Custom OpenCL™ Kernel in OpenVX* Sample

Developer Guide

Intel® Computer Vision SDK (Intel® CV SDK) – Samples

Contents

Legal Information

Introduction

This sample demonstrates an experimental Intel extension that enables using code of regular OpenCL™ kernels as OpenVX* user kernels, referred to as custom OpenCL™ kernels throughout the document.

The custom OpenCL™ kernels are similar to regular OpenVX* user kernels written in C/C++ language, except the kernel itself is written in OpenCL™ code C language, and the kernel is registered in OpenVX* runtime a bit differently than regular user kernels.

This document focuses on the difference between regular OpenVX* user kernels and custom OpenCL™ kernels. The reader should be familiar with regular OpenVX* user kernels. Also the reader should know the basics of the OpenCL™ standard.

The code of the sample consists of the following parts:

- Regular OpenCL™ program (*.cl file) with a single OpenCL kernel, which implements an example of RGB image processing - a good starting point to introduce your own OpenCL code.
- Host code that creates an OpenVX* device kernel library, which is a special notion to represent OpenCL program in OpenVX.
- Registering an OpenCL™ kernel as an OpenVX* user kernel.
- Creating and running an OpenVX* graph with a custom OpenCL™ kernel node.

The OpenVX* extension API described in this document is an Intel experimental feature. The API is subject to change without notice.

Custom OpenCL™ kernels are supported only for the GPU target.

Current extension's API limits the OpenVX* parameters that OpenCL™ kernel can access to `vx_image` and `vx_scalar` objects.

This sample briefly touches the basics of the API. Realistic application of OpenCL custom kernel is demonstrated in the Census Transform sample where it is compared with regular and tiled user kernels running on the CPU. Please refer to the `<SDK_ROOT>/samples/samples/census_transform` directory for the code and documentation.

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.

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/samples/auto_contrast)`. This document requires knowledge of what the user kernels in OpenVX* are and how to implement them. Please refer to other sample that demonstrates regular user kernels usages (`<SDK_ROOT>/samples/samples/video_stabilization`) and, optionally to the sample that demonstrates advanced tiling extension (`<SDK_ROOT>/samples/samples/census_transform`).

Sample Description

The algorithm implemented in this sample serves illustrative purposes. For simplicity reasons, the sample creates a graph consisting of just a single node. This node is implemented as a custom OpenCL™ kernel. The kernel pixelates an input RGB image and writes the result to an output RGB image.

OpenVX* graph for a frame processing is shown on Fig. 1.

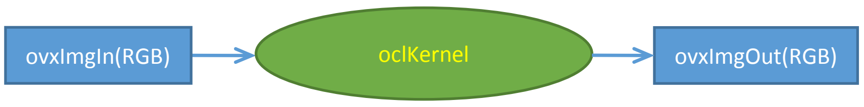


Figure 1: OpenVX* graph implemented in the sample

OpenCL™ Kernel Signature

The OpenVX* user node in Fig. 1 has two parameters:

1. Input RGB image, `vx_image` type
2. Output RGB image, `vx_image` type

OpenVX* data type, such as `vx_image`, cannot be used in OpenCL™ kernels directly because OpenVX* objects are opaque, and OpenCL™ program works with concrete types. So the essential part of custom OpenCL™ kernel extension is how the OpenVX* data types are mapped to OpenCL™ kernel parameters. A single-plane `vx_image`, like RGB image, maps to five OpenCL™ kernel arguments:

1. Pointer to an OpenCL™ kernel buffer that contains vx_image pixels
2. Image width in pixels
3. Image height in pixels
4. The offset between neighbor pixels, or pixel stride, in bytes
5. The offset between neighbor image rows, or row pitch, in bytes.

