Census Transform OpenVX* Sample

Developer Guide

Intel® Computer Vision SDK – Samples

Legal Information

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Contents

Introduction

This Census Transform OpenVX* sample shows you how to use OpenVX in your application. The sample implements the **Cen**sus **T**ransform H**ist**ogram (CENTRIST [5]) algorithm - a well-known visual descriptor calculation algorithm which is used for scene categorization and object recognition tasks. CENTRIST is based on Census transform local binary pattern encoding algorithm. CENTRIST also uses Sobel filtering for image edges/contours emphasizing as a pre-processing stage and histogram calculation for final visual descriptor representation.

CENTRIST algorithm is chosen for illustrative purposes, since it is an algorithm that incorporates well-known computer vision algorithms without too much complex details. Real production object recognition or scene classification pipelines would involve additional recognition stages like cascade detectors or SVM classifiers as well as algorithm training stage. These are not covered here.

This sample 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 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 **Intel OpenVX Advanced Tiling Extension**.
- Using code of regular OpenCL™ kernels directly as OpenVX* user kernels
- Using OpenVX **vx_distribution** data type.
- Adjusting OpenVX **vx_image** color space to fit implementation to previously collected training data.
- 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 and results visualization.

NOTE: This sample uses Intel extension that enables using code of regular OpenCL™ kernels as OpenVX* user kernels, referred to as custom OpenCL™ kernels throughout the document. Look at `<SDK_ROOT>/samples/ocl_custom_kernel` sample for more basics on the API for this feature.

NOTE: In contrast to ocl_custom_kernel sample, this document is focusing on the performance aspect of using OpenCL kernels inside Census Transform OpenVX pipeline.

Brief Introduction to OpenVX*

OpenVX* is a new standard from Khronos*. OpenVX* offers 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* structures *nodes* (functions with *parameters*) and data dependencies in Directed Acyclic *Graphs (DAGs)*. Any graph must be verified by the OpenVX* runtime before execution. You can execute the same graph multiple times, with different data inputs.

Census Transform Based CENTRIST Algorithm as an OpenVX Graph

Figure 1 presents a simplified diagram of the OpenVX* graph that the sample implements

The core of the graph is the Visual Descriptor Calculation block, which consumes gray-scale image (“Grayscale”) and produces a CENTRIST visual descriptor (“Output CT histogram”) and output Census Transform image

("Output CT grayscale frame"). The grayscale image is obtained by conversion of the input RGB image to YUV image followed by extraction of the Y (luminance) channel.

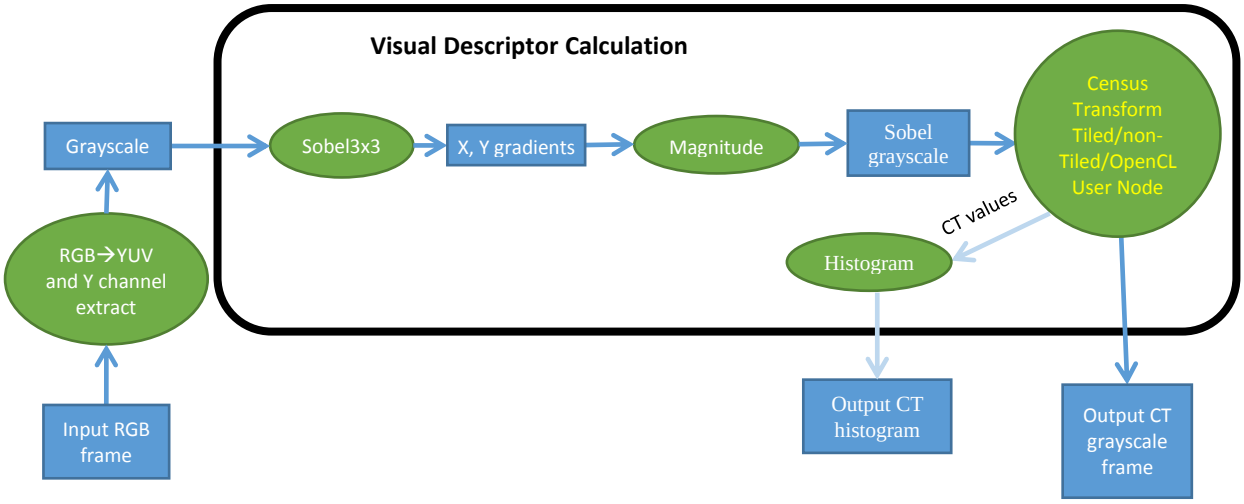


Figure 1: Top-level simplified structure of OpenVX graph implemented in the sample.

The Visual Descriptor Calculation block uses a combination of Sobel filtering to create an image with emphasized contours and Census Transform (CT) binary encoding to create output image containing unsigned char (8-bit) values [0, 255] representing edges patterns. Histogram calculation then applied to sort CT values in 256 bin histogram which is final CENTRIST visual descriptor representing current frame. More complex scenarios calculates CENTRIST for several regions of interest on the input frame. We calculate visual descriptor for the whole frame only to simplify sample pipeline. The same operation performed for all frames of the input video sequence. OpenCV is used to visualize CT encoded frame and corresponding CT histogram.

The complete OpenVX* graph is presented in Figure 2. See `main.cpp` to:

- Create `vx_context` and `vx_graph` instances
- Create virtual and non-virtual data objects
- Register Census Transform user nodes
- Adjust color space
- Create the main loop to process an input video file, frame-by-frame
- Display the results in pop-up GUI windows

See `vx_censustransform_lib.c` and `vx_censustransform_module.c` to:

- Use the `vxCensusTransformNode` function to create a user node and add it to a graph
- Implement user node registration
- Implement user node callback functions.

See `vx_censustransformtiled_lib.c` and `vx_censustransformtiled_module.c` to:

- Use the `vxCensusTransformTiledNode` function to create a user node and add it to a graph
- Implement the user node registration function
- Implement the user node callback functions
- Implement advanced tiling extension specific callback functions

See `vx_censustransformtiled_impl.c` to implement the Census Transform core functionality using SSE intrinsics.

