The OpenVX™ Graph Pipelining, Streaming, and Batch Processing Extension to OpenVX 1.1 and 1.2

The Khronos OpenVX Working Group, Editors: Kedar Chitnis, Jesse Villareal, Radhakrishna Giduthuri, and Frank Brill

Version 1.0.1 (provisional), Mon, 06 Aug 2018 02:31:04 +0000

Table of Contents

Chapter 1. Introduction

1.1. Purpose

Enable multiple initiations of a given graph with different inputs and outputs. Additionally, this extension provides a mechanism for the application to execute a graph such that the application does not need to be involved with data reconfiguration and starting processing of the graph for each set of input/output data.

1.2. Acknowledgements

This specification would not be possible without the contributions from this partial list of the following individuals from the Khronos Working Group and the companies that they represented at the time:

- Kedar Chitnis - Texas Instruments, Inc.
- Jesse Villareal - Texas Instruments, Inc.
- Radhakrishna Giduthuri - AMD
- Tomer Schwartz - Intel
- Frank Brill - Cadence Design Systems
- Thierry Lepley - Cadence Design Systems

1.3. Background and Terminology

This section introduces the concepts of graph pipelining, streaming and batch processing before getting into the details of how OpenVX is extended to support these features.

1.3.1. Graph Pipelining

In order to demonstrate what is meant by pipelined execution, please refer to the following example system which executes the simple graph in a distributed manner:

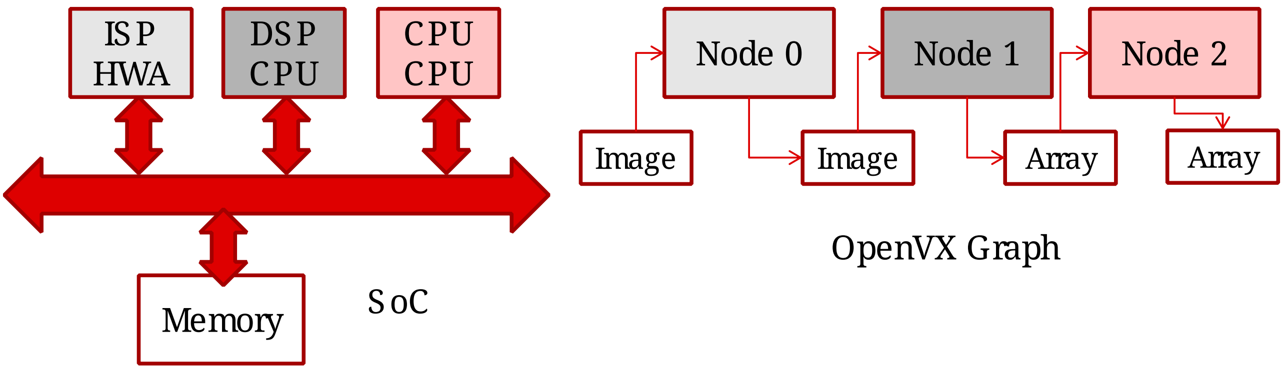

Figure 1. Example SoC and Distributed Graph

In this example, there are three compute units: an Image Signal Processor (ISP) HWA, a Digital

Signal Processor (DSP), and a CPU. The example graph likewise, has three nodes: generically labelled Node 0, Node 1, and Node 2. There could be more or less nodes than compute units, but here, the number of nodes happens to be equal to the number of compute units. In this graph, Node 0 is executed on the ISP, Node 1 is executed on the DSP, and Node 2 is executed on the CPU. Without pipelining enabled, the execution timeline of this graph is shown below:

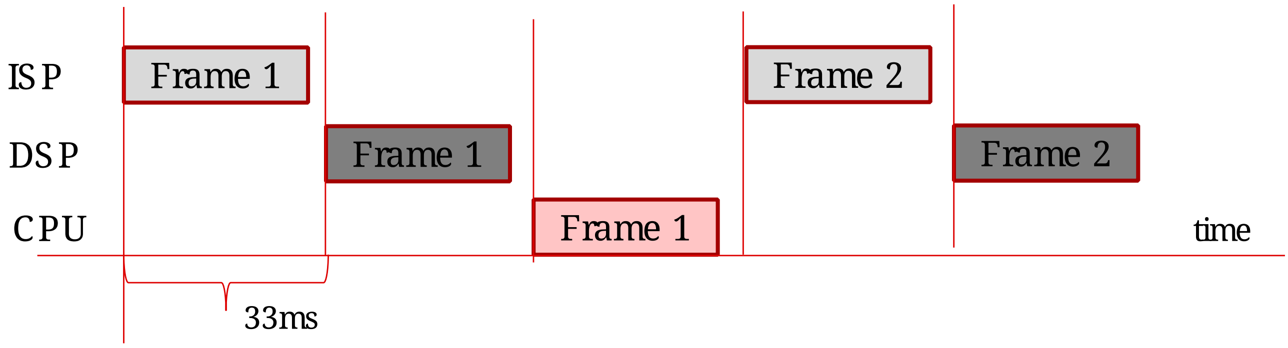

Figure 2. Non-pipelined Execution

Assuming each node takes 33ms to execute, then the full graph takes 99ms to execute. Without this extension, OpenVX requires that a second frame can not start graph execution on this same graph until the first graph execution is completed. This means that the maximum throughput of this example will be one frame completing every 99ms. However, in this example, you can see that each compute unit is only utilized no more than one-third of the time. Furthermore, if the camera input produced a frame every 33ms, then every two out of three frames would need to be "dropped" by the system since this OpenVX graph implementation can not keep up with the input frame rate of the camera.

Pipelining the graph exection will both increase the hardware utilization, and increase the throughput of the OpenVX implementation. These effects can be seen in the timeline of a pipelined execution of the graph below:

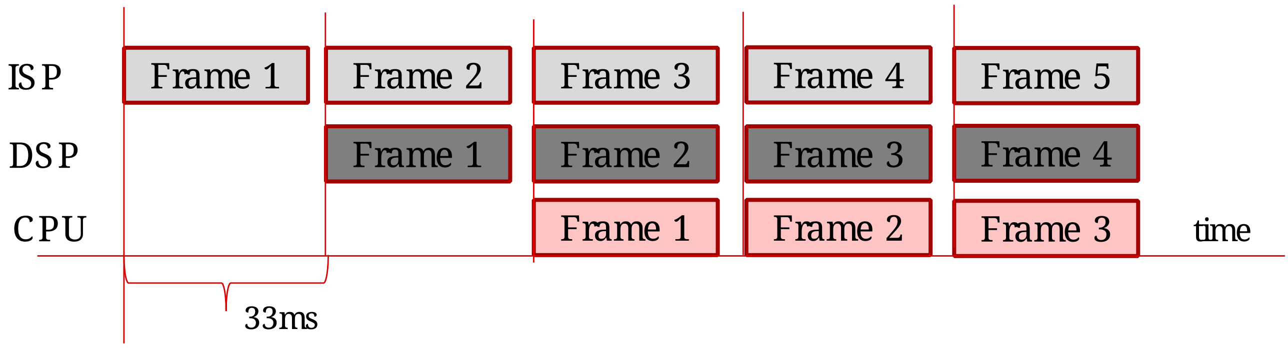

Figure 3. Frame-Level Pipelined Execution

Here, the latency of the graph is still 99ms, but the throughput has been increased to one frame completing every 33ms, allowing the graph to run in real-time with the camera frame-rate.

Now, in this simple example, a lot of assumptions were made in order to illustrate the concept. We assumed that each node took the same amount of time, so pipelining looked like we went from 33% core utilization to 100% core utilization. In practice, this ideal is almost never true. Processing times will vary across both kernels and cores. So although pipelining may bring about increased utilization and throughput, the actual frame rate will be determined by the execution time of the pipeline stage with the longest execution time.

In order to enable pipelining, the implementation must provide a way for the application to update the input and output data for future executions of the graph while previously scheduled graphs are still in the executing state. Likewise, the implementation must allow scheduling and starting of graph executions while previously scheduled graphs are still in the executing state. The Pipelining and Batch Processing section introduces new APIs and gives code examples for how this extension enables this basic pipelining support. The Event handling section extends the controllability and timing of WHEN to exchange frames and schedule new frames using events.

1.3.2. Graph Batch Processing

Batch processing refers to the ability to execute a graph on a group or batch of input and output references. Here the user provides a list of input and output references and a single graph schedule call processes the data without further intervention of the user application. When a batch of input and output references is provided to the implementation, it allows the implementation to potentially parallelize the execution of the graphs on each input/output reference such that overall higher throughput and performance is achieved as compared to sequentially executing the graph for each input/output reference.

