드를 자동적으로 생성하는 기술
    - 쓰레드를 이용한 병렬화, Visual C++과 gcc를 이용하여 간단히 실현하는 것이 가능함
  - ✓GPGPU
    - GPU의 계산 성능을 본래 목적인 그래픽스 계산 외에도 사용하는 방법
    - CUDA, ATI Stream 등을 사용하는 방법
    - 소스코드 레벨에서 이식성이 없음
  - ✓OpenCL
    - 플랫폼에 의존하지 않는 어플리케이션을 작성하기 위한 프레임워크
    - 애플에서 제안

OpenCL 플랫폼

- OpenCL 플랫폼은 1개의 호스트 및 1개 이상의 계산 디바이스로 구성, 한 개의 계산 디바이스는 복수의 CU(Computation Unit)로 구성되고, 한 개의 CU는 복수의 PE(Processing Unit)로 구성



OpenCL 프레임워크

- OpenCL 프레임워크
  - ✓디바이스 드라이버
  - ✓커널용 컴파일러
  - ✓API용 include 파일
  - ✓API 런타임 라이브러리

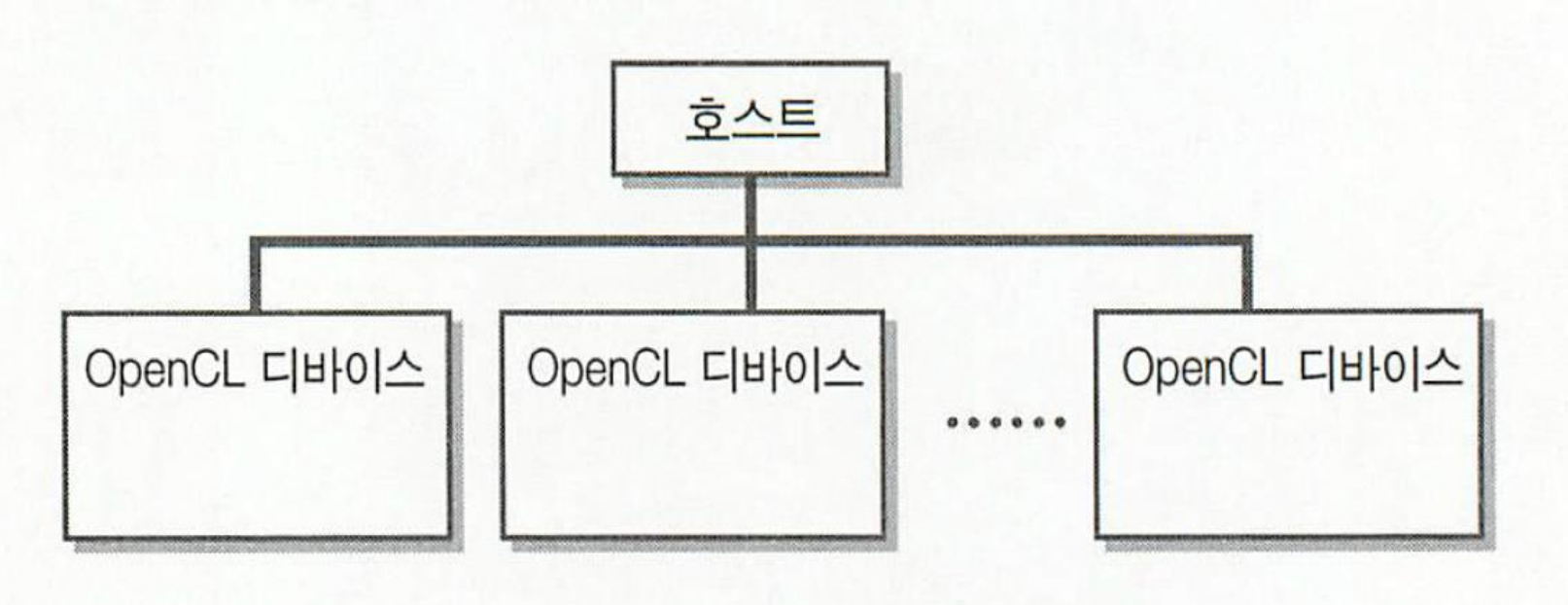
#2. OpenCL 아키텍쳐

OpenCL **아키텍처**

- 플랫폼 모델
- 실행 모델
- 메모리 모델
- 프로그래밍 모델

플랫폼 모델

- OpenCL 어플리케이션을 실행하기 위해 플랫폼을 규정하는 것이 OpenCL 플랫폼 모델
  - ✓OpenCL 어플리케이션을 실행하기 위한 '하드웨어'를 '추상화' 한 것
- "OpenCL 어플리케이션을 작성한다"라는 뜻은 OpenCL 플랫폼 상에서 동작하는 프로그램을 작성하는 것

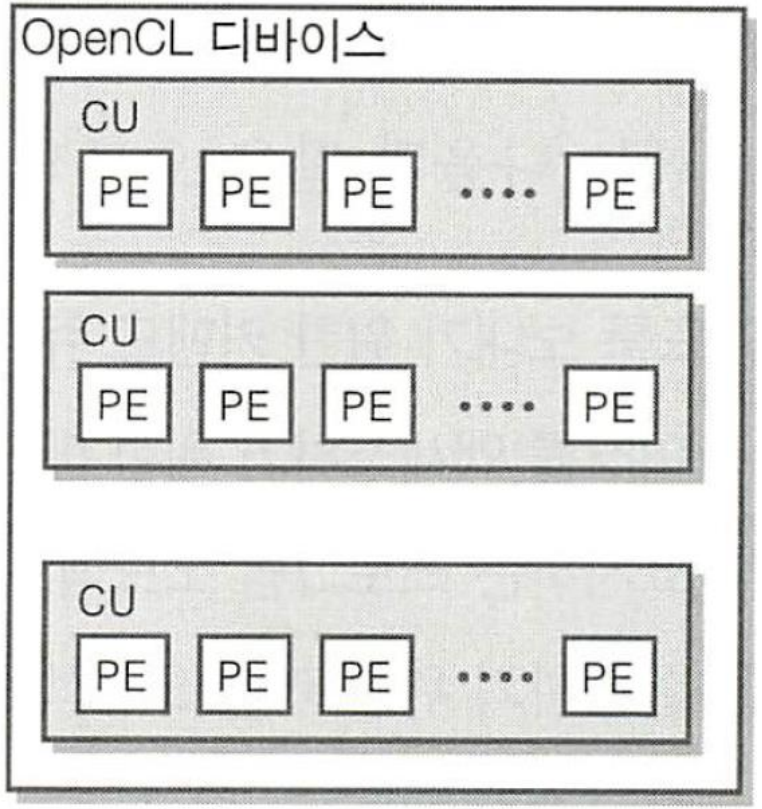


플랫폼 모델 – 호스트와 디바이스

- 호스트
  - ✓OpenCL 어플리케이션을 실행하는 컴퓨터 본체
  - ✓호스트 CPU에서 메인 프로그램(전체 흐름을 제어)
  - ✓C/C++ 언어를 사용해서 작성
  - ✓OpenCL API를 사용해서 OpenCL 디바이스에 커맨드를 전달 및 계산
- 디바이스
  - ✓호스트에 접속된 계산 디바이스로 CPU, GPU, Cell, DSP 등을 나타냄
  - ✓디바이스는 계산에 '집중'
  - ✓하드웨어를 사용하기 위해서 OpenCL 프레임워크가 필요(일반적으로 하드웨어 벤더로부터 제공됨)