Arguments (2) - (5) are used to organize proper access to the data pointer (1) in the kernel body. As the user node has two `vx_image` arguments, the target OpenCL™ kernel should accept 10 arguments:

```
kernel void oclKernel (

    // The kernel signature is completely defined by param_types array passed to
    // vxIntelAddDeviceKernel function in host C++ code. Each OpenVX* parameter is
    // translated to one or multiple arguments of OpenCL kernel.
    // Even if the kernel's body doesn't use all these arguments, they should be
    // defined here anyway, because OpenVX* run-time relies on the order and specific
    // number of parameters, to set them correctly when calling this kernel and
    // translating OpenVX* parameters.

    // OpenVX kernel 0-th parameter has type vx_image, it is mapped to these 5 OpenCL
    // kernel arguments
    // This is input RGB image

    global const uchar* inImgPtr,
    unsigned int        widthInImg,        // width of the input image
    unsigned int        heightInImg,       // height of the input image
    unsigned int        pixelStrideInImg,  // pixel stride in bytes
    unsigned int        rowPitchInImg,     // row stride in bytes

    // OpenVX kernel 1-st parameter has type vx_image, it is mapped to these 5 OpenCL
    // kernel arguments
    // This is output RGB image

    global uchar*       outImgPtr,
    unsigned int        widthOutImg,       // width of the output image
    unsigned int        heightOutImg,      // height of the output image
    unsigned int        pixelStrideOutImg, // pixel stride in bytes
    unsigned int        rowPitchOutImg     // row stride in bytes
)
{
    . . .
}
```

Supported OpenVX* Data Types

The following table summarizes OpenVX* data types that are currently supported as custom OpenCL™ kernel parameters. The table has two columns. The first column is OpenVX* data type; the second column is a set of OpenCL™ kernel arguments that the corresponding OpenVX* data type is translated to.

OpenVX* Data Object	**OpenCL™ Kernel Argument (C code)**

`vx_image`	`global uchar*  ptr,         // pointer to image pixels`
	`uint           width,       // width of the output image`
	`uint           height,      // height of the output image`
	`uint           pixelStride, // pixel stride in bytes`
	`uint           rowPitch,    // row stride in bytes`
	
	`// . . . repeat for all plains (in case of planner images)`

`vx_image (tiled)`	`global uchar*          // pointer to image pixels`
	`int    tile_x,         // x coordinate of the tile`
	`int    tile_y,         // y coordinate of the tile`
	`int    tile_width,     // width of tile`
	`int    tile_height,    // height of tile`
	`int    image_width,    // width of the full image`
	`int    image_height,   // height of the full image`
	`int    pixelStride,    // pixel stride in bytes`
	`int    rowStride,      // row stride in bytes`
	
	`//repeat for every plane (for multi-planar images)`

`vx_array`	`global uchar* ptr,     // pointer to array items`
	`uint   items,          // number of items (not capacity)`
	`uint   itemSize,       // size of an item in bytes`
	`uint   stride          // item stride in bytes`

`vx_matrix`	`global uchar* ptr,     // pointer to matrix items`
	`uint rows,             // number of rows`

	uint cols // number of columns
vx_tensor	global uchar* ptr, // pointer to tensor items long offset, // offset to be added to ptr in bytes int num_dims, // number of dimensions of the tensor global int* dims, // dimensions sizes global int* strides // dimensions strides
vx_scalar (VX_TYPE_INT16)	short
vx_scalar (VX_TYPE_INT32)	int
vx_scalar (VX_TYPE_INT64)	long
vx_scalar (VX_TYPE_UINT16)	ushort
vx_scalar (VX_TYPE_UINT32)	uint
vx_scalar (VX_TYPE_UINT64)	ulong
vx_scalar (VX_TYPE_FLOAT32)	float
vx_scalar (VX_TYPE_FLOAT64)	double
vx_scalar (VX_TYPE_ENUM)	int
vx_scalar (VX_TYPE_SCALAR)	global uchar* ptr, // scalar raw data uchar size // size of the scalar data at ptr (in bytes)
VX_TYPE_FLOAT16_INTEL (as a tensor item type, not as a discrete scalar)	half
vx_threshold	int value_or_upper, // value for VX_THRESHOLD_TYPE_BINARY // upper threshold for VX_THRESHOLD_TYPE_RANGE int lower, // lower, defined for VX_THRESHOLD_TYPE_RANGE only int true_value, // VX_THRESHOLD_TRUE_VALUE attribute int false_value,// VX_THRESHOLD_FALSE_VALUE attribute

