The OpenVX™ OpenCL™ Interop Extension

The Khronos OpenVX Working Group

Version 1.0 (provisional), 25th January 2018

Table of Contents

Copyright and license

Editor

Radhakrishna Giduthuri <radha.giduthuri@amd.com>

Technical Contributors

- Radhakrishna Giduthuri, AMD
- Niclas Danielsson, Axis Communications AB
- Thierry Lepley, Cadence
- Frank Brill, Cadence
- John MacCallum, Imagination
- Ben Ashbaugh, Intel
- Adam Herr, Intel

1. Introduction

This document details an extension to OpenVX 1.2, and references some APIs and symbols that may be found in that API, at https://www.khronos.org/registry/OpenVX/specs/1.2/html/index.html.

Refer to the OpenCL 1.2 specification at https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/.

The name of this extension is ***vx_khr_opencl_interop***.

1.1. Purpose

This OpenVX extension provides a mechanism for interoperation between an OpenVX implementation and an OpenCL application/user-kernel. Efficient communication is key to successful interoperation. The data exchange mechanism needs features such that:

- OpenCL data objects can be imported into the OpenVX environment
- OpenVX data objects can be accessed as OpenCL data objects
- fully asynchronous host-device operations are enabled during data exchange

1.2. Features of the extension specification

This interop extension supports six essential features.

- share common `cl_context` object between OpenVX and the OpenCL application
- share a set of common in-order `cl_command_queue` objects for coordination between OpenVX and the OpenCL application/user-kernel
- mechanism for an OpenCL application to export `cl_mem` buffers to OpenVX
- mechanism for an OpenCL application to reclaim exported `cl_mem` buffers back from OpenVX
- mechanism for an OpenCL application/user-kernel to temporarily map OpenVX data objects into `cl_mem` buffers
- mechanism to copy between `cl_mem` buffers and OpenVX data objects.

1.3. Preserve the asynchronous property of OpenCL

OpenCL has an asynchronous host API where command queues are executed asynchronously unless the application issues a blocking command. A `cl_mem` buffer is a handle to an opaque OpenCL data object. OpenCL kernels that consume such data objects can be enqueued into a command queue *before* the command that produces the content of this data object starts executing. This asynchronous property of the OpenCL API is fundamental, since it allows hiding various overheads/latencies such as creating OpenCL objects or launching kernels on an accelerator. The *vx_khr_opencl_interop* API intends to preserve this asynchronous property by avoiding unnecessary host-accelerator synchronizations when the ownership of data objects switch between OpenCL and OpenVX, or when data are copied between OpenCL and OpenVX.

There are two actors in the *vx_khr_opencl_interop* API:

- the application/user-kernel that uses OpenCL in addition to OpenVX
- the OpenVX implementation

When a `cl_mem` buffer is imported into OpenVX or when an OpenVX object is mapped as an OpenCL object, the ownership of the related data object temporarily changes from one actor to the other. Since the actual production of the data object content is considered as asynchronous, the actor that gets the ownership of the data object can't assume the data content is available. It then has two alternatives:

- launch asynchronous tasks (such as OpenCL kernels), making sure that these tasks will not start executing before the tasks of the other actor that produces the data object content are complete.
- wait until the data is ready before executing a synchronous task that consumes the content of the data object

To ensure the proper coordination between the two actors that exchange data objects, the *vx_khr_opencl_interop* API relies on an OpenCL command queue, referred to in this document as a ***coordination command queue***. A *coordination command queue* is the contract between the two actors exchanging the ownership of the data object. It allows both producer and consumer actors to make sure that any data production task of the former `cl_mem` owner actor is executed before any data consumption tasks of the new owner actor. The *coordination command queue* enables this coordination to happen asynchronously while still supporting synchronization if this is the choice of one of the actors.

1.4. Coordination Command Queues

There are two types of *coordination command queues*:

- A unique **global** *coordination command queue* for coordination between the application OpenCL workload and the OpenVX implementation, outside of OpenVX graphs. The application can provide the global *coordination command queue* when creating the OpenVX context using the `vxCreateContextFromCL` API; otherwise it will be created by the OpenVX implementation.
- **User node** *coordination command queues* for coordination between the user nodes and the OpenVX implementation. These *coordination command queues* will be created and managed by the OpenVX implementation. An OpenVX implementation may use one or more command queues across all the user nodes. The `VX_NODE_CL_COMMAND_QUEUE` node attribute can be queried from within `vx_kernel_initialize_f`, `vx_kernel_f`, and `vx_kernel_deinitialize_f` callbacks, to get access to a user node's *coordination command queue*.