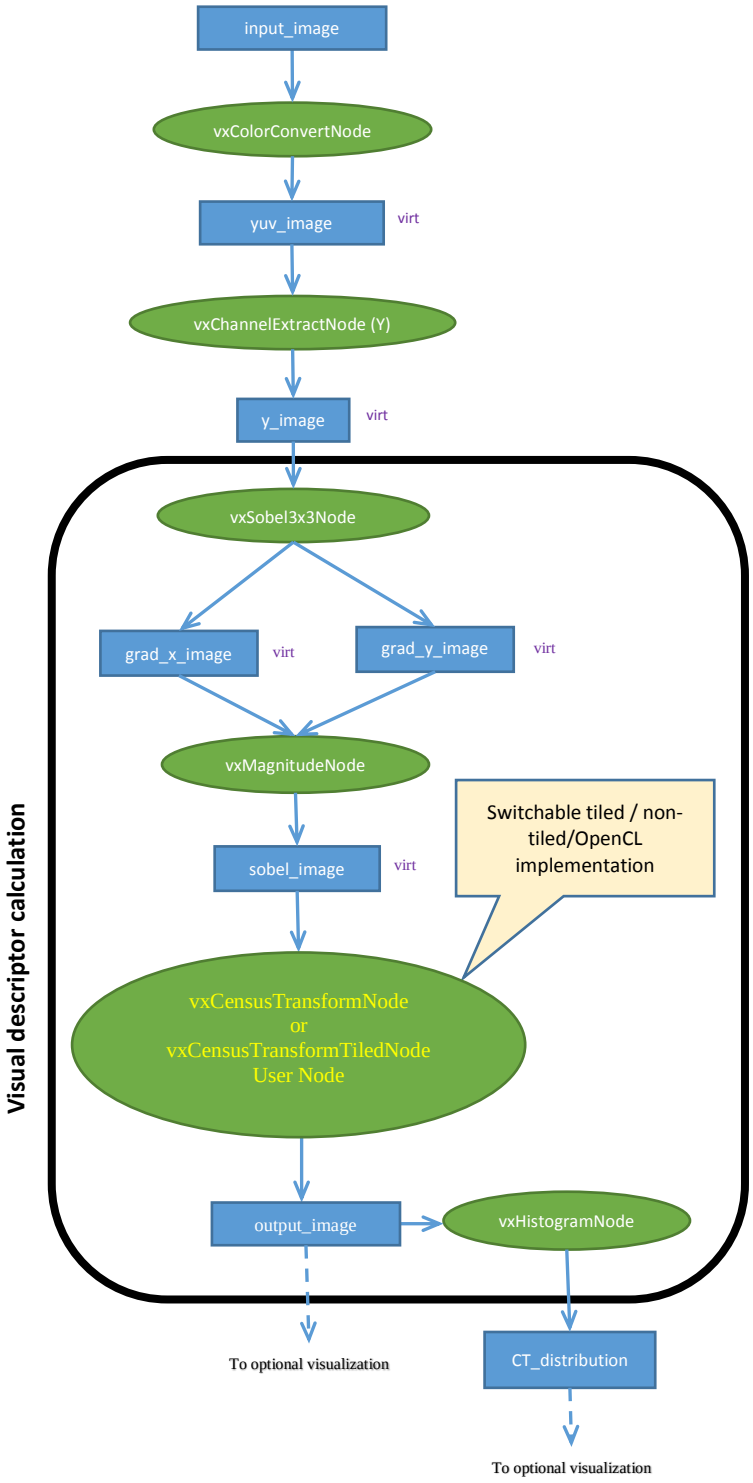


Figure 2: Complete OpenVX graph implemented in the sample.
Legend: nodes are green, data objects are blue, "virt" indicate virtual objects.
Names for the data objects correspond to the names of variables in the source code.

Virtual Data Objects

OpenVX* enables you to define intermediate data objects as virtual. It 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 does 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 is defined as virtual objects. To create a virtual object, the special form of create function is used. For example, in the sample code the object `yuv_image` is created using:

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

Please refer to the Video Stabilization OpenVX* sample for comprehensive description of virtual data objects usage. This sample is available in this SDK under `<SDK_ROOT>/samples/video_stabilization`.

Color Convertion and Color Space Adjustment

In this sample we perform color conversion of RGB `input_image` to YUV image `yuv_image` using `vxColorConvertNode` for further gray scale image retrieval `y_image` with `vxChannelExtractNode`. Several formulas (coefficients) for RGB to YUV channel values conversion are available now. The most popular are ITU 601 STDV (PAL, NTSC, and SECAM) and ITU 709 HDTV. ITU 709 is currently default color space in OpenVX*.

In some cases color space needs to be adjusted. One of the examples is an algorithm result comparison with a reference C implementation of CENTRIST visual descriptor which uses ITU 601 color space. OpenVX implementation output must correspond to the reference implementation. This is why color space needs to be adjusted before performing color conversion using standard OpenVX* node.

The other example is full object recognition pipeline that assumes some training data collected before as input. In this case, color space needs to be adjusted if ITU 601 color space was used at the training stage. Please notice that these examples (comparison with reference implementation and full object recognition pipeline) are out of scope of this sample. Color space check and adjustment functionality was added to Census Transform sample for educational purposes only.

Following code snippet from the sample requests current color space and prints it out:

```
vx_enum space;
vxQueryImage(input_image, VX_IMAGE_SPACE, &space, sizeof(vx_enum));
switch (space) {
       case VX_COLOR_SPACE_NONE:
              cout<<"Current color space is VX_COLOR_SPACE_NONE " <<endl;
              break;
       case VX_COLOR_SPACE_BT601_525:
              cout<<"Current color space is VX_COLOR_SPACE_BT601_525 " <<endl;
              break;
      case VX_COLOR_SPACE_BT601_625:
              cout<<"Current color space is VX_COLOR_SPACE_BT601_625 " <<endl;
              break;
        case VX_COLOR_SPACE_BT709:
              cout<<"Current color space is VX_COLOR_SPACE_BT709 ==
VX_COLOR_SPACE_DEFAULT" <<endl;
```

```
            break;
              default:
              cout<<"Unknown color space " <<endl;

}
```

Here is a code snippet which adjusts color space. You can re-use code from the previous code snippet to check the new color space.

```
vx_enum desired_space = VX_COLOR_SPACE_BT601_625;
vxSetImageAttribute(
       input_image,
       VX_IMAGE_SPACE,
       &desired_space,
       sizeof(vx_enum)
);
vxSetImageAttribute(
       yuv_image,
       VX_IMAGE_SPACE,
       &desired_space,
       sizeof(vx_enum)
);
```

Census Transform Algorithm Overview

The Census Transform (CT) algorithm implemented as part of this sample performs following local binary pattern encoding operation over input S16 local gradients gray scale image pixels. 3x3 pixel region surrounding each pixel of the input image is analyzed. Values of surrounding pixels compared with the current region central pixel. 8-bit binary mask is associated with the central pixel. 8 neighbor pixels contributes in this 8-bit mask depending on their values. Binary value 1 is assigned to the neighbor pixel bit in the binary pattern mask if its value is greater or equal to the current central pixel value. Binary value 0 is assigned otherwise. As result 8-bit length mask formed by 8 neighbor pixel values is obtained. 3x3 region raw wise scan from the left to right forms the final 8-bit code representing central pixel. And finally decimal value representing this 8-bit code is assigned as the output U8 CT image pixel value. The same operation is performed for all pixels of the input image to fill corresponding pixels of output CT image. Following picture illustrates CT encoding for a one pixel of the input image:

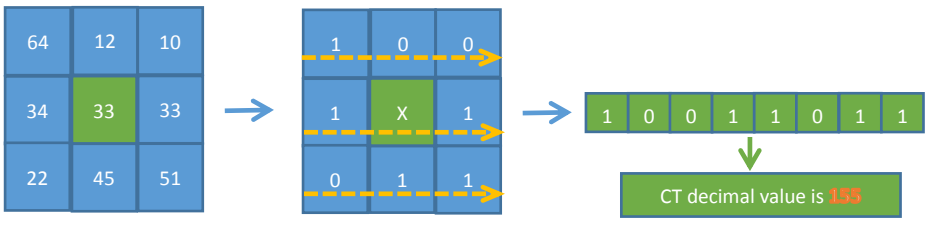


Figure 3: 3x3 Census Transform for a single output pixel implemented in this sample.

You can find C++ SSE optimized Census Transform algorithm implementation function `censustransform` defined in `vx_censustransform_impl.c`. The OpenVX kernel, which is described later in this document, is implemented on top this core function.

Please notice that there are several alternative algorithm modifications exist. 5x5 region can be used to perform similar CT operation. Quite the same idea is used in various modifications of Local Binary Pattern (LBP) algorithms [6]. Comparison rules (less equal, greater equal, less, greater), scanning order (raw-wise, clock-wise, counter clock-wise), window size (3x3, 5x5) and comparison threshold may vary.

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 kernels to process images and combine the kernels with the rest of the pipeline implemented as an OpenVX* graph.

OpenVX* supports client-defined functions that are executed as nodes from inside the graph.

There are three user kernels in the sample. All kernels implement exactly the same algorithm. Two of the kernels are implemented in C code and one of the kernels is written in OpenCL C. One of the C kernels is a standard OpenVX user node and the second C kernel is a user node implemented with Intel tiling extension.

