oneAPI란 무엇인가?

ManyCoreSoft
박정호

MANYCORESOFT
서울특별시 관악구 관악로 1 서울대학교 138동 308호(우편번호 08826)
Tel. +82-2-882-1837 Fax. +82-2-883-1837 E-mail. contact@manycoresoft.co.kr

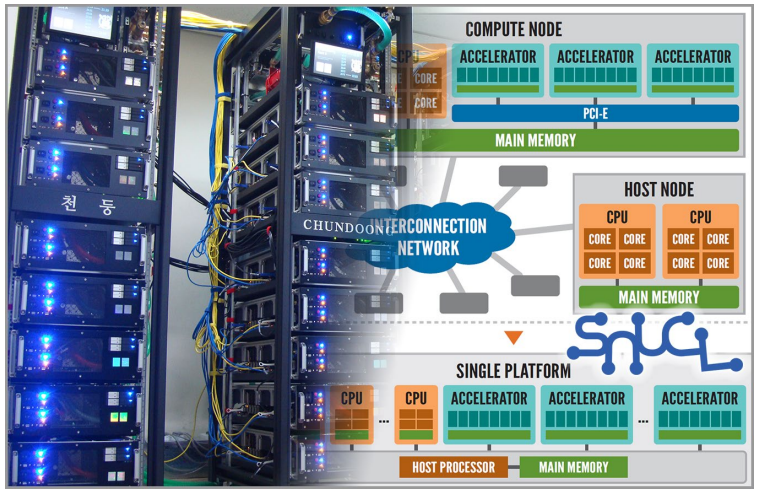
매니코어소프트

매니코어소프트는 서울대학교 멀티코어 컴퓨팅 연구실에서 출발한 고성능컴퓨팅(HPC) 전문 기업입니다

Multicore Computing Research Laboratory
멀티코어 컴퓨팅 연구실

SEOUL NATIONAL UNIVERSITY

고성능컴퓨팅 기술 연구, 개발



MANYCORESOFT

세계 최고 수준의 고성능컴퓨팅 전문기업

S/W 연구개발

AI 클라우드 인프라
AI 가속기 개발 (FPGA)

MOREH

H/W 연구개발

DEEP Gadget

수랭식 GPU 서버

toss, ILDONG, inno.N, 종근당, 위메프, ETRI, KCB, 대웅제약, YONSEI UNIVERSITY HEALTH SYSTEM, KIMS, SNUH, SEOUL NATIONAL UNIVERSITY, KNU, 호서대학교 HOSEO UNIVERSITY, DEEPING SOURCE, kit, MTS MOBILE TOTAL SERVICE, ...

컨설팅

병렬화/최적화 지원

AMD, SK hynix, HANCOM 한글과컴퓨터그룹, koscom, KCB, KIOST, ...

MANYCORESOFT

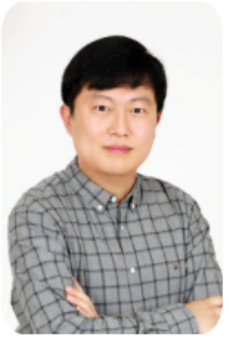
매니코어소프트

핵심 구성원

매니코어소프트의 구성원들은 고성능컴퓨팅 기술에 대한 전문성과 노하우를 바탕으로 기술 혁신을 선도하고 있습니다.

박정호 Jungho Park
Chief Executive Officer

박정호 박사는 매니코어소프트의 공동 창업자이자 대표이사입니다. 서울대학교 컴퓨터공학부를 졸업하고 같은 학부에서 박사 학위를 받았습니다. GPU 컴퓨팅의 태동기였던 2009년부터 지금까지 다양한 분야(AI, 보안, ...)의 애플리케이션을 GPU로 가속하고 최적화하는 일을 해 왔습니다.



조강원 Gangwon Jo
Chief Technology Officer

조강원 박사는 매니코어소프트의 공동 창업자이자 기술이사입니다. 서울대학교 컴퓨터공학부를 수석 졸업하고 같은 학부에서 박사 학위를 받았습니다. 박사과정 동안 GPU와 FPGA를 위한 소프트웨어 플랫폼을 개발하는 연구를 했고 이것이 매니코어소프트가 제공하는 솔루션의 밑바탕이 되었습니다.

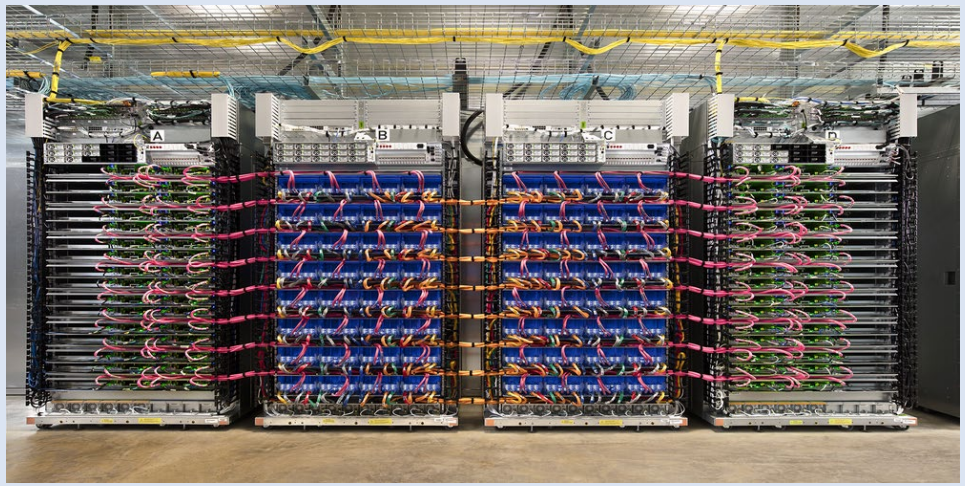
이재진 Jaejin Lee
Co-Founder & Advisor

이재진 교수는 매니코어소프트의 공동 창업자이자 고문입니다. 서울대학교 물리학부를 졸업하고 University of Illinois Urbana-Champaign에서 박사 학위를 받았습니다. 2002년부터 서울대학교 컴퓨터공학부 교수로 재직하면서 세계적으로 이종 컴퓨팅 분야의 연구개발을 선도해 왔습니다.

고성능컴퓨팅(HPC)의 활용

딥 러닝 | 시뮬레이션 | 검색 엔진 | 데이터 마이닝 | ……

High Performance Computing / High Throughput Computing



MANYCORESOFT

oneAPI 프로젝트

- 목적
  - **통합된 프로그래밍 모델**을 제공하여 다양한 아키텍처에서 개발을 쉽게 하자
- 세부사항
  - 다양한 아키텍처에서 공통된 개발 경험 제공
    - 병렬성을 간편하게 표현할 수 있는 통합된 프로그래밍 언어와 라이브러리 제공
  - 아키텍처에 관계없이 높은 수준의 성능 보장
  - CPU, GPU, AI Chip, FPGA 등 지원
  - 산업 표준과 오픈 표준을 기반으로 함
- 2020년 9월에 1.0 표준이 발표됨

왜 통합 프로그래밍 모델이 필요한가?

- 이번 시간의 주제!
  - 가속기의 등장과 병렬 프로그래밍의 중요성
  - 프로그래밍 장벽과 파편화
  - 통합 이종 프로그래밍 환경의 필요성

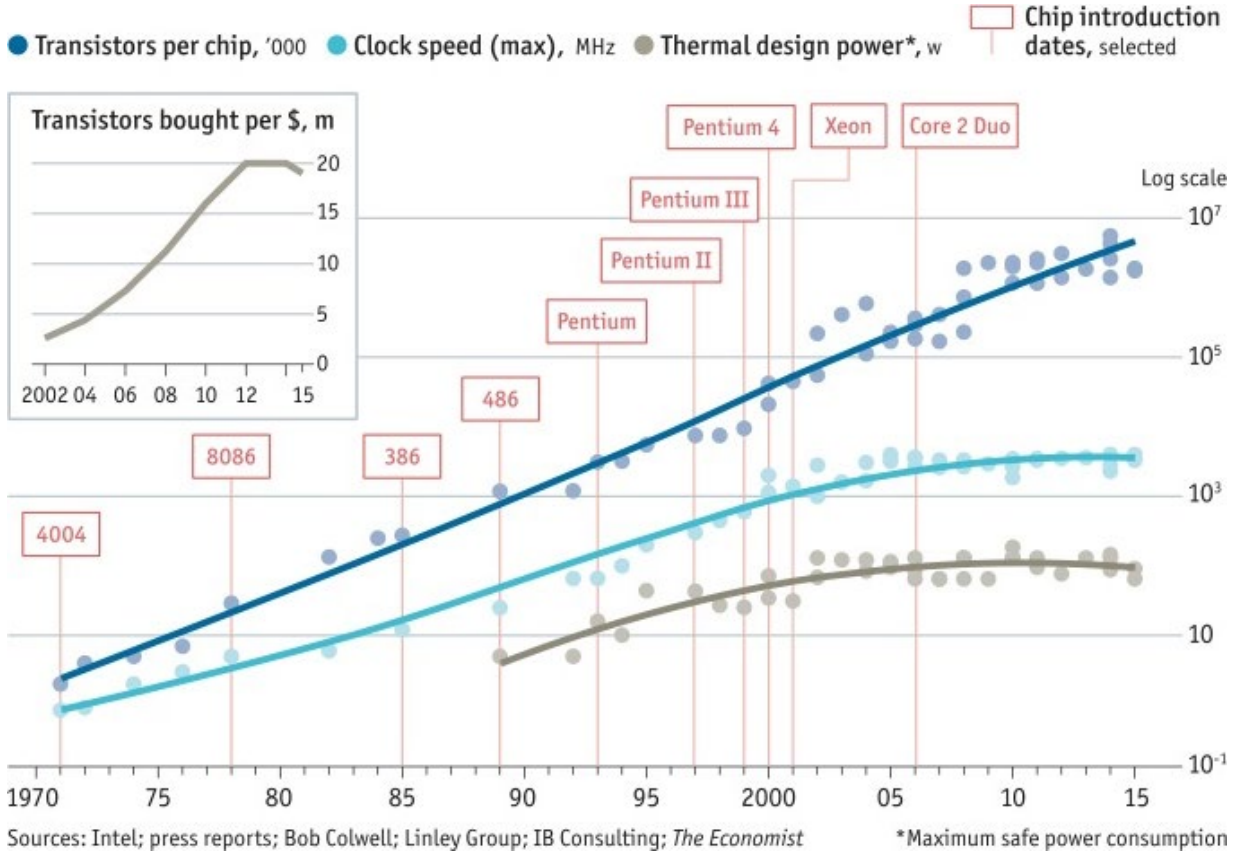
- 그 후에 oneAPI를 사용하여 프로그래밍 하는 방법에 대해 배우자

무어의 법칙

칩 하나에 집적된 트랜지스터 수가 2년에 2배씩 증가

과거: CPU 코어
성능이 2배씩 증가

현재: CPU 코어
개수가 점차 증가



MANYCORESOFT

전력 장벽(Power Wall)