When ownership of an OpenCL object changes from actor A (producer of data) to actor B (consumer of data), the Actor A must make sure that its tasks producing the exchanged `cl_mem` buffers are enqueued into the *coordination command queue* before the ownership change.

Note: In OpenCL, command queues are associated with a target device. In the context of the *vx_khr_opencl_interop* API, the device to which the *coordination command queue* is associated is not important, since it does not prevent actors from using other command queues targeting other devices. The application can manage any dependencies between command queues using OpenCL markers (or barriers) and events.

1.5. Usage Example of the *vx_khr_opencl_interop* API

In this example, OpenVX is used to process an image provided by a camera in the host memory. Then OpenCL is used to post-process the two outputs of the OpenVX graph on different OpenCL devices. The example goes through the following steps:

- create a `vx_context` using a `cl_context` supplied by the application
- create multiple `cl_mem` buffers for the input capture device to cycle through
- create graph output images using `vxCreateImageFromHandle()` with the memory type as `VX_MEMORY_TYPE_OPENCL_BUFFER`

- import a pre-verified OpenVX graph
- execute the OpenVX graph
- reclaim the OpenCL buffer and enqueue post-processing OpenCL kernels

```
#include <VX/vx_khr_opencl_interop.h>

  ...
  // Create an OpenCL context with two OpenCL devices.
  cl_context_properties ctxprop[] = {
      CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0, 0
  };
  cl_device_id device_list[2] = { device_id1, device_id2 };
  cl_context opencl_context;
  opencl_context = clCreateContext(ctxprop, 2, device_list, NULL, NULL, NULL);

  // Create in-order command queues for two different compute devices.
  cl_command_queue global_command_queue, second_command_queue;
  global_command_queue = clCreateCommandQueue(opencl_context, device_id1, 0, NULL);
  second_command_queue = clCreateCommandQueue(opencl_context, device_id2, 0, NULL);

  // Compile and get the OpenCL kernels.
  cl_program opencl_program = clCreateProgramWithSource(opencl_context, 1,
                                    (const char **) &KernelSource, NULL, NULL);
  clBuildProgram(opencl_program, 2, device_list, NULL, NULL, NULL);
  cl_kernel opencl_kernel1 = clCreateKernel(opencl_program, "my_kernel1", NULL);
  cl_kernel opencl_kernel2 = clCreateKernel(opencl_program, "my_kernel2", NULL);

  // Allocate OpenCL buffers.
  cl_mem opencl_buf1 = clCreateBuffer(opencl_context, CL_MEM_READ_WRITE, 640*480,
                                      NULL, NULL);
  cl_mem opencl_buf2 = clCreateBuffer(opencl_context, CL_MEM_READ_WRITE, 640*480,
                                      NULL, NULL);

  // Create the OpenVX context by specifying the OpenCL context and
  // the global coordination command queue.
  vx_context context = vxCreateContextFromCL(opencl_context, global_command_queue);

  // Get data input data from the camera capture API.
  void *camera_buffer = getBufferFromCamera();

  // Create the OpenVX input image (from camera host memory buffer).
  vx_imagepatch_addressing_t addr = {
      640, 480, 1, 640, VX_SCALE_UNITY, VX_SCALE_UNITY, 1, 1
  };
  vx_image input = vxCreateImageFromHandle(context, VX_DF_IMAGE_U8, &addr,
        &camera_buffer, VX_MEMORY_HOST);

  // Create OpenVX output images (from the opencl buffers).
  vx_image output1 = vxCreateImageFromHandle(context,
      VX_DF_IMAGE_U8, &addr, &opencl_buf1, VX_MEMORY_TYPE_OPENCL_BUFFER);
  vx_image output2 = vxCreateImageFromHandle(context,
      VX_DF_IMAGE_U8, &addr, &opencl_buf2, VX_MEMORY_TYPE_OPENCL_BUFFER);

  // Import a pre-verified OpenVX graph from memory at 'ptr' and size of 'len'
bytes.
```

```
  vx_reference refs[4] = { NULL, input, output1, output2 };
  vx_enum uses[4] = {
        VX_IX_USE_EXPORT_VALUES,      // graph
        VX_IX_USE_APPLICATION_CREATE, // input
        VX_IX_USE_APPLICATION_CREATE, // output1
        VX_IX_USE_APPLICATION_CREATE  // output2
  };
  vx_import import = vxImportObjectsFromMemory(context, 4, refs, uses, ptr, len);
  vx_graph graph = (vx_graph)refs[0];

  // Execute the OpenVX graph.
  vxProcessGraph(graph);

  // Reclaim the OpenCL buffers for some post-processing. The OpenVX workload to
write
  // into these buffers may still be pending with the global coordination command
queue
  vxSwapImageHandle(output1, NULL, &opencl_buf1, 1);
  vxSwapImageHandle(output2, NULL, &opencl_buf2, 1);

  // Buffers can be used directly with the global command queue if target used is
the same.
  // Below kernel1 will be scheduled on device_id1 using the global command queue.
  {
    clSetKernelArg(kernel1, 0, sizeof(cl_mem), (void *)&opencl_buf1);
    size_t localWorkSize[2] = { 16, 16 };
    size_t globalWorkSize[2] = { 640, 480 };
    clEnqueueNDRangeKernel(global_command_queue, kernel1, 2, NULL,
        globalWorkSize, localWorkSize, 0, NULL, NULL)
  }

  // To use buffer with another command queue that has a different target device
from
  // the global coordination command queue, a sync point needs to be used. The code
below uses
  // a marker and an event to sync between two command queues and make sure
  // that the second command queue waits for the global coordination command queue.
  // Then kernel2 is scheduled on device_id2 using a second command queue.
  {
    cl_event event;
    clEnqueueMarker(global_command_queue, &event);
    clSetKernelArg(kernel2, 0, sizeof(cl_mem), (void *)&opencl_buf2);
    size_t localWorkSize[2] = { 1, 1 };
    size_t globalWorkSize[2] = { 640, 480 };
    clEnqueueNDRangeKernel(second_command_queue, kernel2, 2, NULL,
        globalWorkSize, localWorkSize, 1, &event, NULL);
  }
  ...
```