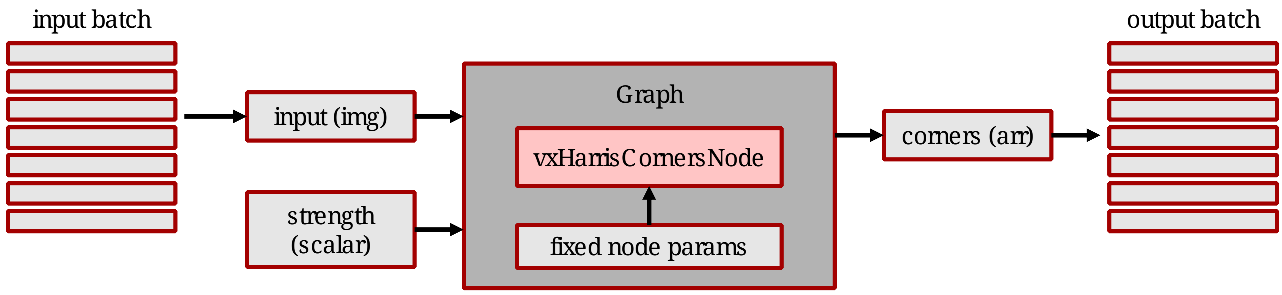

Figure 4. Graph Batch Processing

The Pipelining and Batch Processing section introduces new APIs and gives code examples for how this extension enables batch processing support.

1.3.3. Graph Streaming

Graph streaming refers to the ability of the OpenVX implementation to automatically handle graph input and output updates and re-schedule each frame without intervention from the application. The concept of graph streaming is orthogonal to graph pipelining. Pipelining can be enabled or disabled on a graph which has streaming enabled or disabled, and vice-versa.

In order to enable graph streaming, the implementation must provide a way for the application to enter and exit this streaming mode. Additionally, the implementation must somehow manage the input and output swapping with upstream and downstream components outside of the OpenVX implementation. This can be handled with the concept of SOURCE nodes and SINK nodes.

A SOURCE node is a node which coordinates the supply of input into the graph from upstream components (such as a camera), and the SINK node is a node which coordinates the handoff of output from the graph into downstream components (such as a display).

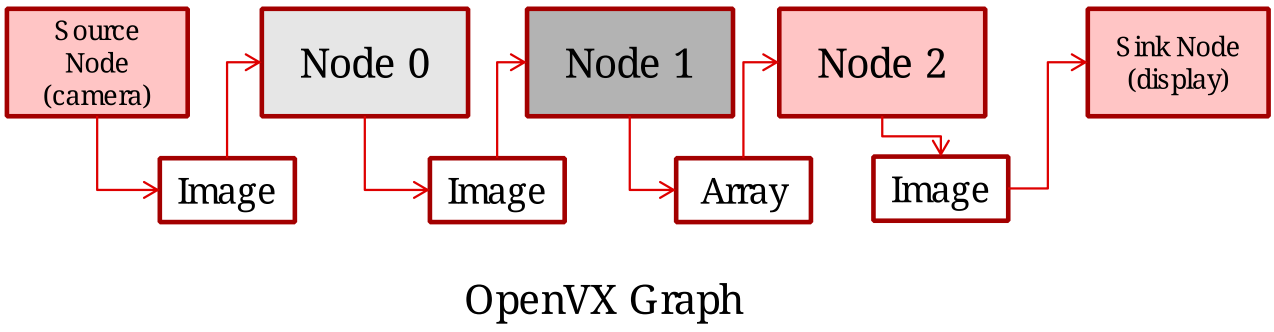

Figure 5. Source/Sink Nodes added for Graph Streaming

The Streaming section introduces new APIs and gives code examples for how this extension enables this basic streaming support.

Chapter 2. Design Overview

2.1. Data reference

In this extension, the term *data reference* is used frequently. In this section we define this term.

Data references are OpenVX references to any of the OpenVX data types listed below,

- `VX_TYPE_LUT`
- `VX_TYPE_DISTRIBUTION`
- `VX_TYPE_PYRAMID`
- `VX_TYPE_THRESHOLD`
- `VX_TYPE_MATRIX`
- `VX_TYPE_CONVOLUTION`
- `VX_TYPE_SCALAR`
- `VX_TYPE_ARRAY`
- `VX_TYPE_IMAGE`
- `VX_TYPE_REMAP`
- `VX_TYPE_OBJECT_ARRAY`
- `VX_TYPE_TENSOR` (OpenVX 1.2 and above)

The APIs which operate on data references take as input a `vx_reference` type. An application can pass any of the above defined data type references to such an API.

2.2. Pipelining and Batch Processing

Pipelining and Batch Processing APIs allow an application to construct a graph which can be executed in a pipelined fashion (see Graph Pipelining), or batch processing fashion (see Graph Batch Processing).

2.2.1. Graph Parameter Queues

The concept of OpenVX "Graph Parameters" is defined in the main OpenVX spec as a means to expose external ports of a graph. Graph parameters enable the abstraction of the remaining graph ports which are not connected as graph parameters. Since graph pipelining and batching is concerned primarily with controlling the flow of data to and from the graph, OpenVX graph parameters provide a useful construct for enabling pipelining and batching.

This extension introduces the concept of *graph parameter queueing* to enable assigning multiple data objects to a graph parameter (either at once, or spaced in time) without needing to wait for the previous graph completion(s). At runtime, the application can utilize the `vxGraphParameterEnqueueReadyRef` function to enqueue a number of data references into a graph parameter to be used by the graph. Likewise, the application can use the `vxGraphParameterDequeueDoneRef` function to dequeue a number of data references from a graph parameter after the graph is done using them (thus, making them available for the application). The `vxGraphParameterCheckDoneRef` function is a non-blocking call that can be used to determine if there are references available for dequeuing, and if so, how many.

In order for the implementation to know which graph parameters it needs to support queuing on, the application should configure this by calling `vxSetGraphScheduleConfig` before calling `vxVerifyGraph` or `vxScheduleGraph`.

2.2.2. Graph Schedule Configuration

The graph schedule configuration function (`vxSetGraphScheduleConfig`) allows users to enable enqueuing of multiple input and output references to a graph parameter. It also allows users to control how the graph gets scheduled based on the references enqueued by the user.

The *graph_schedule_mode* parameter defines two modes of graph scheduling:

1. `VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL`
   - Here the application enqueues the references to be processed at a graph parameter
   - Later when application calls `vxScheduleGraph`, all the previously enqueued references get processed.
   - Enqueuing multiple references and calling a single `vxScheduleGraph` allows implementation flexibility to optimize the execution of the multiple graph executions based on the number of the enqueued references.
2. `VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO`
   - Here also, the user enqueues the references that they want to process at a graph parameter
   - However here user does not explicitly call `vxScheduleGraph`
   - `vxVerifyGraph` *must* be called in this mode (since `vxScheduleGraph` is not called).
   - The implementation automatically triggers graph execution when it has enough enqueued references to start a graph execution
   - Enqueuing multiple references without calling `vxScheduleGraph` allows the implementation to start a graph execution as soon as minimal input or output references are available.

In both of these modes, `vxProcessGraph` is not allowed. The next two sections show how the graph schedule configuration, along with reference enqueue and dequeue is used to realize the graph pipelining and batch processing use-cases.

2.2.3. Example Graph pipelining application

Graph pipelining allow users to schedule a graph multiple times, without having to wait for a graph execution to complete. Each such execution of the graph operates on different input or output references.

In a typical pipeline execution model, there is a pipe-up phase where new inputs are enqueued and graph is scheduled multiple times until the pipeline is full. Once the pipeline is full, then outputs begin to be filled as often as inputs are enqueued (as shown in Frame-Level Pipelined Execution).

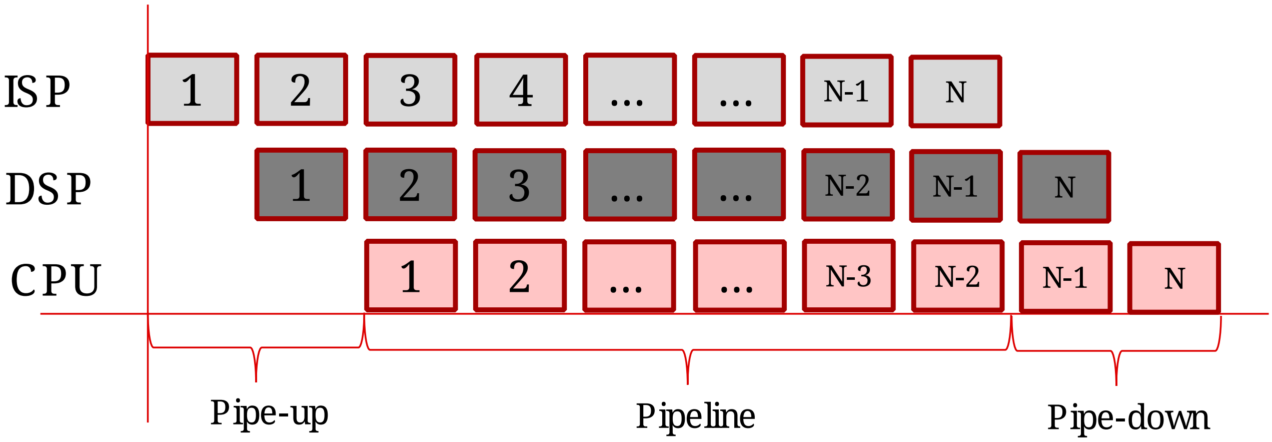

Figure 6. 3 Phases of pipeline: pipe-up, pipeline, and pipe-down

In order for the graph to be executed in a pipelined fashion, the steps outlined below need to be followed by an application:

