Video Stabilization OpenVX* Sample

Developer Guide

Intel® Computer Vision SDK – Samples

Contents

Legal Information

Introduction

This Video stabilization OpenVX* sample teaches how to use OpenVX* in your application. It implements a simplified video stabilization algorithm based on Harris Corners detection and Lucas-Kanade optical flow.

Video stabilization algorithm is chosen for illustrative purposes, as a well-known computer vision algorithm without too much complex details. Real production video stabilization pipelines would involve more advanced algorithms for feature point filtering to obtain more robust results. These are not covered here.

This sample enables you to write complete OpenVX* application. It introduces *advanced* features of OpenVX* required for a *real* application. If you need a detailed step-by-step introduction to the *basics* of OpenVX* development, see the *Auto Contrast sample* available in this SDK `(<SDK_ROOT>/samples/auto_contrast).`

The following OpenVX* topics are covered in the sample:

- Expressing user-defined logic as a part of a graph via **user nodes.**
- Using **vx_delay** to pass data from one graph iteration to another.
- **Virtual and non-virtual data** objects and their role in the graph.
- Using Khronos 1.1 Targets API for **heterogeneous compute**.
- Obtaining and interpreting OpenVX* **performance information.**

Additionally, the sample features basic **interoperability with OpenCV** through data sharing. OpenCV is used for reading the data from a video file. It is also demonstrated how to plug debug visualization with OpenCV right into data processed by the graph.

Brief Introduction to OpenVX*

OpenVX* is a new standard from Khronos*, offering a set of optimized primitives low-level image processing and computer visions primitives. OpenVX* is a specification across multiple vendors and platforms. Relatively high abstraction of OpenVX* notions of resources and execution enables hardware vendors to optimize implementation with a strong focus on a particular platform.

Computer vision algorithms are commonly expressed using dataflow graphs. OpenVX* also structures *nodes* (functions with *parameters*) and data dependencies in directed acyclic *graphs*. Any graph must be verified by the OpenVX* runtime before execution. The same graph can be executed multiple times, with different data inputs.

Video Stabilization Algorithm as an OpenVX* Graph

Figure 1 presents a simplified diagram for the OpenVX* graph implemented in the sample.

The core of the graph is the Detection and Tracking block, which consumes gray-scale image ("Grayscale") and produces a transform matrix ("Matrix"). The grayscale image is obtained by conversion of the input RGB image to NV12 image followed by extraction of the Y (luminance) channel.

The main output of the Detection and Tracking block is the transform matrix, which is consumed by WarpAffine nodes per each input channel (R, G, and B) to produce a final, stabilized image.

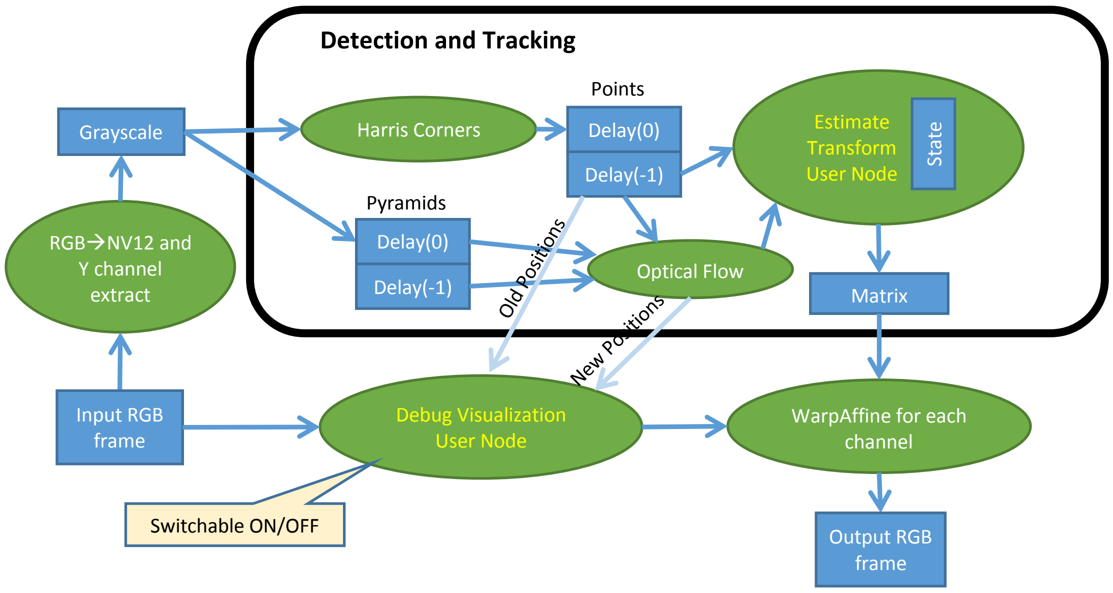


Figure 1: Top-level simplified structure of OpenVX* graph implemented in the sample. For the compactness, some connected nodes are grouped in blocks.

The Detection and Tracking block uses a combination of the Harris Corners detector and the Lucas-Kanade Optical Flow method to recognize global movements between two neighbor frames in a video stream. The sequence of the local transformations between neighbor frames is filtered via Kalman filter to obtain final stabilization transformation matrix for the current frame.

At each iteration, the Harris Corners node detects points that will be used at the next iteration. The Optical Flow node estimates new positions for points detected at the previous iteration.

The complete OpenVX* graph is presented in Figure 2. See `video_stabilization_openvx.cpp` where the following functions are implemented:

- **`video_stabilization_openvx`** function contains code for
    - creation of `vx_context` and `vx_graph` instances;
    - creation of all non-virtual data objects;
    - the main loop to process an input video file, frame-by-frame;
    - output results in pop-up GUI windows.
- **`buildVideoStabilizationGraph`** function contains code for
    - creation of all virtual data objects;
    - population of the graph with nodes to build the pipeline presented on Figure 2.