2. Additional Functionality of Existing OpenVX APIs

This extension defines only one new API function (`vxCreateContextFromCL`), but modifies and extends several existing OpenVX APIs. This section summarizes the extension of the existing APIs.

2.1. Global OpenCL context

The `vxCreateContext()` function is extended to create a OpenCL context that can be queried using a new `VX_CONTEXT_CL_CONTEXT` attribute of `vx_context`. Alternatively, the new `vxCreateContextFromCL` function can be used to create an OpenVX context with a specific global OpenCL context.

2.2. Global Coordination Command Queue

The `vxCreateContext()` function is extended to create the in-order global *coordination command queue* that can be queried using a new `VX_CONTEXT_CL_COMMAND_QUEUE` attribute of `vx_context`. Alternatively, the new `vxCreateContextFromCL` function can be used to create an OpenVX context with a specific in-order global OpenCL command queue. The command queue must be in the specified global OpenCL context.

2.3. User kernels and nodes

2.3.1. User kernel using the *vx_khr_opencl_interop* extension

A user-defined kernel that makes use of the *vx_khr_opencl_interop* extension must be declared by setting the `VX_KERNEL_USE_OPENCL` kernel attribute to `vx_true_e` at kernel registration time before calling `vxFinalizeKernel`. This attribute is read-only after the `vxFinalizeKernel` call.

2.3.2. User node *coordination command queue*

Only user nodes created from user kernels declared as `VX_KERNEL_USE_OPENCL` can use the *vx_khr_opencl_interop* API. The *coordination command queue* is given by the OpenVX implementation and must be queried by the user node using the `VX_NODE_CL_COMMAND_QUEUE` node attribute.

The value of `VX_NODE_CL_COMMAND_QUEUE` must be set by the OpenVX implementation prior to calling the `vx_kernel_initialize_f` callback and it must not change until the `vx_kernel_deinitialize_f` callback is invoked.

2.4. OpenCL buffer memory type

A new `VX_MEMORY_TYPE_OPENCL_BUFFER` enum for `vx_memory_type_e` is accepted by the following APIs:

- all *vxCreateXxxxFromHandle* APIs, such as, `vxCreateImageFromHandle()`
- all *vxCopyXxxx* APIs, such as, `vxCopyImagePatch()`
- all *vxMapXxxx* APIs, such as, `vxMapImagePatch()`

The *vxCreateXxxxFromHandle* APIs and corresponding *vxSwapXxxx* APIs accept `cl_mem` buffers in `ptrs` when `VX_MEMORY_TYPE_OPENCL_BUFFER` is used

- The caller (i.e., the application) must make sure that the content of the `cl_mem` buffer is available to any command enqueued into the *coordination command queue* at the time the *vxCreateXxxxFromHandle* APIs are called
- The contents of reclaimed `cl_mem` buffers are available to any command enqueued into the *coordination command queue* after the *vxSwapXxxx* call returns.

