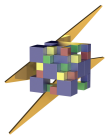
Lecture 21

OpenCL vs CUDA

이재진

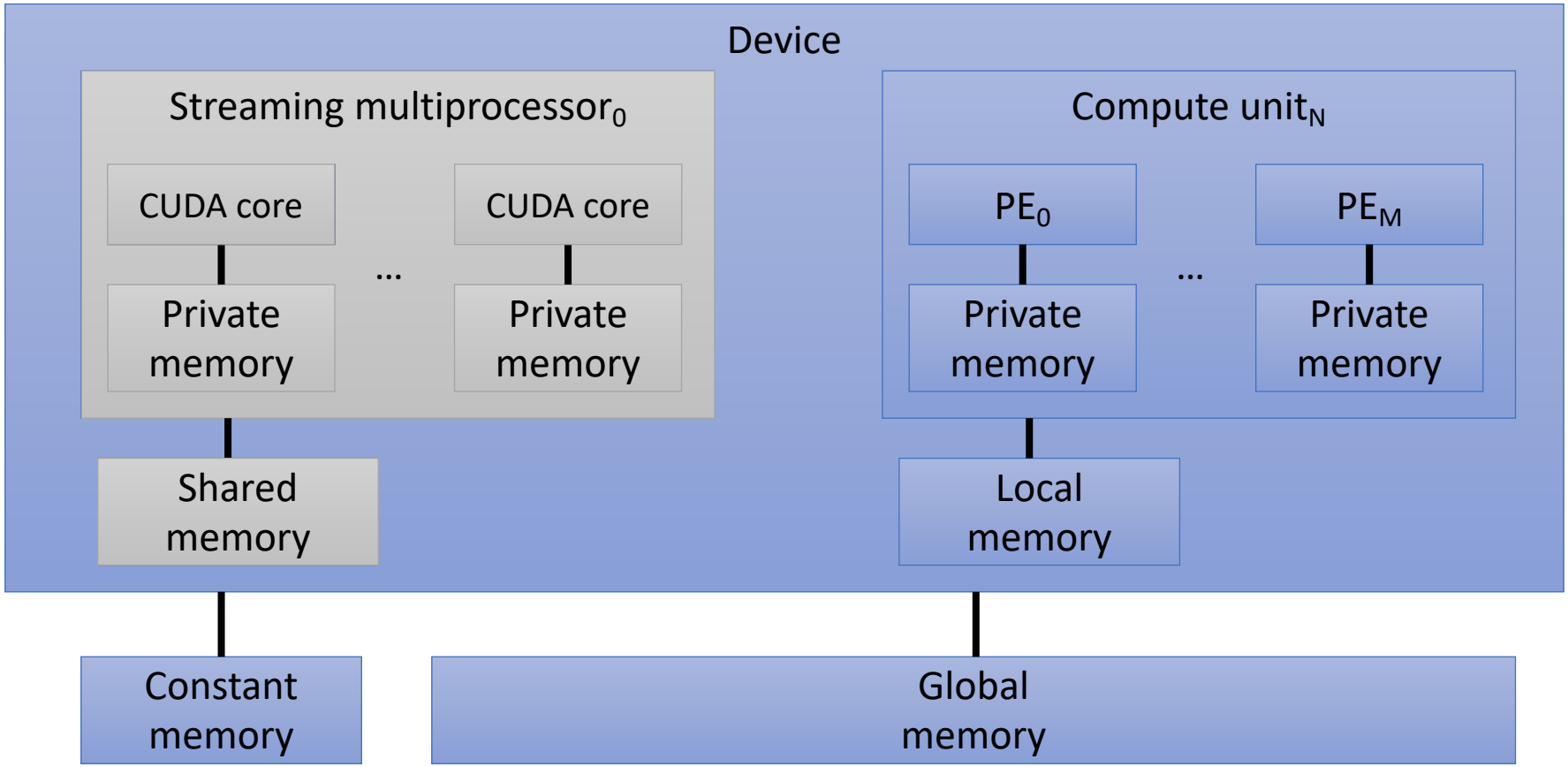
서울대학교 컴퓨터공학부

http://aces.snu.ac.kr



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

Platform Model



OpenCL과 CUDA의 차이점

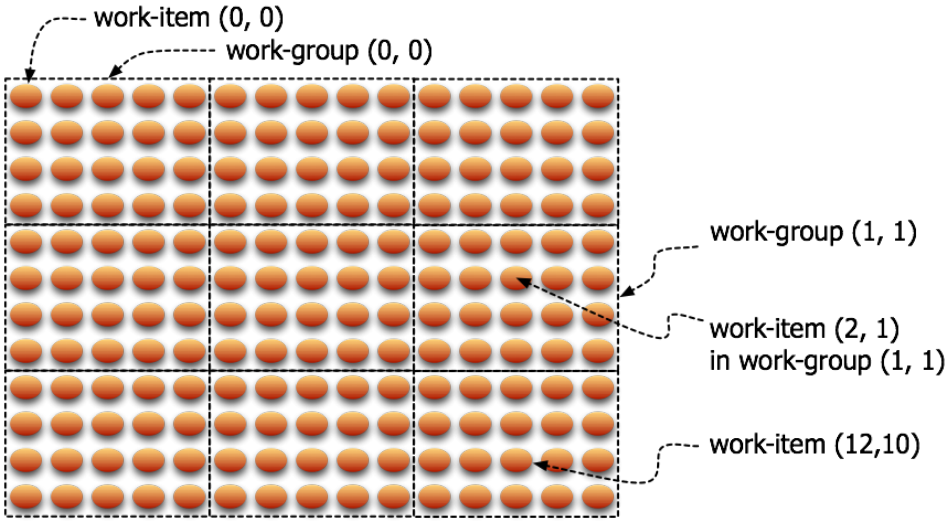
- 플랫폼
- NDRange와 grid
- 호스트 코드와 디바이스 코드
- 디바이스 코드 빌드 방식
- 함수 타입
- 서브 디바이스
- 포인터
- C++ 지원
- 벡터 타입
- 디바이스 빌트인 함수
- 디바이스 메모리 할당 방법

OpenCL's Platform Model

- OpenCL
  - 다양한 회사의 하드웨어를 지원하기 위해 플랫폼 모델을 도입
    - AMD OpenCL 플랫폼
    - Intel OpenCL 플랫폼
    - NVIDIA OpenCL 플랫폼
- CUDA
  - NVIDIA의 GPU만 지원
  - 플랫폼의 개념이 필요하지 않음

NDRange와 grid

- NDRange
  - gws[2] = {15, 12}
  - lws[2] = {5, 4}
- Grid
  - thread_blocks = {3, 3}
  - threads = {5, 4}



호스트 코드와 디바이스 코드

- OpenCL
  - 호스트 코드와 디바이스 코드가 분리되어 있음
  - 디바이스 코드
    - 호스트 프로그램에서는 문자열
    - clBuildProgram을 통해 바이너리 코드를 생성
- CUDA
  - 호스트 코드와 디바이스 코드가 하나의 소스코드를 구성

호스트 코드와 디바이스 코드 (contd.)

- OpenCL

```
// 디바이스 코드
// 호스트 코드에서는 문자열
const char* source_code = "__kernel void …";

int main() {
  …
  // 문자열을 통해 프로그램을 생성 후, 빌드
  cl_program program;
  program = clCreateProgramWithSource(context, 1,
      &source_code, &source_len, &err);
  err = clBuildProgram(program, 1, &dev,
      build_opts, NULL, NULL);

  // 빌드된 커널을 실행
  size_t lws[2] = {16, 16};
  size_t gws[2] = {COL_B, ROW_A};
  clEnqueueNDRangeKernel(cmd_queue, kernel, 2,
      NULL, gws, lws, 0, NULL, NULL);
  …
}
```

- CUDA

```
// 디바이스 코드
__global__ void mat_mul(…) {
}

// 호스트 코드
int main() {
  …

  // 커널을 실행
  mat_mul<<<nblocks, nthreads>>>(…);

  …
}
```

