The OpenVX™ Buffer Aliasing Extension

The Khronos® OpenVX Working Group, Editor: Jesse Villarreal

Version 1.0 (provisional), Wed, 13 Feb 2019 16:07:12 +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 add more robust support for in-place processing optimizations in OpenVX through the use of buffer aliasing. The intended audience for this extension is OpenVX framework vendors, and user kernel writers. The APIs added from this extension have no impact or bearing on the application software (graph creators).

1.2. Background

Before discussing the design for supporting in-place processing in OpenVX, this section defines some terms and introduces relevant concepts.

1.2.1. In-Place Processing

Typically, a conventional processing operation reads data from one or more input memory locations, processes it, and then writes the output to a different memory location. In this situation, the contents of the input memory is not changed by the processing operation since it is read and not written to.

In-place processing, however, is a computer science optimization technique wherein the output of the processing operation overwrites some or all of the input data. In this case the input memory may be changed by the processing operation since it is both read and written to. This approach has some tradeoffs:

PROS:

- Uses less memory than conventional processing approach (since there is no need for a separate output buffer)
- May have faster memory access speeds due to cache benefits of writing to cache lines that are already in the cache due to a recent read.

CONS:

- Input is overwritten, so can not be available to other processing operations that may have needed it.
- Some software pipelined compilers may have optimization penalties due to loop-carried dependencies depending on the order and access pattern of reads and writes of the aliased buffers.

These tradeoffs should be considered when making a decision on whether it is appropriate to use in-place processing. A conservative policy could be to never use in-place processing since it may always be functionally correct, but it also may leave some performance on the table. When performance is a high priority, in-place processing could be a useful technique if it is used carefully and done in a way that guarantees functional correctness.

1.2.2. Use Cases

Within the scope of this extension, there are broadly two categories of use cases to consider: dense processing, and sparse processing.

Dense Processing

An example of a dense processing operation could be a pixel-wise NOT operation. Every pixel in the input image is fetched, processed with a NOT operation, and then written to the output. In the context of in-place processing, the entire input image will be overwritten by the NOT of each input pixel. Obviously, there is a memory usage benefit from using in-place processing. However, from a performance improvement perspective, the NOT operation itself does not benefit from in-place processing other than perhaps a cache access benefit.

Sparse Processing

An example of a sparse processing operation could be a function which conditionally overwrites only parts of the input based on contents of the input. In this case, only a sparse subset of data in the input is overwritten (updated) by the operation. If in-place processing were prohibited, then the function would first need to copy the entire input buffer to the output before starting the conditional overwrite of this output buffer. Not using in-place processing imposes a potentially severe performance penalty of an entire buffer copy. Or, put the other way, in-place processing brings about a significant performance improvement by saving a buffer copy operation.

Comparison

Both dense and sparse processing can benefit from the memory savings and the cache performance savings of in-place processing. However, sparse processing has a significantly higher performance optimization benefit from the use of in-place processing compared to dense processing.

1.2.3. Limitations in Existing Specification

The current OpenVX spec has limited support for bidirectional parameters. Currently, only the built-in accumulation kernels use bidirectional parameters. User kernels are forbidden from using bidirectional parameters.

Bidirectional parameters are problematic when the data object connected to a bidirectional node parameter is attached to more than one node. The graph scheduler can not ascertain the proper order of operations.

Chapter 2. Design Overview

Due to the problematic nature of bidirectional parameters in a graph programming model, this extension takes a different approach: buffer aliasing.

2.1. Buffer Aliasing

Buffer aliasing in the context of OpenVX refers to having multiple data objects internally access the same data buffer in memory. This aliasing of data buffers is abstracted from the application by the framework. As before, the application still creates separate data objects to connect to nodes of the graph, but during graph verification the graph scheduler can make decisions about aliasing some of the data objects together based on additional information provided from the kernel implementations using the new API introduced in this extension. In this way, the application program doesn't need to consider anything from this extension.

2.2. Control Flow

The control flow is straight forward. This API adds two new functions to be optionally called from user kernel implementations:

`vxAliasParameterIndexHint`

- Notifies framework that the kernel supports buffer aliasing of specified parameters
- Is called from within the `vx_publish_kernels_f` callback, for applicable kernels in between the call to the `vxAddUserKernel` function and the `vxFinalizeKernel` function for the corresponding kernel.

`vxIsParameterAliased`

- Query framework if the specified parameters are aliased
- Is called from the `vx_kernel_initialize_f` or `vx_kernel_f` callback function

The idea is that the kernels can hint about its buffer aliasing capabilities to the framework when they are being loaded into the context. Then when an application includes the kernel into a graph, the framework's graph scheduler/optimizer can evaluate if the buffer can be aliased legally at graph verification time. Then, when the framework calls the `vx_kernel_initialize_f` or `vx_kernel_f` callback of the kernel, the kernel can call `vxIsParameterAliased` API to understand what decision the framework made, and process the kernel accordingly.