- CPU의 계산 능력 ∝ CPU 동작 속도
- 소비전력 ∝ CPU 동작 속도
  - CPU 동작 속도는 무한히 증가시킬 수 없음
    - $P_{dynamic} = \alpha C V_{DD}^2 f A$
  - 동작 속도는 3GHz ~ 4GHz에서 더 이상 증가시킬 수 없음
    - 왜? 동작 전압을 낮추는 꼼수가 더 이상 통하지 않아서
- 발열 ∝ 소비전력
- 그래서 코어를 늘리기로 함

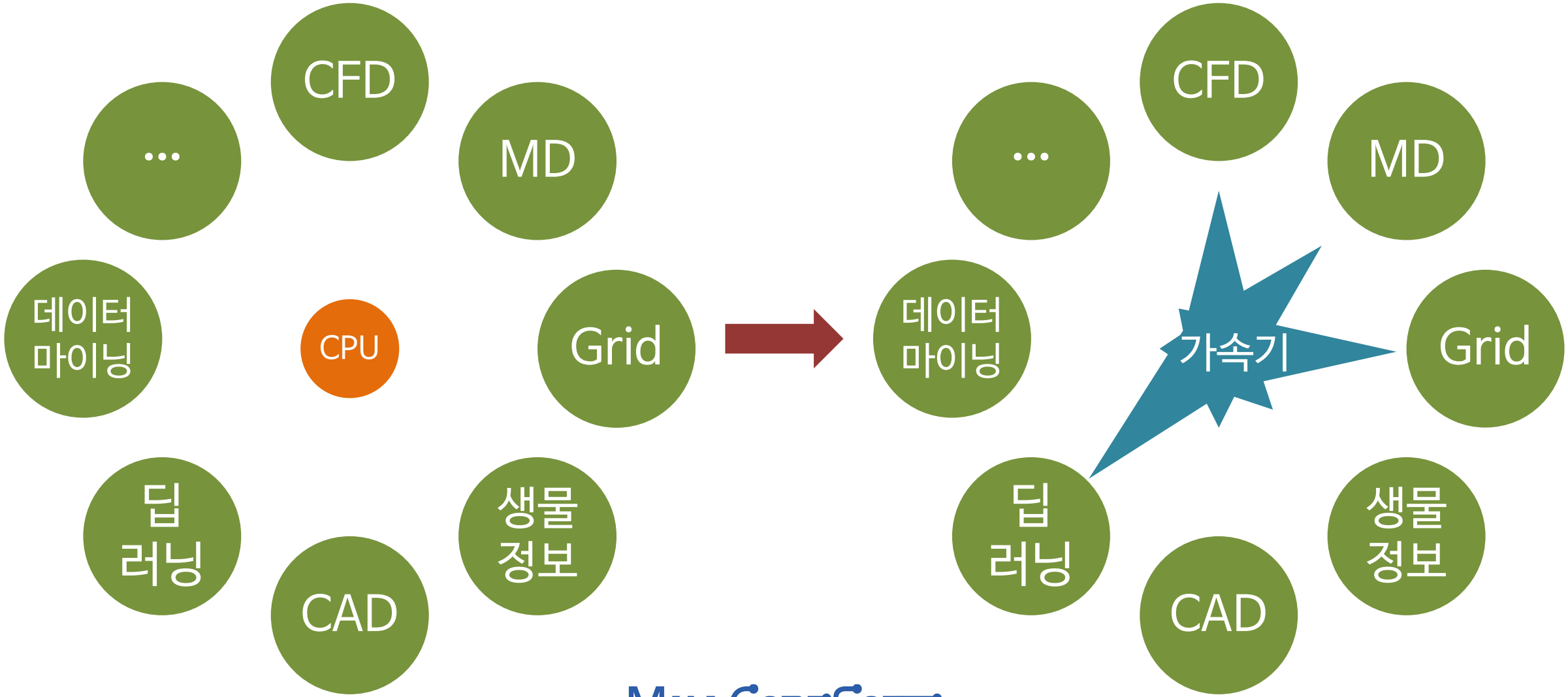
“The Free Lunch is Over”

MANYCORESOFT

멀티코어? 매니코어?

- 멀티코어(Multicore)
  - 두 개 이상의 프로세서를 탑재하고 있는 하나의 칩
  - 프로세서는 코어라 불림
- 매니코어(Manycore)
  - 아주 많은 코어를 탑재하고 있는 하나의 칩
  - 추가적인 소프트웨어 노력이 필요한 정도로 코어가 많은 칩
  - 고전적으로는 16코어 이상을 매니코어라고 함
- 전력 장벽에 대한 해결책

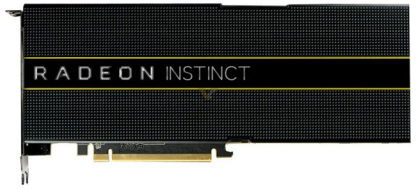
하지만, 일부만 잘 할 수는 있다

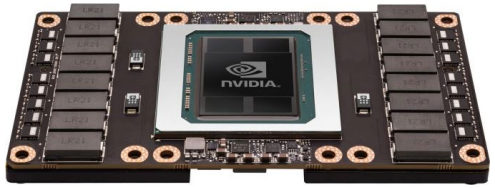


가속기(Accelerator)

- 특정 패턴의 계산에 특화되어 있음
  - CPU의 역할을 완전히 대체할 수 없음: 운영체제 실행, I/O, 순차 처리, …
  - 모든 애플리케이션에서 성능이 빨라지는 것이 아님
- CPU에서 사용하던 프로그램을 그대로 실행시킬 수 없음
  - OpenCL, CUDA, SYCL, OpenACC, …
- CPU와 같은 칩에 있을 수도 있고, 별도로 장착할 수도 있음
  - 예: PCIe, NVLink, …

GPU





- 현재 널리 사용되고 있는 대표적인 가속기
  - ~ 7,000 cores, 20 TFLOPS, 40 GB HBM2, 400 W
- 단순한, 독립적인 계산 작업을 매우 많이 할 때 유리함
- 소프트웨어 에코시스템이 (그나마) 잘 마련되어 있음
  - 프로그래밍 언어: OpenCL, CUDA, OpenACC, …
  - 라이브러리: BLAS, cuDNN, FFT, OpenCV, FFmpeg, …
  - 소프트웨어: NAMD, VASP, GROMACS, LAMMPS, AMBER, MATLAB, Hashcat, Caffe, TensorFlow, Theano, Torch, …

OpenCL

NVIDIA CUDA

FPGA

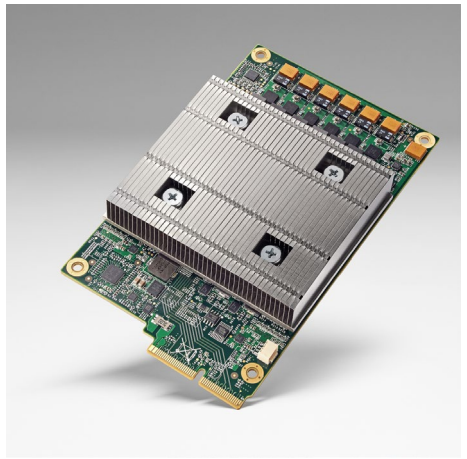
- 내장된 회로를 바꿀 수 있는 반도체
  - 기존에는 ASIC의 대용으로, 혹은 ASIC 생산 전 테스트를 위해 주로 사용됨
  - 주요 벤더: Xilinx와 Intel
- 최근 범용 계산의 가속기로 주목받기 시작
  - FPGA의 크기가 커지고, 범용 계산을 빠르게 수행하기 위한 구성 요소(DSP 등)가 많이 추가됨
  - 최신 FPGA는 이론 성능이 10 TFLOPS 내외
  - 소비전력이 매우 낮음(~ 25 W)

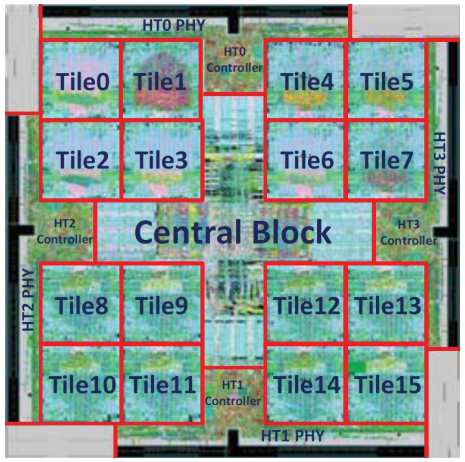
Xilinx UltraScale+ 보드
(BittWare XUPP3R)

