The OpenVX™ User Data Object Extension

The Khronos® OpenVX Working Group, Editor: Jesse Villarreal

Version 1.0 (provisional), Wed, 13 Feb 2019 16:07:15 +0000

Table of Contents

Chapter 1. Introduction

1.1. Purpose

This document details an extension to OpenVX 1.2.1, and references some APIs and symbols that may be found in that API, at https://www.khronos.org/registry/OpenVX/specs/1.2.1/OpenVX_Specification_1_2_1.html.

This extension is intended to define the new User Data Object support for OpenVX. This data object is intended to offer a standard mechanism to enable user kernels with a data object parameter which is not natively supported by existing data objects in the OpenVX spec.

1.2. Background

To set the stage for the describing the problem that user data objects solve, the following is a brief list of relevant points from the current specification:

Data Objects

- OpenVX contains several, specific, built-in data objects (`vx_image`, `vx_array`, etc), but no explicit means for user to extend to use their own custom data types as data objects used for node parameters.

User structs

- User structs were originally created to handle the use of additional fixed-size data structure types needed by `vx_array` for user kernels.
- The model for such structures were taken from existing data structures like `vx_keypoints_t` or `vx_coordinates_t`, which are simple, fixed-sized, data structures which could appear as a list contained in a `vx_array`.
- In OpenVX 1.2, the scope of user structs was expanded to `vx_scalar`, which was meant to handle a single instance of a data structure as opposed to an array of them.
- The function `vxRegisterUserStruct` simply informed the framework that it needed a `vx_enum` which would be associated with a fixed buffer size to house the user struct. This enum was needed as a backward-compatible way to create an array or scalar using this type, as the create functions require passing a type enum as an argument in order to properly allocate enough memory for each element in the array.
- The user kernel has no way to know which specific data structure this enum is assigned to since it is assigned at run-time. The most it can determine is the size of the structure. This can be indirectly used to determine which user structure was passed to the user kernel assuming all user structures are different sizes. However, if two or more different user structs have the same size, then the user structure can't be sure which one was passed.

1.2.1. Limitations in Existing Specification

If a user kernel wants to use a new, custom, data object that is not already included in OpenVX, there is currently no mechanism to extend OpenVX framework to do this. The closest mechanism available within the existing specification is "user structs" which can be used with `vx_array` or `vx_scalar`. However, there are at least 3 limitations of user structs that make it less than ideal for use as a custom user kernel node parameter data object:

Static Size

- The existing mechanism for user struct assumes a fixed size, and does not directly support a variable size. The data structure of a particular type of custom object may have a variable size, similar to an `vx_image` type. The image size changes depending on the width and height parameters.

- There are two possible workarounds for this:
  1. Call `vxRegisterUserStruct` for each instance of the data structure which has a different size.
     - This can result in a prohibitively high number of user structs which need to be registered and maintained, even though many of them are different sizes of the same type
  2. Call `vxRegisterUserStruct` using the worst-case memory requirement
     - This can result in a prohibitively large waste of memory, which is further limited in embedded applications.

Memory Access Performance

- If using `vx_scalar`, there is no Map/Unmap, only Copy
  - For large data objects, a copy may reduce performance each time the application needs to access data.
- If using `vx_array`, there is at least Map/Unmap, but the whole data object needs to be mapped/unmapped as there is no granularity of a range within a user struct.
  - If the application only needs to access a small part of the object, mapping the entire object may also reduce performance.

Misnomer

- Using `vx_scalar` to contain a custom object represented by a complex data structure is a misnomer and a bit confusing.

Chapter 2. Design Overview

The new APIs added in this extension for supporting user data objects follow the same pattern as existing Data Object APIs in OpenVX (i.e. create, release, query, copy, map, and unmap).

2.1. Example Code

This section demonstrates the usage of the new APIs in example code.