There are three flavors of a scalar application in the table above. All three types are different in its expression in the kernel signature:

- Scalars that are defined as `vx_scalar` and declared in OpenVX kernel with a specific primitive type, e.g. `VX_TYPE_INT32` in `param_types` array (see the section Registering a custom OpenCL™ kernel later in this document). Such scalars are passed to the OpenCL kernel as a single argument of one of the native types (e.g. `int`). Such parameters are read-only data for the device side and cannot be updated, so should be declared as `VX_INPUT` only.
- Scalars that are defined as `vx_scalar`, but declared as `VX_TYPE_SCALAR` in `param_types` when the OpenCL custom kernel is created. In this case, type and size of a scalar is not known when the kernel is defined and it fully depends on specific OpenVX data object bound to this parameter when an OpenVX graph is being constructed. Such parameter is represented as two arguments in the OpenCL

kernel signature. The first parameter is a pointer to scalar data, and the second parameter is the size of the data in bytes. Such parameters can be updated in the kernel, so they can be declared `VX_INPUT` and `VX_OUTPUT` as well.

- `VX_TYPE_FLOAT16_INTEL` is a special type that cannot be used in the kernel signature, but can be used as an item of aggregated types, such as `vx_tensor`. It is mapped to the `half` OpenCL native type.

OpenVX* Device kernel library

In order to use OpenCL™ kernels in OpenVX* graph, an OpenCL™ program with the kernels should be registered in the OpenVX* run-time. The API described in this document is an experimental Intel feature. Declarations of new functions are provided in the `vx_intel_volatile.h` header file as a part of Intel® Computer Vision SDK. It should be included in the application source code:

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

The OpenCL™ program is loaded from a text file `source.cl` to `oclSource` as a string and passed to the OpenVX* run-time “as-is”. The OpenVX* run-time compiles the program and do all necessary steps to initialize the OpenCL™ Runtime behind the scene, so you do not operate any OpenCL™ code directly in the program - all the work is done on the side of the OpenVX*.

```
vx_device_kernel_library kernelLibrary = vxAddDeviceLibraryIntel (
    ovxContext,                 // OpenVX context
    oclSource.length(),         // Size in bytes of the OpenCL source
    oclSource.c_str(),          // Pointer to the program source or binary

    "",                 // Pointer to the compilation flags string
                        // Convenient way to pass host-side defined
                        // constants as macros
                        // to OpenCL program.
                        // Ignored if library_type == VX_OPENCL_LIBRARY_BINARY_INTEL

   VX_OPENCL_LIBRARY_SOURCE_INTEL,
// VX_OPENCL_LIBRARY_SOURCE_INTEL or VX_OPENCL_LIBRARY_BINARY_INTEL
   "intel.gpu"                  // Default vx_target_intel name that should run
                                // kernels; see OpenVX Target API for reference
);
```

`vxAddDeviceLibraryIntel` may accept programs in source and binary formats. When passing a program in the source form, the compilation flags string can be specified to pass necessary knobs to the compiler and specialize the OpenCL™ program with user macros. It is assumed that behind the scenes, this flag string will be passed to the compiler as a dedicated parameter of the `clBuildProgram`. Please refer to the OpenCL™ standard specification document to learn how to form this string. In this sample, we do not need any parameters, so the string is empty.

The last argument `"intel.gpu"` specifies a target device which will run the kernels. For standard and vendor extension kernels, the OpenVX* platform may support multiple targets (see the *hetero_basic sample* to see how the complete list of targets can be queried and used). However, only the `"intel.gpu"` target can be used at the moment for custom OpenCL™ kernels. If other than `"intel.gpu"` is passed to this parameter, the OpenVX* run time will return `VX_ERROR_NOT_SUPPORTED` status and the kernel will not be created.