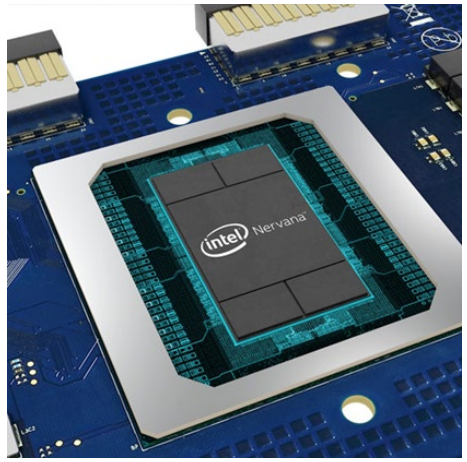
Intel Programmable
Acceleration Card

새로운 가속기의 연구개발

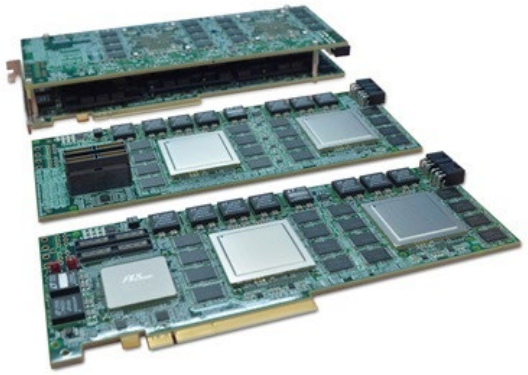
딥 러닝 전용 가속기 (NPU)







PEZY-SC



Sunway SW26010

MPE*4 + CPE*256

중국 Sunway TaihuLight
슈퍼컴퓨터에 사용됨
(93 PFLOPS, 2018년 TOP500 3위)

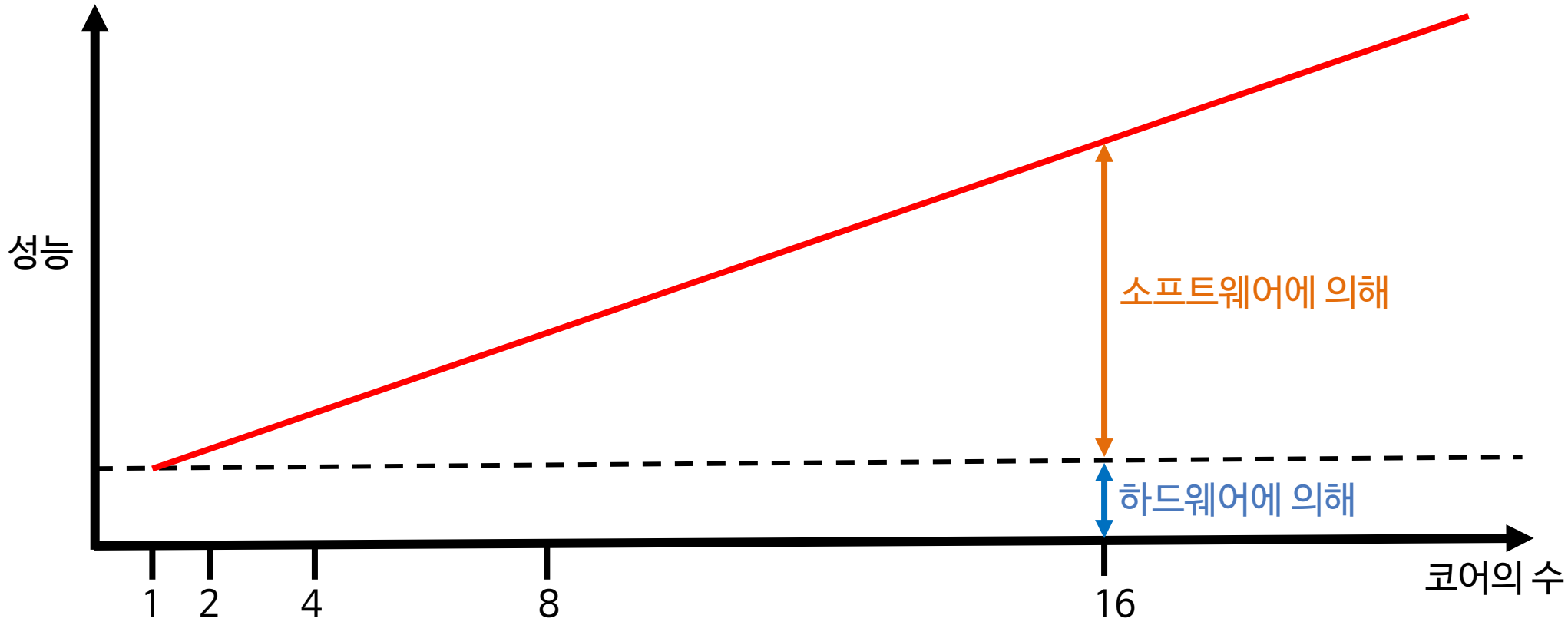
MANYCORESOFT

패러다임 시프트: 이종 시스템으로

- CPU → CPU + GPU → CPU + GPU + FPGA + …
- 응용 분야에 따라 시스템 구성이 달라질 것
  - 어떤 가속기를 얼마나 사용할 것인지
- 응용 소프트웨어 개발과 유지보수의 어려움 증가
  - 특정 가속기를 지원하도록 새로 개발하거나 수정해야 함
  - 프로그램의 병렬성에 대한 이해가 절대적으로 필요

  => 프로그램 생산성 감소

병렬 프로그래밍의 중요성



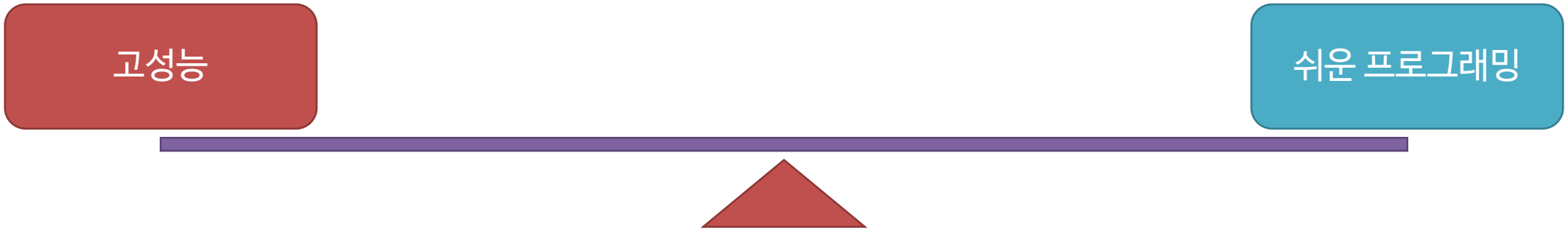
프로그래밍 장벽

- 가속기에 든 병렬성을 효율적으로 이용하는 소프트웨어를 어떻게 쉽게 만드는가?
  - 의존성(dependence), 데이터 분배, 작업 분배, 메모리 일관성 (memory consistency) 등의 복잡한 문제를 극복해야함
- 전통적인 멀티프로세서 시스템에 지난 40년간 풀리지 않는 문제

병렬 프로그래밍 모델

- 응용 프로그램을 개발할 때 프로그래머와 멀티코어 프로세서간의 인터페이스
  - 언어, 라이브러리, 언어 확장, 컴파일러 지시자 등

- **고성능**과 **쉬운 프로그래밍**간의 균형을 맞추는 것이 중요



MANYCORESOFT

병렬 프로그래밍 모델의 종류

- CPU
  - pThreads (POSIX threads)
  - OpenMP
  - Clik
  - TBB
- GPU
  - OpenCL
  - CUDA
  - OpenACC
  - SYCL
- FPGA
  - OpenCL
  - Verilog
- Cluster
  - Message Passing Interface (MPI)
  - PGAS
  - MapReduce
- …

프로그래밍 환경 파편화

- 이종 시스템에서 다양한 종류의 가속기(프로세서)를 사용하려면?
  - 각각 다른 프로그래밍 모델과 환경을 공부하고 프로그래밍 해야함
  - 프로그래밍 + 유지보수의 어려움