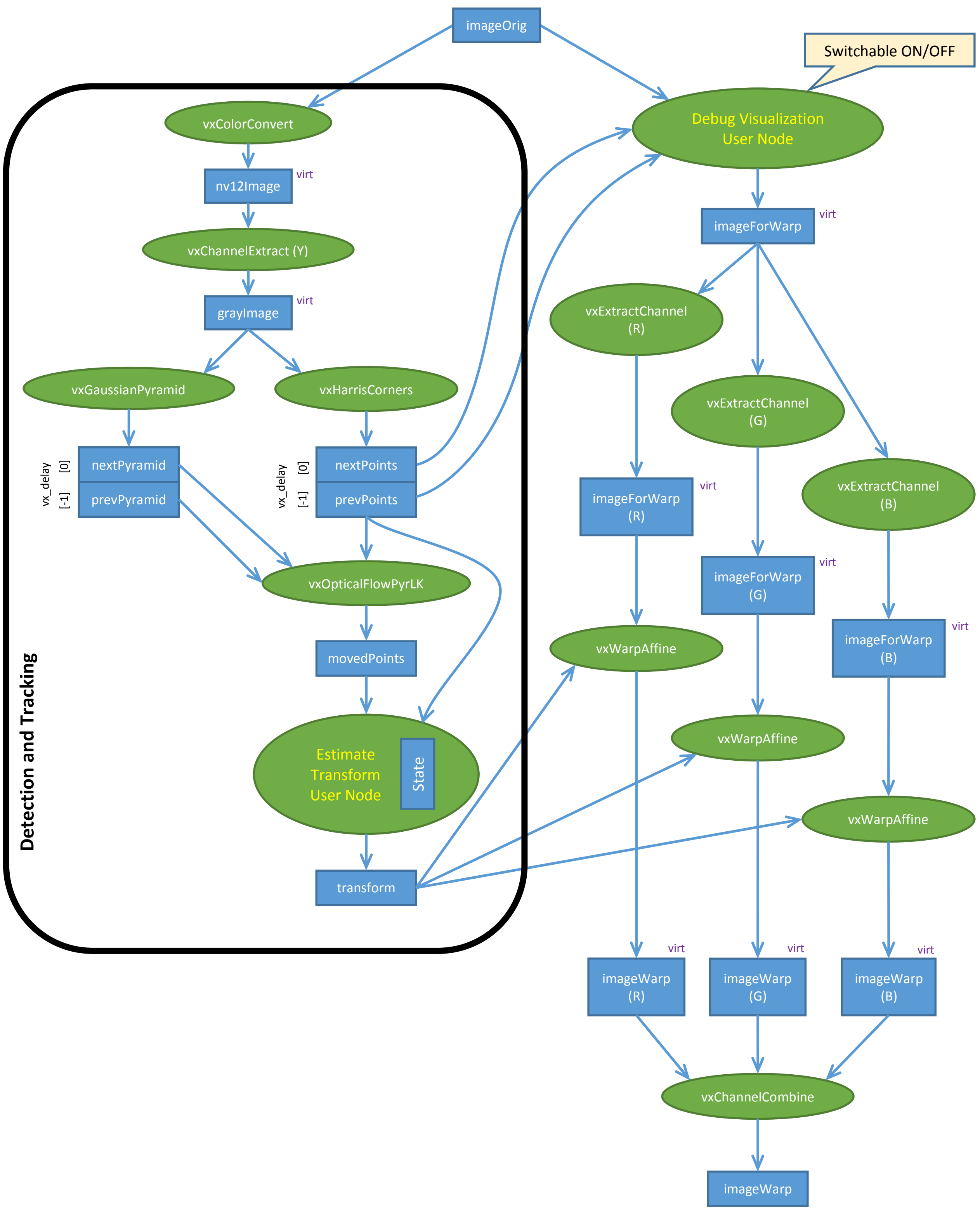


Figure 2: Complete OpenVX* graph implemented in the sample.
Legend: nodes are green, data objects are blue, "virt" indicate virtual objects.
Names for the data objects correspond to the names of variables in the source code.

Using vx_delay to Implement a Loop Dependency in the Graph

A delay is an opaque object that contains a manually-controlled list of objects that is stored in a ring buffer. Each element in the delay is accessed by an index and can be passed as a node parameter in a graph.

To create a `vx_delay` object, you need to define the number of elements, and an exemplar object that will be used as a template for each delay element:

```
vx_pyramid pyramidExemplar = vxCreatePyramid(context, …);

vx_delay pyramidRing = vxCreateDelay(context, (vx_reference)pyramidExemplar, 2);

vxReleasePyramid(&pyramidExemplar);  // delete it if it is not
                                     //  used anywhere else
```

The code above creates `pyramidRing` which is a delay object consisting of two pyramids with metadata (type, size etc.) taken from `pyramidExemplar`. The `pyramidExemplar` is a template object that is not needed after creation of the delay. The exemplar object itself is not linked with a delay, therefore it can be released immediately, if not used anywhere else in the application. Only metadata of an exemplar object is used when the delay is created.

Indices of a delay are in a range from `0` to `(-N + 1)`, where `N` is size of a delay. To access each of the elements, use the `vxGetReferenceFromDelay` function. It returns a generic `vx_reference`. You need to explicitly convert the `vx_reference` element to a delay element type before you can use it in other APIs (e.g. to be passed to a node).

```
(vx_pyramid)vxGetReferenceFromDelay(pyramidRing, -1)  // -1-th element

(vx_pyramid)vxGetReferenceFromDelay(pyramidRing, 0)  // 0-th element
```

The following diagram illustrates indices in a delay for 2 and N elements:

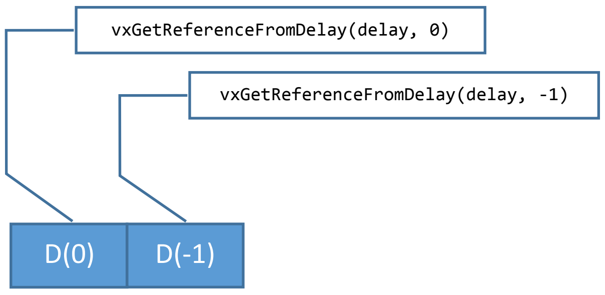

vxGetReferenceFromDelay(delay, 0)
vxGetReferenceFromDelay(delay, -1)
D(0)
D(-1)

Figure 3: A vx_delay object with two elements D(0) and D(-1).
The correspondence between references returned by vxGetReferenceFromDelay and target vx_delay element is shown.

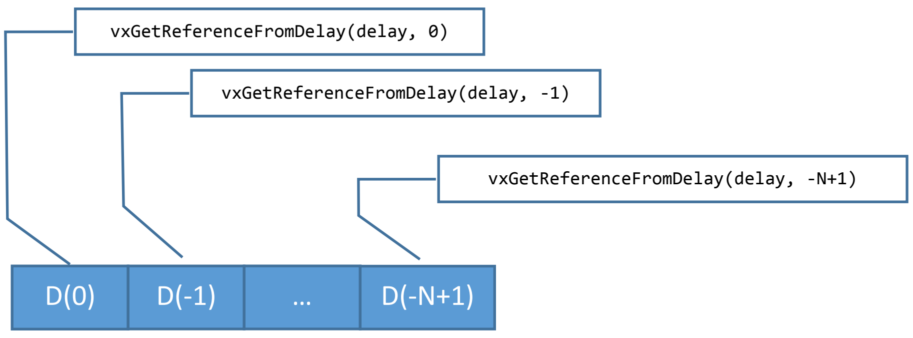

vxGetReferenceFromDelay(delay, 0)
vxGetReferenceFromDelay(delay, -1)
vxGetReferenceFromDelay(delay, -N+1)
D(0)
D(-1)
...
D(-N+1)

Figure 4: A vx_delay object with N elements D(0), D(-1), ..., D(-N+1).
The correspondence between references returned by vxGetReferenceFromDelay and target vx_delay element is shown.

The Delay is designed to explicitly address feedback between graph executions by doing aging for a delay. Aging is an operation when all elements in a ring are shifted by one:

```
vxAgeDelay(pyramidRing);
```

During this operation no new elements are added nor are elements deleted, only indices are changed. When a delay is aged, it automatically updates references to its elements in all graphs where they were used as parameters.

The next diagram illustrates how mapping of indices to elements is changed after `vxAgeDelay` for delays from the previous diagram. `vxGetReferenceFromDelay` returns different data elements D than before `vxAgeDelay`.

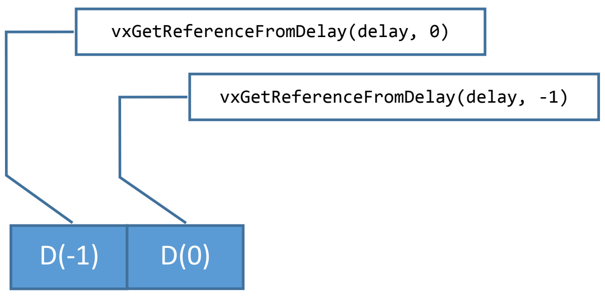

vxGetReferenceFromDelay(delay, 0)
vxGetReferenceFromDelay(delay, -1)
D(-1)
D(0)

Figure 5: A vx_delay object with two elements after vxAgeDelay is called.
The new correspondence between references returned by vxGetReferenceFromDelay and target vx_delay element is shown.