A device kernel library created serves as a container for OpenCL kernels. However, they are not made available automatically in the OpenVX* run-time as custom OpenCL™ kernels. To use them, each kernel should be registered in the OpenVX* run-time.

Registering a custom OpenCL™ kernel

In order to use an OpenCL™ kernel for node creation, it should be registered with the `vxAddDeviceKernelIntel` function first:

```
// For each OpenVX* node parameter, parameter type and direction are defined in
// the following two arrays. Index in the arrays corresponds to an index of
// a parameter. So we have a node with two parameters here: both images, the first
// one is an input parameter and the second one is an output parameter
vx_enum param_types[] = { VX_TYPE_IMAGE, VX_TYPE_IMAGE };
vx_enum param_directions[] = { VX_INPUT, VX_OUTPUT };

// Call vxAddDeviceKernelIntel serves the similar purpose as a regular
// vxAddUserKernel function call but accepts OpenCL kernel specific arguments
vx_kernel oclKernel = vxAddDeviceKernelIntel (
    ovxContext,                     // OpenVX context

    // The name of the kernel in OpenVX nomenclature
    "com.intel.sample.ocl_custom_kernel.oclKernel",

    VX_KERNEL_SAMPLE_OCL_CUSTOM,    // Enum value for OpenVX kernel, formed in the
                                    // same way as for regular user nodes, see
                                    // how this constant is formed in the beginning of
                                    // ocl_custom_kernel.cpp file

    kernelLibrary,                  // Kernel library, just created earlier here
                                    // in the code

    "oclKernel",                    // OpenCL kernel name as it appears in OpenCL
                                    // program

    2,                              // Number of OpenVX Kernel parameters

    param_types,                    // Types of parameters: array of type enums

    param_directions,               // Directions for each parameter: array of
                                    // direction enums

    VX_BORDER_MODE_UNDEFINED,       // Border mode: this sample doesn't care;
                                    // other values are not supported yet

    // The following 3 parameters are similar to regular user nodes
    oclCustomKernelValidate,        // Input/output parameter validator (required)
    oclCustomKernelInitialize,      // Node initialization (optional)
    oclCustomKernelDeinitialize     // Node deinitialization (optional)
);
```

Input and output validator, initialization and deinitialization callback functions (`oclCustomKernelValidate`, `oclCustomKernelInitialize`, `oclCustomKernelDeinitialize`) serve the same purpose as for regular OpenVX* user kernels and are implemented similarly. There is no any OpenCL™ code specifics there. See the `ocl_custom_kernel.cpp` file for implementation details. Remember that input and output validators are obligatory to be implemented, they are not optional in contrast to the initialization and deinitialization callback functions, which can be omitted (replaced by a `nullptr`).

Unlike regular OpenVX* user kernels, all information about kernel parameters are passed directly to `vxAddDeviceKernelIntel` through the `param_types` and `param_directions` arguments. This makes the separate parameter registration unnecessary. Compare with regular user kernels parameters that should be added separately by calling `vxAddParameterToKernel`.

In real applications, the calls to `vxAddDeviceLibraryIntel` and `vxAddDeviceKernelIntel` as well as all OpenCL™ program preparation should be made in the `vxPublishKernels` entry point of a separately compiled shared library. It enables using custom OpenCL™ kernels in multiple programs easily and hide implementation details. Also it enables tools such as Vision Algorithm Designer (VAD) to consume such kernels and use them for graph construction as regular kernel nodes.

Creating node with custom OpenCL™ Kernel

After obtaining a `vx_kernel` object for the OpenCL* kernel with `vxAddDeviceKernelIntel`, the next steps are similar to regular OpenVX* kernels (user defined or standard). The node is created as a generic node, and parameters are set with the `vxSetParameterByIndex` function.

```
vx_node node = vxCreateGenericNode(ovxGraph, oclKernel);
vxSetParameterByIndex(node, 0, (vx_reference)ovxImgIn);
vxSetParameterByIndex(node, 1, (vx_reference)ovxImgOut);
```

In real applications, this code should be a part of a vx-function for simpler creation of a node similar to the standard kernels and vendor extension kernels.