- **Census Transform Node –** This node takes the magnitude of local gradients gray scale image of type `vx_image (sobel_image)` with data type `VX_DF_IMAGE_S16` and outputs Census Transform (CT) image of type `vx_image (output_image)` containing `VX_DF_IMAGE_U8` CT values for each pixel of the input image.

- **Census Transform Tiled Node** implements exactly the same functionality as **Census Transform Node** but uses Intel advanced OpenVX* tiling extension to achieve additional performance gain. Implementation details which differs this advanced tiled node from simple node will be discussed in the next chapter.

- **Census Transform OpenCL Node** implements the same algorithm as first two nodes, but written completely in OpenCL C. OpenCL program with the kernel is compiled by OpenVX run-time internally. The OpenCL program with the kernel is located at `vx_censustransform_opencl_impl.cl` file.

All the kernels, standard, tiled and OpenCL custom kernel, have common set of call-back functions implementing their functionality. In the most cases implementations are almost identical. Additional tiled kernel specific call-back functions will be discussed in the next chapter.

The rest of this chapter describes user C kernels (a regular one and tiled one).

To define a user kernel, you need to define five call-back functions (four callbacks in case of OpenCL kernel):

- **Kernel function** where target image processing logic is defined. Applicable for C kernels only (for OpenCL C kernel, the kernel in OpenCL program serves this role). Refer to `CensusTransformKernel` and `CensusTransformTilingKernel` defined in `vx_censustransform_module.c` and `vx_censustransformtiled_module.c` respectively. Please notice that `CensusTransformKernel` operates with the whole image `vx_image` while `CensusTransformTilingKernel` operates with image tiles `vx_tile_intel_t`.

- **Input and output validator function** 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 `CensusTransformValidator` defined in `vx_censustransform_module.c`.

- **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. But the Census Transform kernel doesn't need any initialization. Refer to `CensusTransformInitialize` and `CensusTransformTiledInitialize` defined in `vx_censustransform_module.c` and `vx_censustransformtiled_module.c` respectively. NULL can be passed to user node registration function as alternative when no initialization is required. Empty initialization function was

implemented in this sample for educational purposes only. Such empty function with a correct signature can be used as a template for further sample code functionality extension.

- **Deinitialization function** that is called when a node instance is destroyed from a graph. It frees resources that are allocated in Initialization function. But the Census Transform kernel doesn't need any deinitialization. Refer to `CensusTransformDeinitialize` and `CensusTransformTiledDeinitialize` defined in `vx_censustransform_module.c` and `vx_censustransformtiled_module.c` respectively. NULL can be passed to user node registration function as alternative when no deinitialization is required. Empty deinitialization function was implemented in this sample for educational purposes only. Such empty function with a correct signature can be used as a template for further sample code functionality extension.

The sample registers Census Transform user node, by calling OpenVX add kernel API `vxAddUserKernel` with passing all 4 corresponding callbacks described above and some other parameters. Refer to `PublishCensusTransformKernel` function defined in `vx_censustransform_module.c` for details. Tiled kernel implementation specific will be discussed in the next chapter.

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 is very similar to creation of any standard nodes. Declaration of `vx-` an `vxu-` functions can be found in `vx_user_census_nodes.h` file and they are defined in `vx_censustransform_lib.cpp` and `vx_censustransformtiled_lib.cpp` files.

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

```
vxCensusTransformNode(
    graph_handle,         // graph where to insert an instance of the node
    sobel_image,          // input image parameter
    output_image          // output image parameter
);
```

And quite the same in the case of tiled node usage:

```
vxCensusTransformTiledNode(
    graph_handle,         // graph where to insert an instance of the node
    sobel_image,          // input image parameter
    output_image          // output image parameter
);
```

A `vxu`-function is used to call a user kernel outside of a graph. It is called *immediate* 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 immediate mode supposes that the implementation of a `vxu`-function will build an OpenVX graph behind the scene. This graph would contain 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 every time when a vxu-function is called. See `vxuCensusTransformNode` or `vxuCensusTransformTiled` functions source code for more details about how `vxu`-functions are usually constructed.

Input and Output Validators for User-Defined Kernels

Input and output validator is a special function defined for each user-kernel. Both regular user node and Intel advanced tiling-based user node in this sample do require validators. Regular user node requires single Input/output validator function according to OpenVX 1.1 standard. Intel advanced tiling-based user node uses the same validator function. They are called by the OpenVX runtime whenever validation is needed for user-node parameters, particularly when `vxVerifyGraph` function is called for a graph where the user-nodes are instantiated.
An input and output validator is called for each kernel input or output parameter. It accepts a node reference parameters array, number of parameters and metadata to check output parameters. The validator 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. The output validation code actually sets requirements for the output parameters attributes without querying them and let the OpenVX runtime to verify them. The requirements for output parameters are set by calling the vxSetMetaFormatAttribute function. After exiting an input and output 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 an output 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 skip definition of all virtual object attributes during such objects creation. Here is `CensusTransformValidator` for one of the nodes from the sample:

```
typedef enum _census_transform_params_e {
    CENSUSTRANSFORM_PARAM_INPUT = 0,
    CENSUSTRANSFORM_PARAM_OUTPUT
} census_transform_params_e;

vx_status VX_CALLBACK CensusTransformValidator(vx_node node, const vx_reference
parameters[], vx_uint32 num, vx_meta_format metas[])
{
    vx_status status = VX_ERROR_INVALID_PARAMETERS;
    if(num!=2) //check that number of parameters is correct
    {
        return status;
    }

    vx_df_image df_image = 0;
    // Verify that input image is a 1-channel S16 image
    if(vxQueryImage((vx_image)parameters[CENSUSTRANSFORM_PARAM_INPUT],
VX_IMAGE_FORMAT, &df_image, sizeof(df_image)) == VX_SUCCESS)
    {
        if (df_image == VX_DF_IMAGE_S16)
        {
            status = VX_SUCCESS;
        }
        else
        {
            status = VX_ERROR_INVALID_VALUE;
            vxAddLogEntry((vx_reference)node, status, "CT Validation failed: invalid
input image format\n");
        }
    }

    vx_uint32 output_width = 0;
    vx_uint32 output_height = 0;
    vx_uint32 input_width = 0;
```

```
    vx_uint32 input_height = 0;

    //Query the input image
    status = vxQueryImage((vx_image)parameters[CENSUSTRANSFORM_PARAM_INPUT],
VX_IMAGE_WIDTH, &input_width, sizeof(input_width));
    status |= vxQueryImage((vx_image)parameters[CENSUSTRANSFORM_PARAM_INPUT],
VX_IMAGE_HEIGHT, &input_height, sizeof(input_height));

    //this node will actually output w-2xh-2
    // since it only processes 'valid' pixels
    output_width = input_width-2;
    output_height = input_height-2;

    vx_df_image  format = VX_DF_IMAGE_U8;
    //Input is of S16 type and output image is of type U8
    //Set width and height for validation as well
    status |= vxSetMetaFormatAttribute(metas[CENSUSTRANSFORM_PARAM_OUTPUT],
VX_IMAGE_WIDTH, &output_width, sizeof(output_width));
    status |= vxSetMetaFormatAttribute(metas[CENSUSTRANSFORM_PARAM_OUTPUT],
VX_IMAGE_HEIGHT, &output_height, sizeof(output_height));
    status |= vxSetMetaFormatAttribute(metas[CENSUSTRANSFORM_PARAM_OUTPUT],
VX_IMAGE_FORMAT, &format, sizeof(format));

    return status;
}
```

Figure 4 presents a simplified sample timeline with focus on Census Transform user node and corresponding callback functions.