플랫폼 모델 – 디바이스

- 디바이스는 구성
  - ✓1개 이상의 계산 유닛(CU)으로 구성
  - ✓CU는 1개 이상의 처리 엘리먼트(PE)
  - ✓실제 계산은 PE에서 진행, PE는 병렬동작함
- OpenCL 계산 수행을 수행하기 위해서 '커널'이라 불리는 계산 프로그램을 작성하고 '호스트'에 의해 빌드하고, 디바이스에 다운로드함
- 커널은 PE에서 병렬 진행됨



실행모델 – 호스트와 커널

- 호스트에서 동작하는 프로그램 ➔ '호스트 프로그램'
  - ✓호스트 프로그램은 '컨텍스트'라 불리는 '오브젝트'를 'OpenCL API'를 사용해서 작성
    - "컨텍스트, 오브젝트, OpenCL API" 세 가지 단어를 꼭 기억하자!
  - ✓컨텍스트는 다음과 같은 오브젝트를 정의(OpenCL API로)
    - 디바이스에 커맨드를 보내기 위한 커맨드 큐
    - 디바이스 상의 데이터를 액세스하기 위한 메모리 오브젝트
    - 커널 프로그램을 표시하는 프로그램 오브젝트
    - 커널 프로그램을 실행하기 위한 커널 오브젝트
- 디바이스에서 동작하는 프로그램 ➔ '커널'

	호스트 프로그램	커널
역할	OpenCL 어플리케이션 전체를 기술	계산 프로그램 기술
동작하는 장소	호스트 CPU	디바이스 PE
소스 프로그램 언어	C 혹은 C++언어 OpenCL API를 사용	OpenCL C 언어
빌드 방법	일반 프로그램 빌드 방식과 동일	OpenCL API를 사용하여 호스트 프로그램 실행 시에 빌드(사전에 빌드해 놓은 것도 가능)

실행모델 – 인덱스 공간

- 호스트 프로그램이 커널의 실행을 스케줄할 때 인덱스 공간이 정의됨
  - ✓커널의 인스턴스를 '워크 아이템'이라고 함
    - 커널의 실행은 디바이스의 PE에서 실행됨을 기억하자
  - ✓'워크 아이템'은 인덱스 공간의 각 지점에서 실행
    - 이란 말은 '인덱스 공간'은 디바이스에 존재한다는 뜻
- OpenCL 1.0에서 지원하는 인덱스 공간은 NDRange라 불림
  - ✓N차원의 인덱스 공간을 나타냄

워크 아이템	커널의 인스턴스. 디바이스의 PE 상에서 실행
워크 그룹	복수 워크 아이템을 합한 것 같은 그룹에 속하는 워크 아이템은 같은 CU 상에서 실행
글로벌 ID	인덱스 공간에서 정해지는 ID. 워크 아이템에 할당
워크 그룹 ID	인덱스 공간에서 정해지는 ID. 워크 그룹에 할당
로컬 ID	워크 그룹내에서 정해지는 ID. 워크 아이템에 할당

N=1의 경우, n개의 워크 아이템을 실행 :

```
for (i = 0; i < n; i++)  {
        // 워크 아이템(i)을 실행
}
```

N=2의 경우, n * m개의 워크 아이템을 실행

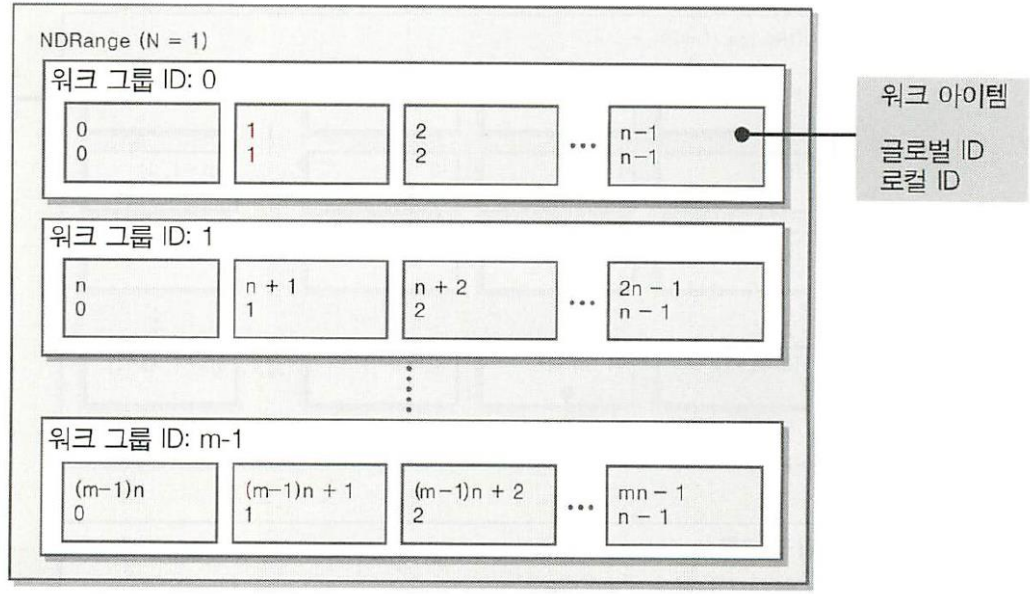
```
for (i = 0; i < n; i++)  {
        for (j = 0; j < m; j++)  {
          // 워크 아이템(i, j)을 실행
        }
}
```

N=3인 경우, n * m * l개의 워크 아이템을 실행

```
for (i = 0; i < n; i++)  {
      for (j = 0; j < m; j++)  {
            for (k = 0; k < l; k++)  {
                // 워크 아이템(i, j, k)을 실행
            }
      }
}
```

