

Language

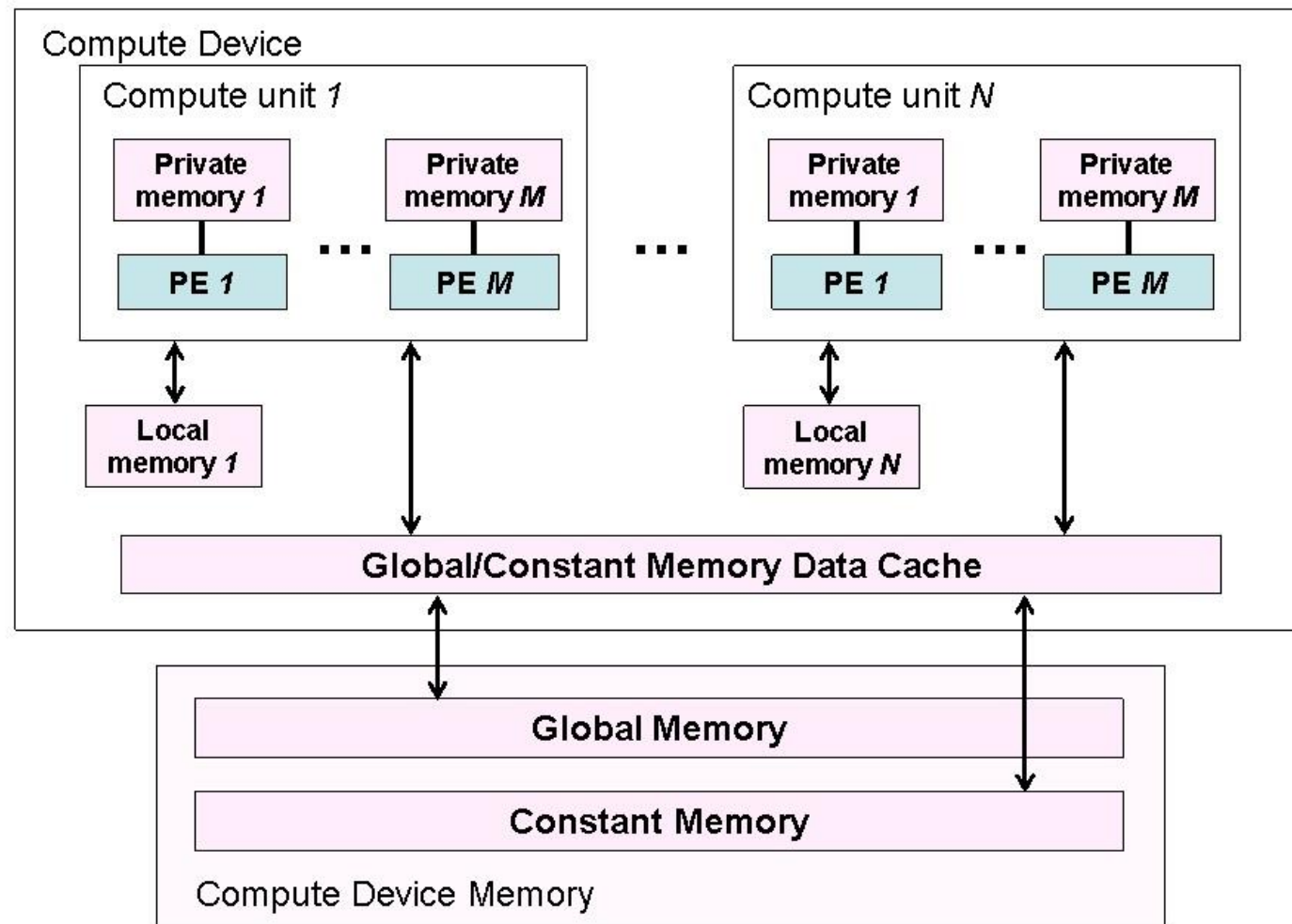
1. OpenCL (OPEN Computing Language)