1. Create a graph and add nodes to the graph as usual.
2. For data references which need to be enqueued and dequeued by the application, add them as graph parameters.
3. Call `vxSetGraphScheduleConfig` with the parameters as follows:
   - Set scheduling mode (`VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL` or `VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO`).
   - List the graph parameters on which enqueue / dequeue operations are required.
   - For these parameters specify the list of references that could be enqueued later.
4. All other data references created in, and associated with, the graph are made specific to the graph. A data reference can be made specific to a graph by either creating it as virtual or by exporting and re-importing the graph using the import/export extension.
5. Delays in the graph, if any, MUST be set to auto-age using `vxRegisterAutoAging`.
6. Verify the graph using `vxVerifyGraph`.
7. Now data reference enqueue / dequeue can be done on associated graph parameters using `vxGraphParameterEnqueueReadyRef` and `vxGraphParameterDequeueDoneRef`.
8. Graph execution on enqueued parameters depends on the scheduling mode chosen:
   - `VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL`: User manually schedules the graph on the full set of all enqueued parameters by calling `vxScheduleGraph`. This gives more control to the application to limit when the graph execution on enqueued parameters can begin.
   - `VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO`: Implementation automatically schedules graph as long as enough data is enqueued to it. This gives more control to the implementation to decide when the graph execution on enqueued parameters can begin.
9. `vxGraphParameterCheckDoneRef` can be used to determine when to dequeue graph parameters for completed graph executions.
10. In order to gracefully end graph pipelining, the application should cease enqueing graph parameters, and call `vxWaitGraph` to wait for the in-flight graph executions to complete. When

the call returns, call vxGraphParameterDequeueDoneRef on all the graph parameters to return control of the buffers to the application.

The following code offers an example of the process outlined above, using VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO scheduling mode.

```
/*
 * index of graph parameter data reference which is used to provide input to the graph
 */
#define GRAPH_PARAMETER_IN  (0u)
/*
 * index of graph parameter data reference which is used to provide output to the
graph
 */
#define GRAPH_PARAMETER_OUT (1u)
/*
 * max parameters to this graph
 */
#define GRAPH_PARAMETER_MAX (2u)

/*
 * Utility API used to add a graph parameter from a node, node parameter index
 */
void add_graph_parameter_by_node_index(vx_graph graph, vx_node node,
                                       vx_uint32 node_parameter_index)
{
    vx_parameter parameter = vxGetParameterByIndex(node, node_parameter_index);
    vxAddParameterToGraph(graph, parameter);
    vxReleaseParameter(&parameter);
}

/*
 * Utility API used to create graph with graph parameter for input and output
 *
 * The following graph is created,
 * IN_IMG -> EXTRACT_NODE -> TMP_IMG -> CONVERT_DEPTH_NODE -> OUT_IMG
 *                                          ^
 *                                          |
 *                                      SHIFT_SCALAR
 *
 * IN_IMG and OUT_IMG are graph parameters.
 * TMP_IMG is a virtual image
 */
static vx_graph create_graph(vx_context context, vx_uint32 width, vx_uint32 height)
{
    vx_graph graph;
    vx_node n0, n1;
    vx_image tmp_img;
    vx_int32 shift;
    vx_scalar s0;
```

```
    graph = vxCreateGraph(context);

    /* create intermediate virtual image */
    tmp_img = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);

    /* create first node, input is NULL this will be made as graph parameter */
    n0 = vxChannelExtractNode(graph, NULL, VX_CHANNEL_G, tmp_img);

    /* create a scalar object required for second node */
    shift = 8;
    s0 = vxCreateScalar(context, VX_TYPE_INT32, &shift);

    /* create second node, output is NULL since this will be made as graph parameter
     */
    n1 = vxConvertDepthNode(graph, tmp_img, NULL, VX_CONVERT_POLICY_SATURATE, s0);

    /* add graph parameters */
    add_graph_parameter_by_node_index(graph, n0, 0);
    add_graph_parameter_by_node_index(graph, n1, 1);

    vxReleaseScalar(&s0);
    vxReleaseNode(&n0);
    vxReleaseNode(&n1);
    vxReleaseImage(&tmp_img);

    return graph;
}

/*
 * Utility API used to fill data and enqueue input to graph
 */
static void enqueue_input(vx_graph graph,
                          vx_uint32 width, vx_uint32 height, vx_image in_img)
{
    vx_rectangle_t rect = { 0, 0, width, height};
    vx_imagepatch_addressing_t imagepatch_addr;
    vx_map_id map_id;
    void *user_ptr;

    if(in_img!=NULL)
    {
        /* Fill input data using Copy/Map/SwapHandles */
        vxMapImagePatch(in_img, &rect, 0, &map_id, &imagepatch_addr, &user_ptr,
                        VX_WRITE_ONLY, VX_MEMORY_TYPE_NONE, VX_NOGAP_X);
        /* ... */
        vxUnmapImagePatch(in_img, map_id);
        vxGraphParameterEnqueueReadyRef(graph, GRAPH_PARAMETER_IN,
                                        (vx_reference*)&in_img, 1);
    }
}
```

```
/*
 * Utility API used to fill input to graph
 */
static void dequeue_input(vx_graph graph, vx_image *in_img)
{
    vx_uint32 num_refs;

    *in_img = NULL;

    /* Get consumed input reference */
    vxGraphParameterDequeueDoneRef(graph, GRAPH_PARAMETER_IN,
                                   (vx_reference*)in_img, 1, &num_refs);
}

/*
 * Utility API used to enqueue output to graph
 */
static void enqueue_output(vx_graph graph, vx_image out_img)
{
    if(out_img!=NULL)
    {
        vxGraphParameterEnqueueReadyRef(graph, GRAPH_PARAMETER_OUT,
                                        (vx_reference*)&out_img, 1);
    }
}

static vx_bool is_output_available(vx_graph graph)
{
    vx_uint32 num_refs;

    vxGraphParameterCheckDoneRef(graph, GRAPH_PARAMETER_OUT, &num_refs);

    return (num_refs > 0);
}

/*
 * Utility API used to dequeue output and consume it
 */
static void dequeue_output(vx_graph graph,
                           vx_uint32 width, vx_uint32 height, vx_image *out_img)
{
    vx_rectangle_t rect = { 0, 0, width, height};
    vx_imagepatch_addressing_t imagepatch_addr;
    vx_map_id map_id;
    void *user_ptr;
    vx_uint32 num_refs;

    *out_img = NULL;

    /* Get output reference and consume new data,
```

```
     * waits until a reference is available
     */
    vxGraphParameterDequeueDoneRef(graph, GRAPH_PARAMETER_OUT,
                                   (vx_reference*)out_img, 1, &num_refs);
    if(*out_img!=NULL)
    {
        /* Consume output data using Copy/Map/SwapHandles */
        vxMapImagePatch(*out_img, &rect, 0, &map_id, &imagepatch_addr, &user_ptr,
                        VX_READ_ONLY, VX_MEMORY_TYPE_NONE, VX_NOGAP_X);
        /* ... */
        vxUnmapImagePatch(*out_img, map_id);
    }
}

/* Max number of input references */
#define GRAPH_PARAMETER_IN_MAX_REFS   (2u)
/* Max number of output references */
#define GRAPH_PARAMETER_OUT_MAX_REFS   (2u)

/* execute graph in a pipelined manner
 */
void vx_khr_pipelining()
{
    vx_uint32 width = 640, height = 480, i;
    vx_context context;
    vx_graph graph;
    vx_image in_refs[GRAPH_PARAMETER_IN_MAX_REFS];
    vx_image out_refs[GRAPH_PARAMETER_IN_MAX_REFS];
    vx_image in_img, out_img;
    vx_graph_parameter_queue_params_t graph_parameters_queue_params_list
[GRAPH_PARAMETER_MAX];

    context = vxCreateContext();
    graph = create_graph(context, width, height);

    create_data_refs(context, in_refs, out_refs, GRAPH_PARAMETER_IN_MAX_REFS,
                     GRAPH_PARAMETER_OUT_MAX_REFS, width, height);

    graph_parameters_queue_params_list[0].graph_parameter_index =
            GRAPH_PARAMETER_IN;
    graph_parameters_queue_params_list[0].refs_list_size =
            GRAPH_PARAMETER_IN_MAX_REFS;
    graph_parameters_queue_params_list[0].refs_list =
            (vx_reference*)&in_refs[0];
    graph_parameters_queue_params_list[1].graph_parameter_index =
            GRAPH_PARAMETER_OUT;
    graph_parameters_queue_params_list[1].refs_list_size =
            GRAPH_PARAMETER_OUT_MAX_REFS;
    graph_parameters_queue_params_list[1].refs_list =
            (vx_reference*)&out_refs[0];
```

```
vxSetGraphScheduleConfig(graph,
        VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO,
        GRAPH_PARAMETER_MAX,
        graph_parameters_queue_params_list
        );

vxVerifyGraph(graph);

/* enqueue input and output to trigger graph */
for(i=0; i<GRAPH_PARAMETER_IN_MAX_REFS; i++)
{
    enqueue_input(graph, width, height, in_refs[i]);
}
for(i=0; i<GRAPH_PARAMETER_OUT_MAX_REFS; i++)
{
    enqueue_output(graph, out_refs[i]);
}

while(1)
{
    /* wait for input to be available, dequeue it -
     * BLOCKs until input can be dequeued
     */
    dequeue_input(graph, &in_img);

    /* wait for output to be available, dequeue output and process it -
     * BLOCKs until output can be dequeued
     */
    dequeue_output(graph, width, height, &out_img);

    /* recycle input - fill new data and re-enqueue*/
    enqueue_input(graph, width, height, in_img);

    /* recycle output */
    enqueue_output(graph, out_img);

    if(CheckExit())
    {
        /* App wants to exit, break from main loop */
        break;
    }
}

/*
 * wait until all previous graph executions have completed
 */
vxWaitGraph(graph);

/* flush output references, only required
 * if need to consume last few references
 */
```

```
    while( is_output_available(graph) )
    {
      dequeue_output(graph, width, height, &out_img);
    }

    vxReleaseGraph(&graph);
    release_data_refs(in_refs, out_refs, GRAPH_PARAMETER_IN_MAX_REFS,
                      GRAPH_PARAMETER_OUT_MAX_REFS);
    vxReleaseContext(&context);
}
```