2.1.1. Create and Release

The following example shows how the User Data Object create and release APIs could be used to connect two custom user kernels together in a graph. The example also defines a data structure used within the user data object: user_custom_data_t.

```
/*
 * The custom data object structure for a user custom kernel which contains
 * a custom object of 2 images with embedded metadata (as an example).
 */
typedef struct {
    uint16_t  mode;            /*!< Indicates the contents of this buffer
                                    (so kernel knows how to parse) */
    uint16_t  source_data;     /*!< Indicates the source data corresponding
                                    to this data (for algorithm usage) */
    uint16_t  byte_size[2];    /*!< Number of bytes per pixel of each image
                                    in payload */
    uint16_t  width[2];        /*!< Widths of each image in payload */
    uint16_t  heights[2];      /*!< Heights of each image in payload */
    uint16_t  meta_size[2];    /*!< Number of bytes of meta information for
                                    each image in payload */
    uint8_t   payload[];       /*!< Payload of the custom data */
} user_custom_data_t;

/*
 * Utility API used to create the graph below using user data object in between two
 * user defined nodes:
 *
 * The following graph is created,
 * IMG1 -> vxCustomPreprocessCombineImagesNode -> TMP_USER_DATA_OBJECT
 * IMG2 ->                                          '-> vxCustomClassifier -> CLASS
 *
 */
static vx_graph create_graph(vx_context context,
                             vx_uint32 byte_size[],
                             vx_uint32 width[],
                             vx_uint32 height[],
                             vx_uint32 meta_size[])
{
    vx_graph graph;
    vx_node n0, n1;
    vx_image img1, img2;
    vx_user_data_object tmp_user_obj;
    vx_uint32 payload_size;
    vx_int32 class;
    vx_scalar s0;

    graph = vxCreateGraph(context);
```

```
    /* create input images */
    img1 = vxCreateImage(context, width[0], height[0], getDataFormat(byte_size[0]));
    img2 = vxCreateImage(context, width[1], height[1], getDataFormat(byte_size[1]));

    /* calculate payload size of custom object */
    payload_size = calculate_payload(byte_size, width, height, meta_size);

    /* create intermediate custom object */
    tmp_user_obj = vxCreateUserDataObject(context,
                                    "user_custom_data_t",
                                    sizeof(user_custom_data_t) + payload_size,
                                    NULL);

    /* create first node: input is 2 images and source string, output is combined
     * images data object with mode, source information, and meta information added
     * in custom format for downstream processing on second node */
    n0 = vxCustomPreprocessCombineImagesNode(graph, img1, img2, "survellance",
                                            tmp_user_obj);

    /* create a scalar object required for second node */
    class = 0;
    s0 = vxCreateScalar(context, VX_TYPE_INT32, &class);

    /* create second node: input is output of first node, output is classification
     * scalar */
    n1 = vxCustomClassifier(graph, tmp_user_obj, s0);

    vxReleaseScalar(&s0);
    vxReleaseNode(&n0);
    vxReleaseNode(&n1);
    vxReleaseImage(&img1);
    vxReleaseImage(&img2);
    vxReleaseUserDataObject(&tmp_user_obj);

    return graph;
}
```