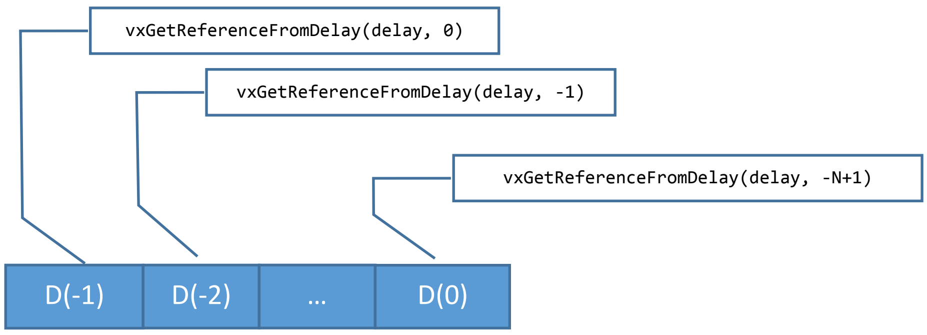

vxGetReferenceFromDelay(delay, 0)
vxGetReferenceFromDelay(delay, -1)
vxGetReferenceFromDelay(delay, -N+1)
D(-1)
D(-2)
...
D(0)

Figure 6: A vx_delay object with N elements D(0), D(-1), ..., D(-N+1) after vxAgeDelay is called. The new correspondence between references returned by vxGetReferenceFromDelay and target vx_delay element is shown.

Virtual Data Objects

An essential feature of OpenVX* is the ability to define intermediate data objects as *virtual*. This ability is applicable for `vx_image`, `vx_array` and `vx_pyramid`. Virtual data objects serve as connections between nodes inside the graph for passing data between nodes. The OpenVX* implementation will not necessarily keep a virtual object in memory or allocate only a smaller portion of the memory in comparison to the amount needed for non-virtual object with the same properties. Therefore, the content of a virtual data object cannot be accessed outside of the graph.

In the sample, all intermediate data are defined as virtual objects. To create a virtual object, the special form of create function is used. For example, in the sample code `grayImage` is created by

```
vx_image grayImage = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
```

Notice, no image size nor image data type are provided to this function. They are optional. These attributes will be deduced from the nodes in the graph by calling the sequence of output validators when the graph is verified by `vxVerifyGraph`. See the Input and Output Validators for User-Defined Kernels section for example of such deduction.

Optionally, you should specify some attributes for virtual objects. For example:

```
vx_image nv12Image = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_NV12);
```

Object `nv12Image` is created with explicitly defined `VX_DF_IMAGE_NV12`. It is needed because `nv12Image` is used as an output parameter for `vxColorConvertNode`. `vxColorConvert` node requires that the type of output is determined, because this type defines which conversion should be performed in the node. Image dimensions are not needed here because they will be deduced from input image dimensions. In essence, you are required to provide certain attributes for a virtual object, if those attributes are essential and cannot be deduced. Otherwise, the graph does not pass validation.

Graph input and output data entries cannot be virtual because they need to be accessed outside of the graph. In addition, the delay objects cannot be virtual because they keep data between different graph executions. Therefore, the following data in the sample are not virtual:

- Input image (`imageOrig`)
- Output image (`imageWarp`)
- `vx_delay` object for pyramids (`pyramidRing`)
- `vx_delay` object for feature points (`pointsRing`)

Summary on benefits of data virtual objects:

- They provide more opportunities for the run-time to produce efficient graph execution with lower memory bandwidth, resulting in improved performance and reduced power consumption.
- Easy to use and more generic than real data objects as many of the attributes are not required.

Extending OpenVX* with User Kernels

It is a common situation that standard OpenVX* kernels and kernels provided by a vendor through extensions are not enough to implement a particular computer vision pipeline. In this case, you might need to write additional kernel to process images and combine the kernel with the rest of the pipeline implemented as an OpenVX* graph.

OpenVX* supports the concept of client-defined functions that shall be executed as nodes from inside the graph.

There are two user kernels in the sample:

- **EstimateTransform Node** takes two sets of feature points as `vx_array` objects: one set for the previous frame and one for the current frame. Points are filtered and global transformation matrix is reconstructed with the help of OpenCV library, using `estimateRigidTransform` and `KalmanFilter`. This node encapsulates a state object that holds information between executions of the node to keep the `KalmanFilter` state. The state is stored as a node attribute `VX_NODE_LOCAL_DATA_PTR`.

- **DrawDebugVisualization Node** draws feature points and their movements on the intermediate image. This node is optional (turned on by using command line knob `'--visualization'`). This illustrates how you can implement debug visualization in OpenVX*. Actual drawing on `vx_image` is implemented with OpenCV library.

To define a user kernel, you need to define five call functions:

- **Kernel function** where target image processing logic is defined. Refer to `EstimateTransformKernel` and `DebugVisualizationKernel` defined in `video_stabilization_user_nodes_module.cpp.`

- **Input/output validator** that is used by OpenVX* runtime to validate input and output parameters for a node in a graph as a part of `vxValidateGraph` function call. Refer to `EstimateTransformValidator` and `DebugVisualizationValidator` defined in `video_stabilization_user_nodes_module.cpp.`

- **Initializing function** that is called each time a node instance is created in a graph. It may pre-allocate memory for a kernel execution to avoid doing it each time the kernel is invoked. For example, the `EstimateTransform` node initializes an internal state that contains the history of video frame movements, and updates each time the node kernel is invoked. Refer to `EstimateTransformInitialize` function defined in `video_stabilization_user_nodes_module.cpp.`

- **Deinitialization** that is called when a node instance is destroyed from a graph. It frees resources that are allocated in Initialization function. Refer to `EstimateTransformDeinitialize` function defined in `video_stabilization_user_nodes_module.cpp.`

All the functions above are compiled in a separate `video_stabilization_user_nodes_module.so` to be loaded in the main application by `vxLoadKernels` function. Compilation in a separate module is a regular practice for libraries of user nodes that can be reusable in other applications. Vision Algorithm Designer consumes dynamically loadable libraries and enables you to construct a graph with user-defined kernels.

To be correctly loaded by `vxLoadKernels`, the module with user kernels should define `vxPublishKernels`. OpenVX* runtime looks for this specific function while the module is loaded, and calls it. The function should register all user nodes, by calling `vxAddUserKernel` with passing all four callbacks described above and some other parameters. Refer to `vxPublishKernels` function defined in `video_stabilization_user_nodes_module.so` for details.

Besides the module, you should provide `vx` and `vxu` functions to be used during graph construction, or to call in the immediate mode correspondingly. Having these functions, creation of the user nodes are very similar to creation of any standard nodes. Declaration of `vx` an `vxu` functions can be found in the `video_stabilization_user_nodes_lib.hpp` file and they are defined in the `video_stabilization_user_nodes_lib.cpp` file.

After defining `vx`-function for EstimateTransform node, you can insert the `vxEstimateTransformNode` user node into a graph, with this code:

```
vxEstimateTransformNode(
    graph,          // graph where to insert an instance of the node
    prevPoints,     // one of the node parameters
    movedPoints,    // one of the node parameters
    frameRect,      // one of the node parameters
    transform       // one of the node parameters
);
```

A `vxu`-function is used to call a user kernel outside of a graph. It is called *intermediate* mode. You can use the `vxu`-function when there is no need to construct a graph for a rare kernel execution, or for debugging purposes. In the sample code this function is provided but it is never used, because the sample uses *graph* mode.

NOTE: The intermediate mode supposes that the implementation of `vxu`-function will build an OpenVX* graph behind the scenes. This graph contains a single node with a user-kernel and `vxVerifyGraph` and `vxProcessGraph` functions are called. In most cases it means that extra time is spent on graph verification with calling input/output validators. See `vxuEstimateTransform` or `vxuDebugVisualization` functions source code for more details about how `vxu`-functions are usually constructed.

Input and Output Validator for User-Defined Kernels

Input and output validator is special function defined for each user-kernel. They are called whenever validation is needed for user-node parameters, particularly when `vxVerifyGraph` function is called for a graph where the user-nodes are instantiated.