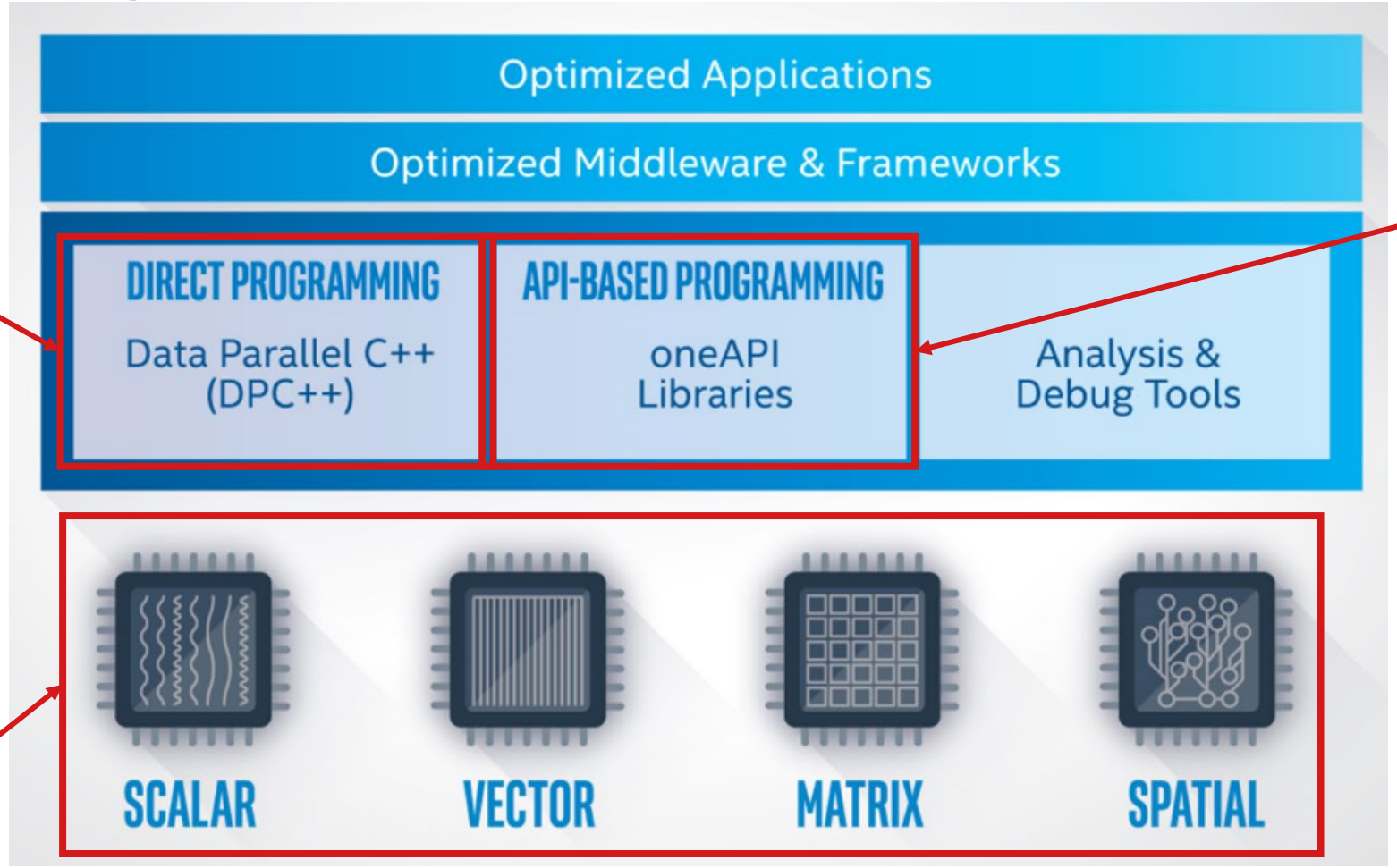
통합 이종 프로그래밍 모델

- 프로그래밍의 복잡성 해결
  - 프로그래밍 장벽 극복 = 쉬운 프로그래밍 모델
  - 프로그래밍 파편화 극복 = 통합 프로그래밍 모델
- 코드이식성 + 성능이식성 보장
  - 코드 이식성: 같은 코드가 여러 다른 프로세서에서 동일하게 작동함
  - 성능 이식성: 같은 코드가 여러 다른 프로세서에서 비슷한 성능을 냄

oneAPI 프로젝트

- 목적
  - 통합된 프로그래밍 모델을 제공하여 다양한 아키텍처에서 개발을 쉽게 하자
- 세부사항
  - **다양한 아키텍처에서 공통된 개발 경험 제공 = 코드 이식성**
    - 병렬성을 간편하게 표현할 수 있는 통합된 프로그래밍 언어와 라이브러리 제공
  - **아키텍처에 관계없이 높은 수준의 성능 보장 = 성능 이식성**
  - CPU, GPU, AI Chip, FPGA 등 지원
  - 산업 표준과 오픈 표준을 기반으로 함

oneAPI System Architecture



코드 이식성

성능 이식성

다양한 프로세서
지원 (가속기)

DPC++

- oneAPI Data Parallel C++ (DPC++)
  - C++, SYCL 기반의 프로그래밍 모델
    - SYCL=C++에 병렬화를 표현할 수 있는 것을 추가한 프로그래밍 모델
  - Unified Shared Memory (USM) 지원
    - SYCL에는 없는 대표적인 추가 기능
    - Host와 Device 사이의 메모리 관리를 쉽게 함
    - 프로그래밍 편의성

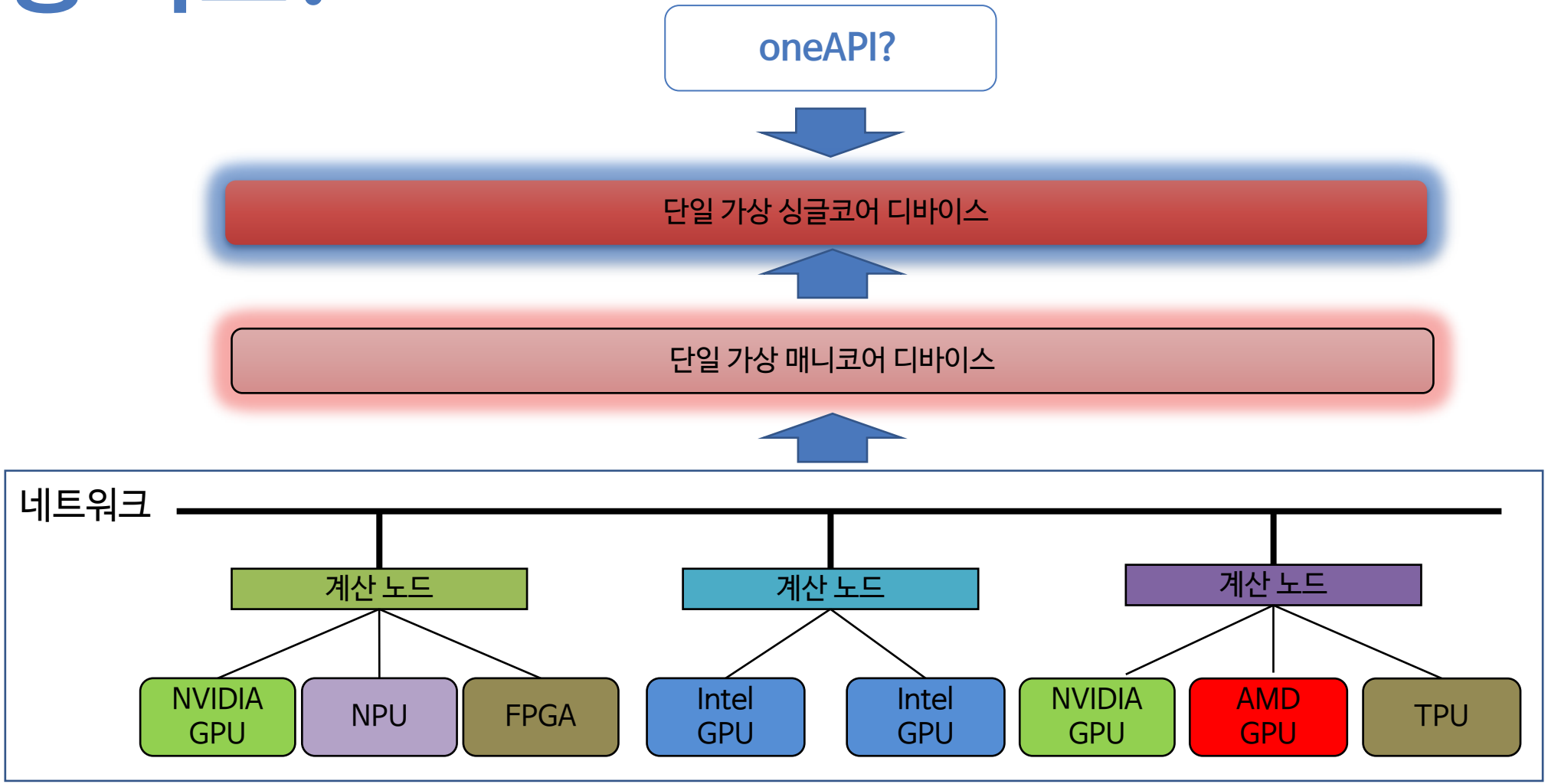
oneAPI 라이브러리

- oneDPL (=STL)
  - Parallel STL
- oneDNN (=cuDNN)
  - oneAPI deep learning primitives
- oneCCL (=nccl)
  - oneAPI collective communication library
- …

oneAPI의 한계

- 지원하는 device의 한계
  - 현재 x86 CPU, Intel 내장형 GPU, Intel FPGA를 지원
  - Intel외에 다른 벤더의 하드웨어에 대한 지원의 부족
- 프로그래밍 장벽
  - DPC++는 CUDA와 OpenCL등 보다는 쉽지만 순차 프로그래밍 보다는 어려움
- 멀티 디바이스와 클러스터 지원
  - 일반적으로 멀티 디바이스를 따로 관리해 줘야함
  - 클러스터 내 여러 하드웨어에 대한 지원 부족