- 다양한 device, platform을 지원해준다
- CUDA에 비해 코드가 복잡하다

2. CUDA (Compute Unified Device Architecture)

- NVIDIA 제품만 지원한다
- OpenCL에 비해 코드가 간결하다
- 다양한 Library들이 존재한다

OpenCL 구조



OpenCL 동작 과정

Setup 단계	
Platform 정보 읽기	AMD OpenCL , Intel OpenCL , NVIDIA OpenCL
Device 정보 읽기	NVIDIA or AMD GPU , FPGA , Intel Xeon
Context 생성	Kernel이 실제로 실행되는 환경을 생성하는 단계
OpenCL program compile	OpenCL application 실행 중에 kernel compile
Command queue 생성	Device 마다 하나 이상의 queue가 필요
Memory object 생성	Device의 global 혹은 constant memory에 공간 할당
Kernel code 실행	
Data를 GPU memory로 전송	입력 값을 전송
Kernel의 argument 설정	전송 된 입력 값들 중 kernel argument를 구분해 줌
Kernel 실행	동시에 병렬로 실행
Data를 GPU memory에서 전송	결과 값을 읽음

OpenCL simple code

```
err = clGetPlatformIDs(1, &platform, NULL);
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, NUM_DEVICES, &device, NULL);
context = clCreateContext(NULL, NUM_DEVICES, &device, NULL, NULL, &err);
queue = clCreateCommandQueue(context, device, 0, &err);

kernel_source = get_source_code("kernel.cl", &kernel_source_size);
program = clCreateProgramWithSource(context, 1, (const char **)&kernel_source, &kernel_source_size, &err);
err = clBuildProgram(program, NUM_DEVICES, &device, "", NULL, NULL);
```

```
kernel = clCreateKernel(program, "vec_add", &err);
```

```
int A[16384], B[16384], C[16384];
for (int idx = 0; idx < 16384; ++idx) {
    A[idx] = rand() % 100;
    B[idx] = rand() % 100;
}
```

```
cl_mem buffer_A, buffer_B, buffer_C;
```

```
buffer_A = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) * 16384, NULL, &err);
buffer_B = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) * 16384, NULL, &err);
buffer_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int) * 16384, NULL, &err);
```

```
__kernel void vec_add(__global int *A, __global int *B, __global int *C)
{
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
```

Kernel code in kernel.cl

```
err = clEnqueueWriteBuffer(queue, buffer_A, CL_FALSE, 0, sizeof(int) * 16384, A, 0, NULL, NULL);
err = clEnqueueWriteBuffer(queue, buffer_B, CL_FALSE, 0, sizeof(int) * 16384, B, 0, NULL, NULL);
```

```
err = clSetKernelArg(kernel, 0, CL_MEM_SIZE, &buffer_A);
err = clSetKernelArg(kernel, 1, CL_MEM_SIZE, &buffer_B);
err = clSetKernelArg(kernel, 2, CL_MEM_SIZE, &buffer_C);
```

```
size_t global_size = 16384;
size_t local_size = 256;
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);

err = clEnqueueReadBuffer(queue, buffer_C, M2S_TRUE, 0, sizeof(int) * 16384, C, 0, NULL, NULL);
```

Setup Phase

Running kernel

문제점

1. 프로그래머가 device의 대부분을 책임진다.

- Device의 global memory를 얼마나 쓸지 할당해야 한다. → clCreateBuffer
- CPU와 GPU간의 Data 전송을 직접 명시해야 한다. → clEnqueueReadBuffer / clEnqueueWriteBuffer
- 만약 어떤 문제로 data 전송과 kernel 실행 코드가 섞이게 되면 더욱 복잡한 코드가 된다.

2. 좋은 Performance를 얻기 위해서는 device들의 특성을 알고 있어야한다.