A validator accepts a node reference, input parameters, number of parameters and output parameters meta format. The input validation code should do necessary checks for a parameter attributes to make sure that the data object is a valid to be used as a particular input parameter for this user-kernel. In contrast with the input validation code, which queries the attributes for a particular input parameter and verify that they are valid, the output validation code sets requirements for the attributes without querying them. The requirements are set by calling the `vxSetMetaFormatAttribute` function. After exiting a validator, the OpenVX* runtime will use the attributes requirements:

- To *match* them to actual attributes for a particular parameter if the parameter is a real data object (non-virtual) or
- To *define* the attributes if a particular parameter is a virtual data object with no defined attributes

This mechanism implies that requirements set by a validator shape the actual attributes for virtual data objects used in a graph. This is an important part of data object attributes deduction during the graph validation process. This process enables you to not define all virtual object attributes.

Here is a fragment of `DebugVisualizationValidator` for one of the nodes from the sample:

```
vx_status VX_CALLBACK DebugVisualizationValidator(vx_node node, const vx_reference
parameters[], vx_uint32 num, vx_meta_format metas[])
{
    vx_status status = VX_ERROR_INVALID_PARAMETERS;
    if(num!=4) //check that number of parameters is correct
    {
        return status;
    }

    vx_df_image imageFormat;
    if(vxQueryImage((vx_image)parameters[DEBUG_VISUALIZATION_PARAM_INPUT],
VX_IMAGE_FORMAT, &imageFormat, sizeof(imageFormat)) == VX_SUCCESS &&
        imageFormat == VX_DF_IMAGE_RGB
        )
    {
        status = VX_SUCCESS;
    }
     // verify other input parameters here
     …
     // set meta format for output parameters
     vxSetMetaFormatAttribute(metas[DEBUG_VISUALIZATION_PARAM_OUTPUT],
VX_IMAGE_FORMAT, &imageFormat, sizeof(imageFormat));
     …
        status = VX_SUCCESS;
    }

    return status;
}
```

Passing a Struct as a User Kernel Parameter

The OpenVX* standard 1.0.1 specification does not provide an easy way to pass a structure object as a node parameter. It is possible for simple scalar parameters of type int or float, but not for the structures like `vx_rectangle_t` or `vx_keypoint_t`, or user-defined structures.

In the sample code, `vx_rectangle_t` object should be passed to the EstimateTransform user node to perform filtering out of feature points that are tracked outside of the valid frame rectangle. In this case an instance of `vx_array` is created with element type set to `VX_TYPE_RECTANGLE`. A single element is added:

```
    vx_rectangle_t frameRectVal;
    frameRectVal.start_x = 0;
    frameRectVal.start_y = 0;
    frameRectVal.end_x = inputWidth;
    frameRectVal.end_y = inputHeight;
    vx_array frameRect = vxCreateArray(context, VX_TYPE_RECTANGLE, 1);
    vxAddArrayItems(frameRect, 1, &frameRectVal, sizeof(frameRectVal));
```

Then the `frameRect` `vx_array` object is passed to a user-kernel where the `vx_rectangle_t` object is retrieved:

```
    vx_rectangle_t frameRectValue;
    vx_size stride = sizeof(vx_rectangle_t);
    vx_rectangle_t* pRectValue = 0;
    vx_map_id map_id;
    vxMapArrayRange(frameRect, 0, 1, &map_id, &stride, (void**)&pRectValue,
VX_READ_ONLY, VX_MEMORY_TYPE_HOST, 0);
    frameRectValue = *pRectValue;
    vxUnmapArrayRange(frameRect, map_id);
```

The EstimateTransform validator code verifies that this is indeed a single-element array of `VX_TYPE_RECTANGLE` type:

```
vx_status VX_CALLBACK EstimateTransformValidator(vx_node node, const vx_reference
parameters[], vx_uint32 num, vx_meta_format metas[])
{
    . . .
    else if (index == ESTIMATE_TRANSFORM_PARAM_RECT)
    {

        vx_enum itemType;
        vx_size numItems;

        if (
            vxQueryArray((vx_array)parameters[ESTIMATE_TRANSFORM_PARAM_RECT],
VX_ARRAY_ATTRIBUTE_ITEMTYPE,
                         &itemType, sizeof(itemType)) == VX_SUCCESS &&
            vxQueryArray((vx_array)parameters[ESTIMATE_TRANSFORM_PARAM_RECT],
VX_ARRAY_ATTRIBUTE_NUMITEMS,
                         &numItems, sizeof(numItems)) == VX_SUCCESS &&
            itemType == VX_TYPE_RECTANGLE &&
            numItems == 1
        )
        {
            status = VX_SUCCESS;
        }
        else
        {
            status = VX_ERROR_INVALID_VALUE;
        }
    . . .
}
```

Note, that it is possible to re-implement EstimateTransform node to pass an input image instead of an explicit frame rectangle. In this case, the kernel code will extract the valid rectangle from the image. The sample code uses explicit passing of `vx_rectangle_t` for illustrative purposes.

Using Targets API for Heterogeneous Computing

Intel® Computer Vision SDK provides the API to schedule parts of an OpenVX* graph to different compute units (e.g. CPU, GPU or IPU). In the context of OpenVX*, such a compute unit is named *target*. The set of available targets depends on a platform used to run an application.

A developer might want to schedule nodes to a particular target to improve performance or power consumption. For example, for certain parameters (e.g. input image size) some nodes are doing better on the GPU than on CPU and vice versa. Then for a graph that includes such nodes, the developer can experiment with running different parts of the pipeline on the GPU while measuring performance and/or power consumption metrics.

Intel Computer Vision SDK provides OpenVX* 1.1 implementation that has Targets API for assigning selected nodes in an OpenVX* graph to a particular target, overriding the default run-time choice. This API enables heterogeneous usages of Intel platforms that is a way to better hardware utilization.

For better understanding of Targets API, see OpenVX* Heterogeneous Basic sample, available in this SDK `(<SDK_ROOT>/samples/hetero_basic).`

NOTE: In contrast to Heterogeneous Basic sample, which teach you how to use the Targets API in source code, the Video Stabilization sample provides a handy tool that allow switching targets for nodes through a configuration file. It enables developers to easily experiment with different mappings of nodes to targets by just supplying different configuration files. It doesn't require code re-compilation.

NOTE: The functionality described in this section is not a part of OpenVX* standard, it is just an example code that illustrates how dynamic heterogeneous schedules can be organized in an application.

Let us describe the format of the configuration file. There are several examples of such files in the directory of the sample. To specify which configuration file to pick up, use the `--hetero-config` command line parameter. Here is the content of one of the configuration files, `hetero.config.gpu-harris-3-warps.txt`, which brings one of the best performance results on the experimental system that were used for performance evaluation:

```
vxColorConvertNode          intel.cpu
vxChannelExtractNode        intel.cpu
vxHarrisCornersNode         intel.gpu
vxGaussianPyramidNode       intel.cpu
vxOpticalFlowPyrLKNode      intel.cpu
vxChannelExtractNode(R)     intel.cpu
vxChannelExtractNode(G)     intel.cpu
vxChannelExtractNode(B)     intel.cpu
vxWarpAffineNode(R)         intel.gpu
vxWarpAffineNode(G)         intel.gpu
vxWarpAffineNode(B)         intel.gpu
vxChannelCombineNode(warp)  intel.cpu
```

Each row in the file is a mapping for a single node. It has two parts: a name of a node and a name of a target that will be used for this node. Name of the node and the target name are separated with one or multiple spaces. The name of the node is defined in the source code of the sample and it is user-defined. The name shouldn't contain spaces. Here is a snippet from `video_stabilization_openvx.cpp` file where name `vxColorConvertNode` is assigned to the specific node in the graph:

```
outNodes.push_back(
    vxColorConvertNode(graph, imageOrig, nv12Image)
);
CHECK_VX_STATUS(vxGetStatus((vx_reference)outNodes.back()));
if(vx_target target = heteroConfig.getTargetByNodeName("vxColorConvertNode"))
{
    CHECK_VX_STATUS(vxSetNodeTarget(outNodes.back(), target, 0));
}
```

The name of the target should be from the list of targets available on the platform. The list of supported targets is pre-defined, but each availability of each particular target depends on platform used. The sample prints the list of the supported targets in the beginning as a part of its output for convenience:

```
. . .
[ INFO ]     Target[0] name: intel.cpu
[ INFO ]     Target[1] name: intel.gpu
[ INFO ]     Target[3] name: intel.ipu
. . .
```

See the description of supported targets in CV SDK User Guide.