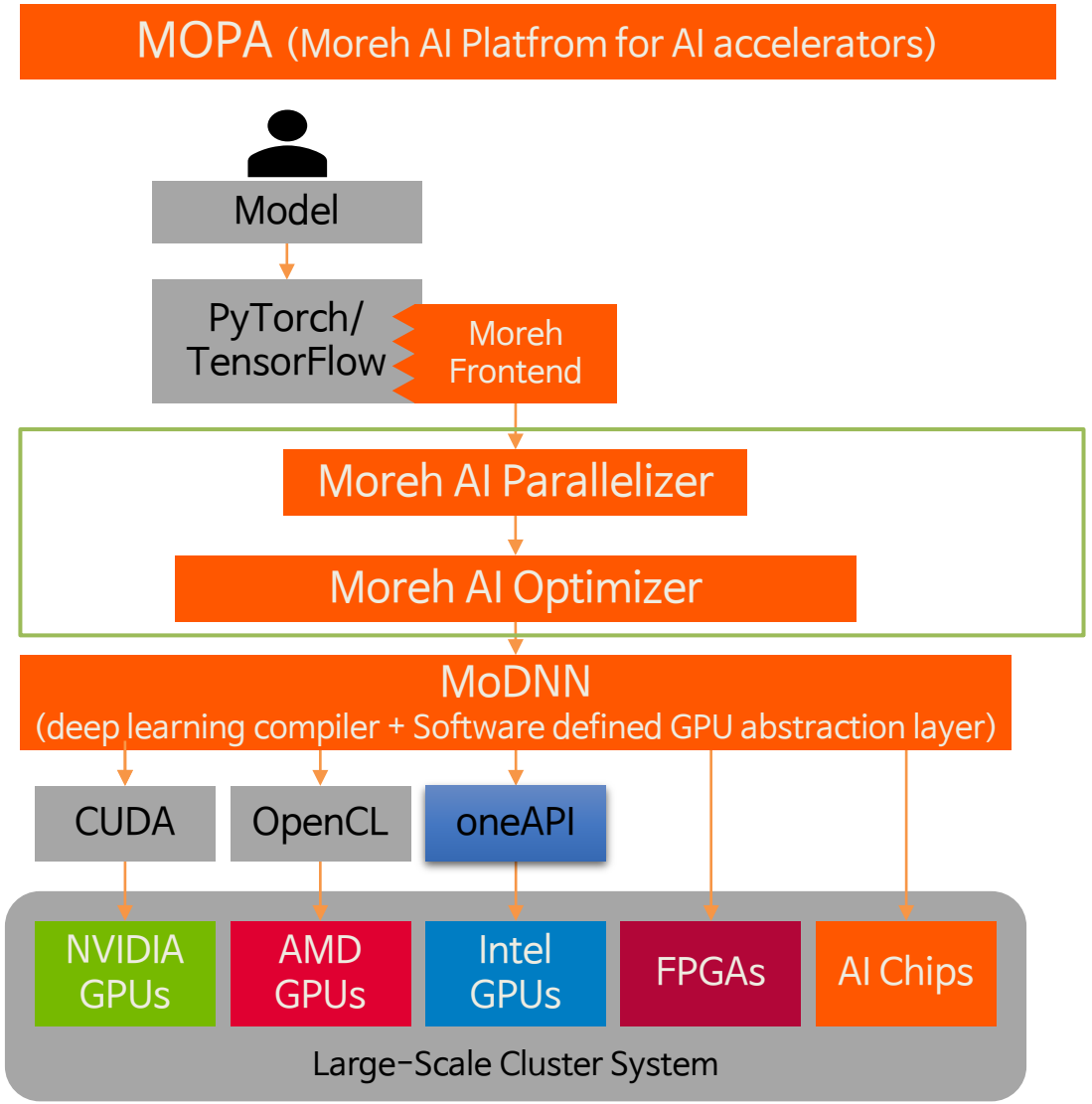
최종 목표?



MANYCORESOFT

활용

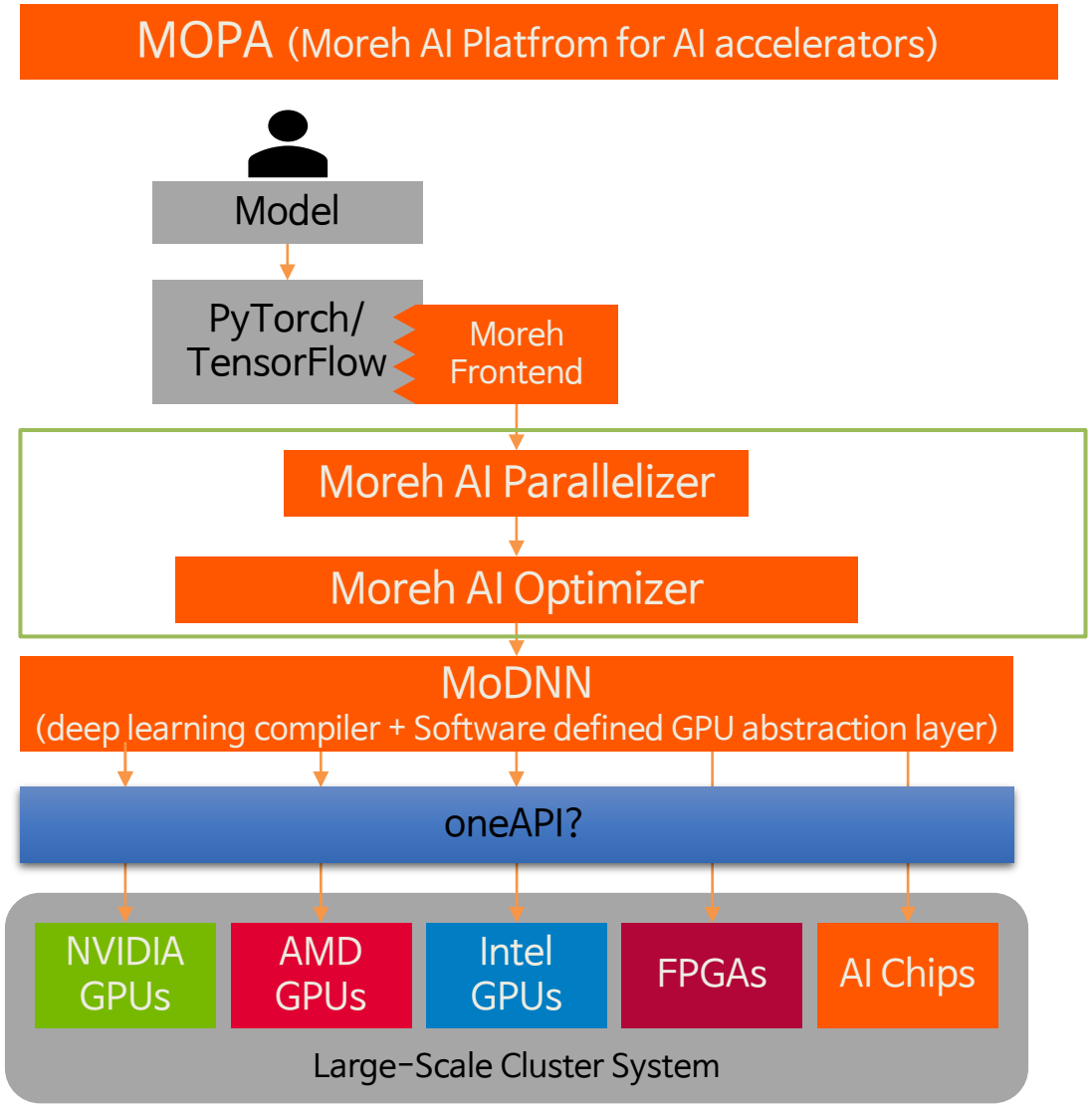
MOREH



MANYCORESOFT

활용

MOREH



MANYCORESOFT

oneAPI를 이용한 병렬 프로그래밍

MANYCORESOFT
서울특별시 관악구 관악로 1 서울대학교 138동 308호(우편번호 08826)
Tel. +82-2-882-1837 Fax. +82-2-883-1837 E-mail. contact@manycoresoft.co.kr

Instructor

조강원 | 매니코어소프트 기술이사

서울대학교 컴퓨터공학 박사(2020)

국내 최초 GPU 슈퍼컴퓨터 '천둥' 개발(2012)
클러스터용 OpenCL 프레임워크 'SnuCL' 개발(2013)
FPGA용 OpenCL 컴파일러 'SOFF' 개발(2020)

가속기 프로그래밍 여름학교/겨울학교 강사(2015-19)
취업연계 R&D 교육센터 강사(2014)

매니코어소프트는
서울대학교 멀티코어 컴퓨팅 연구실
출신 구성원들이 창업한
고성능컴퓨팅(HPC) 전문 기업입니다.

프로그래밍 솔루션 개발, 판매

HPC 시스템 설계, 구축

프로그래밍 컨설팅 및 교육 서비스

MANYCORESOFT

Agenda

- DPC++ 프로그래밍 모델 개요
- Device 선택하기 및 Queue 생성
- Kernel 실행
- Buffer와 Accessor
- Host-Device 간 동기화
- DPC++의 추가 기능들: USM, sub-group, …

SYCL

- 이종 시스템을 위한 프로그램을 작성하기 위한 공개 표준
  - Pronounced ‘sickle’
  - Khronos Group에서 표준 정의: https://www.khronos.org/sycl/
  - CUDA(2007) → OpenCL(2009) → SYCL(2014)
- “C++ single-source heterogeneous programming”
  - C++: modern C++을 사용해 가속기 쪽 코드 작성
  - Single-source: 가속기 쪽 코드와 호스트 쪽 코드를 같은 파일에 작성, 컴파일

Data Parallel C++ (DPC++)

- Intel에서 정의한 SYCL의 extension
  - Unified shared memory, in-order queues, reduction, subgroups, ...
  - 이 중 상당수는 SYCL 2020 버전에 정식 추가 예정
- oneAPI의 프로그래밍 언어로 채택됨
  - DPC++로 작성된 프로그램은 CPU, GPU, FPGA 등 다양한 가속기에 모두 실행 가능 (code portability)