2.2.4. Example Batch processing application

In order for the graph to be executed in batch processing mode, the steps outlined below need to be followed by an application:

1. Create a graph and add nodes to the graph as usual.
2. For data references which need to be "batched" by the application, add them as graph parameters.
3. Call vxSetGraphScheduleConfig with the parameters as follows:
   - Set scheduling mode (VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL or VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO).
   - List the graph parameters which will be batch processed.
   - For these parameters specify the list of references that could be enqueued later for batch processing.
4. All other data references created in, and associated with the graph are made specific to the graph. A data reference can be made specific to a graph by either creating it as virtual or by exporting and re-importing the graph using the import/export extension.
5. Delays in the graph, if any, MUST be set to auto-age using vxRegisterAutoAging.
6. Verify the graph using vxVerifyGraph.
7. To execute the graph:
   - Enqueue the data references which need to be processed in a batch using vxGraphParameterEnqueueReadyRef.
   - If scheduling mode was set to VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL, use vxScheduleGraph to trigger the batch processing.
   - Use vxWaitGraph to wait for the batch processing to complete.
   - Dequeue the processed data references using vxGraphParameterDequeueDoneRef.

The following code offers a example of the process outlined above using VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL scheduling mode.

```
/* Max batch size supported by application */
#define GRAPH_PARAMETER_MAX_BATCH_SIZE  (10u)
```

```
/* execute graph in a batch-processing manner
 */
void vx_khr_batch_processing()
{
    vx_uint32 width = 640, height = 480, actual_batch_size;
    vx_context context;
    vx_graph graph;
    vx_image in_refs[GRAPH_PARAMETER_MAX_BATCH_SIZE];
    vx_image out_refs[GRAPH_PARAMETER_MAX_BATCH_SIZE];
    vx_graph_parameter_queue_params_t graph_parameters_queue_params_list
[GRAPH_PARAMETER_MAX];

    context = vxCreateContext();
    graph = create_graph(context, width, height);

    create_data_refs(context, in_refs, out_refs, GRAPH_PARAMETER_MAX_BATCH_SIZE,
                     GRAPH_PARAMETER_MAX_BATCH_SIZE, width, height);

    graph_parameters_queue_params_list[0].graph_parameter_index = GRAPH_PARAMETER_IN;
    graph_parameters_queue_params_list[0].refs_list_size =
            GRAPH_PARAMETER_MAX_BATCH_SIZE;
    graph_parameters_queue_params_list[0].refs_list = (vx_reference*)&in_refs[0];
    graph_parameters_queue_params_list[1].graph_parameter_index = GRAPH_PARAMETER_OUT;
    graph_parameters_queue_params_list[1].refs_list_size =
            GRAPH_PARAMETER_MAX_BATCH_SIZE;
    graph_parameters_queue_params_list[1].refs_list = (vx_reference*)&out_refs[0];

    vxSetGraphScheduleConfig(graph,
            VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL,
            GRAPH_PARAMETER_MAX,
            graph_parameters_queue_params_list
            );

    vxVerifyGraph(graph);

    while(1)
    {
        /* read next batch of input and output */
        get_input_output_batch(in_refs, out_refs,
                GRAPH_PARAMETER_MAX_BATCH_SIZE,
                &actual_batch_size);

        vxGraphParameterEnqueueReadyRef(graph,
            GRAPH_PARAMETER_IN,
            (vx_reference*)&in_refs[0],
            actual_batch_size);

        vxGraphParameterEnqueueReadyRef(
            graph,
            GRAPH_PARAMETER_OUT,
```

```
            (vx_reference*)&out_refs[0],
            actual_batch_size);

        /* trigger processing of previously enqueued input and output */
        vxScheduleGraph(graph);
        /* wait for the batch processing to complete */
        vxWaitGraph(graph);

        /* dequeue the processed input and output data */
        vxGraphParameterDequeueDoneRef(graph,
            GRAPH_PARAMETER_IN,
            (vx_reference*)&in_refs[0],
            GRAPH_PARAMETER_MAX_BATCH_SIZE,
            &actual_batch_size);

        vxGraphParameterDequeueDoneRef(
            graph,
            GRAPH_PARAMETER_OUT,
            (vx_reference*)&out_refs[0],
            GRAPH_PARAMETER_MAX_BATCH_SIZE,
            &actual_batch_size);

        if(CheckExit())
        {
            /* App wants to exit, break from main loop */
            break;
        }
    }

    vxReleaseGraph(&graph);
    release_data_refs(in_refs, out_refs, GRAPH_PARAMETER_MAX_BATCH_SIZE,
                      GRAPH_PARAMETER_MAX_BATCH_SIZE);
    vxReleaseContext(&context);
}
```

2.3. Streaming

OpenVX APIs allow a user to construct a graph with source nodes and sink nodes. A source node is a node which takes no input and only outputs data to one or more data references. A sink node is a node which takes one or more data references as input but produces no output. For such a graph, graph execution can be started in streaming mode, wherein, user intervention is not needed to re-schedule the graph each time.

2.3.1. Source/sink user nodes

Source/sink user nodes are implemented using the existing user kernel OpenVX API.

The following is an example of streaming user source node where the data references are coming from a vendor specific capture device component:

```c
static vx_status user_node_source_validate(
                vx_node node,
                const vx_reference parameters[],
                vx_uint32 num,
                vx_meta_format metas[])
{
    /* if any verification checks do here */
    return VX_SUCCESS;
}

static vx_status user_node_source_init(
                vx_node node,
                const vx_reference parameters[],
                vx_uint32 num)
{
    vx_image img = (vx_image)parameters[0];
    vx_uint32 width, height, i;
    vx_enum df;

    vxQueryImage(img, VX_IMAGE_WIDTH, &width, sizeof(vx_uint32));
    vxQueryImage(img, VX_IMAGE_HEIGHT, &height, sizeof(vx_uint32));
    vxQueryImage(img, VX_IMAGE_FORMAT, &df, sizeof(vx_enum));

    CaptureDeviceOpen(&capture_dev, width, height, df);
    /* allocate images for priming the capture device.
     * Typically capture devices need some image references to be
     * primed in order to start capturing data.
     */
    CaptureDeviceAllocHandles(capture_dev, capture_refs_prime,
                              MAX_CAPTURE_REFS_PRIME);
    /* prime image references to capture device */
    for(i=0; i<MAX_CAPTURE_REFS_PRIME; i++)
    {
        CaptureDeviceSwapHandles(capture_dev, capture_refs_prime[i], NULL);
    }
    /* start capturing data to primed image references */
    CaptureDeviceStart(capture_dev);

    return VX_SUCCESS;
}

static vx_status user_node_source_run(
                    vx_node node,
                    vx_reference parameters[],
                    vx_uint32 num)
{
    vx_reference empty_ref, full_ref;

    empty_ref = parameters[0];

    /* swap a 'empty' image reference with a captured image reference filled with data
```

```
     * If this is one of the first few calls to CaptureDeviceSwapHandle, then full_buf
     * would be one of the image references primed during user_node_source_init
     */
    CaptureDeviceSwapHandles(capture_dev, empty_ref, &full_ref);

    parameters[0] = full_ref;

    return VX_SUCCESS;
}

static vx_status user_node_source_deinit(
                    vx_node node,
                    const vx_reference parameters[],
                    vx_uint32 num)
{
    CaptureDeviceStop(capture_dev);
    CaptureDeviceFreeHandles(capture_dev, capture_refs_prime, MAX_CAPTURE_REFS_PRIME);
    CaptureDeviceClose(&capture_dev);

    return VX_SUCCESS;
}

/* Add user node as streaming node */
static void user_node_source_add(vx_context context)
{
    vxAllocateUserKernelId(context, &user_node_source_kernel_id);

    user_node_source_kernel = vxAddUserKernel(
            context,
            "user_kernel.source",
            user_node_source_kernel_id,
            (vx_kernel_f)user_node_source_run,
            1,
            user_node_source_validate,
            user_node_source_init,
            user_node_source_deinit
            );

    vxAddParameterToKernel(user_node_source_kernel,
        0,
        VX_OUTPUT,
        VX_TYPE_IMAGE,
        VX_PARAMETER_STATE_REQUIRED
        );

    vxFinalizeKernel(user_node_source_kernel);
}

/* Boiler plate code of standard OpenVX API, nothing specific to streaming API */
static void user_node_source_remove()
{
```

```
    vxRemoveKernel(user_node_source_kernel);
}

/* Boiler plate code of standard OpenVX API, nothing specific to streaming API */
static vx_node user_node_source_create_node(vx_graph graph, vx_image output)
{
    vx_node node = NULL;

    node = vxCreateGenericNode(graph, user_node_source_kernel);
    vxSetParameterByIndex(node, 0, (vx_reference)output);

    return node;
}
```