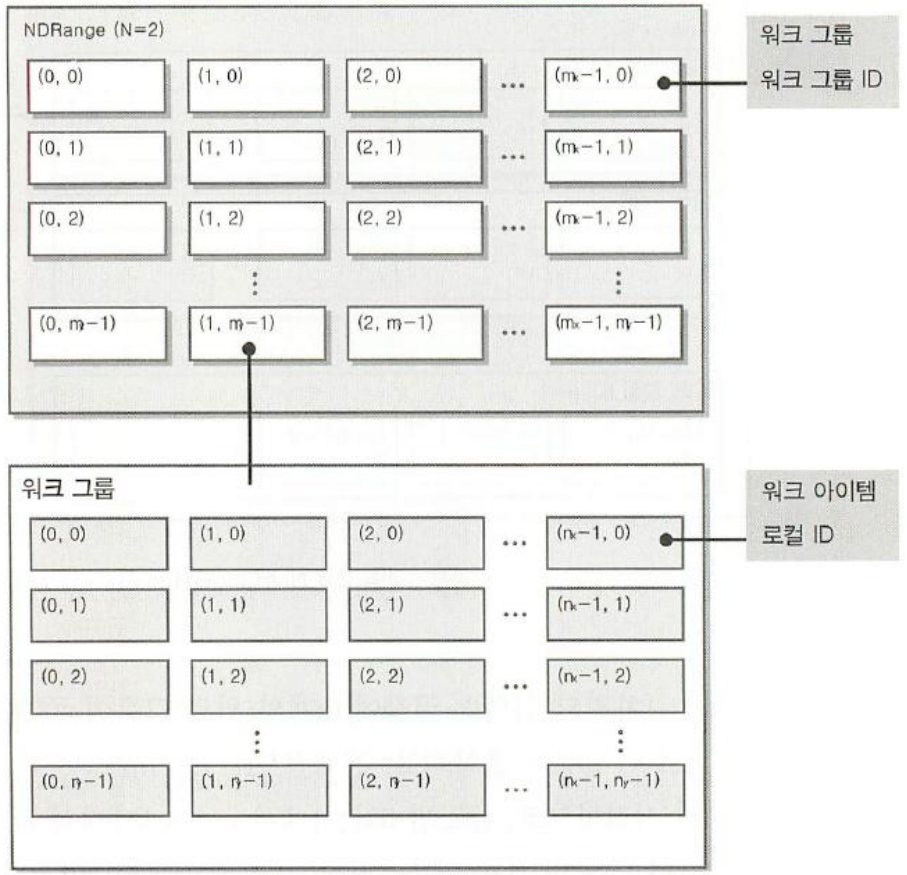
NDRange의 특징

- 몇차원인가?
  - ✓1차원, 1DRange
- 몇 개의 워크 그룹이 존재하는가?
  - ✓m개
- 워크 그룹은 몇 개의 워크 아이템이 존재하는가?
  - ✓n개
- 전체적으로 몇 개의 워크 아이템이 존재하는가?
  - ✓m*n
- 인덱스는?
  - ✓m*n-1



Ex) NDRange를 **이해해 보자**!

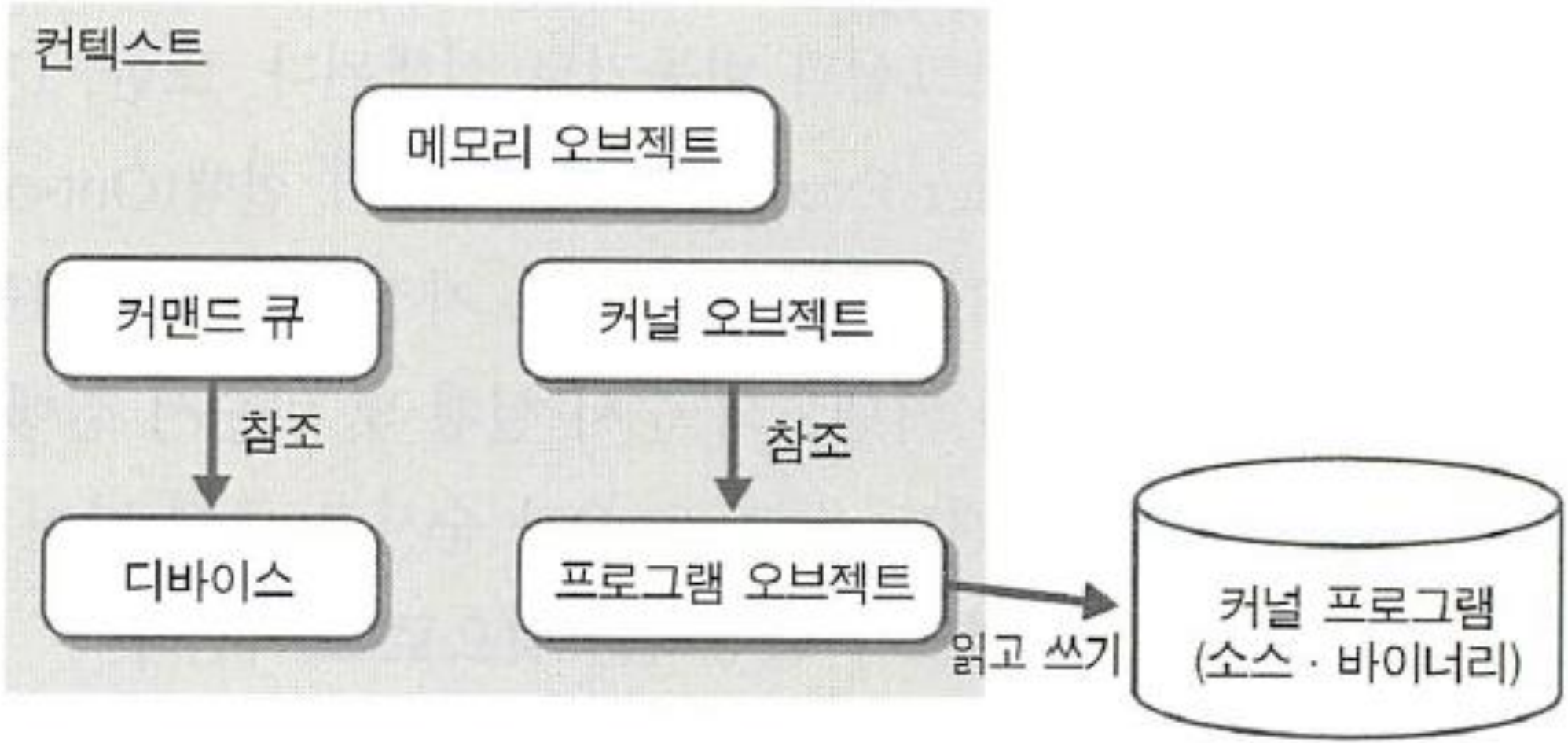
- 몇차원인가?
  - ✓
- 몇 개의 워크 그룹이 존재하는가?
  - ✓
- 워크 그룹은 몇 개의 워크 아이템이 존재하는가?
  - ✓
- 전체적으로 몇 개의 워크 아이템이 존재하는가?
  - ✓
- 인덱스는?
  - ✓