- OpenCL은 다양한 device들을 지원한다. (GPU, FPGA, Xeon, ...)
- Device들은 서로 다른 스펙을 가지고 있다.
 - GTX 1060 1280 cores, global memory 6GB, local memory 1.5 MB
 - GTX 1080 2560 cores, global memory 8GB, local memory 2 MB
 - GTX 1080 Ti 3584 cores, global memory 11GB, local memory 2.75 MB
 - ...
- 같은 코드라도 warp 수, work group 개수 등이 다르다면 같은 device에서도 성능차이가 난다.

M2S

1. 공유 가능한 부분은 공유한다.

- Platform, Context, Program 등의 하나를 공유 할 수 있다면 여러 device들이 공유한다.

2. 공유 할 수 없는 자원들은 struct로 묶는다.

- Device_id, Command_queue 등은 device 마다 고유해야 한다.
- Struct로 묶어 하나로 관리한다.

```
typedef struct M2S_DEVICE {  
    cl_uint      num_entries;  
    cl_device_id *devices;  
    // device hint ratio  
    size_t       *device_hint;  
}m2s_device_id;
```

- Device마다 힌트를 생성한다. ← 핵심 point

M2S

3. 최대한 OpenCL API와 비슷하게 제작한다.

- OpenCL과 비슷한 API로 만들어 하나의 device를 다루고 있다고 생각할 수 있도록 한다.
- OpenCL의 API들과 1:1 대응

clGetPlatformIDs	:	m2sGetPlatformIDs
clGetDeviceIDs	:	m2sGetDeviceIDs
clCreateContext	:	m2sCreateContext

등과 같이 앞부분을 m2s로 바꾼 후, 기능도 OpenCL과 똑같이 만들도록 노력한다.

M2S – Hint 생성

```
cl_uint num_devices;
err = clGetDeviceIDs(platforms[idx], CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
devices[idx] = (cl_device_id *)malloc(sizeof(cl_device_id) * num_devices);
err = clGetDeviceIDs(platforms[idx], CL_DEVICE_TYPE_ALL, num_devices, devices[idx], NULL);
for (cl_uint jdx = 0; jdx < num_devices; ++jdx)
{
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_NAME, 1024, name, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_TYPE, sizeof(cl_device_type), &type, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_VENDOR, 1024, name, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &cu, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t), &wg, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(cl_ulong), &mem, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_LOCAL_MEM_SIZE, sizeof(cl_ulong), &mem, NULL);
    err = clGetDeviceInfo(devices[idx][jdx], CL_DEVICE_MAX_MEM_ALLOC_SIZE, sizeof(cl_ulong), &mem, NULL);
}
```

- OpenCL에서는 Device에 대한 정보를 알 수 있도록 API를 제공한다.
- 현재 github에 구현되어 있는 코드는 아직 1/n로 동작하게 되어있다. ← 수정 필요

M2S – data 전송 및 kernel 실행

- **Device 힌트에 따라서 적절하게 나눈다.**

- 힌트에 따라서 아래와 같이 코드를 반복한다.

```
size_t current_size = 0;
size_t total_size = 0;

for (m2s_uint idx = 1; idx < num; ++idx)
{
    current_size = size * device->device_hint[idx] / 100;


    err = clEnqueueReadBuffer(command_queue.queues[idx], buffer.mems[idx], CL_FALSE, 0, current_size, (void *)((int *)ptr + total_size), 0, NULL, NULL);
    if (err != CL_SUCCESS) {
        printf("EnqueueWriteBuffer Error: cannot write buffer\n");
        return err;
    }

    total_size += current_size;
}
```

M2S – Simple Test (Vector Addition)

현재까지 완성된 M2S로 간단한 vector addition 실험

- 하지만 현재 GPU 부족으로 single-gpu system 상에서 밖에 실험 못 함
- Single-gpu system에서는 data나 task를 나눌 필요가 없기 때문에 잘 동작함

 Microsoft Visual Studio 디버그 콘솔

```
verification success
```

```
D:\workspace\m2s_opencv\Debug\m2s_opencv.exe(12316 프로세스)이(가) 0 코드로 인해 종료되었습니다.  
이 창을 닫으려면 아무 키나 누르세요.
```