NOTE: If a node name is not listed in the configuration file or the configuration file maps a node to a target that is not available on the platform, this node will not be assigned to any target. The latter means that the default target chosen by the OpenVX* run-time will be used in this case.

Debug Visualization as a User OpenVX* Kernel

Debug visualization (like the one implemented in DebugVisualization node) is frequently used when developing and debugging a visual pipeline. But it is rarely used in the final products. So why do we need to implement it as an OpenVX* user node and spend extra efforts to provide validators, `vx` and `vxu`-function etc? There are several reasons that are applicable for most programs, not specifically for this sample code:

- Avoid breaking a single graph into several pieces. If DebugVisualization is not called as a user-node, the graph should be divided into two pieces, with `vxWarpAffine` and final `vxChannelCombine` nodes in the second graph. It is not convenient because diverges code from its product form significantly: some data objects become not local for one graph and should be defined as non-virtuals, two graphs should be verified and processed separately etc. The code start to be less maintainable.
- Enable tools that support visual graph construction. DebugVisualization user node can be loaded in Vision Algorithm Designer and the client can construct a graph in the convenient environment. It may be reused easily in other applications.

Introduction to OpenCV Interoperability

The sample uses OpenCV for:

- Reading video frame by frame from a file with the help of `cv::VideoCapture`.
- Implementing core video stabilization function by using `cv::estimateRigidTransform` and `cv::KalmanFilter`.
- Visualizing input and output frames by creating pop-up GUI windows and drawing debug information.
- Reference implementation of the complete video stabilization pipeline.

OpenCV can be used together with OpenVX* code by careful sharing of data objects. In this sample the memory is shared between vx_image and `cv::Mat`. It is achieved in two steps:

1. Create `vx_image` by `vxCreateImageFromHandle` function that accepts a pointer to a region of native memory. This pointer can be retrieved from `cv::Mat::data` field.
2. Call `vxMapImagePatch/vxUnmapImagePatch` to handle ownership and updates between `cv::Mat` and `vx_image` objects.

In the sample, the input `frame` object is defined as `cv::Mat`, and then the `vx_image` instance is created based on the memory allocated in `frame` object. Notice that before calling `vxCreateImageFromHandle`, you need to initialize an object of `vx_imagepatch_addressing_t` type to describe `frame` object memory format:

```
    cv::Mat frame;

    // . . . read frame from video file to determine image dimensions

    vx_imagepatch_addressing_t frameFormat;
    frameFormat.dim_x = inputWidth;
    frameFormat.dim_y = inputHeight;
    frameFormat.stride_x = 3;   // for three channels: R, G, and B
    frameFormat.stride_y = frame.step;  // number of bytes each matrix row occupies
    frameFormat.scale_x = VX_SCALE_UNITY;
    frameFormat.scale_y = VX_SCALE_UNITY;
    frameFormat.step_x = 1;
    frameFormat.step_y = 1;
```

Then call `vxCreateImageFromHandle`:

```
    vx_image imageOrig = vxCreateImageFromHandle(
        context,
        VX_DF_IMAGE_RGB,
        &frameFormat,
        (void**)&frame.data,      // a pointer to data from cv::Mat object
        VX_MEMORY_TYPE_HOST
    );
```

`imageOrig` may share the same memory that `cv::Mat::data` pointer refers to. But the OpenVX* implementation is not required to do that. Since the `vx_image` object is opaque, after you call `vxCreateImageFromHandle`, do not perform any manipulations with the original `cv::Mat frame` object. Instead you can used it in the OpenVX* context by running a graph that reads/write data in that image object.

When you need to access the memory on the OpenCV side, for reading the next frame, map `imageOrig` by calling of `vxMapImagePatch`:

```
vx_imagepatch_addressing_t outPatchAddr;
void* mappedArea = 0;
vx_map_id map_id;
vxMapImagePatch(
    imageOrig,      // vx_image to map
    &frameRect,     // rectangle in image pixel coordinates that should be mapped
    0,              // plane index, there is only one plane in our image: 0-th
    &map_id         // mapping id
    &outPatchAddr,  // output addressing structure like
```

```
                    // used in vxCreateImageFromHandle
    &mappedArea,    // a pointer to a pointer to mapped memory
    VX_WRITE_ONLY,  // which access type is needed:
                    //           read only, write only or read/write
    VX_MEMORY_TYPE_HOST, // type of the memory
    0                    // Map/Unmap operation enumeration flag (reserved)
);
```

As `imageOrig` was created with `vxCreateImageFromHandle`, it is guaranteed that `vxMapImagePatch` will return a `mappedArea` pointer that is equal to the `frame.data` pointer that was used when `imageOrig` was created. So after this call, the `cv::Mat` object `frame` can be accessed and modified:

```
cap >> frame;
```

When the frame update is complete, you return ownership back to OpenVX* space using `vxUnmapImagePatch`:

```
vxUnmapImagePatch(imageOrig, map_id);
```

After this call, the ownership for the memory underneath `imageOrig` is transferred back to OpenVX*, and `imageOrig` can be processed in the `vxProcessGraph`.

NOTE: See the appropriate section in the Developer Guide for more information on the OpenCV* provided with the Intel® Computer Vision SDK. This includes details on environment setup and build instructions for applications using OpenCV. The Developer Guide also explains details on the samples using CMake* to automatically detect the installed Intel OpenCV package, and setup the paths for include directories, inputs for the linker and so on. It also describes alternative build strategies.

OpenCV Reference Implementation

Besides OpenVX* implementation of the video stabilization pipeline, there is a complete OpenCV code that implements almost the same algorithm. It is defined in the `video_stabilization_opencv.cpp` source file. The difference between OpenVX* and OpenCV versions is not critical for functional and performance comparison of these two versions.

Switch between the versions by using `--impl openvx` or `--impl opencv` command line arguments.

Building the Sample

See the root `README` file for all samples and another `README` file located in `video_stabilization` sample directory for complete instructions about how to build the sample.

Running the Sample and Understanding the Output

The sample is a command line application. It can create GUI windows with visualization of input/output video frames and the debug visualization. The behavior is controlled by providing command line parameters. To get the complete list of command line parameters, run:

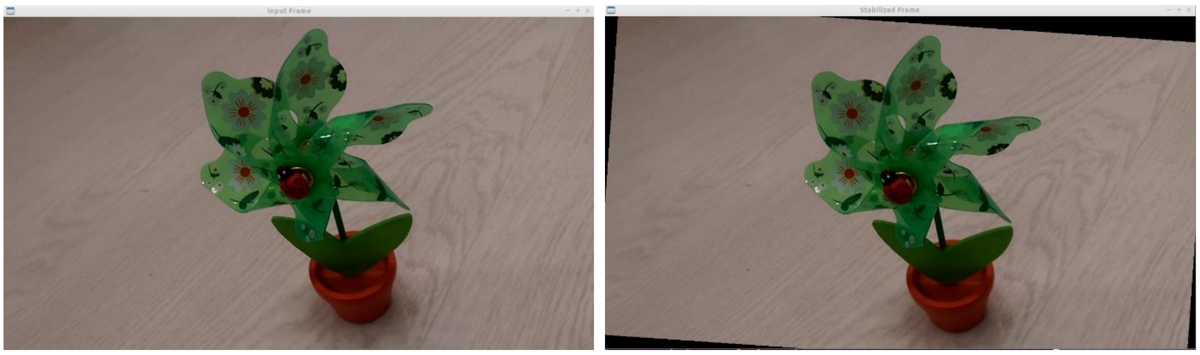
```
$ ./video_stabilization --help
```

The following section provides a few examples how to run the sample in different configurations.