Likewise, the following is an example of streaming user sink node where the data references are going to a vendor specific display device component:

```
/* Boiler plate code of standard OpenVX API, nothing specific to streaming API */
static vx_status user_node_sink_validate(
                    vx_node node,
                     const vx_reference parameters[],
                     vx_uint32 num,
                     vx_meta_format metas[])
{
    /* if any verification checks do here */
    return VX_SUCCESS;
}

static vx_status user_node_sink_init(
                    vx_node node,
                    const vx_reference parameters[],
                    vx_uint32 num)
{
    vx_image img = (vx_image)parameters[0];
    vx_uint32 width, height;
    vx_enum df;

    vxQueryImage(img, VX_IMAGE_WIDTH, &width, sizeof(vx_uint32));
    vxQueryImage(img, VX_IMAGE_HEIGHT, &height, sizeof(vx_uint32));
    vxQueryImage(img, VX_IMAGE_FORMAT, &df, sizeof(vx_enum));

    DisplayDeviceOpen(&display_dev, width, height, df);

    return VX_SUCCESS;
}

static vx_status user_node_sink_run(
                    vx_node node,
                    vx_reference parameters[],
                    vx_uint32 num)
```

```
{
    vx_reference new_ref, old_ref;

    new_ref = parameters[0];

    /* swap input reference with reference currently held by display if this is
     * first call to DisplayDeviceSwapHandle, then out_ref could be NULL
     * reference when returned via parameters to framework is ignored by framework
     * non-NULL reference when returned via parameters to framework is recycled
     * by framework for subsequent graph execution
     */
    DisplayDeviceSwapHandles(display_dev, new_ref, &old_ref);

    parameters[0] = old_ref;

    return VX_SUCCESS;
}

static vx_status user_node_sink_deinit(
                vx_node node,
                const vx_reference parameters[],
                vx_uint32 num)
{
    DisplayDeviceClose(&display_dev);

    return VX_SUCCESS;
}

/* Add user node as streaming node */
static void user_node_sink_add(vx_context context)
{
    vxAllocateUserKernelId(context, &user_node_sink_kernel_id);

    user_node_sink_kernel = vxAddUserKernel(
            context,
            "user_kernel.sink",
            user_node_sink_kernel_id,
            (vx_kernel_f)user_node_sink_run,
            1,
            user_node_sink_validate,
            user_node_sink_init,
            user_node_sink_deinit
            );

    vxAddParameterToKernel(user_node_sink_kernel,
        0,
        VX_INPUT,
        VX_TYPE_IMAGE,
        VX_PARAMETER_STATE_REQUIRED
        );
```

```
    vxFinalizeKernel(user_node_sink_kernel);
}

/* Boiler plate code of standard OpenVX API, nothing specific to streaming API */
static void user_node_sink_remove()
{
    vxRemoveKernel(user_node_sink_kernel);
}

/* Boiler plate code of standard OpenVX API, nothing specific to streaming API */
static vx_node user_node_sink_create_node(vx_graph graph, vx_image input)
{
    vx_node node = NULL;

    node = vxCreateGenericNode(graph, user_node_sink_kernel);
    vxSetParameterByIndex(node, 0, (vx_reference)input);

    return node;
}
```

In both these examples, the user node “swaps” the reference provided by the implementation with another “compatible” reference. This allows user nodes to implement zero-copy capture and display functions.

2.3.2. Graph streaming application

To execute a graph in streaming mode, the following steps need to followed by an application:

- Create a graph with source and sink nodes.
- All data references created in and associated with the graph are made specific to the graph. A data reference can be made specific to a graph by either creating it as virtual or by exporting and re-importing the graph using the import/export extension.
- Verify the graph using `vxVerifyGraph`
- Start the streaming mode of graph execution using `vxStartGraphStreaming`
- Now the graph gets re-scheduled continuously.
  - The implementation automatically decides the re-schedule trigger condition.
- Sometimes a user node may want to stop the continuous graph execution due to end of stream or error condition detected within its node execution. In this case the user node should return an error status. When a error status is returned by the user node, the continuous graph execution is stopped.
- Application can use `vxWaitGraph` to wait for streaming graph execution to stop on its own.
- Alternatively, user application can explicitly stop the streaming mode of execution using `vxStopGraphStreaming`.
- In all cases, the continuous mode of graph execution is stopped at an implementation-defined logical boundary (e.g. after all previous graph executions have completed).

The following example code demonstrates how one can use these APIs in an application,

```
/*
 * Utility API used to create graph with source and sink nodes
 */
static vx_graph create_graph(vx_context context, vx_uint32 width, vx_uint32 height)
{
    vx_graph graph;
    vx_node n0, n1, node_source, node_sink;
    vx_image in_img, tmp_img, out_img;
    vx_int32 shift;
    vx_scalar s0;

    graph = vxCreateGraph(context);

    in_img = vxCreateVirtualImage(graph, width, height, VX_DF_IMAGE_RGB);

    /* create source node */
    node_source = user_node_source_create_node(graph, in_img);

    /* create intermediate virtual image */
    tmp_img = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);

    /* create first node, input is NULL since this will be made as graph parameter */
    n0 = vxChannelExtractNode(graph, in_img, VX_CHANNEL_G, tmp_img);

    out_img = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_S16);

    /* create a scalar object required for second node */
    shift = 8;
    s0 = vxCreateScalar(context, VX_TYPE_INT32, &shift);

    /* create second node, output is NULL since this will be made as graph parameter
     */
    n1 = vxConvertDepthNode(graph, tmp_img, out_img, VX_CONVERT_POLICY_SATURATE, s0);

    /* create sink node */
    node_sink = user_node_sink_create_node(graph, out_img);

    vxReleaseScalar(&s0);
    vxReleaseNode(&n0);
    vxReleaseNode(&n1);
    vxReleaseNode(&node_source);
    vxReleaseNode(&node_sink);
    vxReleaseImage(&tmp_img);
    vxReleaseImage(&in_img);
    vxReleaseImage(&out_img);

    return graph;
}
```

```c
void vx_khr_streaming_sample()
{
    vx_uint32 width = 640, height = 480;
    vx_context context = vxCreateContext();
    vx_graph graph;

    /* add user kernels to context */
    user_node_source_add(context);
    user_node_sink_add(context);

    graph = create_graph(context, width, height);

    vxVerifyGraph(graph);

    /* execute graph in streaming mode,
     * graph is retriggered when input reference is consumed by a graph execution
     */
    vxStartGraphStreaming(graph);

    /* wait until user wants to exit */
    WaitExit();

    /* stop graph streaming */
    vxStopGraphStreaming(graph);

    vxReleaseGraph(&graph);

    /* remove user kernels from context */
    user_node_source_remove();
    user_node_sink_remove();

    vxReleaseContext(&context);
}
```

2.4. Event handling

Event handling APIs allow users to register conditions on a graph, based on which events are generated by the implementation. User applications can then wait for events and take appropriate action based on the received event. User-specified events can also be generated by the application so that all events can be handled at a centralized location. This simplifies the application state machine, and in the case of graph pipelining, it allows optimized scheduling of the graph.

2.4.1. Motivation for event handling

1. Pipelining without events would need blocking calls on the data producers, consumers, and the graph itself. If there were multiple graphs or multiple data producers/consumers pipelined at different rates, one can see how the application logic can easily get complicated.
2. Applications need a mechanism to allow input references to be dequeued before the full graph

execution is completed. This allows implementations to have larger pipeline depths but at the same time have fewer queued references at a graph parameter.