2.1.2. Query and Map/Unmap or Copy

The following code shows an example of how the query API and map/unmap or copy functions could be used to read a field from the user data object.

```
/*
 * Utility API used to read the source data from vx_user_data_object
 * of type user_custom_data_t
 * - Uses map API
 * - Uses query API
 */
static vx_status get_source_data_using_map(vx_user_data_object usr_obj,
                                           uint16_t *source_data)
{
    vx_status status = VX_FAILURE;
    uint16_t *pSourceData = NULL;
    vx_map_id map_id;

    vx_char obj_name[VX_MAX_REFERENCE_NAME];

    vxQueryUserDataObject(usr_obj, VX_USER_DATA_OBJECT_NAME, &obj_name,
                          sizeof(obj_name));
```

```
    if(strncmp(obj_name, "user_custom_data_t", VX_MAX_REFERENCE_NAME) == 0)
    {
        vx_size offset = offsetof(user_custom_data_t, source_data);
        vx_size size   = sizeof(((user_custom_data_t *)0)->source_data;

        status = vxMapUserDataObject(usr_obj, offset, size, &map_id,
                                     (void **)&pSourceData, VX_READ_ONLY,
                                     VX_MEMORY_TYPE_HOST, 0);
        if( VX_SUCCESS == status )
        {
            *source_data = *pSourceData;
            status = vxUnmapUserDataObject(map_id);
        }
    }

    return status;
}

/*
 * Utility API used to read the source data from vx_user_data_object of
 * type user_custom_data_t
 * - Uses copy API
 * - Uses query API
 */
static vx_status get_source_data_using_copy(vx_user_data_object usr_obj,
                                            uint16_t *source_data)
{
    vx_status status = VX_FAILURE;
    uint16_t *pSourceData = NULL;
    vx_map_id map_id;

    vx_char obj_name[VX_MAX_REFERENCE_NAME];

    vxQueryUserDataObject(usr_obj, VX_USER_DATA_OBJECT_NAME, &obj_name,
                          sizeof(obj_name));

    if(strncmp(obj_name, "user_custom_data_t", VX_MAX_REFERENCE_NAME) == 0)
    {
        vx_size offset = offsetof(user_custom_data_t, source_data);
        vx_size size   = sizeof(((user_custom_data_t *)0)->source_data;

        status = vxCopyUserDataObject(usr_obj, offset, size, source_data,
                                  VX_READ_ONLY, VX_MEMORY_TYPE_HOST);
    }

    return status;
}
```

2.2. Changes to the OpenVX Specification

VX_TYPE_USER_DATA_OBJECT should be added to the list of supported delay objects and supported object arrays.

Chapter 3. Module Documentation

Macros

- VX_TYPE_USER_DATA_OBJECT

Typedefs

- vx_user_data_object

Enumerations

- vx_user_data_object_attribute_e

Functions

- vxCreateUserDataObject
- vxCreateVirtualUserDataObject
- vxReleaseUserDataObject
- vxQueryUserDataObject
- vxCopyUserDataObject
- vxMapUserDataObject
- vxUnmapUserDataObject

3.1. Macros

3.1.1. VX_TYPE_USER_DATA_OBJECT

The object type enumeration for user data object.

```
#define VX_TYPE_USER_DATA_OBJECT 0x816
```

3.2. Typedefs

3.2.1. vx_user_data_object

The User Data Object. User Data Object is a strongly-typed container for other data structures

```
typedef struct _vx_user_data_object *vx_user_data_object;
```

3.3. Enumerations

3.3.1. vx_user_data_object_attribute_e

The user data object attributes.

```
enum vx_user_data_object_attribute_e {
    VX_USER_DATA_OBJECT_NAME = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS, VX_TYPE_USER_DATA_OBJECT) + 0x0,
    VX_USER_DATA_OBJECT_SIZE = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS, VX_TYPE_USER_DATA_OBJECT) + 0x1,
};
```

Enumerator

- VX_USER_DATA_OBJECT_NAME - The type name of the user data object. Read-only. Use a vx_char [VX_MAX_REFERENCE_NAME] array.
- VX_USER_DATA_OBJECT_SIZE - The number of bytes in the user data object. Read-only. Use a vx_size parameter.

3.4. Functions

3.4.1. vxCreateUserDataObject

Creates a reference to a User Data Object.

```
vx_user_data_object vxCreateUserDataObject(
    vx_context                              context,
    const vx_char*                          type_name,
    vx_size                                 size,
    const void*                             ptr);
```

User data objects can be used to pass a user kernel defined data structure or blob of memory as a parameter to a user kernel.