By the default, the sample reads the video file `toy_flower.mp4` that is located in the current directory. So calling sample without parameters:

```
$ ./video_stabilization
```

Will open `toy_flower.mp4` and create two pop-up windows playing input and stabilized output frame streams:



You can exit from the sample by pressing Esc when one of the GUI windows is in focus.

To run the sample in pure command line mode without pop-up windows (for example when using a remote terminal) the visualization mode can be disabled:

```
$ ./video_stabilization --visualization 0
```

When the sample finishes execution, it prints performance statistics out to the console in milliseconds (in the example below --hetero-config command line option is also used to enable heterogeneous compute for OpenVX graph; consider reviewing of Using Targets API for Heterogeneous Computing section in this document for more details):

```
$ ./video_stabilization --visualization 0 --hetero-config hetero.config.gpu-harris-3-warps.txt

Video frame size: 1280x720
[ INFO ] Number of supported targets: 3
[ INFO ]     Target[0] name: intel.cpu
[ INFO ]     Target[1] name: intel.gpu
[ INFO ]     Target[2] name: intel.ipu
[ INFO ] Node vxColorConvertNode is assigned to intel.cpu
[ INFO ] Node vxChannelExtractNode is assigned to intel.cpu
[ INFO ] Node vxHarrisCornersNode is assigned to intel.gpu
[ INFO ] Node vxGaussianPyramidNode is assigned to intel.cpu
[ INFO ] Node vxOpticalFlowPyrLKNode is assigned to intel.cpu
[ INFO ] Node vxEstimateTransformNode not assigned to any specific target (missed in
config)
[ INFO ] Node vxChannelExtractNode(R) is assigned to intel.cpu
[ INFO ] Node vxChannelExtractNode(G) is assigned to intel.cpu
[ INFO ] Node vxChannelExtractNode(B) is assigned to intel.cpu
[ INFO ] Node vxWarpAffineNode(R) is assigned to intel.gpu
[ INFO ] Node vxWarpAffineNode(G) is assigned to intel.gpu
[ INFO ] Node vxWarpAffineNode(B) is assigned to intel.gpu
[ INFO ] Node vxChannelCombineNode(warp) is assigned to intel.cpu
1 warning generated.
Reached end of video file
Processed 166 iterations
0.42 ms by estimateTransform_lib averaged by 166 samples
0.44 ms by EstimateTransformKernel averaged by 166 samples
Sample was finished successfully
18.93 ms by ProcessFrame averaged by 166 samples
5.98 ms by ReadFrame averaged by 167 samples
24.72 ms by Frame averaged by 166 samples
3676.94 ms by vxVerifyGraph averaged by 1 samples
18.77 ms by vxProcessGraph averaged by 166 samples
```

In case when OpenVX* implementation is chosen, there is a big section in the output that shows how nodes were assigned to targets. **hetero.config.gpu-harris-3-warps.txt** file was used for heterogeneous configuration to produce the output above, then a part of the nodes was assigned to **intel.cpu** target and another part is assigned to **intel.gpu** target according to the mapping defined in **hetero.config.gpu-last-part.txt**. Also in the beginning there is a list of the names of all available targets on the platform.

The main metrics that are applicable for both the OpenVX* and OpenCV implementations are:

- **ProcessFrame** captures the average time of frame processing without frame reading time and frame visualization time. For OpenVX* it includes `vxProcessGraph` and `vxAgeDelay` calls.
- **ReadFrame** captures the average time of next frame preparation. In case of OpenCV it is just reading the next frame from `VideoCapture`. In case of OpenVX* it also includes the time spent in `vxMapImagePatch`/`vxUnmapImagePatch`.
- **Frame** is average of a total frame time that includes everything.

There are two specific metrics for the OpenVX* implementation:

- **vxVerifyGraph** is the average time spent in `vxVerifyGraph` function call that is called once in the beginning of the sample.
- **vxProcessGraph** is the pure `vxProcessGraph` API call average time.

Debug Visualization

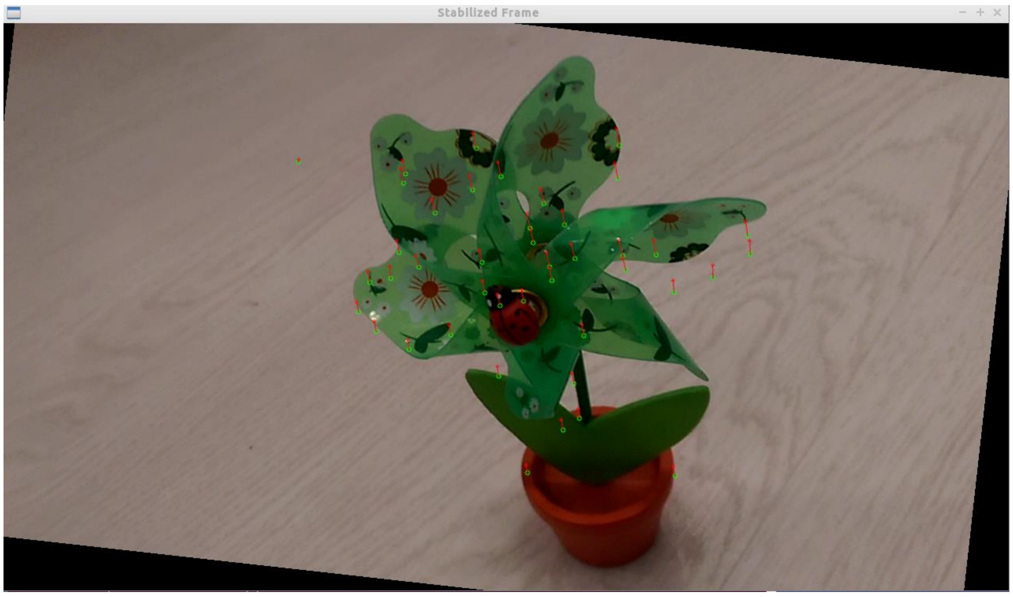
You can turn on debug visualization by setting `--visualization` knob to value `2`:

```
$ ./video_stabilization --visualization 2
```

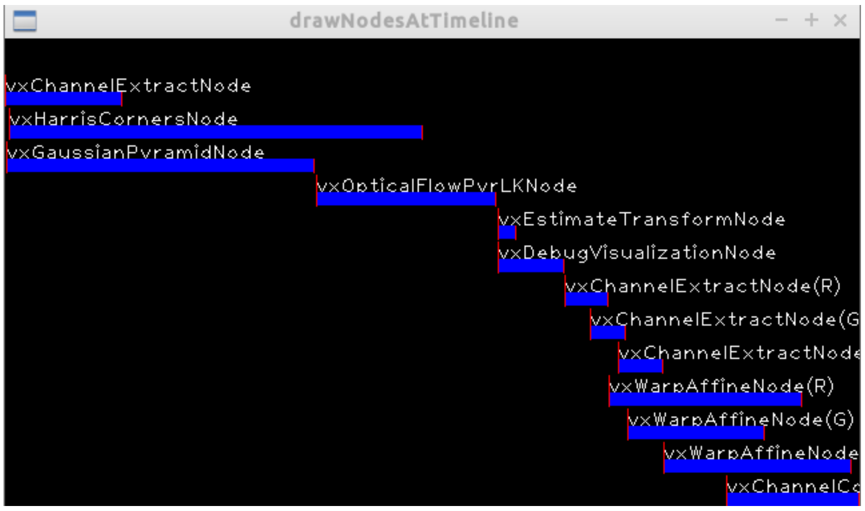
It adds the following to the sample output:

- Draws feature points from the previous frame, their estimated positions on the current frame and motion vectors. It is enabled by adding DebugVisualization user-defined node in the graph.
- Creates an additional pop-up GUI window that visualizes OpenVX* graph execution performance numbers obtained by `vxQueryNode` with `VX_NODE_ATTRIBUTE_PERFORMANCE` parameter.