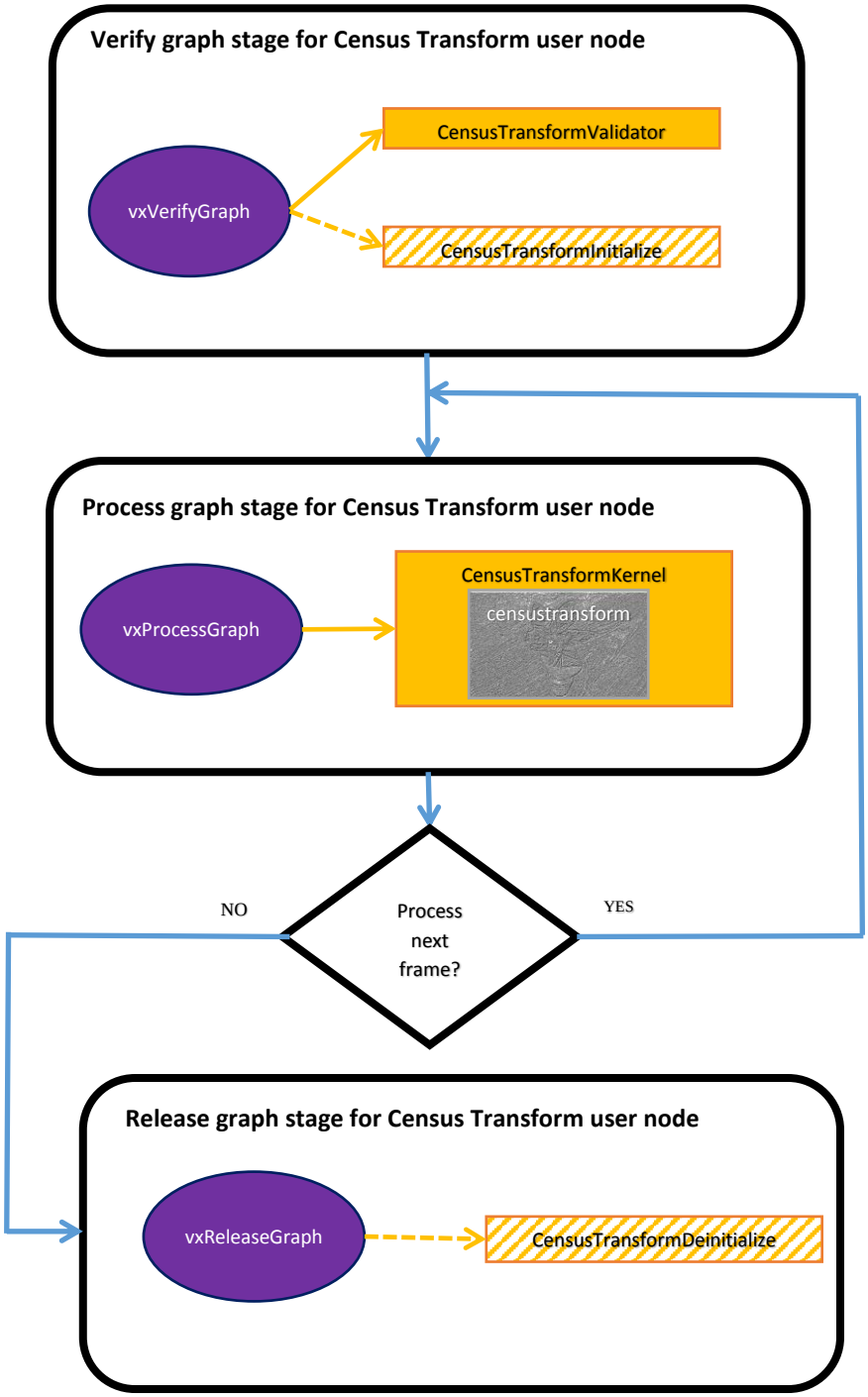


Figure 4: Census Transform user node timeline.
Legend: OpenVX API calls are violet, CT user nodes call back functions are orange.
Names for the callback functions rectangles correspond to the names in the source code. Optional callback function blocks are with line pattern fill.

Intel OpenVX Advanced Tiling Extension

Khronos* OpenVX* work group has defined “The OpenVX ™ User Kernel Tiling Extension” which enables to break the image data into smaller rectangular units (Tiles). This enables optimized computation on these tiles with faster memory access or parallel execution of a user node on multiple image tiles simultaneously.

The Intel OpenVX Advanced Tiling Extension is designed to provide additional support for kernels that exhibit complex tile dependencies, beyond what the standard Khronos "OpenVX User Kernel Tiling Extension" supports. For example, the Advanced Tiling Extension enables you to create tiled user kernels for:

- Geometric operations (for example, transformation, rotation, scaling)
- Kernel output tiles that need to be produced serially (error diffusion)
- Kernels for which the produced output tile depends on differing tiles from multiple input images

The Intel® Computer Vision SDK* comes with support for user kernel tiling. The Vision SDK supports Intel OpenVX Advanced Tiling Extension. This sample uses advanced tiling extension to implement **Census Transform Tiled** node. Besides 4 standard user node callback functions implementing kernel, input/output validation and initialization and de-initialization callback functions, the advanced tiling extension requires additional callback functions. The most of them are optional and implemented in this sample as templates with correct signatures for educational purposes. Here is the list of additional callback functions:

- **Tile mapping function** that is used by OpenVX runtime to map input and output tiles. Refer to `CensusTransformTileMapping` defined in `vx_censustransformtiled_module.c`. As well you can find relevant code snippet and detailed description below in this chapter.

  Pre-Process function. If set, this function is called once at the beginning of each call to `vxProcessGraph`, but before any nodes in the graph have started processing. It is intended to be used to perform any required per-`vxProcessGraph` initialization, which may be required for certain kernels (e.g. custom memory allocation). Census transform tiled implementation requires no pre-processing. Pre-processing function stub implementation added to the sample for educational purposes only. Refer to `CensusTransformTiledPreProcessing` defined in `vx_censustransformtiled_module.c`. NULL can be passed to the tiled user node registration function as the alternative when no pre-processing is required.

  Post-Process function. If set, this function is called once at the end of each call to `vxProcessGraph`, after all nodes in the graph have completed processing. It is intended to perform any required per-`vxProcessGraph` de-initialization or data aggregation which may be required for certain kernels. Census transform tiled implementation requires no post-processing. Post-processing function stub implementation added to the sample for educational purposes only. Refer to `CensusTransformTiledPostProcessing` defined in `vx_censustransformtiled_module.c`. NULL can be passed to the tiled user node registration function as the alternative when no post-processing is required.

  Set Tile Dimensions function. If set, this function is called within `vxVerifyGraph` to give a kernel the ability to override the current output tile size chosen by the runtime with a custom tile size. For example, a kernel may be designed to work only with tile width which is equal to the output image width. The function can update tile dimensions using a combination of the kernel parameters and current tile dimensions, which are passed as an input parameter. Census transform tiled implementation requires no specific tile dimensions size. The sample uses tile dimensions chosen by runtime. Tile dimensions setup function stub implementation added to the sample for educational purposes only. Refer to `CensusTransformTiledSetTileDimensions` defined in `vx_censustransformtiled_module.c`. NULL can be passed to the tiled user node registration function as the alternative when no specific tile dimensions setup is required.

The sample registers Census Transform user kernel, by calling `vxAddAdvancedTilingKernel` with passing all 9 corresponding callbacks to the OpenVX runtime upon the kernel registration:

- Four standard user nodes callback and slightly modified kernel function operating with `vx_tile_intel_t` instead of `vx_image` (described in the previous section)
- plus tile mapping callback function (which is specific for Intel OpenVX advanced tiling extension)
- Pre and Post Processing callbacks and Tile Dimension Setup callback function are optional and not required by this sample. NULL values can be passed to `vxAddAdvancedTilingKernel` instead of these 3 callbacks.