Data type information in polymorphic kernels

An OpenCL program, which is registered as an OpenVX device library with the `vxAddDeviceLibraryIntel` function, can use several pre-defined in the compile-time macros to retrieve data types of parameters. It enables specialization of the kernel for specific data formats used in node parameters.

Consider the kernel used in this tutorial. It is defined with the following parameter types:

```
vx_enum param_types[] = { VX_TYPE_IMAGE, VX_TYPE_IMAGE };
```

And it is not known which specific data format is used for the image at the moment when the kernel is created. It is only known when a node is instantiated in an OpenVX graph and parameters are set. The same is true for other aggregator types, such as arrays, tensors and matrices.

The Intel OpenVX run-time provides a dedicated mechanism to retrieve this information in the OpenCL program when it is compiled inside OpenVX context as a custom kernel.

For example, if vx_tensor parameter is passed to the kernel, it can contain `VX_TYPE_FLOAT32` or `VX_TYPE_FLOAT16_INTEL` items depending on how `vx_tensor` is created. Having a single kernel, both types can be handled inside the kernel using the `VX_PARAMETER_ELEMENT_CL_TYPE` pre-defined macro. For example, for a custom kernel with OpenVX parameters defined as:

```
vx_enum param_types[] = { VX_TYPE_TENSOR, VX_TYPE_TENSOR };
```

If the polymorphic kernel feature is not used, the kernel can be implemented with hard-coded types that work only for `float` tensor elements:

```
kernel void mykernel (
    global float* inptr,
    . . .
    global float* outptr,
    . . .
)
{
    . . .
    float inval = inptr[i];
    float outval = do_something_with(inval);
    outptr[i] = outval;
}
```

In this case, you need to make sure that no other types except `float` are passed to this kernel by providing corresponding code in the validator function. In case if the algorithm implemented in this kernel should be applied for tensors with other type of items, you need to implement another custom kernel with different name. Instead of doing that, the kernel can be implemented with use of the compile-time information about types. In this case, the kernel is transformed to this one:

```
#if VX_CL_KERNEL(mykernel)  // limit compilation scope to this kernel only

kernel void mykernel (
    global VX_PARAMETER_ELEMENT_CL_TYPE(0)* inptr,
    . . .
    global VX_PARAMETER_ELEMENT_CL_TYPE(1)* outptr,
    . . .
)
{
    // VX_PARAMETER_ELEMENT_CL_TYPE macro can be also used here for defining
    // private variables, converts, etc.
    . . .
    VX_PARAMETER_ELEMENT_CL_TYPE(0) inval = inptr[i];
    VX_PARAMETER_ELEMENT_CL_TYPE(1) outval = do_something_with(inval);
    outptr[i] = outval;
}

#endif
```

The macro `VX_PARAMETER_ELEMENT_CL_TYPE(INDEX)` is substituted with corresponding OpenCL native data type that represents the data item for the `INDEX`-th kernel parameter. For example, if input and output parameters for the OpenVX kernels are both vx_tensors of 32-bit floats (`VX_TYPE_FLOAT32`), then both `VX_PARAMETER_ELEMENT_CL_TYPE(0)` and `VX_PARAMETER_ELEMENT_CL_TYPE(1)` are defined as float. If one of them, for example first one, is an vx_tensor of `VX_TYPE_FLOAT16_INTEL items,` then `VX_PARAMETER_ELEMENT_CL_TYPE(0)` is defined as `half`.

`VX_CL_KERNEL` is used to limit compilation scope to `mykernel` only. It is necessary because `VX_PARAMETER_ELEMENT_CL_TYPE` can be defined differently for different kernel in the graph. The run-time builds the program for each kernel individually and the code between `#if VX_CL_KERNEL(mykernel)` and `#endif` is built only when a node with the kernel `mykernel` is created during `vxVerifyGraph` call.