실행모델 – 컨텍스트

- 컨텍스트는 '호스트 프로그램'에 의해 작성되고, 커널을 실행하기 위한 환경
  - ✓호스트 프로그램이 OpenCL API를 사용하여 컨텍스트를 작성
  - ✓C/C++를 사용해서 '전체 프로그램 흐름과 구조'를 작성

오브젝트	설명
커맨드 큐	디바이스에 커맨드를 보내기 위한 큐
메모리	디바이스상의 데이터를 액세스하기 위한 오브젝트
프로그램	커널을 실행하는 프로그램 소스 및 실행 형식
커널	OpenCL 디바이스에서 동작하는 프로그램

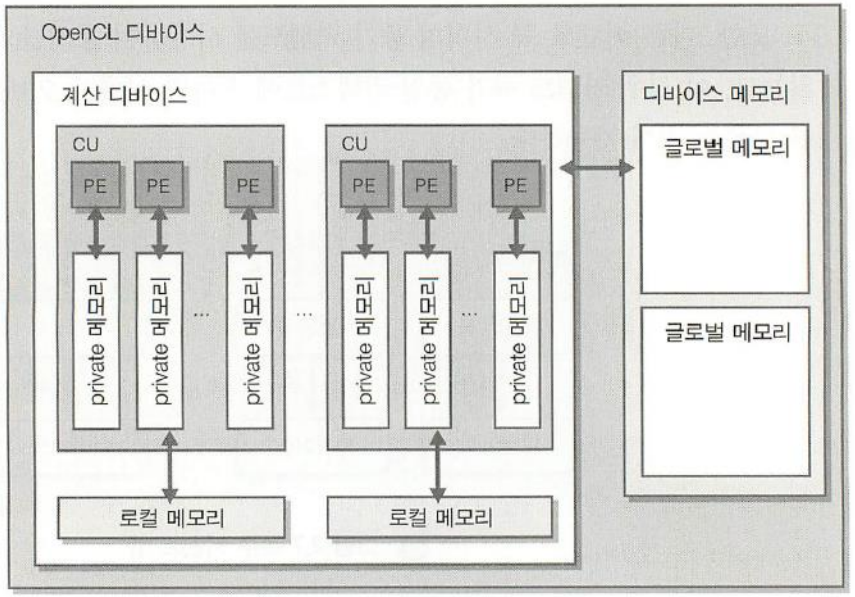

컨텍스트
메모리 오브젝트
커맨드 큐
참조
디바이스
커널 오브젝트
참조
프로그램 오브젝트
읽고 쓰기
커널 프로그램
(소스 · 바이너리)

실행모델 – 커맨드 큐

- 호스트 프로그램은 큐에 커맨드를 발생하여 디바이스를 제어함
- 커맨드를 디바이스 상에서 실행하기 위해 커맨드를 스케쥴함
  - ✓큐에 넣어진 커맨드는 호스트 프로그램과 비동기로 실행
    - 순서 실행(In-order Execution)
    - 비순서 실행(Out-of-order Execution)
- 컨텍스트(!)가 복수 디바이스를 포함하는 경우 각 디바이스에 대해 커맨드 큐를 작성가능
  - ✓큐에 넣어진 커맨드는 커맨드 큐마다 독립적으로 병렬로 동작
  - ✓동기를 다루는 것도 가능

메모리 모델

- PE(PE는 어디에 속하요?)에서 커널을 실행하는 워크 아이템(!)은 4가지 메모리(이 메모리를 호칭하는 이름은?) 영역으로 액세스 됨
  - ✓글로벌 메모리 : 모든 워크 그룹내에 속하는 모든 워크 아이템이 읽고 쓰여지는 메모리 영역
    - Constant 메모리 : 글로벌 메모리 내에 임의의 영역, '호스트 프로그램이 메모리 오브젝트를 할당하고, 초기화 후 Constant 메모리'에 위치
  - ✓로컬 메모리 : 각 워크 그룹에 대한 로컬 메모리 영역으로 이 영역은 워크 그룹 내 워크 아이템으로 공유
  - ✓Private 메모리 : 워크 아이템에 대한 Private 한 메모리 영역, 다른 워크 아이템에서 참조하는 것은 불가



프로그래밍 모델

- 데이터 병렬, 타스크 병렬 이라는 2가지 프로그래밍 모델을 지원함
  - ✓데이터 병렬은 데이터를 워크 아이템으로 지정하고 병렬로 처리
  - ✓타스크 병렬은 커널의 인스턴스를 인덱스 공간과 무관하게 동작하는 모델로 생각, 복수의 커널 실행 타스크를 큐에 넣고 커널 중에 벡터 형을 사용한 연산을 수행하는 것으로 실현

프로그래밍 모델 - 동기

- 워크 그룹 내에서 워크 아이템 동기화
  - ✓그룹 배리어를 사용, 그룹에 속한 모든 워크 아이템이 barrier 함수를 호출할 때까지 barrier 함수는 반환되지 않음
  - ✓다시말해, barrier 다음에 오는 명령어가 실행되는 시점은 모든 워크 아이템에 대해 barrier 이전에 기술된 명령어가 실행되는 것이 보증
- 컨텍스트 내에서 커맨드 동기
  - ✓커맨드 큐 베리어를 사용하는 방법 v.s. 이벤트를 가지는 방법
    - 커맨드 큐 배리어 ➔ 한 개의 커맨드 큐에서 동기를 다루기 위해서 사용
    - 이벤티를 가지는 방법 ➔ 커맨드 큐가 달라도 동기를 다루는 것이 가능

#3. 프로그램의 작성 순서

샘플 프로그램

- 2개의 float형 배열(in1, in2)에 대해 요소 마다 더하기를 행하고, 결과를 float형 배열(out)에 저장, 배열의 크기는 10000으로 각각의 배열은 적당한 값을 지정

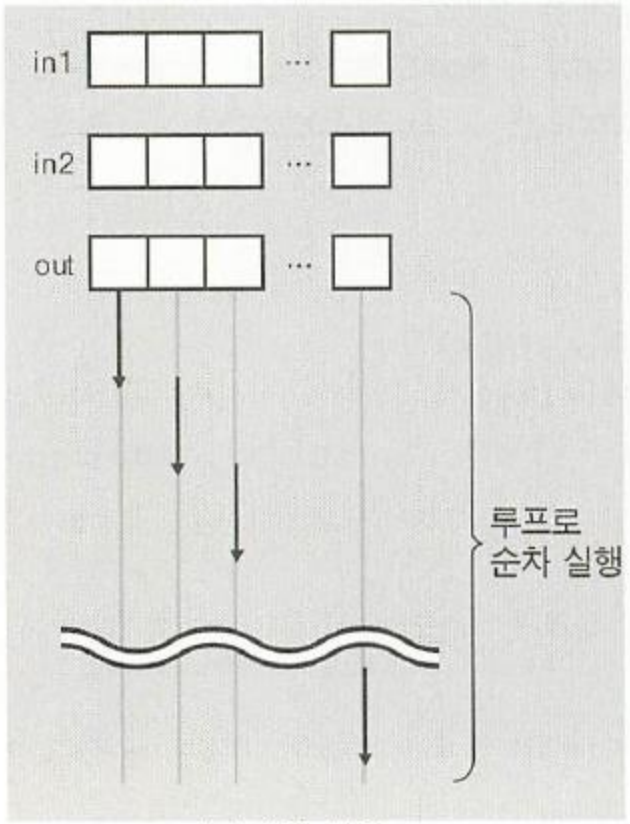
C/C++로 코드 작성 시작

작성시간 최대(10분) 고고!