2.4.2. Event handling application

Event handling APIs allow user the flexibility to do early dequeue of input references, and late enqueue of output references. It enables applications to effectively block at a single centralized location for both implementation-generated events as well as user-generated events. Event handling allows the graph to produce events which can then be used by the application. For example, if the thread had an event handler that is used to manage multiple graphs, consumers, and producers, then the events produced by the implementation could feed into this manager. Likewise, early dequeue of input can be achieved, if the event handler could use the graph parameter consumed events to trigger calls to vxGraphParameterEnqueueReadyRef, vxGraphParameterDequeueDoneRef.

The following code offers an example of the event handling.

```
/* Utility API to clear any pending events */
static void clear_pending_events(vx_context context)
{
    vx_event_t event;

    /* do not block */
    while(vxWaitEvent(context, &event, vx_true_e)==VX_SUCCESS)
        ;
}

/* execute graph in a pipelined manner with events used
 * to schedule the graph execution
 */
void vx_khr_pipelining_with_events()
{
    vx_uint32 width = 640, height = 480, i;
    vx_context context;
    vx_graph graph;
    vx_image in_refs[GRAPH_PARAMETER_IN_MAX_REFS];
    vx_image out_refs[GRAPH_PARAMETER_IN_MAX_REFS];
    vx_image in_img, out_img;
    vx_graph_parameter_queue_params_t graph_parameters_queue_params_list
[GRAPH_PARAMETER_MAX];

    context = vxCreateContext();
    graph = create_graph(context, width, height);

    create_data_refs(context, in_refs, out_refs, GRAPH_PARAMETER_IN_MAX_REFS,
                     GRAPH_PARAMETER_OUT_MAX_REFS, width, height);

    graph_parameters_queue_params_list[0].graph_parameter_index = GRAPH_PARAMETER_IN;
    graph_parameters_queue_params_list[0].refs_list_size =
                    GRAPH_PARAMETER_IN_MAX_REFS;
```

```
    graph_parameters_queue_params_list[0].refs_list = (vx_reference*)&in_refs[0];
    graph_parameters_queue_params_list[1].graph_parameter_index = GRAPH_PARAMETER_OUT;
    graph_parameters_queue_params_list[1].refs_list_size =
                    GRAPH_PARAMETER_OUT_MAX_REFS;
    graph_parameters_queue_params_list[1].refs_list = (vx_reference*)&out_refs[0];

    vxSetGraphScheduleConfig(graph,
            VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO,
            GRAPH_PARAMETER_MAX,
            graph_parameters_queue_params_list
            );

    /* register events for input consumed and output consumed */
    vxRegisterEvent((vx_reference)graph, VX_EVENT_GRAPH_PARAMETER_CONSUMED,
                    GRAPH_PARAMETER_IN);
    vxRegisterEvent((vx_reference)graph, VX_EVENT_GRAPH_PARAMETER_CONSUMED,
                    GRAPH_PARAMETER_OUT);

    vxVerifyGraph(graph);

    /* disable events generation */
    vxEnableEvents(context);
    /* clear pending events.
     * Not strictly required but- it's a good practice to clear any
     * pending events from last execution before waiting on new events */
    clear_pending_events(context);

    /* enqueue input and output to trigger graph */
    for(i=0; i<GRAPH_PARAMETER_IN_MAX_REFS; i++)
    {
        enqueue_input(graph, width, height, in_refs[i]);
    }
    for(i=0; i<GRAPH_PARAMETER_OUT_MAX_REFS; i++)
    {
        enqueue_output(graph, out_refs[i]);
    }

    while(1)
    {
        vx_event_t event;

        /* wait for events, block until event is received */
        vxWaitEvent(context, &event, vx_false_e);

        /* event for input data ready for recycling, i.e early input release */
        if(event.type == VX_EVENT_GRAPH_PARAMETER_CONSUMED
            && event.event_info.graph_parameter_consumed.graph == graph
            && event.event_info.graph_parameter_consumed.graph_parameter_index
                    == GRAPH_PARAMETER_IN
            )
        {
```

```
            /* dequeue consumed input, fill new data and re-enqueue */
            dequeue_input(graph, &in_img);
            enqueue_input(graph, width, height, in_img);
        }
        else
        /* event for output data ready for recycling, i.e output release */
        if(event.type == VX_EVENT_GRAPH_PARAMETER_CONSUMED
            && event.event_info.graph_parameter_consumed.graph == graph
            && event.event_info.graph_parameter_consumed.graph_parameter_index
                    == GRAPH_PARAMETER_OUT
            )
        {
            /* dequeue output reference, consume generated data and
             * re-enqueue output reference
             */
            dequeue_output(graph, width, height, &out_img);
            enqueue_output(graph, out_img);
        }
        else
        if(event.type == VX_EVENT_USER && event.event_info.user_event.user_event_id
                                            == 0xDEADBEEF /* app code for exit */
            )
        {
            /* App wants to exit, break from main loop */
            break;
        }
    }

    /*
     * wait until all previous graph executions have completed
     */
    vxWaitGraph(graph);

    /* flush output references, only required if need to consume last few references
     */
    do {
      dequeue_output(graph, width, height, &out_img);
    } while(out_img!=NULL);

    vxReleaseGraph(&graph);
    release_data_refs(in_refs, out_refs, GRAPH_PARAMETER_IN_MAX_REFS,
                      GRAPH_PARAMETER_OUT_MAX_REFS);

    /* disable events generation */
    vxDisableEvents(context);
    /* clear pending events.
     * Not strictly required but- it's a good practice to clear any
     * pending events from last execution before exiting application */
    clear_pending_events(context);

    vxReleaseContext(&context);
```

```
}
```

Chapter 3. Module Documentation

3.1. Pipelining and Batch Processing

Data Structures

- vx_graph_parameter_queue_params_t

Enumerations

- vx_graph_schedule_mode_enum_e
- vx_graph_schedule_mode_type_e
- vx_graph_attribute_pipelining_e

Functions

- vxSetGraphScheduleConfig
- vxGraphParameterEnqueueReadyRef
- vxGraphParameterDequeueDoneRef
- vxGraphParameterCheckDoneRef

This section lists the APIs required for graph pipelining and batch processing.

3.1.1. Data Structures

vx_graph_parameter_queue_params_t

Queueing parameters for a specific graph parameter.

```
typedef struct _vx_graph_parameter_queue_params_t {
    uint32_t          graph_parameter_index;
    vx_uint32         refs_list_size;
    vx_reference *    refs_list;
} vx_graph_parameter_queue_params_t;
```

- *graph_parameter_index* - Index of graph parameter to which these properties apply
- *refs_list_size* - Number of elements in array *refs_list*
- *refs_list* - Array of references that could be enqueued at a later point of time at this graph parameter

See vxSetGraphScheduleConfig for additional details.

3.1.2. Enumerations

vx_graph_schedule_mode_enum_e

Extra enums.

```
enum vx_graph_schedule_mode_enum_e {
    VX_ENUM_GRAPH_SCHEDULE_MODE_TYPE = 0x1E,
};
```

Enumerator

- VX_ENUM_GRAPH_SCHEDULE_MODE_TYPE - Graph schedule mode type enumeration.

vx_graph_schedule_mode_type_e

Type of graph scheduling mode.

```
enum vx_graph_schedule_mode_type_e {
    VX_GRAPH_SCHEDULE_MODE_NORMAL = ((( VX_ID_KHRONOS ) << 20) | (
VX_ENUM_GRAPH_SCHEDULE_MODE_TYPE << 12)) + 0x0,
    VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO = ((( VX_ID_KHRONOS ) << 20) | (
VX_ENUM_GRAPH_SCHEDULE_MODE_TYPE << 12)) + 0x1,
    VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL = ((( VX_ID_KHRONOS ) << 20) | (
VX_ENUM_GRAPH_SCHEDULE_MODE_TYPE << 12)) + 0x2,
};
```

See vxSetGraphScheduleConfig and vxGraphParameterEnqueueReadyRef for details about each mode.

Enumerator

- VX_GRAPH_SCHEDULE_MODE_NORMAL - Schedule graph in non-queueing mode.
- VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO - Schedule graph in queueing mode with auto scheduling.
- VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL - Schedule graph in queueing mode with manual scheduling.

vx_graph_attribute_pipelining_e

The graph attributes added by this extension.

```
enum vx_graph_attribute_pipelining_e {
    VX_GRAPH_SCHEDULE_MODE = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS, VX_TYPE_GRAPH) + 0x5,
};
```

Enumerator

- VX_GRAPH_SCHEDULE_MODE - Returns the schedule mode of a graph. Read-only. Use a vx_enum parameter. See vx_graph_schedule_mode_type_e enum.

3.1.3. Functions

vxSetGraphScheduleConfig

Sets the graph scheduler config.

```
VX_API_CALL vx_status VX_API_ENTRY vxSetGraphScheduleConfig(
    vx_graph                                    graph,
    vx_enum                                     graph_schedule_mode,
    uint32_t                                    graph_parameters_list_size,
    const vx_graph_parameter_queue_params_t     graph_parameters_queue_params_list[]);
```

This API is used to set the graph scheduler config to allow user to schedule multiple instances of a graph for execution.

For legacy applications that don't need graph pipelining or batch processing, this API need not be used.