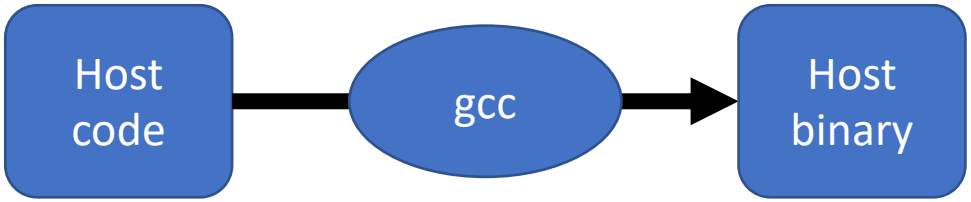
디바이스 코드 빌드 방식

- OpenCL
  - 디바이스 코드가 별도로 빌드됨
  - 런타임 빌드
    - 디바이스 코드를 clBuildProgram을 통해 빌드
  - 런타임 로드
    - 이미 빌드된 바이너리를 clCreateProgramWithBinary를 통해 로드

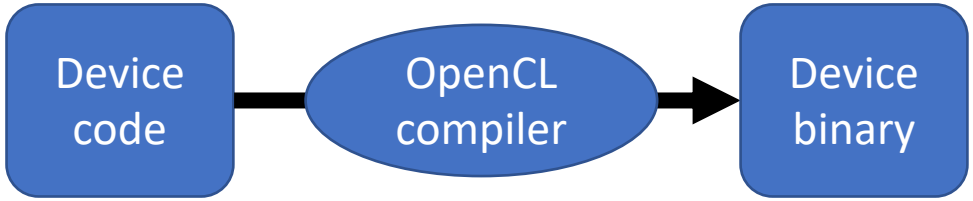
- CUDA
  - 디바이스 코드가 호스트 코드와 함께 빌드됨
  - nvcc program.cu

디바이스 코드 빌드 방식 (cont'd)

- OpenCL
  - 호스트 코드 빌드

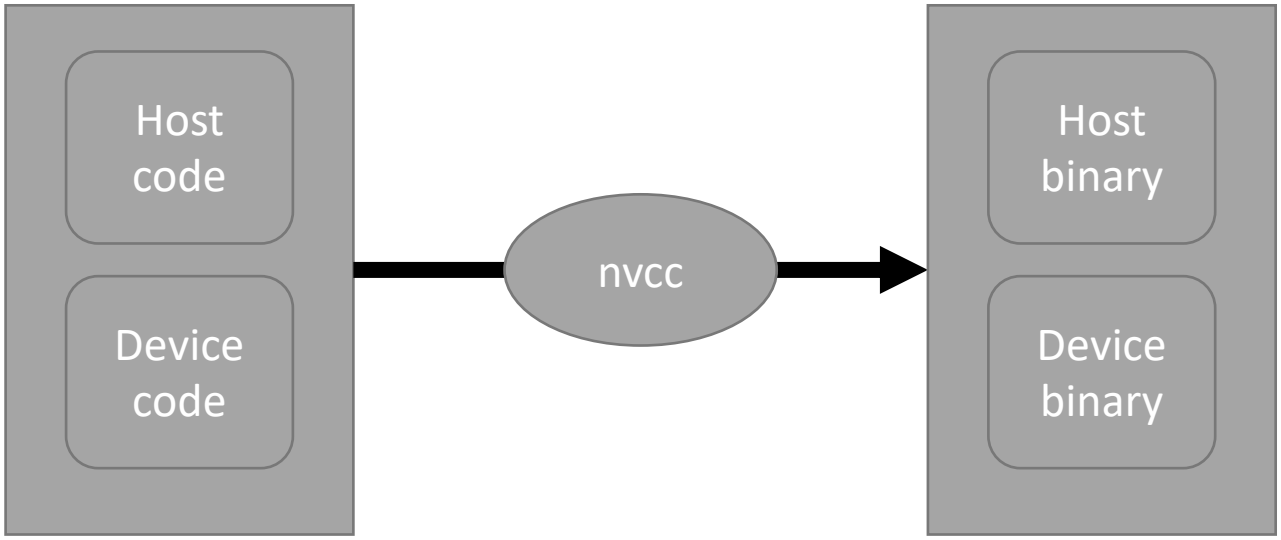


  - 디바이스 코드 빌드



디바이스 코드 빌드 방식 (cont'd)