Parameters

- [in] *context* - The reference to the overall Context.
- [in] *type_name* - Pointer to the '\0' terminated string that identifies the type of object. The string is copied by the function so that it stays the property of the caller. The length of the string shall be lower than VX_MAX_REFERENCE_NAME bytes. The string passed here is what shall be returned when passing the VX_USER_DATA_OBJECT_NAME attribute enum to the vxQueryUserDataObject function. In the case where NULL is passed to type_name, then the query of the VX_USER_DATA_OBJECT_NAME attribute enum will return a single character '\0' string.
- [in] *size* - The number of bytes required to store this instance of the user data object.
- [in] *ptr* - The pointer to the initial value of the user data object. If NULL, then entire size bytes of the user data object is initialized to all 0s, otherwise, *size* bytes is copied into the object from ptr to initialize the object

Returns: A user data object reference vx_user_data_object. Any possible errors preventing a successful creation should be checked using vxGetStatus

3.4.2. vxCreateVirtualUserDataObject

Creates an opaque reference to a virtual User Data Object with no direct user access.

```
vx_user_data_object vxCreateVirtualUserDataObject(
    vx_graph                                graph,
    const vx_char*                          type_name,
    vx_size                                 size);
```

Virtual User Data Objects are useful when the User Data Object is used as internal graph edge. Virtual User Data Objects are scoped within the parent graph only.

Parameters

- [in] *graph* - The reference to the parent graph.
- [in] *type_name* - Pointer to the '\0' terminated string that identifies the type of object. The string is copied by the function so that it stays the property of the caller. The length of the string shall be lower than VX_MAX_REFERENCE_NAME bytes. The string passed here is what shall be returned when passing the VX_USER_DATA_OBJECT_NAME attribute enum to the vxQueryUserDataObject function. In the case where NULL is passed to type_name, then the query of the VX_USER_DATA_OBJECT_NAME attribute enum will return a single character '\0' string.
- [in] *size* - The number of bytes required to store this instance of the user data object.

Returns: A user data object reference vx_user_data_object. Any possible errors preventing a successful creation should be checked using vxGetStatus

3.4.3. vxReleaseUserDataObject

Releases a reference of a User data object. The object may not be garbage collected until its total reference count is zero. After returning from this function the reference is zeroed.

```
vx_status vxReleaseUserDataObject(
    vx_user_data_object*                    user_data_object);
```

Parameters

- [in] *user_data_object* - The pointer to the User Data Object to release.

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors; any other value indicates failure.
- VX_ERROR_INVALID_REFERENCE - *user_data_object* is not a valid vx_user_data_object reference.

3.4.4. vxQueryUserDataObject

Queries the User data object for some specific information.

```
vx_status vxQueryUserDataObject(
    vx_user_data_object                     user_data_object,
    vx_enum                                 attribute,
    void*                                   ptr,
    vx_size                                 size);
```

Parameters

- [in] *user_data_object* - The reference to the User data object.
- [in] *attribute* - The attribute to query. Use a vx_user_data_object_attribute_e.
- [out] *ptr* - The location at which to store the resulting value.
- [in] *size* - The size in bytes of the container to which *ptr* points.

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors; any other value indicates failure.
- VX_ERROR_INVALID_REFERENCE - *user_data_object* is not a valid vx_user_data_object reference.
- VX_ERROR_NOT_SUPPORTED - If the *attribute* is not a value supported on this implementation.
- VX_ERROR_INVALID_PARAMETERS - If any of the other parameters are incorrect.

3.4.5. vxCopyUserDataObject

Allows the application to copy a subset from/into a user data object.

```
vx_status vxCopyUserDataObject(
    vx_user_data_object                     user_data_object,
    vx_size                                 offset,
    vx_size                                 size,
    void*                                   user_ptr,
    vx_enum                                 usage,
    vx_enum                                 user_mem_type);
```

Parameters

- [in] *user_data_object* - The reference to the user data object that is the source or the destination of the copy.
- [in] *offset* - The byte offset into the user data object to copy.
- [in] *size* - The number of bytes to copy. The size must be within the bounds of the user data object: 0 ≤ (*offset* + *size*) ≤ *size* of the user data object. If zero, then copy until the end of the object.
- [in] *user_ptr* - The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the user data object if the copy was requested in write mode. The accessible memory must be large enough to contain the specified size.
- [in] *usage* - This declares the effect of the copy with regard to the user data object using the vx_accessor_e enumeration. Only VX_READ_ONLY and VX_WRITE_ONLY are supported:
  - VX_READ_ONLY means that data are copied from the user data object into the user memory.
  - VX_WRITE_ONLY means that data are copied into the user data object from the user memory.
- [in] *user_mem_type* - A vx_memory_type_e enumeration that specifies the memory type of the memory referenced by the *user_addr*.