Using this API, the application specifies the graph schedule mode, as well as queueing parameters for all graph parameters that need to allow enqueueing of references. A single monolithic API is provided instead of discrete APIs, since this allows the implementation to get all information related to scheduling in one shot and then optimize the subsequent graph scheduling based on this information. **This API MUST be called before graph verify**, since in this case it allows implementations the opportunity to optimize resources based on information provided by the application.

graph_schedule_mode selects how input and output references are provided to a graph and how the next graph schedule is triggered by an implementation.

Below scheduling modes are supported:

When graph schedule mode is `VX_GRAPH_SCHEDULE_MODE_QUEUE_AUTO`:

- Application needs to explicitly call `vxVerifyGraph` before enqueing data references
- Application should not call `vxScheduleGraph` or `vxProcessGraph`
- When enough references are enqueued at various graph parameters, the implementation could trigger the next graph schedule.
- Here, not all graph parameters need to have enqueued references for a graph schedule to begin. An implementation is expected to execute the graph as much as possible until a enqueued reference is not available at which time it will stall the graph until the reference becomes available. This allows application to schedule a graph even when all parameters references are not yet available, i.e do a "late" enqueue. However, exact behaviour is implementation specific.

When graph schedule mode is `VX_GRAPH_SCHEDULE_MODE_QUEUE_MANUAL`:

- Application needs to explicitly call `vxScheduleGraph`
- Application should not call `vxProcessGraph`
- References for all graph parameters of the graph needs to enqueued before `vxScheduleGraph` is called on the graph else an error is returned by `vxScheduleGraph`

- Application can enqueue multiple references at the same graph parameter. When `vxScheduleGraph` is called, all enqueued references get processed in a “batch”.
- User can use `vxWaitGraph` to wait for the previous `vxScheduleGraph` to complete.

When graph schedule mode is `VX_GRAPH_SCHEDULE_MODE_NORMAL`:

- *graph_parameters_list_size* MUST be 0 and
- *graph_parameters_queue_params_list* MUST be NULL
- This mode is equivalent to non-queueing scheduling mode as defined by OpenVX v1.2 and earlier.

By default all graphs are in `VX_GRAPH_SCHEDULE_MODE_NORMAL` mode until this API is called.

graph_parameters_queue_params_list allows to specify below information:

- For the graph parameter index that is specified, it enables queueing mode of operation
- Further it allows the application to specify the list of references that it could later enqueue at this graph parameter.

For graph parameters listed in *graph_parameters_queue_params_list*, application MUST use `vxGraphParameterEnqueueReadyRef` to set references at the graph parameter. Using other data access API’s on these parameters or corresponding data objects will return an error. For graph parameters not listed in *graph_parameters_queue_params_list* application MUST use the `vxSetGraphParameterByIndex` to set the reference at the graph parameter. Using other data access API’s on these parameters or corresponding data objects will return an error.

This API also allows application to provide a list of references which could be later enqueued at the graph parameter. This allows implementation to do meta-data checking up front rather than during each reference enqueue.

When this API is called before `vxVerifyGraph`, the *refs_list* field can be NULL, if the reference handles are not available yet at the application. However *refs_list_size* MUST always be specified by the application. Application can call `vxSetGraphScheduleConfig` again after verify graph with all parameters remaining the same except with *refs_list* field providing the list of references that can be enqueued at the graph parameter.

Parameters

- `[in]` *graph* - Graph reference
- `[in]` *graph_schedule_mode* - Graph schedule mode. See `vx_graph_schedule_mode_type_e`
- `[in]` *graph_parameters_list_size* - Number of elements in *graph_parameters_queue_params_list*
- `[in]` *graph_parameters_queue_params_list* - Array containing queuing properties at graph parameters that need to support queueing.

Returns: A `vx_status_e` enumeration.

Return Values

- VX_SUCCESS - No errors.
- VX_ERROR_INVALID_REFERENCE - *graph* is not a valid reference
- VX_ERROR_INVALID_PARAMETERS - Invalid graph parameter queueing parameters
- VX_FAILURE - Any other failure.

vxGraphParameterEnqueueReadyRef

Enqueues new references into a graph parameter for processing.

```
VX_API_CALL vx_status VX_API_ENTRY vxGraphParameterEnqueueReadyRef(
    vx_graph                                    graph,
    vx_uint32                                   graph_parameter_index,
    vx_reference *                              refs,
    vx_uint32                                   num_refs);
```

This new reference will take effect on the next graph schedule.

In case of a graph parameter which is input to a graph, this function provides a data reference with new input data to the graph. In case of a graph parameter which is not input to a graph, this function provides a "empty" reference into which a graph execution can write new data into.

This function essentially transfers ownership of the reference from the application to the graph.

User MUST use vxGraphParameterDequeueDoneRef to get back the processed or consumed references.

The references that are enqueued MUST be the references listed during vxSetGraphScheduleConfig. If a reference outside this list is provided then behaviour is undefined.

Parameters

- [in] *graph* - Graph reference
- [in] *graph_parameter_index* - Graph parameter index
- [in] *refs* - The array of references to enqueue into the graph parameter
- [in] *num_refs* - Number of references to enqueue

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors.
- VX_ERROR_INVALID_REFERENCE - *graph* is not a valid reference OR reference is not a valid reference
- VX_ERROR_INVALID_PARAMETERS - *graph_parameter_index* is NOT a valid graph parameter index
- VX_FAILURE - Reference could not be enqueued.

vxGraphParameterDequeueDoneRef

Dequeues "consumed" references from a graph parameter.

```
VX_API_CALL vx_status VX_API_ENTRY vxGraphParameterDequeueDoneRef(
    vx_graph                                    graph,
    vx_uint32                                   graph_parameter_index,
    vx_reference *                              refs,
    vx_uint32                                   max_refs,
    vx_uint32 *                                 num_refs);
```

This function dequeues references from a graph parameter of a graph. The reference that is dequeued is a reference that had been previously enqueued into a graph, and after subsequent graph execution is considered as processed or consumed by the graph. This function essentially transfers ownership of the reference from the graph to the application.

IMPORTANT : This API will block until at least one reference is dequeued.

In case of a graph parameter which is input to a graph, this function provides a "consumed" buffer to the application so that new input data can filled and later enqueued to the graph. In case of a graph parameter which is not input to a graph, this function provides a reference filled with new data based on graph execution. User can then use this newly generated data with their application. Typically when this new data is consumed by the application the "empty" reference is again enqueued to the graph.

This API returns an array of references up to a maximum of *max_refs*. Application MUST ensure the array pointer (*refs*) passed as input can hold *max_refs*. *num_refs* is actual number of references returned and will be less than or equal to *max_refs*.

Parameters

- `[in]` *graph* - Graph reference
- `[in]` *graph_parameter_index* - Graph parameter index
- `[in]` *refs* - Pointer to an array of max elements *max_refs*
- `[out]` *refs* - Dequeued references filled in the array
- `[in]` *max_refs* - Max number of references to dequeue
- `[out]` *num_refs* - Actual number of references dequeued.

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors.
- `VX_ERROR_INVALID_REFERENCE` - *graph* is not a valid reference
- `VX_ERROR_INVALID_PARAMETERS` - *graph_parameter_index* is NOT a valid graph parameter index
- `VX_FAILURE` - Reference could not be dequeued.

vxGraphParameterCheckDoneRef

Checks and returns the number of references that are ready for dequeue.

```
VX_API_CALL vx_status VX_API_ENTRY vxGraphParameterCheckDoneRef(
    vx_graph                                    graph,
    vx_uint32                                   graph_parameter_index,
    vx_uint32 *                                 num_refs);
```

This function checks the number of references that can be dequeued and returns the value to the application.

See also `vxGraphParameterDequeueDoneRef`.

Parameters

- `[in]` *graph* - Graph reference
- `[in]` *graph_parameter_index* - Graph parameter index
- `[out]` *num_refs* - Number of references that can be dequeued using

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors.
- `VX_ERROR_INVALID_REFERENCE` - *graph* is not a valid reference
- `VX_ERROR_INVALID_PARAMETERS` - *graph_parameter_index* is NOT a valid graph parameter index
- `VX_FAILURE` - Any other failure.

3.2. Streaming

Functions

- `vxStartGraphStreaming`
- `vxStopGraphStreaming`

This section lists the APIs required for graph streaming.

3.2.1. Functions

vxStartGraphStreaming

Start streaming mode of graph execution.

```
VX_API_CALL vx_status VX_API_ENTRY vxStartGraphStreaming(
    vx_graph                                    graph);
```

In streaming mode of graph execution, once a application starts graph execution further intervention of the application is not needed to re-schedule a graph; i.e. a graph re-schedules itself and executes continuously until streaming mode of execution is stopped.

When this API is called, the framework schedules the graph via `vxScheduleGraph` and returns. This graph gets re-scheduled continuously until `vxStopGraphStreaming` is called by the user or any of the graph nodes return error during execution.

The graph MUST be verified via `vxVerifyGraph` before calling this API. Also user application MUST ensure no previous executions of the graph are scheduled before calling this API.