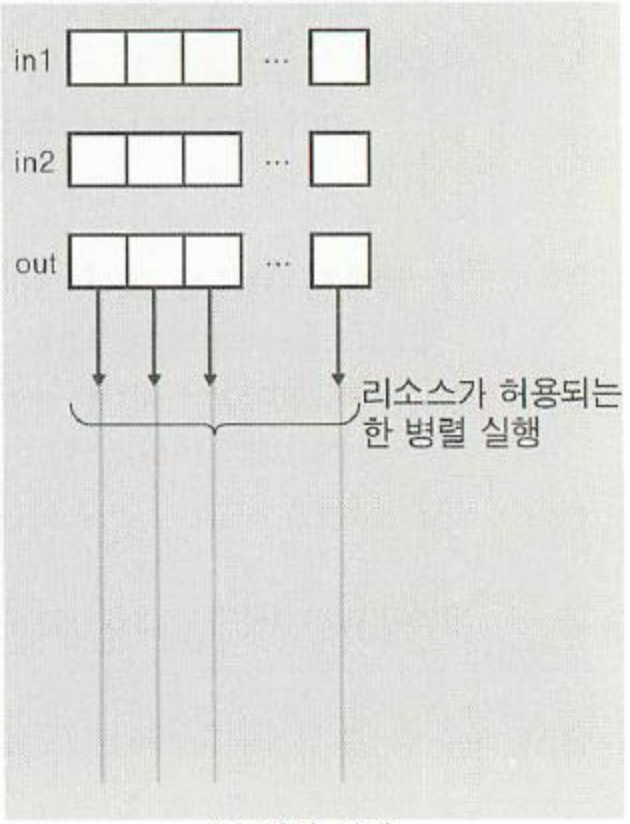
```
#include <stdio.h>
static const size_t NumElements = 10000;

int main() {

    float in1[NumElements];
    float in2[NumElements];
    float out[NumElements];

    // 샘플 값 설정
    for(int i = 0; i < NumElements; i++) {
        in1[i] = float(i) * 10.0f;
        in2[i] = float(i) / 10.0f;
        out[i] = 0.0f;
    } //for i
```

```
    for(int index = 0; index < NumElements; index++) {
         out[index] = in1[index] + in2[index]
    } //for index

    printf("index: in1, in2, out \n");
    for(int i = 0; i < 100; i++) {
         printf("%d: %f, %f, %f \n", i, in1[i], in2[i],
                                                 out[i]);
    } // for i

    return 0;

} // main
```

루프 병렬화



(a) 순차 실행



(b) 병렬 실행

- OpenCL 병렬 처리를 하는 경우 병렬로 수행하는 처리에 대해 커널 함수로 정의하고 이를 워크 아이템 상에서 실행
- 워크 아이템은 인덱스 공간에서 실행되지만 워크 아이템을 인덱스 공간에서 실행하기 위해서는 인덱스 공간의 차원 및 크기를 결정할 필요가 있음
- 인덱스 공간의 차원은 워크 아이템에 할당된 인덱스 차원에 대응, 인덱스 공간의 크기는 워크 아이템이 실행될 횟수에 대응
  - ✓커널 함수에서 수행하는 처리
  - ✓인덱스 공간의 차원
  - ✓워크 아이템 수

병렬화에서 고려할 점	일반적인 해결책	예
커널 함수	루프에서 반복 처리	배열의 한 요소를 가산하는 처리
인덱스 공간의 차원	처리하는 데이터의 차원	1차원
워크 아이템 수	루프의 반복 총수	배열의 요소 수

OpenCL 프로그램 'Flow'

1. 커널 함수 작성
2. 호스트 프로그램 작성
   1) 컨텍스트 작성
   2) 디바이스 얻기
   3) 커맨드 큐 작성
   4) 프로그램 오브젝트 작성
   5) 프로그램 빌드
   6) 커널의 작성
   7) 메모리 오브젝트 작성
   8) 커널에 전달하는 인수 지정
   9) 커널 실행
   10) 결과 취득
   11) 리소스 해방

1. 커널 함수 작성

- 커널 함수 ➔ '디바이스 상에서 병렬로 실행하며 처리하는 것'
- 커널 함수는 OpenCL C 프로그래밍 언어를 사용하여 기술
  - ✓OpenCL C는 C++의 '부분집합'으로 병렬 처리용으로 확장한 언어

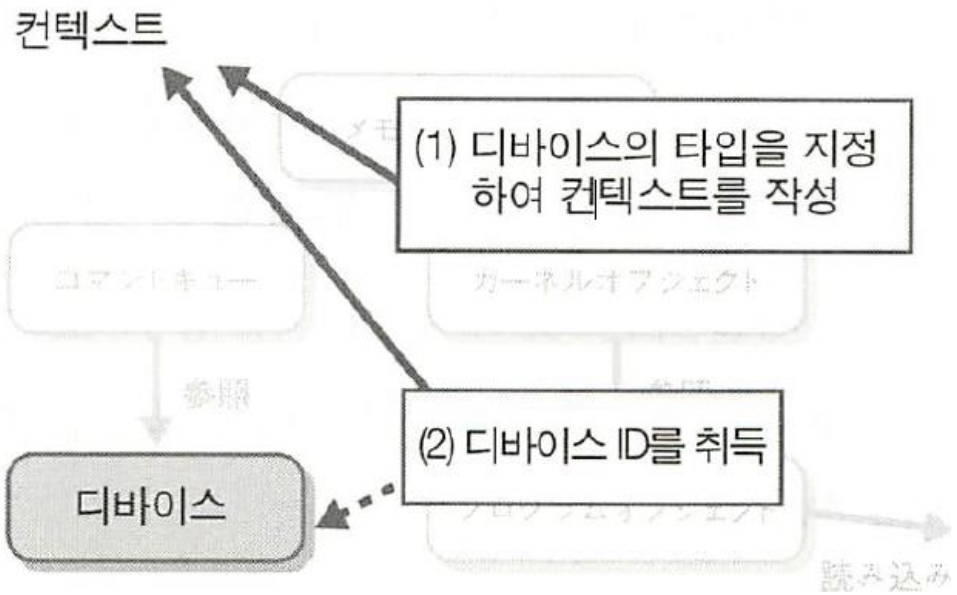
```
__kernel void
addVector(__global const float *int,
          __global const float *in2,
          __global float *out)
{
     int index = get_global_id(0); // 글로벌 ID
     out[index] = in1[index] + in2[index]; // 덧셈
}
```