Refer to `PublishCensusTransformTiledKernel` function defined in `vx_censustransformtiled_module.c` for details.

Tile Mapping Function and Tiling Kernel

Advanced tiling extension allows flexible source to destination tile mapping in contrast to Khronos tiling extension "The OpenVX ™ User Kernel Tiling Extension". For each given output tile, it is required that the kernel creator describe the input tile dependencies through the provided tile mapping function. Within the `vxVerifyGraph`, the OpenVX runtime will call this function for each tile in the output image. The tile attributes (`x, y, width, height`) of the destination tile (`dstRectIn`) are passed as input parameters. The purpose of this function is to set the input tile attributes (`srcRectOut`), which are required to produce the given output tile attributes (`dstRectIn`). If the kernel has multiple `vx_image` input parameters, this tile mapping function will be called separately for each one. The passed in parameter, `param_num`, specifies which input parameter index that the runtime is requesting a tile mapping for. The following code snippet illustrates the single mandatory tile mapping function which is required by the advanced tiling in addition to the standard user node callbacks.

```
static vx_status CensusTransformTileMapping( vx_node node,
                                     vx_reference parameters[],
                                     const vx_tile_t_attributes_intel_t* dstRectIn,
                                     vx_tile_t_attributes_intel_t* srcRectOut,
                                     vx_uint32 param_num )
{
    // To produce a dst tile, a source tile which is 2 columns / 2 lines
    // larger is required.
    // dst image size is srcWidth-2, srcHeight-2, so no need to adjust
    // x, y because of smaller dst tile size
    srcRectOut->x = dstRectIn->x;
    srcRectOut->y = dstRectIn->y;
    srcRectOut->tile_block.width = dstRectIn->tile_block.width + 2;
    srcRectOut->tile_block.height = dstRectIn->tile_block.height + 2;
    return VX_SUCCESS;
}
```

The following code snippet illustrates tiling kernel which retrieves tile data `vx_tile_intel_t` containing address for source and destination along with tile data strides, width and height. Please notice that in case of non-tiled user node `CensusTransformKernel` kernel receives the whole input and output images of type `vx_image`. Core Census Transform implementation function `censustransfrom` (defined `vx_censustransform.c`) is exactly the same as in case of Census Transform non-tiled kernel callback function implementation. The `censustransfrom` implementation makes no difference between image tile and the whole image and used in both tiled and non-tiled nodes. Refer to `CensusTransformTilingKernel` function defined in `vx_censustransformtiled_module.c`.

```
static vx_status CensusTransformTilingKernel(vx_node node,
                                             void * parameters[],
                                             vx_uint32 num,
                                             void * tile_memory,
                                             vx_size tile_memory_size)
{
```

```
    vx_tile_intel_t *pInTile =
(vx_tile_intel_t*)parameters[CENSUSTRANSFORMTILED_PARAM_INPUT];
    vx_tile_intel_t *pOutTile =
(vx_tile_t*)parameters[CENSUSTRANSFORMTILED_PARAM_OUTPUT];


    vx_int16 *pSrc = (vx_int16 *)pInTile->base[0];
    vx_uint8 *pDst = pOutTile->base[0];


    vx_int32 srcStride = pInTile->addr[0].stride_y;
    vx_int32 dstStride = pOutTile->addr[0].stride_y;


    vx_uint32 dst_tile_width = pOutTile->addr[0].dim_x;
    vx_uint32 dst_tile_height = pOutTile->addr[0].dim_y;


    return censustransform(pSrc, srcStride, pDst, dstStride,
                           dst_tile_width, dst_tile_height);
}
```

Figure 5 presents a simplified sample timeline with focus on Census Transform Tiled user node and corresponding call back functions.

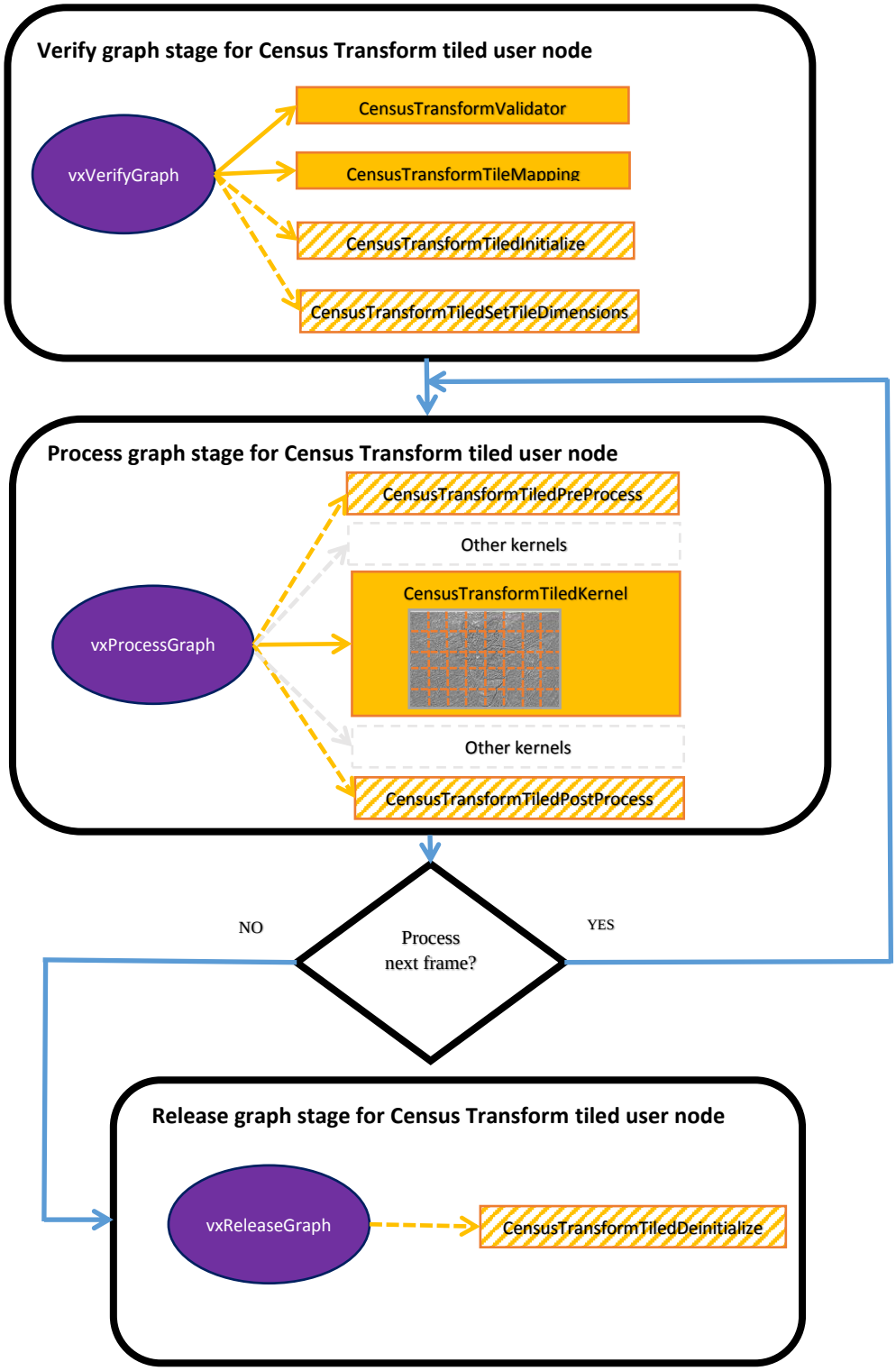


Figure 5: Census Transform Tiled user node timeline.
Legend: OpenVX API calls are violet, CT user nodes call back functions are orange.
Names for the implemented callback functions rectangles correspond to the names in the source code. Optional callback function blocks are with line pattern fill.