DPC++의 핵심 아이디어

- SPMD: Single Program, Multiple Data
  - 같은 커널 함수의 여러 인스턴스(**워크-아이템; work-item**)를 동시에 실행
  - 각 워크-아이템은 고유한 ID를 가지며 이를 사용해 서로 다른 데이터 원소를 처리
  - 데이터 병렬성(data parallelism) 활용

순차 코드

```
for (i = 0; i < 1024; i++) {
  C[i] = A[i] + B[i];
}
```

DPC++ 커널 함수

```
h.parallel_for(range<1>(1024), [=](id<1> i) {
  C[i] = A[i] + B[i];
});
```

DPC++ 프로그램 예시

```
#include <CL/sycl.hpp>
using namespace sycl;
void vec_add(int *a, int *b, int *c, int N) {
  queue q;
  buffer<int, 1> buf_a(a, range<1>(N));
  buffer<int, 1> buf_b(b, range<1>(N));
  buffer<int, 1> buf_c(c, range<1>(N));
  q.submit([&](handler &h) {
    auto A = buf_a.get_access<access::mode::read>(h);
    auto B = buf_b.get_access<access::mode::read>(h);
    auto C = buf_c.get_access<access::mode::write>(h);
    h.parallel_for(range<1>(N), [=](id<1> i) {
      C[i] = A[i] + B[i];
    });
  });
}}
```

] 가속기에 명령을 내리기 위한 큐 생성

] 가속기 메모리에 버퍼 할당 및 초기화

] 큐에 새로운 명령 추가

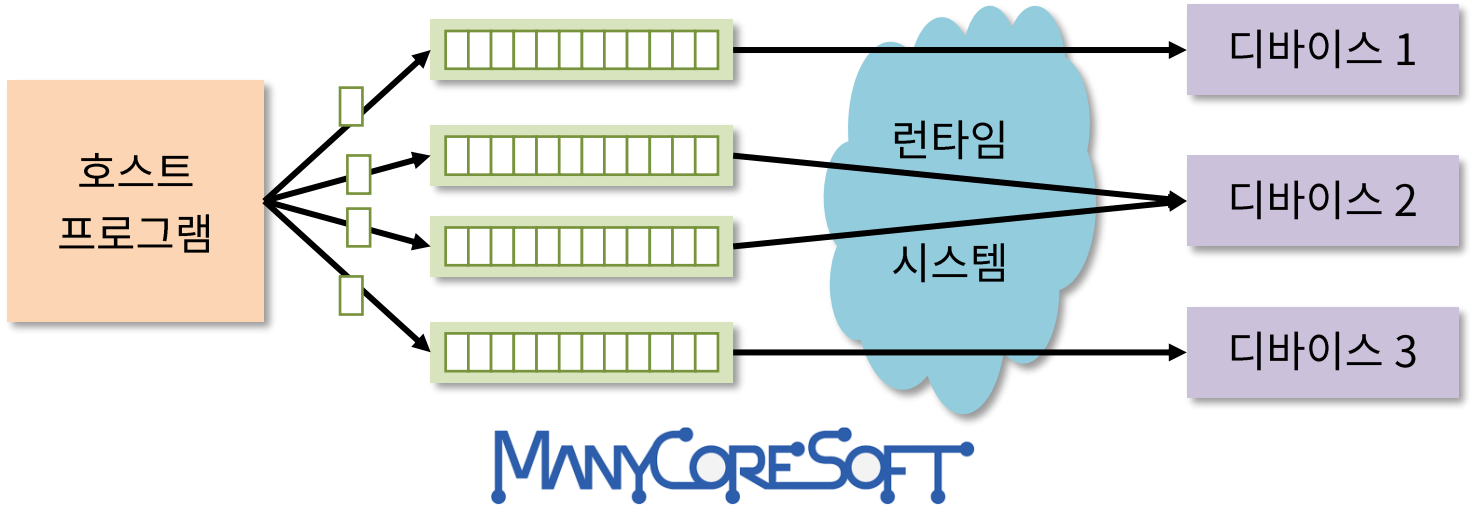
] 어떤 버퍼에 접근할 것인지 지정

] 계산 작업을 병렬로 실행

MANYCORESOFT

디바이스(Device)와 큐(Queue)

- 디바이스는 시스템에 장착된 oneAPI를 지원하는 개별 프로세서를 의미
  - (멀티코어) CPU, integrated GPU, discrete GPU, FPGA, …
- 디바이스에 명령을 내리기 위해 큐를 사용
  - 큐를 만들고 → 큐에 명령을 추가하면 → oneAPI 런타임 시스템이 알아서 실행
  - 여러 디바이스를 함께 사용하려면 각 디바이스마다 따로 큐를 만들어야 함



큐 생성

- 인자 없이 sycl::queue 객체를 만들면 기본 디바이스에 대한 큐 생성
- sycl::queue 객체를 만들 때 인자로 device selector 객체를 전달해서 특정 디바이스에 대한 큐를 만들 수 있음
  - sycl::cpu_selector
  - sycl::gpu_selector
  - sycl::accelerator_selector
  - sycl::intel::fpga_selector
  - sycl::intel::fpga_emulator_selector
  - sycl::host_selector

```
queue q;
q.submit([&](handler &h) {
  ...
}
```

```
cpu_selector selector;
queue q(selector);
```

```
gpu_selector selector;
queue q(selector);
```

```
intel::fpga_selector selector;
queue q(selector);
```

MANYCORESOFT

디바이스 정보

- sycl::device::get_info() 함수를 사용해서 현재 선택된 디바이스에 대한 정보를 얻어올 수 있음
  - 하드웨어 종류: 이름, 제조사 등
  - 아키텍처 정보: 메모리 크기, 캐시 크기, 벡터 유닛 크기 등

```
queue q(selector);
device dev = q.get_device();
std::cout << "Name: " << dev.get_info<info::device::name>() << std::endl;
std::cout << "Vendor: " << dev.get_info<info::device::vendor>() << std::endl;
std::cout << "Memory size: " << dev.get_info<info::device::global_mem_size>()
<< std::endl;
```

디바이스에 명령(Command) 내리기

- `sycl::queue::submit()` 함수를 사용해서 큐에 새로운 명령을 추가할 수 있음
- 명령 = C++ 람다(lambda) 함수
  - `sycl::handler` 객체를 인자로 받음
  - `[&]`: 람다 함수 밖의 변수를 안에서 읽고 쓸 수 있음을 의미
  - 람다 함수 안에 실제 디바이스에서 실행할 명령을 정의

```
int N = 1024;
q.submit([&](handler &h) {

  ...

});
```

명령: Parallel Kernel

- sycl::handler::parallel_for() 함수를 사용해서 커널 함수를 SPMD 방식으로 실행
- sycl::range 객체로 워크-아이템(커널 인스턴스)을 몇 개나 실행할지 지정함
- 커널 함수 역시 람다 함수로 정의
  - 인자로 전달되는 sycl::id 객체는 현재 워크-아이템의 ID를 가지고 있음
  - [=]: 람다 함수 밖의 변수를 안에서 읽을 수 있음을 의미

```
int N = 1024;
q.submit([&](handler &h) {
  ...
  h.parallel_for(range<1>(N), [=](id<1> i) {
    c[i] = a[i] + b[(N-1) - i];
  });
});
```