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors; any other value indicates failure.
- VX_ERROR_OPTIMIZED_AWAY - This is a reference to a virtual user data object that cannot be accessed by the application.
- VX_ERROR_INVALID_REFERENCE - *user_data_object* is not a valid vx_user_data_object reference.
- VX_ERROR_INVALID_PARAMETERS - An other parameter is incorrect.

3.4.6. vxMapUserDataObject

Allows the application to get direct access to a subset of the user data object.

```
vx_status vxMapUserDataObject(
    vx_user_data_object                     user_data_object,
    vx_size                                 offset,
    vx_size                                 size,
    vx_map_id*                              map_id,
    void**                                  ptr,
    vx_enum                                 usage,
    vx_enum                                 mem_type,
    vx_uint32                               flags);
```

Parameters

- [in] *user_data_object* - The reference to the user data object that contains the subset to map.
- [in] *offset* - The byte offset into the user data object to map.
- [in] *size* - The number of bytes to map. The size must be within the bounds of the user data object: 0 ≤ (*offset* + *size*) ≤ *size* of the user data object. If zero, then map until the end of the object.
- [out] *map_id* - The address of a vx_map_id variable where the function returns a map identifier.
  - (*map_id) must eventually be provided as the map_id parameter of a call to vxUnmapUserDataObject.
- [out] *ptr* - The address of a pointer that the function sets to the address where the requested data can be accessed. The returned (**ptr*) address is only valid between the call to the function and the corresponding call to vxUnmapUserDataObject.
- [in] *usage* - This declares the access mode for the user data object subset, using the vx_accessor_e enumeration.
  - VX_READ_ONLY: after the function call, the content of the memory location pointed by (**ptr*) contains the user data object subset data. Writing into this memory location is forbidden and its behavior is implementation specific.
  - VX_READ_AND_WRITE: after the function call, the content of the memory location pointed by (**ptr*) contains the user data object subset data; writing into this memory is allowed only for the location of items and will result in a modification of the affected items in the user data object once the range is unmapped.
  - VX_WRITE_ONLY: after the function call, the memory location pointed by (**ptr*) contains implementation specific data; writing to all data in the subset is required prior to unmapping. Data values not written by the application before unmap may be defined differently in different implementations after unmap, even if they were well defined before map.
- [in] *mem_type* - A vx_memory_type_e enumeration that specifies the type of the memory where the user data object subset is requested to be mapped.
- [in] *flags* - An integer that allows passing options to the map operation. Use the vx_map_flag_e enumeration.

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.
- `VX_ERROR_OPTIMIZED_AWAY` - This is a reference to a virtual user data object that cannot be accessed by the application.
- `VX_ERROR_INVALID_REFERENCE` - *user_data_object* is not a valid `vx_user_data_object` reference.
- `VX_ERROR_INVALID_PARAMETERS` - An other parameter is incorrect.

Postcondition: `vxUnmapUserDataObject` with same (*map_id) value.

3.4.7. vxUnmapUserDataObject

Unmap and commit potential changes to a user data object subset that was previously mapped. Unmapping a user data object subset invalidates the memory location from which the subset could be accessed by the application. Accessing this memory location after the unmap function completes is implementation specific.

```
vx_status vxUnmapUserDataObject(
    vx_user_data_object                     user_data_object,
    vx_map_id                               map_id);
```

Parameters

- `[in]` *user_data_object* - The reference to the user data object to unmap.
- `[out]` *map_id* - The unique map identifier that was returned when calling `vxMapUserDataObject`.

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.
- `VX_ERROR_INVALID_REFERENCE` - The *user_data_object* reference is not actually a `vx_user_data_object` reference.
- `VX_ERROR_INVALID_PARAMETERS` - An other parameter is incorrect.

Precondition: `vxMapUserDataObject` returning the same map_id value

Index