- __kernel : 커널 함수 정의
- __global : 글로벌 메모리 공간에 배치
  ✓ 모든 워크 아이템에서 참조 가능, 워크 아이템 ID 할당([0] ~ [N-1])

✓ get_global_id() : 글로벌 ID 값을 획득하는 함수

2-1. 컨텍스트 작성

- 계산에 사용하는 디바이스 타입을 지정하고 컨텍스트를 작성
- 디바이스 타입
  - ✓CPU, GPU, 액셀러레이터, 시스템에서 디폴트 디바이스, 모든 디바이스
- 실제로 디바이스를 지정하기 위해서는 다비이스가 OpenCL 디바이스로서 구성될 필요가 있음
- clCreateContextFromType

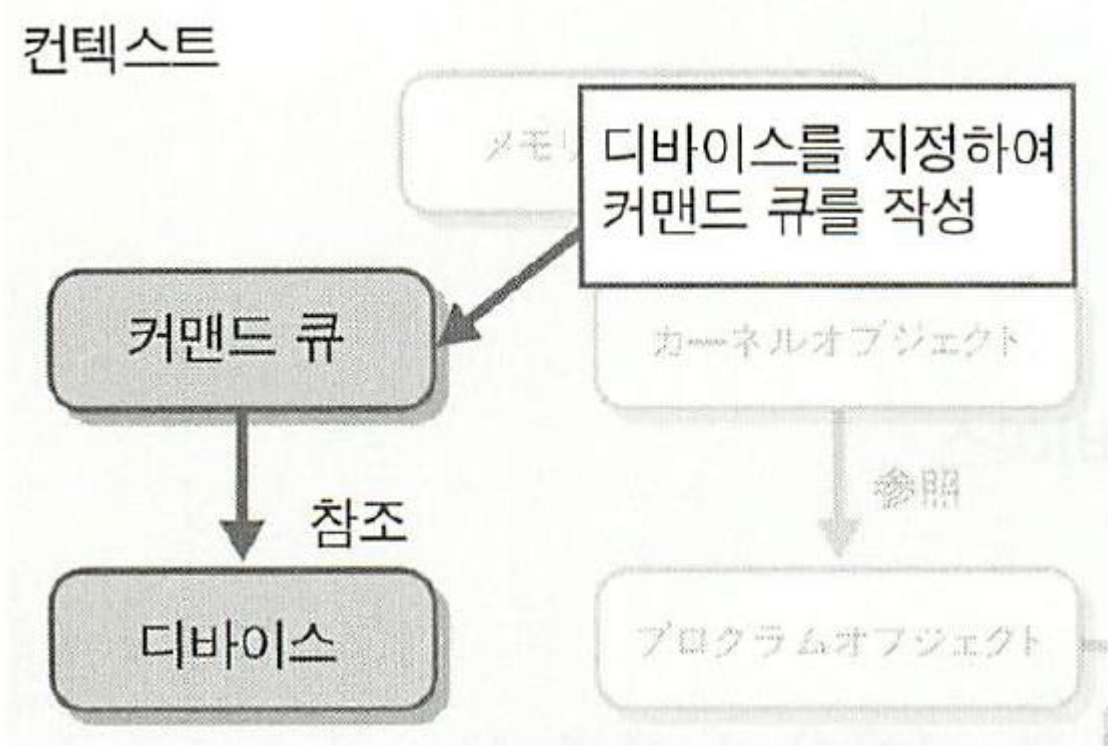


2-2. 디바이스 얻기

- 컨텍스트에 포함되는 디바이스 ID를 획득
- 디바이스 ID는 컨텍스트 정보로 얻는 것이 가능
- clGetContextInfo

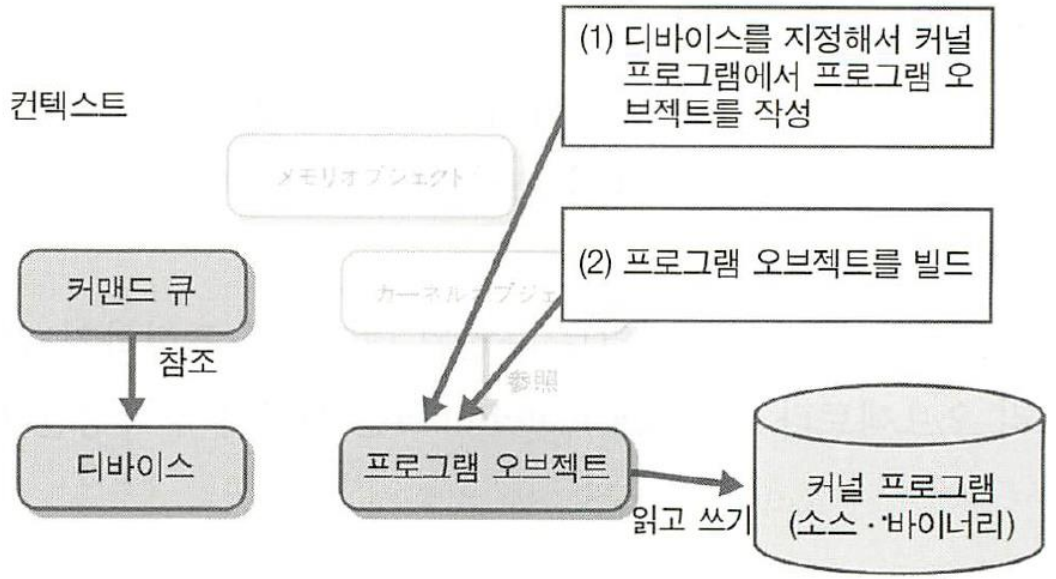
2-3. 커맨드 큐 작성

- 컨텍스트에 대해 커맨드 큐를 작성
- 호스트 프로그램은 커맨드 큐를 사용하여 디바이스 제어
- clCreateCommandQueue



2-4. 프로그램 오브젝트 작성

- OpenCL C언어로 커널의 소스 코드로부터 프로그램 오브젝트를 작성
- 커널의 소스 코드는 호스트 프로그램에 문자열로 채워지는 것이 가능하지만 호스트 프로그램과 다른 파일로 작성하는 것이 좋음
  - ✓분리할 경우, 커널 프로그램을 변경해도 호스트 프로그램 자체는 영향이 없어 편리함
- clCreateProgramWithSource