The feature points and motion vectors are drawn on the stabilized output image. You can see an example of the debug output on the following screen shot:



The window with performance numbers shows time spent in each node in the graph on the timeline. The absolute numbers are not shown, only relative time is presented to compare nodes and observe their place in the schedule of the graph execution:



This window is updated after each frame is processed with `vxProcessGraph`. The complete elapsed time of graph processing is normalized by the window width. It is approximately equivalent to the time reported by **vxProcessGraph** metric in the command line output described above. The left edge of the window is where the graph starts execution, the right edge is where the graph stops execution.

The performance numbers are queried by calling of OpenVX* function `vxQueryNode`:

```
vx_perf_t perf;
vxQueryNode(node, VX_NODE_ATTRIBUTE_PERFORMANCE, &perf, sizeof(perf));
```

The performance information is stored in fields of `perf` structure. There are two fields that are used in building the chart in `drawNodesAtTimeline` window:

- **vx_peft_t::beg** – the first moment when the specified node starts execution during the last `vxProcessGraph` call;
- **vx_perf_t::end** – the latest moment when the specified node ends the execution during the last `vxProcessGraph` call.

The intervals [`perf.beg, perf.end`] are presented as blue bars in `drawNodesAtTimeline` window for all nodes in the graph.

The intervals for different nodes may overlap because of two main reasons (one or both):

- Nodes that are not dependent on each other can be executed in parallel (on different CPU threads). Example: `vxEstimateTransformNode` and `vxDebugVisualizationNode`.
- Nodes process input images by tiles, and the tiles processing by several different nodes are interleaved. Example: `vxChannelExtractNode` node and `vxHarrisCornersNode`.

For more details please refer to the implementation of `IntelVXSample::drawNodesAtTimeline` function in samples common infrastructure file `samples/common/src/perfprof.cpp`.

Tuning for Performance Using Heterogeneous Compute

By carefully offloading part of the nodes in the OpenVX* graph to GPU, it is sometimes possible to achieve better performance results than in pure CPU and pure GPU compute. There are two main reasons for the speedup of CPU/GPU mix of nodes:

1. Some nodes in the graph can work faster on GPU than on CPU and vice versa. The decision which device should run the kernel is made per each node. Even two different instances of the same kernel in the graph may be assigned to different devices to achieve better performance for each of them.
2. If the graph has nodes that can work independently, they can be assigned to different devices and work in parallel.

Let us try to find the best combination of nodes assigned to CPU and GPU that provides the highest performance numbers for video stabilization sample graph. The main metric considered here is vxProcessGraph, which should be minimized.

NOTE: Performance results presented here are highly dependent on platform used. The results are for illustrative purposes and gathered on pre-production systems. Additional tuning should be performed for each particular application, platform and environment to achieve the best results.

To form the performance baseline, let us measure pure CPU and GPU cases. CPU-only mode on a pre-release Apollo Lake system:

```
       ./video_stabilization --visualization 0 --hetero-config hetero.config.cpu-
all.txt

       Video frame size: 1280x720
       [ INFO ] Number of supported targets: 3
       [ INFO ]     Target[0] name: intel.cpu
       [ INFO ]     Target[1] name: intel.gpu
       [ INFO ]     Target[2] name: intel.ipu
       [ INFO ] Node vxColorConvertNode is assigned to intel.cpu
       [ INFO ] Node vxChannelExtractNode is assigned to intel.cpu
       [ INFO ] Node vxHarrisCornersNode is assigned to intel.cpu
       [ INFO ] Node vxGaussianPyramidNode is assigned to intel.cpu
       [ INFO ] Node vxOpticalFlowPyrLKNode is assigned to intel.cpu
       [ INFO ] Node vxEstimateTransformNode not assigned to any specific target
(missed in config)
       [ INFO ] Node vxChannelExtractNode(R) is assigned to intel.cpu
```

```
      [ INFO ] Node vxChannelExtractNode(G) is assigned to intel.cpu
      [ INFO ] Node vxChannelExtractNode(B) is assigned to intel.cpu
      [ INFO ] Node vxWarpAffineNode(R) is assigned to intel.cpu
      [ INFO ] Node vxWarpAffineNode(G) is assigned to intel.cpu
      [ INFO ] Node vxWarpAffineNode(B) is assigned to intel.cpu
      [ INFO ] Node vxChannelCombineNode(warp) is assigned to intel.cpu
      Reached end of video file
      Processed 166 iterations
      0.34 ms by estimateTransform_lib averaged by 166 samples
      0.35 ms by EstimateTransformKernel averaged by 166 samples
      Sample was finished successfully
      30.56 ms by ProcessFrame averaged by 166 samples
      5.05 ms by ReadFrame averaged by 167 samples
      35.42 ms by Frame averaged by 166 samples
      25.05 ms by vxVerifyGraph averaged by 1 samples
      30.40 ms by vxProcessGraph averaged by 166 samples
```

And GPU mode, where all standard nodes except user node is offloaded to GPU (the same system):

```
      ./video_stabilization --visualization 0 --hetero-config hetero.config.gpu-
all.txt

      Video frame size: 1280x720
      [ INFO ] Number of supported targets: 3
      [ INFO ]     Target[0] name: intel.cpu
      [ INFO ]     Target[1] name: intel.gpu
      [ INFO ]     Target[2] name: intel.ipu
      [ INFO ] Node vxColorConvertNode is assigned to intel.gpu
      [ INFO ] Node vxChannelExtractNode is assigned to intel.gpu
      [ INFO ] Node vxHarrisCornersNode is assigned to intel.gpu
      [ INFO ] Node vxGaussianPyramidNode is assigned to intel.gpu
      [ INFO ] Node vxOpticalFlowPyrLKNode is assigned to intel.gpu
      [ INFO ] Node vxEstimateTransformNode not assigned to any specific target
(missed in config)
      [ INFO ] Node vxChannelExtractNode(R) is assigned to intel.gpu
      [ INFO ] Node vxChannelExtractNode(G) is assigned to intel.gpu
      [ INFO ] Node vxChannelExtractNode(B) is assigned to intel.gpu
      [ INFO ] Node vxWarpAffineNode(R) is assigned to intel.gpu
      [ INFO ] Node vxWarpAffineNode(G) is assigned to intel.gpu
      [ INFO ] Node vxWarpAffineNode(B) is assigned to intel.gpu
      [ INFO ] Node vxChannelCombineNode(warp) is assigned to intel.gpu
      1 warning generated.
      Reached end of video file
      Processed 166 iterations
      0.34 ms by estimateTransform_lib averaged by 166 samples
      0.35 ms by EstimateTransformKernel averaged by 166 samples
      Sample was finished successfully
      36.03 ms by ProcessFrame averaged by 166 samples
      6.05 ms by ReadFrame averaged by 167 samples
      41.89 ms by Frame averaged by 166 samples
      9128.67 ms by vxVerifyGraph averaged by 1 samples
      35.87 ms by vxProcessGraph averaged by 166 samples
```

So the best vxProcessGraph time is 30.4 ms so far, and it is achieved with pure CPU execution of the graph.