size_t로 자동 캐스팅됨

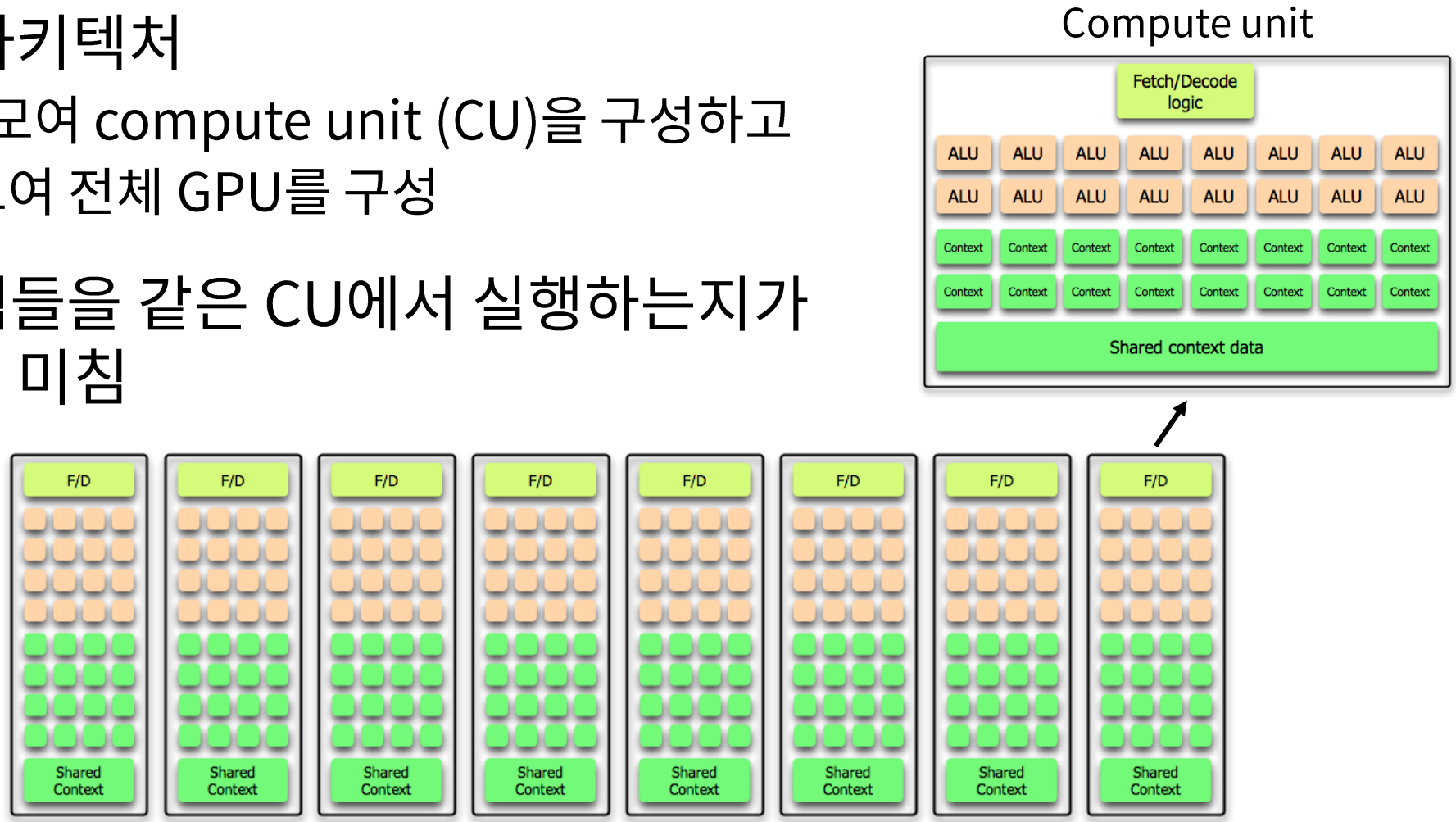
2~3차원 Range와 ID

- 워크-아이템에 2차원 혹은 3차원으로 ID를 부여할 수 있음

```
int N = 64, M = 32;
q.submit([&](handler &h) {
  ...
  h.parallel_for(range<2>(N, M), [=](id<2> i) {
    c[i[0] * M + i[1]] = a[i[0] * M + i[1]] + b[i[1] * N + i[0]];
  });
});
```

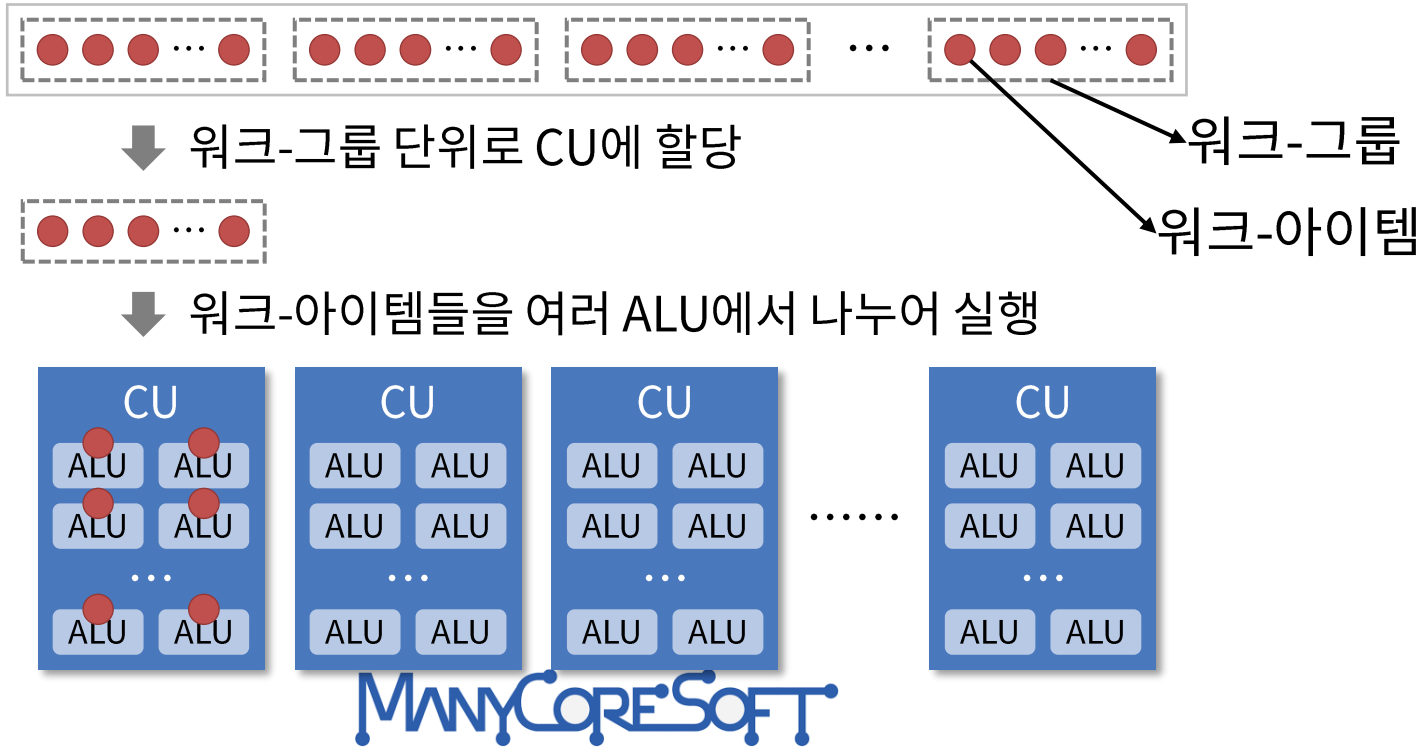
명령: NDRange Kernel

- 통상적인 GPU 아키텍처
  - ALU 여러 개가 모여 compute unit (CU)을 구성하고
  - CU 여러 개가 모여 전체 GPU를 구성
- 어떤 워크-아이템들을 같은 CU에서 실행하는지가 성능에 큰 영향을 미침



명령: NDRange Kernel

- 워크-아이템들을 워크-그룹(work-group) 단위로 묶어서, 같은 워크-그룹 안에 있는 워크-아이템들은 항상 같은 CU에서 실행되도록 함
  - 모든 워크-그룹의 크기는 동일



명령: NDRange Kernel

- sycl::nd_range 객체에 전체 워크-아이템 개수와, 각 워크-그룹 크기를 지정
- 커널 함수에 인자로 전달되는 sycl::nd_item 객체로부터 워크-아이템 ID를 얻어올 수 있음

```
int N = 1024;
q.submit([&](handler &h) {
  ...
  h.parallel_for(nd_range<1>(range<1>(N),
                             range<1>(64)),
                 [=](nd_item<1> item) {
    auto idx = item.get_global_id();
    c[idx] = a[idx] + b[idx];
  });
});
```

버퍼(Buffer)와 Accessor

- 버퍼는 호스트와 디바이스에서 모두 읽고 쓸 수 있는 메모리 영역을 의미
  - Discrete GPU나 FPGA의 경우 별도의 메모리(GDDR or HBM)를 가지고 있으므로 버퍼가 여기에 할당됨
- 버퍼에 대한 accessor를 받아 와서 일반 배열처럼 읽고 쓸 수 있음

※ Pseudo code

```
buffer<> buf;

q.submit([&](handler &h) {
  auto access = buf.get_access(h);
  h.parallel_for(... {
    access[i] = ...;
  });
});

auto access = buf.get_access();
... = access[i];
```

버퍼 할당

- 1~3차원 버퍼를 할당할 수 있음
- 버퍼를 할당할 때 특정 메인 메모리 영역과 연결시킬 수 있음
  - 처음에 버퍼를 메인 메모리 영역에 있던 데이터로 초기화
  - 버퍼가 소멸될 때 버퍼의 내용을 다시 메인 메모리 영역에 복사

```
buffer<int, 1> first(range<1>(1024));
buffer<float, 2> second(range<2>(64, 32));


int host_c[16] = {1, 2, ...};
{
  buffer<int, 1> third(host_c, range<1>(16));
  ...
  (modify the buffer "third")
  ...
}
```

third가 소멸될 때 수정된 값이
다시 host_c에 복사됨

MANYCORESOFT

커널 함수에서 사용할 Accessor 발급

- 디바이스 명령 안에서 커널 함수를 실행(parallel_for())하기 전에 accessor를 먼저 발급받음
- 버퍼 안의 값을 읽을지 쓸지 여부를 알려 줘야 함
  - access::mode::read
  - access::mode::write
  - access::mode::read_write
- 같은 버퍼를 어떤 명령에서는 읽기만 하고, 어떤 버퍼에서는 쓰기만 할 수도 있음

```
buffer<int, 1> buf_a(...);
buffer<int, 1> buf_b(...);
buffer<int, 1> buf_c(...);
q.submit([&](handler &h) {
  auto a = buf_a.get_access<access::mode::read>(h);
  auto b = buf_b.get_access<access::mode::read>(h);
  auto c = buf_c.get_access<access::mode::write>(h);
  h.parallel_for(range<1>(N), [=](id<1> i) {
    c[i] = a[i] + b[i];
  });
});
```

호스트에서 사용할 Accessor 발급

- 호스트에서(= 디바이스 명령 밖에서)도 accessor를 발급받아 버퍼 안의 값을 읽고 쓸 수 있음

```
buffer<int, 1> buf_c(...);
...
{
  auto c = buf_c.get_access<access::mode::read>();
  for (int i = 0; i < N; i++) {
    std::cout << c[i] << std::endl;
  }
}
```