`VX_CL_KERNEL` is not necessary to use if a given OpenCL program defines a single kernel only. In this case definitions of `VX_PARAMETER_ELEMENT_CL_TYPE` and `VX_PARAMETER_ELEMENT_ENUM_TYPE` are not ambiguous. Otherwise, `VX_CL_KERNEL` should be used for each kernel that uses `VX_PARAMETER_ELEMENT_CL_TYPE` and `VX_PARAMETER_ELEMENT_ENUM_TYPE` in a program.

Together with `VX_PARAMETER_ELEMENT_CL_TYPE,` which is translated to a native OpenCL data type, the OpenVX run-time defines `VX_PARAMETER_ELEMENT_ENUM_TYPE(INDEX)` which is defined as an integer constant that identifies the type for the `INDEX`-th parameter. It is dedicated for a conditional compilation in case when different portions of the code should be used for different types. For example:

```
#if VX_PARAMETER_ELEMENT_ENUM_TYPE(0) == VX_TYPE_FLOAT32
    // specialized code for FLOAT32
#elif VX_PARAMETER_ELEMENT_ENUM_TYPE(0) == VX_TYPE_FLOAT16_INTEL
    // specialized code path for FLOAT16
#else
    // something else for other types
#endif
```

Constants VX_TYPE_FLOAT32 and VX_TYPE_FLOAT16_INTEL as well as others for all supported types are pre-defined when OpenCL program is compiled in the OpenVX run-time.

Constants have the same names as for a regular OpenVX program written in C on the host side (please refer to the Khronos OpenVX* specification). However, the values are not guaranteed to match between the host and the device. So the values should not be passed from the host to device and vice-versa. The usage illustrated above is only valid application of the constants in an OpenCL program.

The method of data type deduction described in this section works for all aggregator types (images, arrays, matrices and tensors) as well as for scalar types. However for images, it has limited set of supported data formats. Multi-channel interleaved formats, such as YUV and RGB, are not supported and only VX_DF_IMAGE_U8, VX_DF_IMAGE_S8, VX_DF_IMAGE_U16, VX_DF_IMAGE_S16 are mapped correctly to uchar, char, ushort and short correspondingly.

Accessing Images from OpenCL™ Kernel

OpenCL™ kernel is invoked as a part of vxProcessGraph execution. NDRange that is used to execute the OpenCL™ kernel is determined by the OpenVX* run-time. By default, the global size of NDRange is the same as output image dimensions:

- get_global_size(0) = widthOutImg
- get_global_size(1) = heightOutImg

So, by default, each OpenCL™ kernel work-item is mapped to a single image pixel. In this case byte offset of a pixel can be calculated like this:

```
int dstOffset = get_global_id(1)*rowPitchOutImg + get_global_id(0)*pixelStrideOutImg;
```

The 1:1 mapping of work-items and output image pixels can be changed by setting the VX_OPENCL_WORK_ITEM_XSIZE_INTEL and VX_OPENCL_WORK_ITEM_YSIZE_INTEL attributes for the custom OpenCL™ kernel. These attributes define how many pixels in each dimension are assigned to a single work-item. By default, they both are equal to 1. It can be set to arbitrary values by calling vxSetKernelAttribute. If one or both attributes are greater than 1, each work-item is mapped to a block of pixels of specified size. For example, here is setting of block with four lines and eight pixels in a line for each work-item:

```
vx_size  xAreaSize = 8;
vx_size  yAreaSize = 4;

vxSetKernelAttribute(
    oclKernel,
    VX_OPENCL_WORK_ITEM_XSIZE_INTEL,
    &xAreaSize,
    sizeof(xAreaSize)
);

vxSetKernelAttribute(
    oclKernel,
    VX_OPENCL_WORK_ITEM_YSIZE_INTEL,
    &yAreaSize,
    sizeof(yAreaSize)
);
```

In this case the global size of NDRange will be divided by specified values in each dimension. In case if a dimension of the image is not divisible by `VX_OPENCL_WORK_ITEM_{X,Y}SIZE_INTEL`, an extra one work-item will be added for such a dimension:

