

Migration from OpenCLIPER V1 to V2

La versión 2 de OpenCLIPER incluye nuevas estructuras para almacenar los datos en los distintos tipos de memoria de dispositivo. Este cambio tiene como objetivo simplificar la gestión de memoria dinámica y facilitar futuras ampliaciones del marco de trabajo. Estos cambios internos suponen algunos cambios en la API pública de OpenCLIPER.

Resumen de cambios en la API pública

- No se pueden utilizar "handles" (índices usados en estructuras internas de datos). En su lugar, se utilizan directamente punteros inteligentes compartidos (también en las estructuras *InitParameters* y *LaunchParameters* usadas por las clases derivadas de *Process*).
- No se pueden usar ya las llamadas al método *CLapp::addData* (para asociar objetos de clases derivadas de *Data* al objeto *CLapp*), los objetos se asocian a un objeto *CLapp* y se copian sus datos implícitamente a memoria de dispositivo durante su creación. Dicho método ya no está disponible.
- No se pueden usar ya las llamadas a *CLapp::delData* como parte de la liberación de recursos asociados a un objeto (las tareas necesarias se realizan automáticamente en el destructor de todas las clases derivadas de *Data*). Dicho método ya no está disponible.
- No se pueden usar ya las llamadas a *CLapp::getData* para obtener un objeto de una clase derivada de *Data* a partir de un "handle".
- Los métodos para obtener los punteros a memoria de dispositivo o a memoria de host asociada a memoria de dispositivo, que guarda datos en formato buffer o imagen, ya no forman parte de la clase *NDArray* (dichos punteros no se guardan en ella). En su lugar, se deben usar los métodos del mismo nombre de la clase *Data* que llevan como parámetro el índice del *NDArray* del que se quiere leer memoria. La tabla a continuación muestra dichos métodos en la versión 1 y sus equivalentes de la versión 2.

V1 method	V2 method	function
void* NDArray::getDeviceBuffer()	void* Data::getDeviceBuffer(dimIndexType index);	get pointer to buffer in device memory storing data of 1 NDArray
void* NDArray::getHostBuffer()	void* Data::getHostBuffer(dimIndexType index);	get pointer to buffer in host mapped memory storing data of 1 NDArray
void* NDArray::getDeviceImage()	void* Data::getDeviceImage(dimIndexType index);	get pointer to image in device memory storing data of 1 NDArray

V1 method	V2 method	function
void* NDArray::getHostImage()	void* Data::getHostImage(dimIndexType index);	get pointer to image in host mapped memory storing data of 1 NDArray

- Los constructores de todos los objetos derivados de *Data* llevarán como primer parámetro un argumento de tipo *std::shared_ptr<CLapp>* (para poder asociarlos a este objeto y copiar automáticamente su contenido a memoria de dispositivo).
- La asociación de datos de entrada y de salida a un objeto de una clase derivada de *Process* se realizarán mediante los métodos:
 - *setInput*
 - *setOutput*
 (ya no están disponibles los métodos *setInHandle* ni *setOutHandle*).
- Se han añadido nuevos constructores (de uso opcional) a la clase *CLapp* para realizar en una única llamada la creación, inicialización y carga de uno o varios ficheros que contienen kernels.
- Se ha añadido un nuevo constructor (de uso opcional) a la clase *Process* para realizar en una única llamada la creación y asignación de los datos de entrada y salida.

Guía de migración de OpenCLIPER V1 a V2

Los cambios necesarios para adaptar una aplicación desarrollada en OPenCLIPER V1 a V2 se dividen en cambios necesarios y cambios opcionales (pero recomendados), y son los que se describen a continuación.