Final Step in the Graph: Histogram

The final step of CENTRIST visual descriptor algorithm is calculation of CT values histogram. `vxHistogramNode` is added in the sample graph for this purpose. `vx_distribution` is output for this node. This distribution data object need to be created and properly initialized to receive output CENTRIST values for each frame of the input image. Here is distribution creation function definition:

```
vx_distribution VX_API_CALL vxCreateDistribution (vx_context context,
                                                  vx_size numBins,
                                                   vx_int32 offset,
                                                   vx_uint32 range);
```

The second argument is a number of bins in histogram. We want to sort U8 CT values which are in range [0, 255] in 256 histogram bins. We use 256 as the second argument value. The third argument is the start offset into the range value that marks the beginning of the 1D Distribution. We set it to 0 (no offset). The fourth argument is histogram range. We set it to 256 as we process the U8 image. As result we will sort U8 values [0, 255] into 256 histogram bins to obtain output CENTRIST values.

```
vx_distribution CT_distribution = vxCreateDistribution (context, 256, 0, 256);
```

Initialized distribution data object then passed as input argument in histogram node

```
vxHistogramNode (vx_graph_handle, vx_output_image, CT_distribution)
```

After graph execution calculated histogram values can be retrieved from distribution data object. Following snippet demonstrates distribution data access

```
//Access histogram data
void* histData = NULL;
vx_map_id map_id;
vxMapDistribution (CT_distribution, &map_id, &histData, VX_READ_ONLY,
VX_MEMORY_TYPE_HOST, 0);
//Do some processing with histogram data histData
…
//Unmap distribution resource
vxUnmapDistribution(CT_distribution, map_id);
```

Basic OpenCV Interoperability

The sample uses OpenCV for:

- Reading video frame by frame from a file with the help of `cv::VideoCapture`.
- Visualizing input and output frames by creating pop-up GUI windows and drawing debug information.

OpenCV can be used together with OpenVX code by careful sharing of data pointed by objects. In this sample the memory is shared between `vx_image` and `cv::Mat`. Please refer to Video Stabilization OpenVX sample to see detailed description for `vx_image` and `cv::Mat` mapping procedure. This sample is available in this SDK (<SDK_ROOT>/samples/video_stabilization).

Building the Sample

See the root `README` file for all samples and another `README` file located in `census_transform` 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 output video frames and the CENTRIST visualization. The behavior is controlled by providing command line parameters. To get the complete list of command line parameters, run:

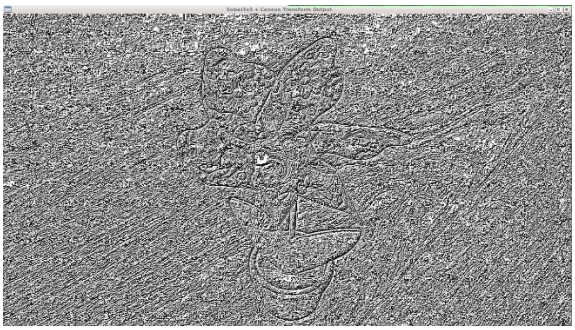
```
$ ./census_transform --help
```

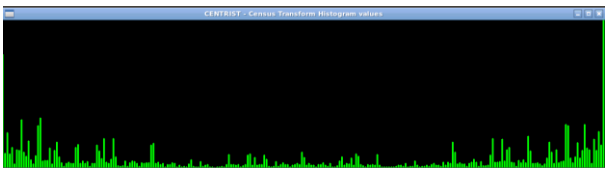
The following section provides a few examples how to run the sample in different configurations.

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

```
$ ./census_transform
```

will open `toy_flower.mp4` and create two pop-up windows playing Census Transform output frame stream and corresponding CENTRIST visual descriptor values histogram:





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:

```
$ ./census_transform --no-show
```

When the sample finishes execution, it prints performance statistics out to the console in milliseconds:

```
$ ./census_transform --no-show

[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
Current color space is VX_COLOR_SPACE_BT709 == VX_COLOR_SPACE_DEFAULT
Switched to VX_COLOR_SPACE_BT601_625
Frames are over
Number of processed frames is : 166
12.53 ms by vxProcessGraph averaged by 166 samples
```

The main performance metrics are times in ms. spent in `vxProcessGraph` which represents average CENTRIST graph processing time per frame.

The following example demonstrate performance output for the same graph but with tiling disabled for CT user node:

```
$ ./census_transform --no-show --no-tiled

[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
Current color space is VX_COLOR_SPACE_BT709 == VX_COLOR_SPACE_DEFAULT
Switched to VX_COLOR_SPACE_BT601_625
Frames are over
Number of processed frames is : 166
17.23 ms by vxProcessGraph averaged by 166 samples
```

Please notice significant performance gap between tiled and non-tiled version: **12.53** ms vs. **17.23** ms. Go to Performance Analysis section below for more details on performance.

NOTE: Console output is presented here for illustrative purposes. The performance numbers may differ significantly depending on the platform, specific system and the environment used. The numbers presented in this particular section are gathered on the pre-production Apollo Lake system.

To use OpenCL™ C kernel to do Census Transform algorithm in the graph, `--opencl` key should be used together with `--no-tiled` key. In this case, `--no-tiled` knob means that OpenCL kernel doesn't process data in tiles defined in OpenVX run-time. Instead a whole input and output images are passed to OpenCL kernel.

```
./census_transform --no-show --no-tiled --opencl

[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
Current color space is VX_COLOR_SPACE_BT709 == VX_COLOR_SPACE_DEFAULT
Switched to VX_COLOR_SPACE_BT601_625
Frames are over
Number of processed frames is : 166
9.78 ms by vxProcessGraph averaged by 166 samples
```

Note that the following line appears in all cases regardless of the use of knob `--opencl`:

```
[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
```

It happens because the OpenCL C kernel is compiled in the beginning of the program together with creation of regular user node kernel and tiled kernel regardless of options provided in the command line. While compilation of the kernel always happens, it is really used in the graph when `--opencl` command line option is provided only.

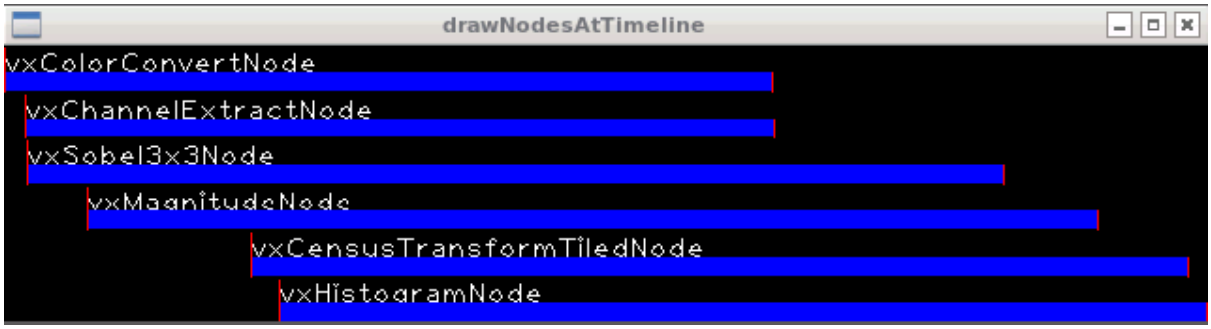
Using `--opencl` without `--no-tiled` knob is not supported in current version of the sample.