- `get_global_size(0) = widthOutImg/8 + (1 if widthOutImg % 8 != 0)`
- `get_global_size(1) = heightOutImg/4 + (1 if heightOutImg % 4 != 0)`

Or, in general:

- `get_global_size(0) = 1 + (widthOutImg - 1)/VX_OPENCL_WORK_ITEM_XSIZE_INTEL`
- `get_global_size(1) = 1 + (heightOutImg - 1)/VX_OPENCL_WORK_ITEM_YSIZE_INTEL`

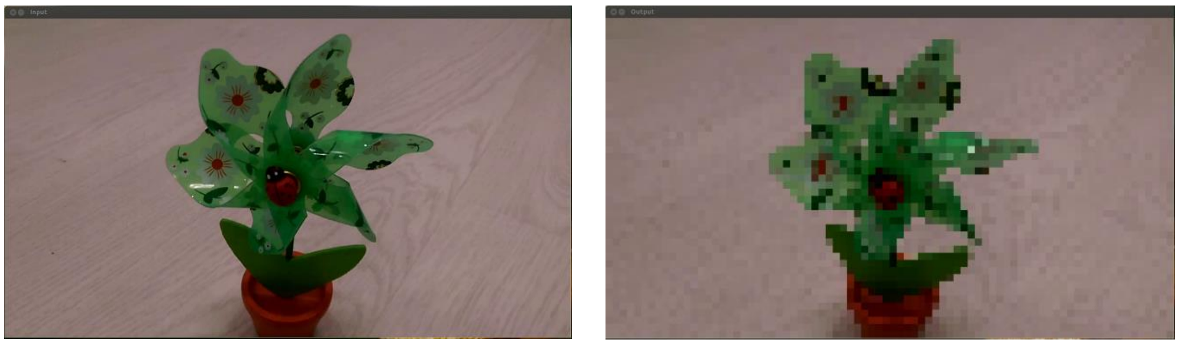
Not default values of `VX_OPENCL_WORK_ITEM_{X,Y}SIZE_INTEL` should affect the OpenCL™ kernel code which accesses image data accordingly. Please refer to `<SDK_ROOT>/samples/samples/census_transform` sample for realistic example, where the OpenCL kernel has special data path for a remainder work-item.

Building the Sample

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

Running the Sample and Understanding the Output

The sample is a command line application with additional visualization functionality. By default, it creates two GUI windows with visualization of input and output video frames as shown on Fig. 2.



Input Image **Output Image**

Fig. 2. Example of pop-up GUI windows when sample is run with default parameters

The behavior is controlled by providing command line parameters. To get the complete list of command line parameters, run:

```
$ ./ocl_custom_kernel --help
```

Here are a few examples how to run the sample in different configurations. By default, the sample loads and processes the `toy_flower.mp4` file, in the sample directory. You can change the input with the `-i/--input` option.

Example command-line:

```
$ ./ocl_custom_kernel --input your_video_file_name
```

To disable visualization, use the `--no-show` parameter:

Alternatively, the output video stream can be written to a file with the `--output` option, for example:

```
$ ./ocl_custom_kernel --output output_file_name
```

The output file video format and other properties (frame rate, codec) is chosen by the input file.

Here is an example of sample output with disabled visualization:

```
$ ./ocl_custom_kernel --no-show
toy_flower.mp4 is opened
Input frame size: 1280x720

Break on ocvCapture.read
Release data...
1.91 ms by ReadFrame averaged by 167 samples
0.73 ms by vxProcessGraph averaged by 166 samples
```

Performance metrics reported by the sample output are:

- **vxProcessGraph** is the pure `vxProcessGraph` API call average time. It includes OpenCL™ kernel execution time.
- **ReadFrame** captures the average time of frame preparation. It includes reading a frame content from `VideoCapture`, time spent in `vxMapImagePatch`/`vxUnmapImagePatch` and some other necessary operations.

References

- OpenVX* 1.0.1 Specification
- OpenVX* 1.1 Specification
- Intel® Computer Vision SDK Developer Guide