Cambios obligatorios

- Nueva forma de creación de objetos de clases derivadas de *Data*, *Process* o de la clase *CLapp*
 - El puntero resultado de la creación se debe guardar en un puntero inteligente de tipo único (clases derivadas de *Process*) o de tipo compartido (resto de clases). Además, se debe usar la clase más específica necesaria, no alguna de sus superclases (para evitar la necesidad de usar casts explícitos de superclase a subclases).
 - Ejemplo 1: creación de un objeto de tipo *XData*

- la línea de la versión 1:

```
Data* pData = new XData(parámetros);
```

- debe ser sustituida por la línea:

```
std::shared_ptr<XData> pData = std::make_shared(parámetros);
```

- Ejemplo 2: creación de un objeto de un tipo derivado de *Process* (no existe *std::make_unique* en C++11, sí en C++17)

- la línea de la versión 1:

```
Process* pProcess = new MyProcess(parámetros);
```

- debe ser sustituida por la línea

```
std::unique_ptr<MyProcess> pProcess(new MyProcess(parámetros));
```

- Nuevo parámetro de tipo *CLapp* obligatorio para la creación de objetos de clases derivadas de *Data*:

- El primer parámetro de los constructores de estas clases debe ser un puntero inteligente compartido que apunte a un objeto de tipo *CLapp* ya creado.
- Ejemplo:

- la líneas de la versión 1:

```
CLapp* pCLapp = new CLapp();
Data* pXData = new XData(parámetros_sin_pCLapp);
```

- deben ser sustituidas por la línea:

```
std::shared_ptr<CLapp> pCLapp = std::make_shared();
std::shared_ptr<XData> pXData =
    std::make_shared(pCLapp, parámetros_sin_pCLapp);
```

- Cambios en la asociación de objetos de clases derivadas de *Datas* a los parámetros de inicialización o de lanzamiento de procesos:

- Todos los parámetros de tipo *DataHandle* de las estructuras de datos *InitParameters* y *LaunchParameters* deben ser sustituidos por parámetros de tipo *std::shared_ptr<Clase>*, donde **Clase** puede ser *XData*, *KData*, *SensitivityMapsData*, *SamplingMasksData*, ...
- Ejemplo:

- las líneas de la versión 1:

```
struct LaunchParameters:Process::LaunchParameters {
    ...
    DataHandle sensitivityMapsDataHandle=INVALIDDATAHANDLE;
    ...
};
```

- debe ser sustituidas por las líneas:

```
struct LaunchParameters:Process::LaunchParameters {
    ...
    std::shared_ptr<SensitivityMapsData> sensitivityMapsData = nullptr;
    ...
};
```

- Nueva forma de asociar datos de entrada y salida a un proceso: se deben usar los métodos *setInput* y

setOutput en lugar de *setInHandle* y *setOutHandle*.

○ Ejemplo:

- las líneas de la versión 1:

```
pProcess->setInHandle(inHandle);  
pProcess->setOutHandle(outHandle);
```

- deben ser sustituidas por:

```
pProcess->setInput(pInData);  
pProcess->setOutput(pOutData);
```

(donde *inHandle* era el handle asociado a *pInData*, obtenido al realizar un *addData(pInData)*, y *outHandle* era el handle asociado a *pOutData*, obtenido de la misma forma).

- Nueva forma de obtener punteros a los objetos que contienen los datos de entrada y salida asociados a un proceso desde el código de éste: se deben usar los métodos *getInput* y *getOutput* de la clase *Process* en lugar del método *getData* de la clase *CLapp* usando como parámetro el "handle" asociado a los datos de entrada o salida.

○ Ejemplo:

- las líneas de la versión 1:

```
... = getApp()->getData(inHandle);  
... = getApp()->getData(outHandle);
```

- deben ser sustituidas por:

```
... = getInput();  
... = getOutput();
```

(donde *inHandle* era el handle asociado a *pInData*, obtenido al realizar un *addData(pInData)*, y *outHandle* era el handle asociado a *pOutData*, obtenido de la misma forma).

- Nueva forma de obtener punteros a memoria que almacena datos desde dentro del código de procesos:

- Los métodos *getHostBuffer*, *getDeviceBuffer*, *getHostImage*, *getDeviceImage* se deben invocar sobre los objetos de clases derivadas de *Data*, no sobre los *NDArrays* de estas.