There may be many situations that arise where buffers can not be legally aliased due to conflicts between multiple consumers of a buffer which need the original data to not be overwritten. The framework should have the global knowledge of the graph to be able to make a decision on whether or not some of the alias hints can be honored or not. In fact, a framework can be minimally compliant with this extension if it simply implements a stub for the `vxAliasParameterIndexHint` API, and returns `vx_false_e` for the `vxIsParameterAliased` API. That would be consistent with the conservative policy of never allowing buffer aliasing, while being compatible with user kernels who utilize this extension.

Chapter 3. Module Documentation

Enumerations

- vx_buffer_aliasing_enum_e
- vx_buffer_aliasing_processing_type_e

Functions

- vxAliasParameterIndexHint
- vxIsParameterAliased

3.1. Enumerations

3.1.1. vx_buffer_aliasing_enum_e

Extra enums.

```
enum vx_buffer_aliasing_enum_e {
    VX_ENUM_BUFFER_ALIASING_TYPE = 0x1F,
};
```

Enumerator

- VX_ENUM_BUFFER_ALIASING_TYPE - Buffer aliasing type enumeration.

3.1.2. vx_buffer_aliasing_processing_type_e

Type of processing the kernel will perform on the buffer

```
enum vx_buffer_aliasing_processing_type_e {
    VX_BUFFER_ALIASING_PROCESSING_TYPE_DENSE = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_BUFFER_ALIASING_TYPE
<< 12)) + 0x0,
    VX_BUFFER_ALIASING_PROCESSING_TYPE_SPARSE = ((( VX_ID_KHRONOS ) << 20) | (
VX_ENUM_BUFFER_ALIASING_TYPE << 12)) + 0x1,
};
```

See also: vxAliasParameterIndexHint

Indicates what type of processing the kernel will perform on the buffer. The framework may use this information to arbitrate between requests. For example, if there are two or three conflicting requests for buffer aliasing, then the framework may choose to prioritize a request which gives a performance improvement as compared with one that only saves memory but otherwise doesn't give a performance improvement. For example, a kernel which performs sparse processing may need to first do a buffer copy before processing if the buffers are not aliased. However a kernel which performs dense processing will not need to do this. So priority of the alias request may be given to the kernel which performs sparse processing.

Enumerator

- VX_BUFFER_ALIASING_PROCESSING_TYPE_DENSE - Dense processing on the buffer that can be aliased

- VX_BUFFER_ALIASING_PROCESSING_TYPE_SPARSE - Sparse processing on the buffer that can be aliased

3.2. Functions

3.2.1. vxAliasParameterIndexHint

Notifies framework that the kernel supports buffer aliasing of specified parameters

```
vx_status vxAliasParameterIndexHint(
    vx_kernel                                   kernel,
    vx_uint32                                   parameter_index_a,
    vx_uint32                                   parameter_index_b,
    vx_enum                                     processing_type);
```

This is intended to be called from within the vx_publish_kernels_f callback, for applicable kernels in between the call to the vxAddUserKernel function and the vxFinalizeKernel function for the corresponding kernel.

If a kernel can not support buffer aliasing of its parameters (for in-place processing), then it should not call this function. However, if a kernel can support buffer aliasing of a pair of its parameters, then it may call this function with the appropriate parameter indices and priority value.

Note that calling this function does not guarantee that the buffers will ultimately be aliased by the framework. The framework may consider this hint as part of performance or memory optimization logic along with other factors such as graph topology, other competing hints, and if the parameters are virtual objects or not.

Parameters

- [in] *kernel* - Kernel reference
- [in] *parameter_index_a* - Index of a kernel parameter to request for aliasing
- [in] *parameter_index_b* - Index of another kernel parameter to request to alias with parameter_index_a
- [in] *processing_type* - Indicate the type of processing on this buffer from the kernel. (See vx_buffer_aliasing_processing_type_e)

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors. failure.
- VX_ERROR_INVALID_REFERENCE - kernel is not a valid vx_kernel reference
- VX_ERROR_INVALID_PARAMETERS - *parameter_index_a* or *parameter_index_b* is NOT a valid kernel parameter index
- VX_FAILURE - *processing_type* is not a supported enumeration value.

3.2.2. vxIsParameterAliased

Query framework if the specified parameters are aliased

```
vx_bool vxIsParameterAliased(
    vx_node                                 node,
    vx_uint32                               parameter_index_a,
    vx_uint32                               parameter_index_b);
```

This is intended to be called from the `vx_kernel_initialize_f` or `vx_kernel_f` callback functions.

If a kernel has called the `vxAliasParameterIndexHint` function during the vx_publish_kernels_f callback, then `vxIsParameterAliased` is the function that can be called in the init or processing function to query if the framework was able to alias the buffers specified. Based on this information, the kernel may execute the kernel differently.

Parameters

- `[in]` *node* - Node reference
- `[in]` *parameter_index_a* - Index of a kernel parameter to query for aliasing
- `[in]` *parameter_index_b* - Index of another kernel parameter to query to alias with parameter_index_a

Returns: A `vx_bool` value.

Return Values

- `vx_true_e` - The parameters are aliased.
- `vx_false_e` - The parameters are not aliased.

Index