- CUDA
  - 호스트 코드와 디바이스 코드가 함께 빌드됨



함수 타입

- OpenCL
  - 호스트 코드와 디바이스 코드가 분리됨
  - 디바이스 코드 내 함수 타입
    - 호스트에서 부를 수 있는 함수 (커널)
    - 호스트에서 부를 수 없는 함수

- CUDA
  - 호스트 코드와 디바이스 코드가 합쳐져 있음
  - 함수 타입
    - 호스트 함수
    - 디바이스 함수
      - 호스트 함수가 부를 수 있는 디바이스 함수 (커널)
      - 호스트 함수가 부를 수 없는 디바이스 함수

함수 타입 (cont'd)

- OpenCL

```
// 호스트에서 부를 수 있는 함수 (커널)
// __kernel 키워드가 있음
// 함수의 return 타입은 void이어야 함
__kernel void opencl_kernel(…) {
}

// 호스트에서 부를 수 없는 함수
// __kernel 키워드가 없음
int add_function(…) {
}
```

- CUDA

```
// 호스트 함수
// 일반적인 함수.
// __host__는 생략가능
__host__ int normal_c_function(…) {
}

// 호스트가 부를 수 있는 디바이스 함수 (커널)
// __global__ 키워드가 있음
// 함수의 return 타입은 void이어야 함
__global__ void cuda_kernel(…) {
}

// 호스트가 부를 수 없는 디바이스 함수
// __device__ 키워드가 있음
__device__ int add_function(…) {
}
```

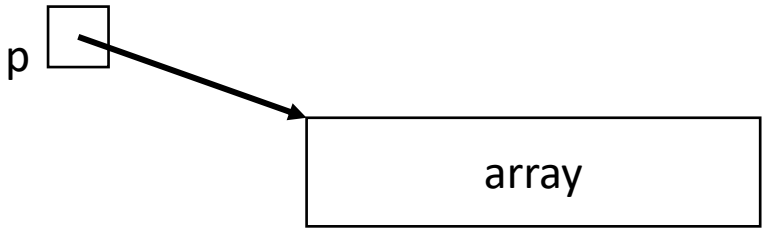
서브 디바이스

- OpenCL
  - 하나의 디바이스에서 여러 개의 서브 디바이스 생성 가능
    - clCreateSubDevices()
    - 디바이스에서 해당 기능을 지원해야 함
      - clGetDeviceInfo()를 통해 확인 가능
- CUDA
  - 서브 디바이스 생성 불가능