○ Ejemplo:

- las líneas de la versión 1:

```
getApp()->getData(handle)->getNDArrays()->at(i)->getDeviceBuffer();
```

- se deben sustituir por :

```
getInput()->getDeviceBuffer(i);
```

- Cambios en la forma de liberar los recursos asociados a los objetos creados:
 - No es posible usar el método *delApp* de la clase *CLapp* (sus tareas se realizan automáticamente en los destructores de las clases).
 - La liberación de objetos guardados en punteros inteligentes compartidos se realiza asignándoles el valor *nullptr*.
 - La liberación de objetos guardados en punteros inteligentes únicos se realiza llamando al método *reset(nullptr)* del puntero.
 - Ejemplo:

- las líneas de la versión 1:

```
delete pProcess;
pCLapp->delData(inHandle);
pCLapp->delData(outHandle);
delete pInData;
delete pOutData;
delete pCLapp;
```

- deben ser sustituidas por:

```
pProcess.reset(nullptr);
pInData = nullptr;
pOutData = nullptr;
pCLapp = nullptr;
```

Cambios opcionales (pero recomendados)

- Simplificación de las tareas de creación de un objeto de la clase *CLapp*, su inicialización y la carga de ficheros que contienen kernels:
 - Se recomienda usar los nuevos constructores de la clase *CLapp* que contienen los parámetros de configuración de plataformas, configuración de dispositivos y nombres del fichero o ficheros que contienen los kernels que deben ser cargados.
 - Ejemplo:

- las líneas de la versión 2 sin simplificaciones:

```
CLapp::PlatformTraits platformTraits;
CLapp::DeviceTraits deviceTraits;
deviceTraits.type = CLapp::DEVICE_TYPE_GPU;
std::shared_ptr<CLapp> pCLapp = std::make_shared<CLapp>();
pCLapp->init(platformTraits, deviceTraits);
pCLapp->loadKernels("examples/negate.cl");
```

- pueden ser sustituidas por:

```
CLapp::PlatformTraits platformTraits;
CLapp::DeviceTraits deviceTraits;
deviceTraits.type = CLapp::DEVICE_TYPE_GPU;
std::shared_ptr<CLapp> pCLapp =
    std::make_shared<CLapp>(platformTraits, deviceTraits,
                           "examples/negate.cl");
```

- Simplificación en las tareas de creación de procesos y la asignación de sus datos de entrada y de salida:

- Se recomienda usar los nuevos constructores de la clase *Process* que contienen como parámetros los punteros a los objetos que contienen los datos de entrada y los de salida del proceso
- Ejemplo:
 - las líneas de la versión 2 sin simplificaciones:

```
std::shared_ptr<XData> pInData = std::make_shared(pCLapp, ...);
std::shared_ptr<XData> pOutData = std::make_shared(pCLapp, ...);
std::unique_ptr<MyProcess> pProcess(new negate(pCLapp));
pProcess->setInput(pInData);
pProcess->setOutput(pOutData);
```

- pueden ser sustituidas por:

```
std::shared_ptr<XData> pInData = std::make_shared(pCLapp, ...);
std::shared_ptr<XData> pOutData = std::make_shared(pCLapp, ...);
std::unique_ptr<MyProcess> pProcess(new Negate(pCLapp, pInData, pOutData))
;
```

- Simplificación en la copia de datos de memoria de dispositivo a memoria de host

- Se recomienda usar el nuevo método *device2Host* de la clase *Data* en lugar del método del mismo nombre de la clase *CLapp*
- Ejemplo:

- la línea de la versión 2 sin simplificaciones:

```
pCLapp->device2Host(pOutData, SyncSource::BUFFER_ONLY);
```

- puede ser sustituida por:

```
pOutData->device2Host(SyncSource::BUFFER_ONLY);
```

Documentation built with MkDocs (<http://www.mkdocs.org/>).