There are two parts in the graph that have inter-node parallelism:

1. Harris Corners node output is not used in the same vxProcessGraph invocation. It will be consumed in the next call of vxProcessGraph only. So this node can work independently with other nodes in the graph except nodes that produce input to Harris Corners. Let's try to offload Harris Corners node to GPU.
2. Warp Affine nodes are instantiated for three channels and they are not dependent on each other. Part of these nodes can be offloaded to GPU.

But the specific heterogeneous configuration that gives the best performance results varies depending on the platform:

1. Using an experimental Apollo Lake system, it turns out that Warp Affine kernel is too slow on CPU in comparison to GPU. In this case all 3 Warp Affine nodes is offloaded to GPU. This configuration is in `hetero.config.gpu-harris-3-warps.txt.`
2. Using an experimental Intel® microarchitecture code name Skylake based system, it turns out that Warp Affine kernel has similar performance on CPU and GPU and offloading one of 3 nodes to GPU gives the best results. This configuration is in `hetero.config.gpu-harris-warp.txt` file.

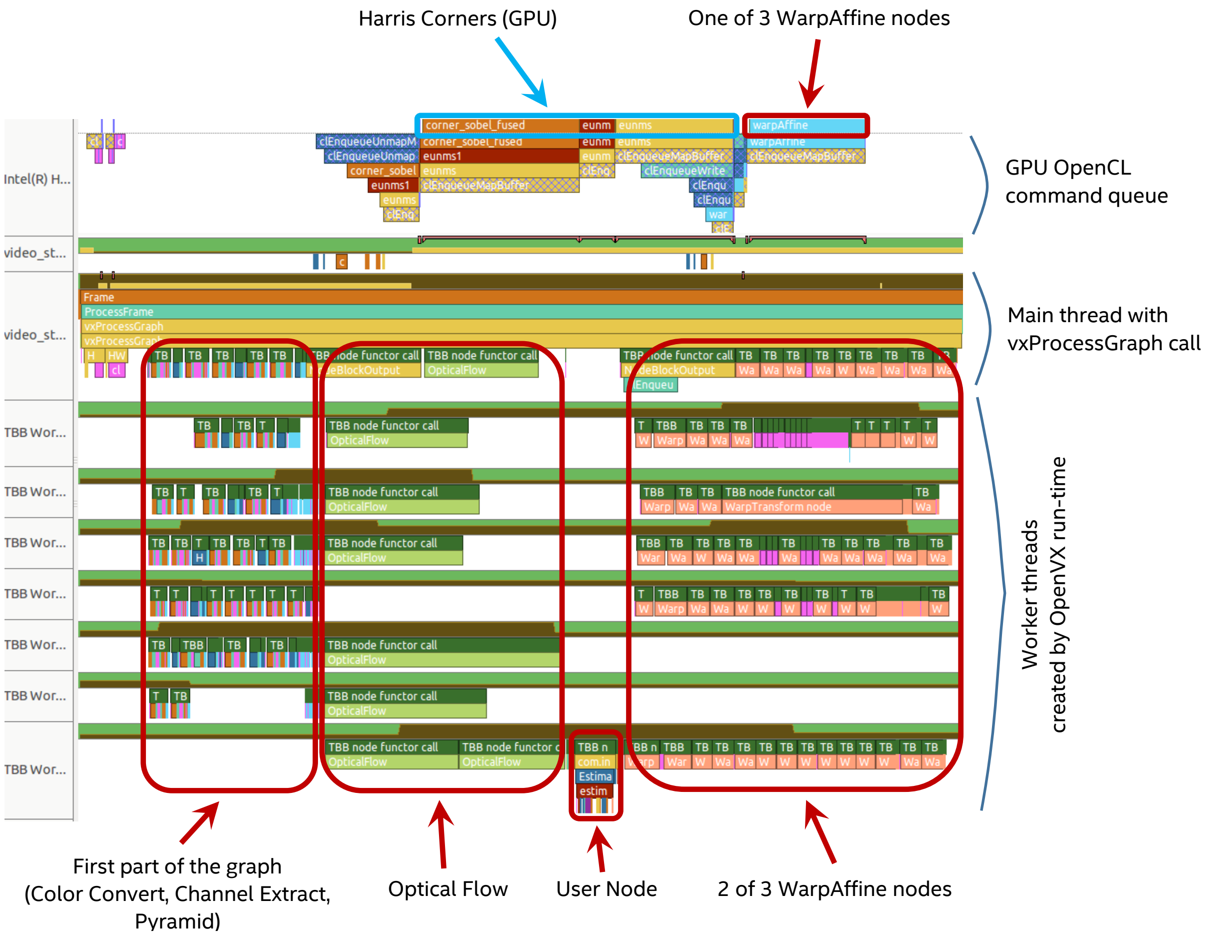
The best mode for pre-release Apollo Lake system is:

```
       ./video_stabilization --visualization 0 --hetero-config hetero.config.gpu-
harris-3-warps.txt

       Video frame size: 1280x720
       [ INFO ] Number of supported targets: 3
       [ INFO ]     Target[0] name: intel.cpu
       [ INFO ]     Target[1] name: intel.gpu
       [ INFO ]     Target[2] name: intel.ipu
       [ INFO ] Node vxColorConvertNode is assigned to intel.cpu
       [ INFO ] Node vxChannelExtractNode is assigned to intel.cpu
       [ INFO ] Node vxHarrisCornersNode is assigned to intel.gpu
       [ INFO ] Node vxGaussianPyramidNode is assigned to intel.cpu
       [ INFO ] Node vxOpticalFlowPyrLKNode is assigned to intel.cpu
       [ INFO ] Node vxEstimateTransformNode not assigned to any specific target
(missed in config)
       [ INFO ] Node vxChannelExtractNode(R) is assigned to intel.cpu
       [ INFO ] Node vxChannelExtractNode(G) is assigned to intel.cpu
       [ INFO ] Node vxChannelExtractNode(B) is assigned to intel.cpu
       [ INFO ] Node vxWarpAffineNode(R) is assigned to intel.gpu
       [ INFO ] Node vxWarpAffineNode(G) is assigned to intel.gpu
       [ INFO ] Node vxWarpAffineNode(B) is assigned to intel.gpu
       [ INFO ] Node vxChannelCombineNode(warp) is assigned to intel.cpu
       1 warning generated.
       Reached end of video file
       Processed 166 iterations
       0.42 ms by estimateTransform_lib averaged by 166 samples
       0.44 ms by EstimateTransformKernel averaged by 166 samples
       Sample was finished successfully
       18.93 ms by ProcessFrame averaged by 166 samples
       5.98 ms by ReadFrame averaged by 167 samples
       24.72 ms by Frame averaged by 166 samples
       3676.94 ms by vxVerifyGraph averaged by 1 samples
       18.77 ms by vxProcessGraph averaged by 166 samples
```

Let us see how the timeline looks like for the best heterogeneous configuration on Intel® microarchitecture code name Skylake system in VTune™ Amplifier. One call of vxProcessGraph on the VTune timeline is presented below. Note how Harris Corners node is running on GPU in parallel with Optical Flow node, user node and partially with Affine Transform. Also one of the 3 Warp Affine nodes running on GPU fits 2 of the rest Warp Affine nodes running in parallel on CPU. All these reduces the vxProcessGraph time. The picture below is captured with the following parameters:

```
       ./video_stabilization --visualization 0 --hetero-config hetero.config.gpu-
harris-warp.txt
```



References

- [OpenVX* 1.1 Specification]
- Intel® Computer Vision SDK Developer Guide
- Vision Algorithm Designer Developer Guide