Debug Visualization

When visualization is not switched off by `--no-show` command line option you will see additional window illustrating performance information.

This mode create additional 'drawNodesAtTimeline' window that shows times spent in each node in the OpenVX graph on the timeline. These numbers are provided by OpenVX engine (see details in video_stabilization sample

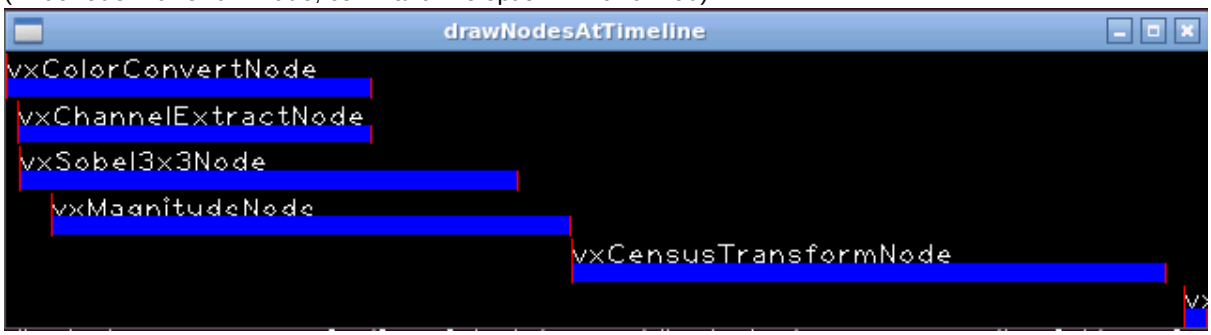
document). The absolute values are not shown. Only contributions of the individual nodes to the overall time are presented to compare nodes performance and inspect their scheduling in the graph execution:



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

Notice that the graph execution visualization normalizes the contribution from the nodes to the overall frame time.

Compare the above screenshot and the contribution from the Census Transform node in the case of tiled node (`vxCensusTransform`**`Tiled`**`Node`) versus graph execution breakdown with non-tiled node (`vxCensusTransformNode`, command line option `--no-tiled`):



Here non-tiled CT user node serializes the pipeline. vxMagnitudeNode and vxHistogramNode are still in multithreaded mode. But their tiles processing can't be overlapped with non-tiled CT node processing because of data dependencies. It also significantly impacts overall performance as it was mentioned before - refer to the execution stats that the sample prints upon exit. So, when non-tiled version of the node is executed, it contributes much more to the graph execution time, and also the overall frame time is significantly larger as can be seen in the sample output (again, refer to the previous section).

Performance Analysis

It is important to understand performance implications of using different types of user kernel in OpenVX. The main performance metric of the OpenVX execution is the time spent in vxProcessGraph. This chapter teachs how to analyze an application that uses OpenVX to find and exploit performance opportunities provided by OpenVX user nodes expressed as regular C, tiled C and OpenCL kernels. Understanding of how vxProcessGraph works with different user nodes is a key to achieve better application performance.

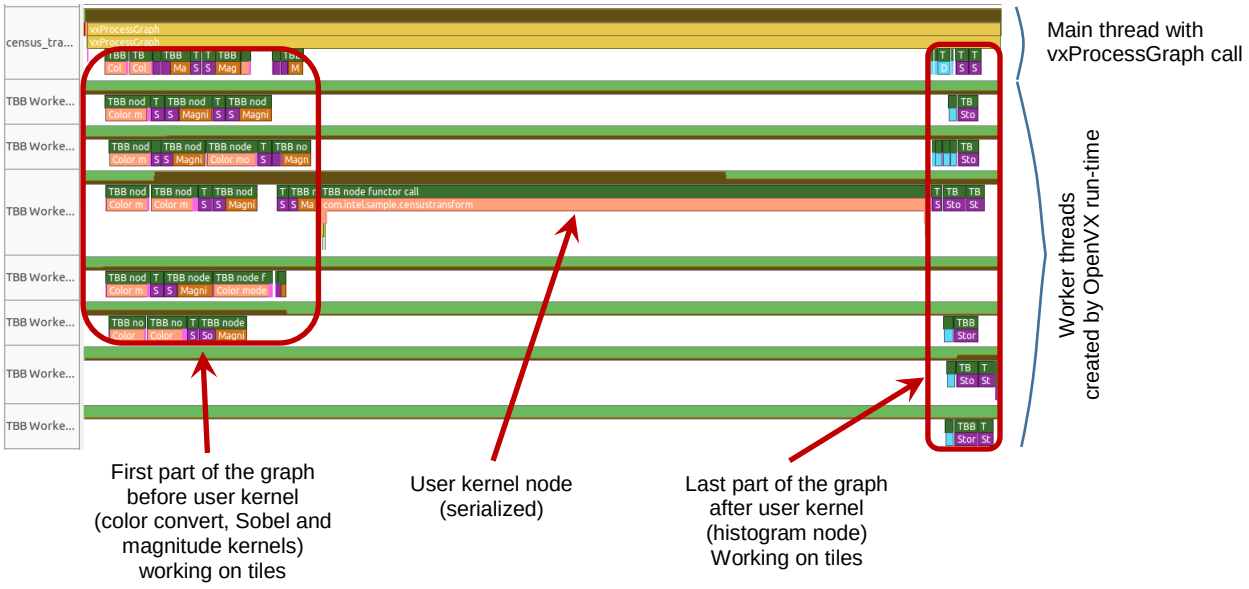
NOTE: Screenshots captured in Intel VTune presented in this section are provided for illustrative purposes. They express brief performance picture that user may expect to see when looking at own application in VTune. The screen shots are gathered on exemplar Skylake and Apollo Lake systems.

CPU user kernels

While non-tiled version of the user kernel written in C may look as the simplest and easy to do option, it doesn't always acceptable from the performance side. A regular user kernel is executed by OpenVX run-time on the host CPU in the serial way. If the OpenVX graph with such a kernel has enough number of other nodes that can run in parallel with a user node, it may have a good CPU utilization because run-time will try to use as many threads as possible to execute other nodes in parallel with user node kernel. But if a user node acts as a barrier in the graph and takes significant amount of time, it may lead to severe platform underutilization.

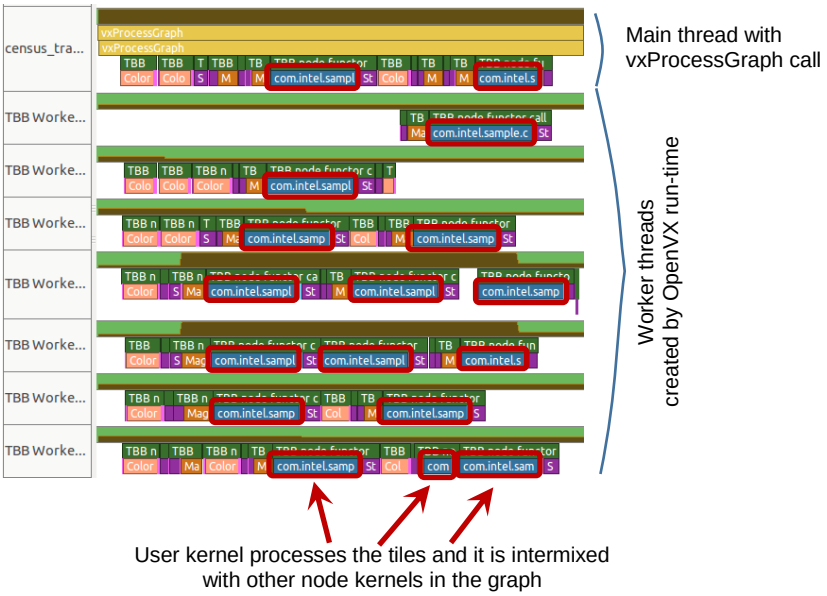
Let's look at the timeline screen shot from Intel VTune that illustrates **non-tiled C kernel** execution of census transform kernel (`--no-tiled` option is used). On the picture below, Instrumentation and Tracing Technology API (ITT, [7]) is used to highlight tile processing of the nodes in the graph.