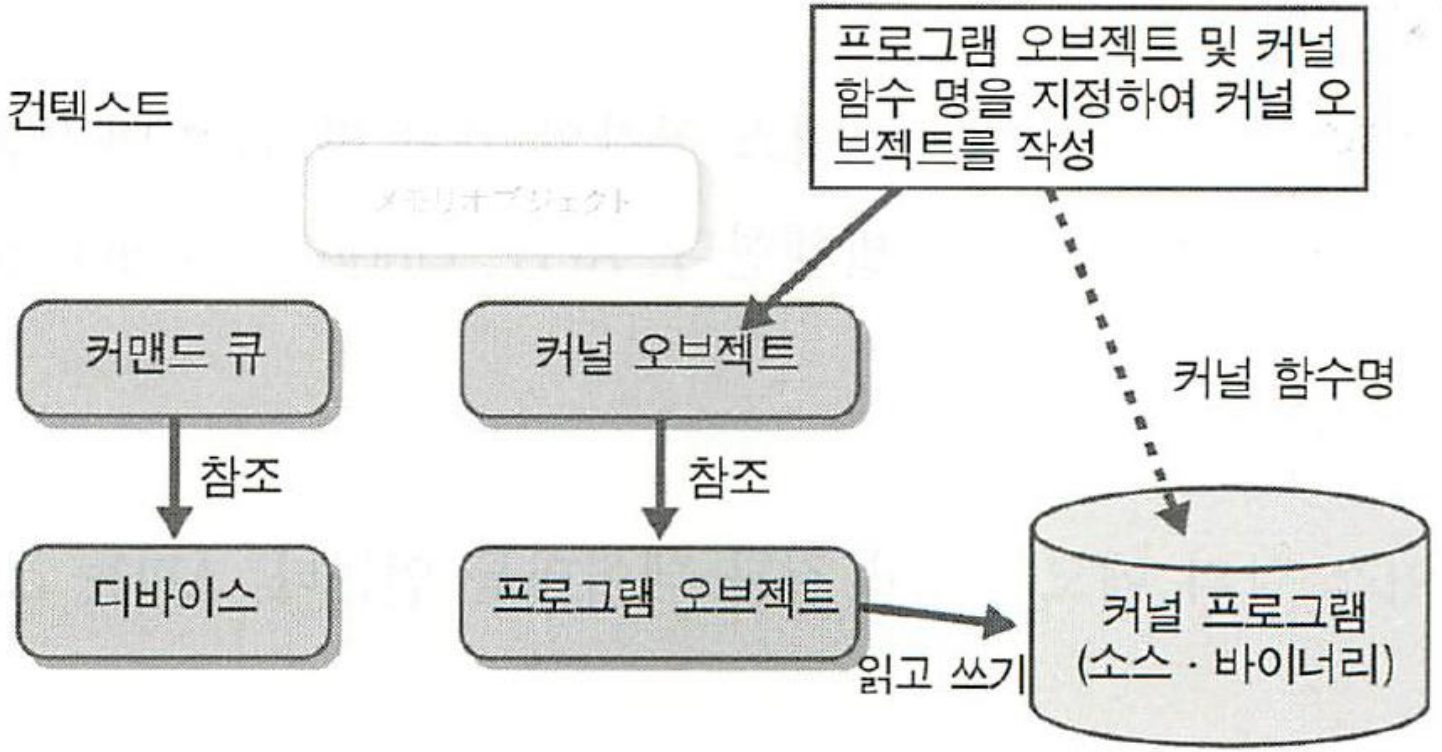


2-5. 프로그램 빌드

- 프로그램을 빌드하는 데는 clBuildProgram
- 빌드 로그를 얻는 데는 clGetProgramBuildInfo
- 해당 자원을 반환을 위해서 clUnloadCompiler

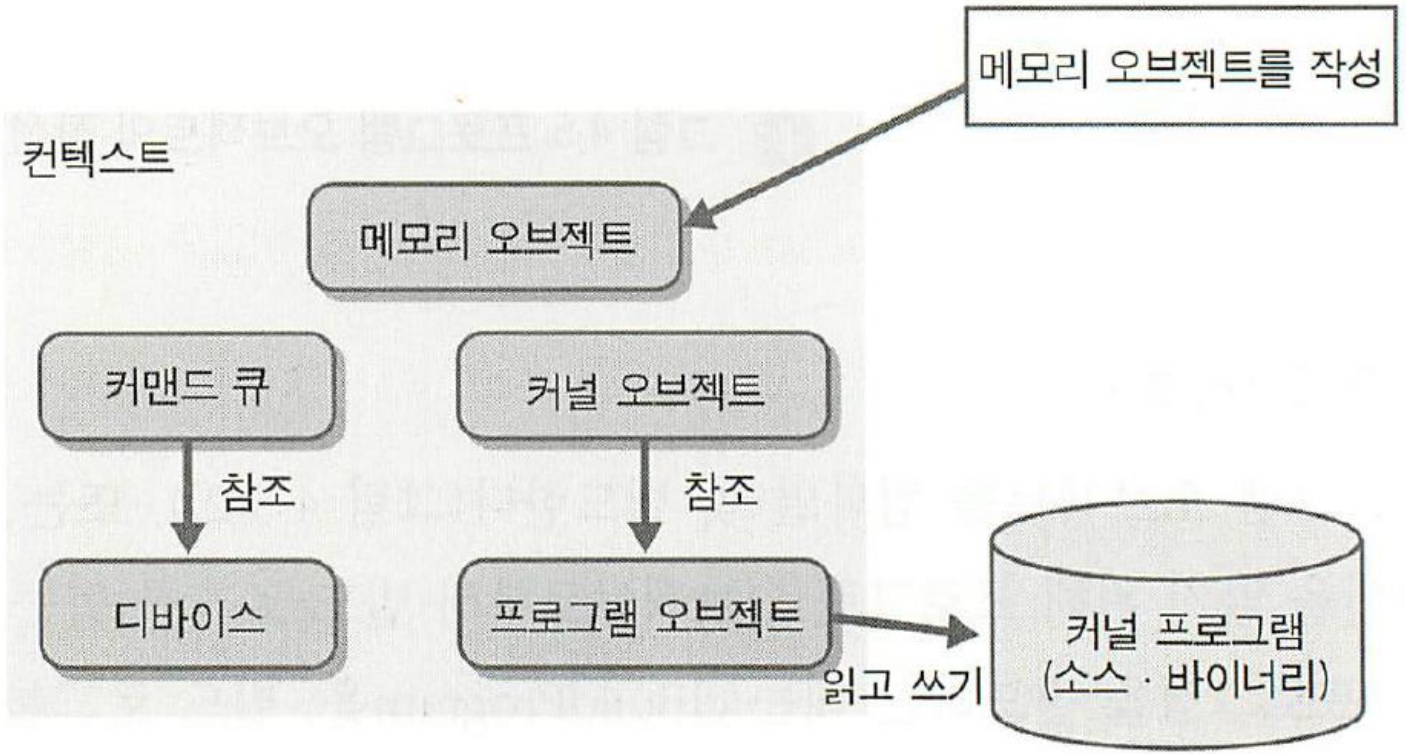
2-6. 커널의 작성

- 빌드된 프로그램 오브젝트를 사용하여 커널 오브젝트를 작성
- 커널 오브젝트를 작성하는 데는 프로그램 오브젝트 및 커널 함수 명을 지정하여 clCreateKernel을 실행