포인터

- 포인터의 두 가지 메모리 영역
  - 포인터가 할당된 메모리 영역
  - 포인터가 가리키는 메모리 영역

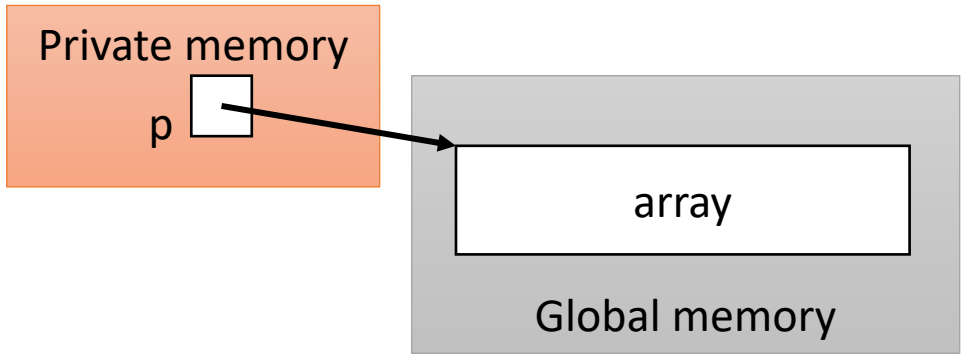
```
int array[100];
int* p = array;
```



OpenCL C에서의 포인터

- Address space qualifier
- p가 가리키는 영역을 표현

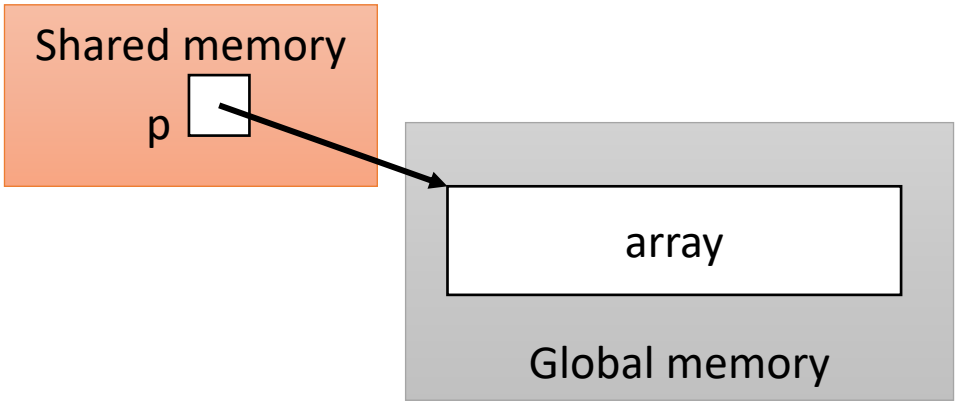
```
__kernel void opencl_kernel(__global int* array) {
  // p는 __private 영역에 있고, p가 가리키는 영역이 __global에 있음을 의미
  __global int* p = array;
}
```



CUDA C에서의 포인터

- Variable type qualifier
- 변수가 선언될 때 해당 변수가 생성되는 메모리 공간을 지정

```
__global__ void cuda_kernel(int* array) {
  // p가 shared 메모리 영역에 할당됨을 의미
  __shared int* p;
  p = array;
}
```



C++ 지원

- OpenCL
  - OpenCL 2.1 이전은 디바이스 코드에서 C++코드를 지원하지 않음
  - OpenCL 2.1에서 C++ 지원

- CUDA
  - C++코드 사용 가능

벡터 타입

- OpenCL
  - 2, 3, 4, 8, 16개의 원소를 가진 벡터 타입 지원
    - 예) char2, long3, ulong16
  - 다양한 접근 방법 제공
    - x, y, z, w, hi, lo, even, odd, si 등

- CUDA
  - 1, 2, 3, 4개의 원소를 가진 벡터 타입 지원
    - 예) char1, long3, float4
  - 접근 방법
    - x, y, z, w만 제공

디바이스 빌트인 함수

- OpenCL
  - 다양한 하드웨어를 지원해야 함
  - 보다 일반적인 빌트인 함수들을 제공
- CUDA
  - NVIDIA GPU의 특성에 맞춘 다양한 함수 제공 가능
  - 하드웨어를 더 세밀하게 조작 가능

디바이스 메모리 할당 방법

		OpenCL	CUDA
Local (shared) 메모리 할당	정적	O	O
	동적	O	O
Constant 메모리 할당	정적	O	O
	동적	O	X
Global 메모리 할당	정적	X	O
	동적	O	O

정적 local (shared) 메모리 할당

- OpenCL

```
__kernel void opencl_kernel() {

  __local int static_local[32];
}
```

- CUDA

```
__global__ void cuda_kernel() {

  __shared__ int static_shared[32];
}
```

동적 local (shared) 메모리 할당

- OpenCL

```
__kernel void opencl_kernel(
    __local int* dynamic_local) {

}
```

```
int main() {
  // 마지막 파라미터는 NULL이어야 함.
  clSetKernelArg(k, 0, 32*sizeof(int), NULL);
}
```

- CUDA

```
__global__ void cuda_kernel() {

  extern __shared__ int static_shared[];
}


int main() {

 // execution configuration의 3번째 파라미터로
 // shared memory의 크기를 지정
  cuda_kernel<<<nblocks, nthreads,
                32*sizeof(int)>>>();

}
```