DPC++ 프로그램 예시 Revisited

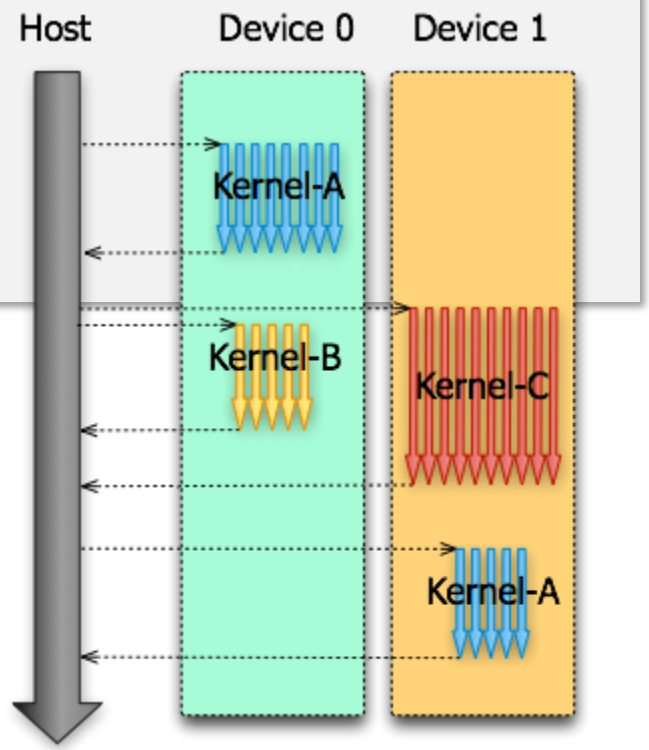
```
#include <CL/sycl.hpp>
using namespace sycl;
void vec_add(int *a, int *b, int *c, int N) {
  queue q;
  buffer<int, 1> buf_a(a, range<1>(N));
  buffer<int, 1> buf_b(b, range<1>(N));
  buffer<int, 1> buf_c(c, range<1>(N));
  q.submit([&](handler &h) {
    auto A = buf_a.get_access<access::mode::read>(h);
    auto B = buf_b.get_access<access::mode::read>(h);
    auto C = buf_c.get_access<access::mode::write>(h);
    h.parallel_for(range<1>(N), [=](id<1> i) {
      C[i] = A[i] + B[i];
    });
  });
}}
```

MANYCORESOFT

DPC++의 동기화 메커니즘

- 호스트-디바이스 사이
  - sycl::queue::submit() 함수는 큐에 명령을 추가한 직후 바로 종료함
  - 호스트 프로그램과 커널은 병렬로 실행
    - 커널은 호스트 프로그램의 개입 없이 oneAPI 런타임 시스템이 알아서 실행
- 같은 디바이스의 명령 사이
  - 큐에 여러 명령을 추가한 경우 이들 사이의 실행 순서는 런타임 시스템이 알아서 정함
- 다른 디바이스의 명령 사이
  - 서로 다른 디바이스의 커널은 병렬로 실행

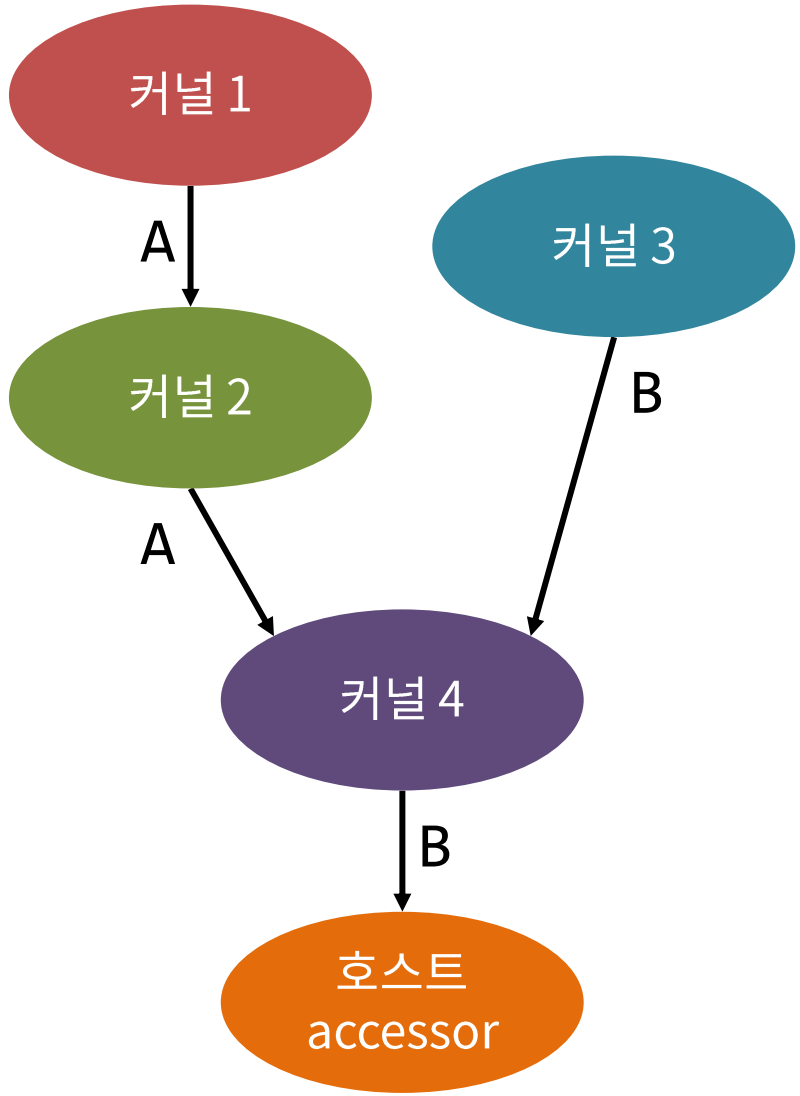
```
int N = 1024;
q.submit([&](handler &h) {
   ...
});
```



DPC++의 동기화 메커니즘

- Accessor 발급받은 것을 보고 내부적으로 data dependence 그래프 생성
  - Read-after-write, write-after-read, write-after-write
- 프로그램의 correctness를 보장하기 위해 필요한 동기화를 알아서 해 줌
  - 같은 디바이스에 대한 여러 명령 사이의 순서 결정
  - 디바이스 명령과 호스트 프로그램 사이의 동기화

```
buffer<int, 1> A(...), B(...);
q.submit([&](handler &h) {
  auto out = A.get_access<access::mode::write>(h);
  h.parallel_for(range<1>(N), [=](id<1> idx) { ... });
});
q.submit([&](handler &h) {
  auto out = A.get_access<access::mode::write>(h);
  h.parallel_for(range<1>(N), [=](id<1> idx) { ... });
});
q.submit([&](handler &h) {
  auto out = B.get_access<access::mode::write>(h);
  h.parallel_for(range<1>(N), [=](id<1> idx) { ... });
});
q.submit([&](handler &h) {
  auto in = A.get_access<access::mode::read>(h);
  auto inout = B.get_access<access::mode::read_write>(h);
  h.parallel_for(range<1>(N), [=](id<1> idx) { ... });
});
auto b = B.get_access<access::mode::read>();
...
```



MANYCORESOFT

DPC++ Unified Shared Memory (USM)

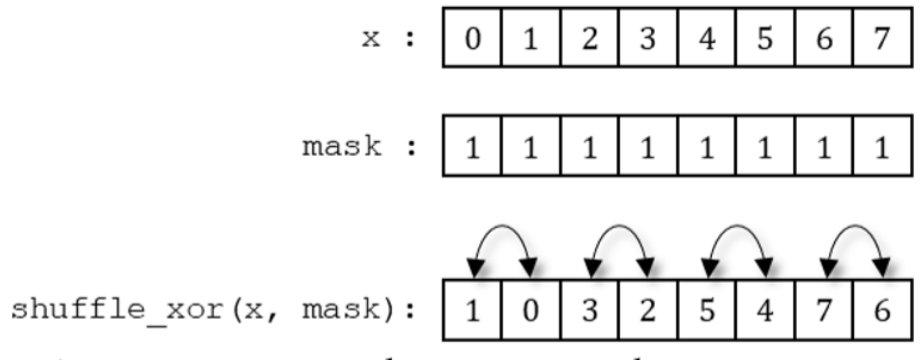
- 기존 순차 프로그램에 pointer와 array가 많은 복잡한 자료구조가 있으면 이걸 버퍼로 옮기기 어려움
  - 버퍼의 '주소'를 알 수 없기 때문
- Unified Shared Memory는 호스트-디바이스가 사이의 공유 메모리 주소 공간 제공
  - `malloc_shared()`로 할당 가능

```
struct Foo { int *p; ... };
Foo *data = malloc_shared<Foo>(1, q);
data->p = malloc_shared<int>(1024, q);
...
q.submit([&](handler &h) {
  h.parallel_for(range<1>(1024), [=](id<1> idx) {
    data->p[idx] += 1;
  });
});
```

DPC++ Sub-Group 내 통신

- DPC++은 워크-아이템 간에 데이터를 교환할 수 있는 범위가 매우 제한적
- Sub-groups은 디바이스의 vector hardware를 공유하는 워크-아이템의 집합
  - 예를 들어 인접한 워크-아이템 32개
- Sub-group 안에서는 하드웨어적으로 빠른 데이터 교환이 가능
  - Shuffle, reduction 등

```
h.parallel_for(nd_range<1>(...),
                    [=](nd_item<1> item) {
  intel::sub_group sg = item.get_sub_group();
  ...
  data[i] = sg.shuffle_xor(data[i], 1);
  ...
});
```



Conclusion

- DPC++은 SYCL 기반으로 SPMD 프로그래밍 모델 제공
- 큐를 만들고 커널 실행 명령을 추가하여 디바이스에서 병렬 연산이 가능
- 버퍼를 할당하여 입출력 데이터를 저장하며 accessor를 통해 접근
- 런타임 시스템은 data dependence 기반으로 알아서 동기화 수행
- DPC++은 USM, sub-group 등 최신 하드웨어의 지원을 받아 성능과 편의성을 높이는 다양한 확장 기능을 제공함

감사합니다!

MANYCORESOFT