2-7. 메모리 오브젝트 작성

- 커널의 인수를 사용하여 메모리 오브젝트를 작성
- 앞선 예에서 2개의 읽기 전용 메모리 오브젝트와 한 개의 쓰기 전용 메모리 오브젝트를 작성하여 커널 함수의 인수로 지정
- 배열 데이터와 같이 직렬로 나열된 데이터를 표현하기 위해 메모리 오브젝트는 버퍼 메모리 오브젝트라 함
- clCreateBuffer



2-8. 커널에 전달하는 인수 지정

- 커널 함수에 전달하는 인수를 지정
- 예에서는 3개의 배열 데이터에 대응한 3개의 버퍼 메모리 오브젝트를 지정
- 커널의 인수를 지정하기 위해서는 clSetKernelArg를 사용

2-9. 커널 실행

- 워크 아이템을 실행하는 인덱스 공간의 차원 및 워크 아이템의 수를 지정하고, 커널 실행 커맨드를 커맨드 큐에 발행
- clnqueueNDrangeKernel

2-10. 결과 취득

- 디바이스 상의 메모리에서 결과 데이터를 얻음
- clEnqueueReadBuffer를 사용

2-11. 리소스 해방

- 작성한 컨텍스트, 커맨드 큐, 프로그램, 커널, 메모리 오브젝트를 해방

'소스코드'로 복습해 보자!

컨텍스트 작성

```
cl_int        status;
cl_context    context;

context = clCreateContextFromType(NULL,                    // 프로퍼티
                                  CL_DEVICE_TYPE_GPU,      // 디바이스 타입
                                  NULL,                    // 콜백함수
                                  NULL,                    // 유지 데이터
                                  &status);                // 실행상태
if (status != CL_SUCCESS) {
        // 컨텍스트 취득에 실패하여 처리를 중단
}
```

디바이스 얻기

```
static const int MaxDevices = 10;
cl_device_id devices[MaxDevices];
size_t size_return;
status = clGetContextInfo(context,                    // 정보를 얻는 컨텍스트
                          CL_CONTEXT_DEVICES,         // 포함하는 디바이스를 얻음
                          sizeof(devices),            // 배열 device의 크기
                          devices,                    // 결과가 설정되는 배열
                          &size_return);              // 실제로 설정된 사이즈

if (status != CL_SUCCESS) {
      // 디바이스 취득에 실패하여 처리를 중단
}
```

커맨드 큐

```
cl_command_queue queue;
queue = clCreateCommandQueue(context,
                             devices[0],
                             0,
                             &status);

if (status != CL_SUCCESS) {
       // 커맨드 큐의 작성이 실패하여 처리를 중단
}
```

프로그램 오브젝트의 작성

```
static const char *sources[] = { // sources[0]에 소스 코드 문자열을 지정
    "__kernel void\n\
    addVector(__global const float *in1,\n\
    __global const float *in2,\n\
    __global float *out)\n\
    {\n\
    int index = get_global_id(0);\n\
    out[index] = in1[index] + in2[index];\n\
}\n"};

cl_program program;
program = clCreateProgramWithSource(context,                    // 컨텍스트
                                    1,                          // 배열 sources의 요소수
                                    (const char**)&sources,     // 소스 코드 문자열의 배열
                                    NULL,                       // 모든 요소가 널로 끝남
                                    &status);                   // 함수의 실행 상태
if (status != CL_SUCCESS) {
        // 프로그램 오브젝트 작성이 실패하여 처리를 중단
}
```

프로그램 빌드

```
status = clBuildProgram(program, // 프로그램 오브젝트
                        1,       // devices에 전달하는 디바이스 수
                        devices, // 프로그램을 실행하는 디바이스ID의 리스트
                        NULL,    // 컴파일러에 전달하는 옵션
                        NULL,    // 콜백 함수
                        NULL);   // 콜백 함수의 인수

if (status != CL_SUCCESS) {
      // 할당 실패
}

clUnloadCompiler();
```

프로그램 빌드(Cont…)

```
size_t size_ret;
char buffer[MaxLogSize + 1];

status = clGetProgramBuildInfo(program,
                               device,
                               CL_PROGRAM_BUILD_LOG,
                               MaxLogSize,
                               buffer,
                               &size_ret);
if (status == CL_SUCCESS) {
        buffer[size_ret] = '\0';
        printf(">>> build log <<<\n");
        printf("%s\n", buffer);
        printf(">>> end of build log <<<\n");
} else {
        // 빌드 로그를 얻는데 실패
}
```

커널 작성

```
cl_kernel kernel;

kernel = clCreateKernel(program, "addVector", &status);
if (status != CL_SUCCESS) {
	// 커널 작성에 실패하여 처리를 중단
}
```

메모리 오브젝트 작성

```
cl_mem memIn1; //memIn2
memIn1 = clCreateBuffer(context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        sizeof(cl_float) * NumElements,
                        In1, //In2
                        &status);
if (status != CL_SUCCESS) {
      // 처리 중단
}

cl_mem memOut;
memOut = clCreateBuffer(context,
                        CL_MEM_WRITE_ONLY,
                        sizeof(cl_float) * NumElements,
                        NULL,
                        &status);
```

인수 설정

```
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memIn1);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memIn2);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memOut);
if (status != CL_SUCCESS) {
    // 처리 중단
}
```

커널 실행

```
size_t globalSize[] = {NumElements};
status = clEnqueueNDRangeKernel(queue,
                                kernel,
                                1,                  // 인덱스 공간
                                NULL,               // 현재 상태
                                globalSize,         // 워크 아이템 수
                                0,                  // 워크 그룹 사이즈
                                0,                  // 이벤트 수
                                NULL,               // 이벤트 리스트
                                NULL);              // 현재 커맨드에 관련된 이벤트

if (status != CL_SUCCESS) {
        // 처리 중단
}
```

결과 얻기

```
status = clEnqueueReadBuffer(queue,
                             memOut,                              // 출력 메모리 오브젝트
                             CL_TRUE,                             // 메모리 복사 종료될 때 까지 유지
                             0,                                   // 선두에서 시작
                             sizeof(cl_float) * NumElements,      // 쓰여질 사이즈
                             Out,                                 // 쓰여질 선두 어드레스
                             0,                                   // 이벤트 수
                             NULL,                                // 이벤트 리스트
                             NULL);                               // 커맨드 관련 리스트

if (status != CL_SUCCESS) {
        // 처리 중단
}
```

리소스 해방

```
clReleaseMemObject(memOut);
clReleaseMemObject(memIn2);
clReleaseMemObject(memIn1);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);

return 0;
```