동적 local (shared) 메모리 할당 (계속)

- OpenCL

```
// 2개의 local memory object를 생성
__kernel void opencl_kernel(
    __local int* dynamic_local1,
    __local int* dynamic_local2) {
}
```

```
int main() {
  // 마지막 파라미터는 NULL이어야 함.
  clSetKernelArg(k, 0, 32*sizeof(int), NULL);
  clSetKernelArg(k, 1, 32*sizeof(int), NULL);
}
```

- CUDA

```
// 2개의 shared memory object를 생성
__global__ void cuda_kernel() {

  extern __shared__ int static_shared[];
  int* static_shared1 = &static_shared[0];
  int* static_shared2 = &static_shared[32];
}


int main() {

  // execution configuratio의 3번째 파라미터로
  // shared 메모리의 크기를 지정
  cuda_kernel<<<nblocks, nthreads,
                64*sizeof(int)>>>();

}
```

정적 constant 메모리 할당

- OpenCL

```
// constant memory object 생성
// 커널 코드의 글로벌 변수로 정의
// 호스트 코드에서 초기화 불가능
__constant int static_constant[32] =
                                {1,2,3,4};
__kernel void opencl_kernel() {
}
```

```
int main() {
  …
}
```

- CUDA

```
// constant memory object 생성
// 글로벌 변수로 정의
__constant int static_constant[32] =
                                {1,2,3,4};

__global__ void cuda_kernel() {
}

int main() {
  …
  int buf[32] = {4,5,6,7};

  // 호스트 코드에서 초기화 가능
  cudaMemcpyToSymbol(static_constant, buf,
                     32*sizeof(int));
}
```

동적 constant 메모리 할당

- OpenCL

```
__kernel void opencl_kernel(
  __constant int* dynamic_constant) {
}
```

```
int main() {
  cl_mem mem = clCreateBuffer(context,
        CL_MEM_READ_ONLY, 32*sizeof(int),
        NULL, &err_code);

  clSetKernelArg(k, 0, sizeof(cl_mem), &mem);
}
```

- CUDA

```
// 동적 할당은 불가능.
// 최대 크기로 할당한 후, 일부만 사용
__constant int static_constant[MAX_CONST_SIZE];

__global__ void cuda_kernel() {
}

int main() {
  …
  int buf[32] = {4,5,6,7};

  // 호스트 코드에서 초기화 가능
  cudaMemcpyToSymbol(static_constant, buf,
                     32*sizeof(int));
}
```

정적 global 메모리 할당

- OpenCL
  - 불가능
  - 동적 메모리 할당을 통해 해결 가능

- CUDA

```
// constant memory object 생성
// 글로벌 변수로 정의
__device__ int static_global[32] = {1,2,3,4};

__global__ void cuda_kernel() {
}

int main() {
  …
  int buf[32] = {4,5,6,7};

  // 호스트 코드에서 초기화 가능
  cudaMemcpyToSymbol(static_global, buf,
                     32*sizeof(int));
}
```

동적 global 메모리 할당

- OpenCL

```
__kernel void opencl_kernel(
  __global int* dynamic_global) {
}
```

```
int main() {
  ...
  cl_mem mem = clCreateBuffer(context,
        CL_MEM_READ_WRITE, 32*sizeof(int),
        NULL, &err_code);

  clSetKernelArg(k, 0, sizeof(cl_mem), &mem);
  ...
}
```

- CUDA

```
__global__ void cuda_kernel(
           int* dynamic_global) {
}

int main() {
  …
  int* d_global;

  cudaMalloc(&d_global, 32*sizeof(int));

  cuda_kernel<<<nblocks, nthreads>>>(d_global);
}
```

용어 정리

OpenCL	CUDA
Work-item	Thread
Work-group	Thread block
NDRange	Grid
Global memory	Global memory
Local memory	Shared memory
Constant memory	Constant memory
Private memory	Register
Processing element	CUDA core
Compute unit	Streaming multiprocessor