The *vxCopyXxxx* APIs accept `cl_mem` buffers as the `user_ptr` when `VX_MEMORY_TYPE_OPENCL_BUFFER` is used.

- if *vxCopyXxxx* is called in read mode, the data copied into the user `cl_mem` buffer is available to any command enqueued into the *coordination command queue* after the vxCopyXxxx call returns.
- if *vxCopyXxxx* is called in write mode, the caller (the application) must make sure that the content of the user `cl_mem` buffer is available to any command enqueued into the *coordination command queue* at the time the *vxCopyXxxx* function is called

The *vxMapXxxx* APIs returns a `cl_mem` buffer in `ptr` when `VX_MEMORY_TYPE_OPENCL_BUFFER` is used. Subsequently, *vxUnmapXxxx* will accept a `cl_mem` buffer as `ptr` when `VX_MEMORY_TYPE_OPENCL_BUFFER` was used for the corresponding *vxMapXxxx* operation.

- if *vxMapXxxx* is requested in read-only mode or read/write mode, the content of the returned

`cl_mem` buffer is available to any command enqueued into the *coordination command queue* after the *vxMapXxxx* call returns.

- if *vxMapXxxx* is requested in write-only mode or read/write mode, the caller (the application) must make sure that the content of the returned `cl_mem` buffer is available to any command enqueued into the *coordination command queue* at the time the corresponding *vxUnmapXxxx* call is performed.

Any data object can be mapped or copied using `VX_MEMORY_TYPE_OPENCL_BUFFER` or `VX_MEMORY_TYPE_HOST`.

3. Constants

The following constants are declared in the header file *VX/vx_khr_opencl_interop.h* :

Table 1. Constants introduced by this extension

Name	Use
`VX_CONTEXT_CL_CONTEXT`	`vx_context` attribute to query the OpenCL context associated with the OpenVX context. Read-only.
`VX_MEMORY_TYPE_OPENCL_BUFFER`	The `vx_memory_type_e` enum to import from the OpenCL buffer.
`VX_CONTEXT_CL_COMMAND_QUEUE`	`vx_context` attribute to query the *coordination command queue* associated with the OpenVX context. Read-only.
`VX_NODE_CL_COMMAND_QUEUE`	`vx_node` attribute to query the `cl_command_queue` associated with a user kernel node. Read-only.
`VX_KERNEL_USE_OPENCL`	`vx_kernel` attribute to specify and query whether a user kernel is using the *vx_khr_opencl_interop* API. Return value is `vx_bool`. The default value of this attribute is `vx_false_e`. This attribute is read-only after the `vxFinalizeKernel` call.

4. The OpenCL Interop API functions

4.1. vxCreateContextFromCL

Create an OpenVX context with specified OpenCL context and global *coordination command queue*.

Declaration in VX/vx_khr_opencl_interop.h

```
vx_context  vxCreateContextFromCL(
        cl_context opencl_context,
        cl_command_queue opencl_command_queue
    );
```

Parameters

opencl_context

[in] The OpenCL context

opencl_command_queue

[in] The global *coordination command queue*

Return Value

- On success, a valid `vx_context` object. Calling `vxGetStatus` with the return value as a parameter will return VX_SUCCESS if the function was successful.

An implementation may provide several different error codes to give useful diagnostic information in the event of failure to create the context.

Description

This function creates a top-level object context for OpenVX and uses the OpenCL context and global *coordination command queue* created by the application for the interop.

This OpenCL context and global *coordination command queue* can be queried using the `VX_CONTEXT_CL_CONTEXT` and `VX_CONTEXT_CL_COMMAND_QUEUE` attributes of

`vx_context`.

If the OpenVX context is created using `vxCreateContext` or `vxCreateContextFromCL` with `opencl_context` as NULL, the OpenCL context used by OpenVX is implementation dependent. If the `opencl_command_queue` is NULL, the global *coordination command queue* used by OpenVX is implementation dependent.

The global *coordination command queue* must be created using the OpenCL context used by OpenVX.

Index