After streaming mode of a graph has been started, the following APIs should **not* be* used on that graph by an application: `vxScheduleGraph`, `vxWaitScheduleGraphDone`, and `vxIsScheduleGraphAllowed`.

`vxWaitGraph` can be used as before to wait for all pending graph executions to complete.

Parameters

- `[in]` *graph* - Reference to the graph to start streaming mode of execution.

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.
- `VX_ERROR_INVALID_REFERENCE` - *graph* is not a valid `vx_graph` reference.

vxStopGraphStreaming

Stop streaming mode of graph execution.

```
VX_API_CALL vx_status VX_API_ENTRY vxStopGraphStreaming(
    vx_graph                                    graph);
```

This function blocks until graph execution is gracefully stopped at a logical boundary, for example, when all internally scheduled graph executions are completed.

Parameters

- `[in]` *graph* - Reference to the graph to stop streaming mode of execution.

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.
- `VX_FAILURE` - Graph is not started in streaming execution mode.
- `VX_ERROR_INVALID_REFERENCE` - graph is not a valid reference.

3.3. Event Handling

Data Structures

- vx_event_graph_parameter_consumed
- vx_event_graph_completed
- vx_event_node_completed
- vx_event_user_event
- vx_event_t
- vx_event_info_t

Enumerations

- vx_event_enum_e
- vx_event_type_e

Functions

- vxDisableEvents
- vxEnableEvents
- vxRegisterEvent
- vxSendUserEvent
- vxWaitEvent

This section lists the APIs required for event driven graph execution

3.3.1. Data Structures

vx_event_graph_parameter_consumed

Parameter structure returned with event of type VX_EVENT_GRAPH_PARAMETER_CONSUMED.

```
typedef struct _vx_event_graph_parameter_consumed {
    vx_graph     graph;
    vx_uint32    graph_parameter_index;
} vx_event_graph_parameter_consumed;
```

- *graph* - graph which generated this event
- *graph_parameter_index* - graph parameter index which generated this event

vx_event_graph_completed

Parameter structure returned with event of type VX_EVENT_GRAPH_COMPLETED.

```
typedef struct _vx_event_graph_completed {
    vx_graph    graph;
} vx_event_graph_completed;
```

* *graph* - graph which generated this event

vx_event_node_completed

Parameter structure returned with event of type VX_EVENT_NODE_COMPLETED.

```
typedef struct _vx_event_node_completed {
    vx_graph    graph;
    vx_node     node;
} vx_event_node_completed;
```

* *graph* - graph which generated this event
* *node* - node which generated this event

vx_event_user_event

Parameter structure returned with event of type VX_EVENT_USER_EVENT.

```
typedef struct _vx_event_user_event {
    vx_uint32     user_event_id;
    void *        user_event_parameter;
} vx_event_user_event;
```

* *user_event_id* - user event ID associated with this event
* *user_event_parameter* - User defined parameter value. This is used to pass additional user defined parameters with a user event.

vx_event_info_t

Parameter structure associated with a event. Depends on type of the event.

```
typedef union _vx_event_info_t {
    vx_event_graph_parameter_consumed    graph_parameter_consumed;
    vx_event_graph_completed             graph_completed;
    vx_event_node_completed              node_completed;
    vx_event_user_event                  user_event;
} vx_event_info_t;
```

* *graph_parameter_consumed* - event information for type VX_EVENT_GRAPH_PARAMETER_CONSUMED
* *graph_completed* - event information for type VX_EVENT_GRAPH_COMPLETED
* *node_completed* - event information for type VX_EVENT_NODE_COMPLETED
* *user_event* - event information for type VX_EVENT_USER

vx_event_t

Data structure which holds event information.

```
typedef struct _vx_event_t {
    vx_enum             type;
    vx_uint64           timestamp;
    vx_event_info_t     event_info;
} vx_event_t;
```

- *type* - see event type `vx_event_type_e`
- *timestamp* - time at which this event was generated, in units of nano-secs
- *event_info* - parameter structure associated with a event. Depends on *type* of the event

3.3.2. Enumerations

vx_event_enum_e

Extra enums.

```
enum vx_event_enum_e {
    VX_ENUM_EVENT_TYPE = 0x1D,
};
```

Enumerator

- `VX_ENUM_EVENT_TYPE` - Event Type enumeration.

vx_event_type_e

Type of event that can be generated during system execution.

```
enum vx_event_type_e {
    VX_EVENT_GRAPH_PARAMETER_CONSUMED = ((( VX_ID_KHRONOS ) << 20) | (
VX_ENUM_EVENT_TYPE << 12)) + 0x0,
    VX_EVENT_GRAPH_COMPLETED = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_EVENT_TYPE <<
12)) + 0x1,
    VX_EVENT_NODE_COMPLETED = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_EVENT_TYPE <<
12)) + 0x2,
    VX_EVENT_USER = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_EVENT_TYPE << 12)) + 0x3,
};
```

Enumerator

- `VX_EVENT_GRAPH_PARAMETER_CONSUMED` - Graph parameter consumed event.

  This event is generated when a data reference at a graph parameter is consumed during a graph execution. It is used to indicate that a given data reference is no longer used by the graph and can be dequeued and accessed by the application.

> *Note*
>
> Graph execution could still be “in progress” for rest of the graph that does not use this data reference.

- `VX_EVENT_GRAPH_COMPLETED` - Graph completion event.

  This event is generated every time a graph execution completes. Graph completion event is generated for both successful execution of a graph or abandoned execution of a graph.

- `VX_EVENT_NODE_COMPLETED` - Node completion event.

  This event is generated every time a node within a graph completes execution.

- `VX_EVENT_USER` - User defined event.

  This event is generated by user application outside of OpenVX framework using the `vxSendUserEvent` API. User events allow application to have single centralized “wait-for” loop to handle both framework generated events as well as user generated events.

3.3.3. Functions

vxWaitEvent

Wait for a single event.

```
VX_API_CALL vx_status VX_API_ENTRY vxWaitEvent(
    vx_context                                  context,
    vx_event_t *                                event,
    vx_bool                                     do_not_block);
```

After `vxDisableEvents` is called, if `vxWaitEvent(.., .., vx_false_e)` is called, `vxWaitEvent` will remain blocked until events are re-enabled using `vxEnableEvents` and a new event is received.

If `vxReleaseContext` is called while a application is blocked on `vxWaitEvent`, the behavior is not defined by OpenVX.

If `vxWaitEvent` is called simultaneously from multiple thread/task contexts then its behaviour is not defined by OpenVX.

Parameters

- `[in]` *context* - OpenVX context
- `[out]` *event* - Data structure which holds information about a received event
- `[in]` *do_not_block* - When value is `vx_true_e` API does not block and only checks for the condition

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - Event received and event information available in *event*
- VX_FAILURE - No event is received

vxEnableEvents

Enable event generation.

```
VX_API_CALL vx_status VX_API_ENTRY vxEnableEvents(
    vx_context                                  context);
```

Parameters

- [in] *context* - OpenVX context

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors; any other value indicates failure.

vxDisableEvents

Disable event generation.

```
VX_API_CALL vx_status VX_API_ENTRY vxDisableEvents(
    vx_context                                  context);
```

When events are disabled, any event generated before this API is called will still be returned via vxWaitEvent API. However no additional events would be returned via vxWaitEvent API until events are enabled again.

Parameters

- [in] *context* - OpenVX context

Returns: A vx_status_e enumeration.

Return Values

- VX_SUCCESS - No errors; any other value indicates failure.

vxSendUserEvent

Generate user defined event.

```
VX_API_CALL vx_status VX_API_ENTRY vxSendUserEvent(
    vx_context                                  context,
    vx_uint32                                   id,
    void *                                      parameter);
```

Parameters

- `[in]` *context* - OpenVX context
- `[in]` *user_event_id* - User defined event ID
- `[in]` *user_event_parameter* - User defined event parameter. NOT used by implementation. Returned to user as part `vx_event_t.user_event_parameter` field

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.

vxRegisterEvent

Register an event to be generated.

```
VX_API_CALL vx_status VX_API_ENTRY vxRegisterEvent(
    vx_reference                                ref,
    vx_event_type_e                             type,
    vx_uint32                                   param);
```

Generation of event may need additional resources and overheads for an implementation. Hence events should be registered for references only when really required by an application.

This API can be called on graph, node or graph parameter. This API MUST be called before doing `vxVerifyGraph` for that graph.

Parameters

- `[in]` *ref* - Reference which will generate the event
- `[in]` *type* - Type or condition on which the event is generated
- `[in]` *param* - Specifies the graph parameter index when *type* is `VX_EVENT_GRAPH_PARAMETER_CONSUMED`

Returns: A `vx_status_e` enumeration.

Return Values

- `VX_SUCCESS` - No errors; any other value indicates failure.
- `VX_ERROR_INVALID_REFERENCE` - *ref* is not a valid `vx_reference` reference.
- `VX_ERROR_NOT_SUPPORTED` - *type* is not valid for the provided reference.