NOTE: Level of ITT instrumentation presented here, where vxProcessGraph internals are presented in details, may not be available in product builds of OpenVX run-time. Please refer to the User Guide and Release Notes.



As it is seen on the picture, graph execution is serialized when user kernel is executed. All nodes before the user node should be completed prior user node starts the execution. And all nodes after user node can start execution after user node finishes only. Beginning and ending of the graph is multi-threaded, but not user kernel execution.

Compare the picture below with timeline of **tiled C kernel** execution (`--no-tiled` option is **not** used):



Using tiled version of regular C kernel unlocks multithreaded execution on the target platform with many hardware threads. For census transform sample graph, it eliminates big single-threaded region from the timeline and leads to much better CPU cores utilization and lesser execution time.

Running on GPU

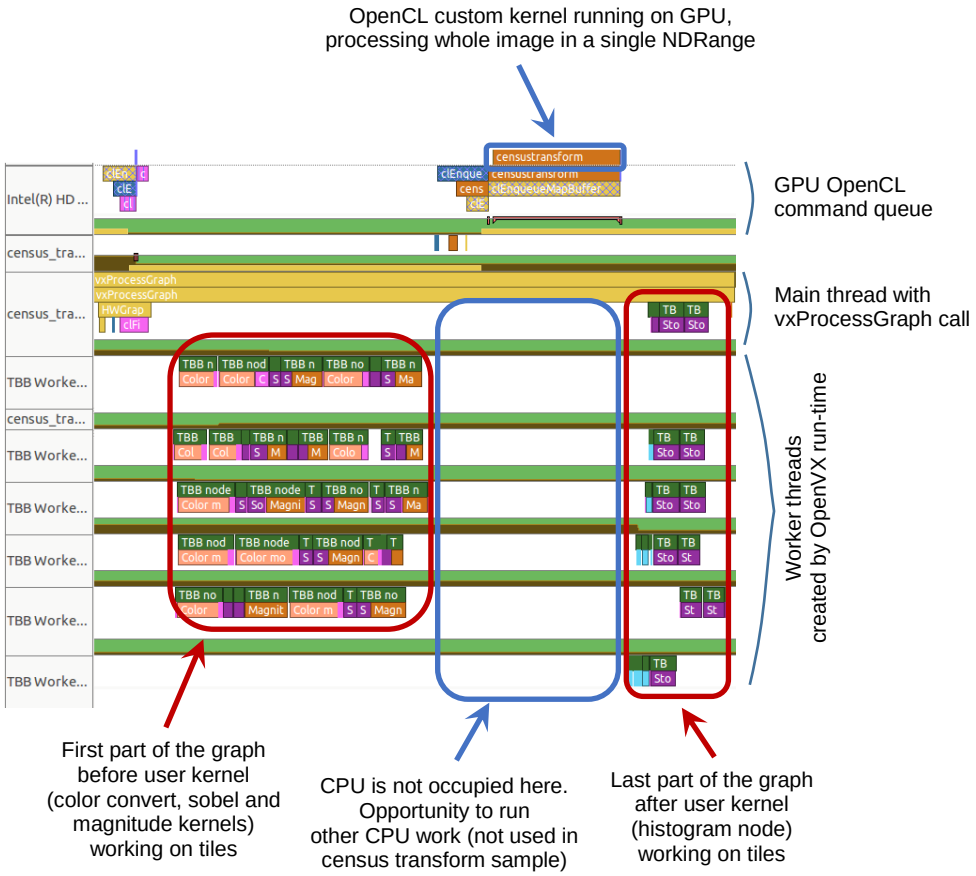
Using OpenCL custom kernel, part of the graph execution is offloaded to GPU. It may have two main sources of potential improvement:

1. Kernel may run faster on GPU than on CPU even if the last one is highly utilized via multi-threading
2. When running on the GPU, extra CPU time is freed for execution other tasks if any.

In the census transform sample, there is no work on CPU to do in parallel with user kernel due to strict data dependencies in the graph. So #2 option from the list above doesn't lead to performance improvements in the sample.

Considering option #1, the resulting performance is highly dependent on the target platform. Using the exemplar Apollo Lake system, the OpenCL kernel turns to run faster than the best CPU execution, so #1 gives better performance here running ~10 ms instead of ~12 ms for the tiled C kernel implementation.

Let's see how the timeline looks like when **OpenCL custom kernel** is used (running with `--opencl --no-tiled` knobs):



Having OpenCL custom kernel running on GPU, it is worth trying to run the complete graph on GPU by defining dedicated environment variable (VX_INTEL_ALLOWED_TARGETS):

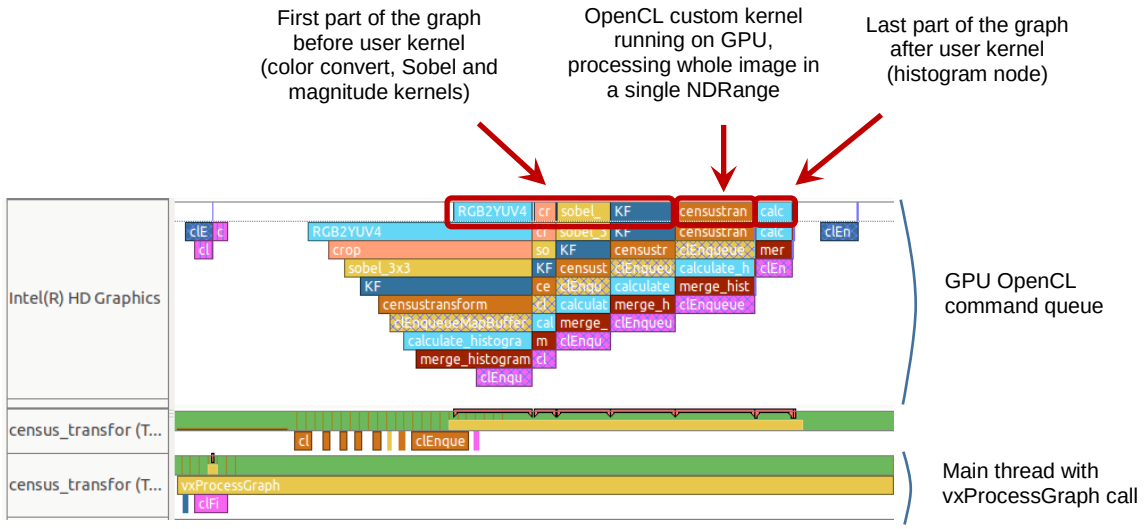
```
export VX_INTEL_ALLOWED_TARGETS=gpu

./census_transform --no-show --no-tiled --opencl

[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
Current color space is VX_COLOR_SPACE_BT709 == VX_COLOR_SPACE_DEFAULT
Switched to VX_COLOR_SPACE_BT601_625
Frames are over
Number of processed frames is : 166
7.35 ms by vxProcessGraph averaged by 166 samples
```

Execution time is reduced from ~10 ms in the mixed CPU/GPU execution to ~7 ms in pure GPU execution. It happens due to the fact that the kernels involved in the census transform graph work slightly faster on GPU than on CPU on our exemplar system. For other kernels, graphs and platforms it may have an opposite effect.

The following picture is illustration of how timeline looks like when all nodes run on GPU:

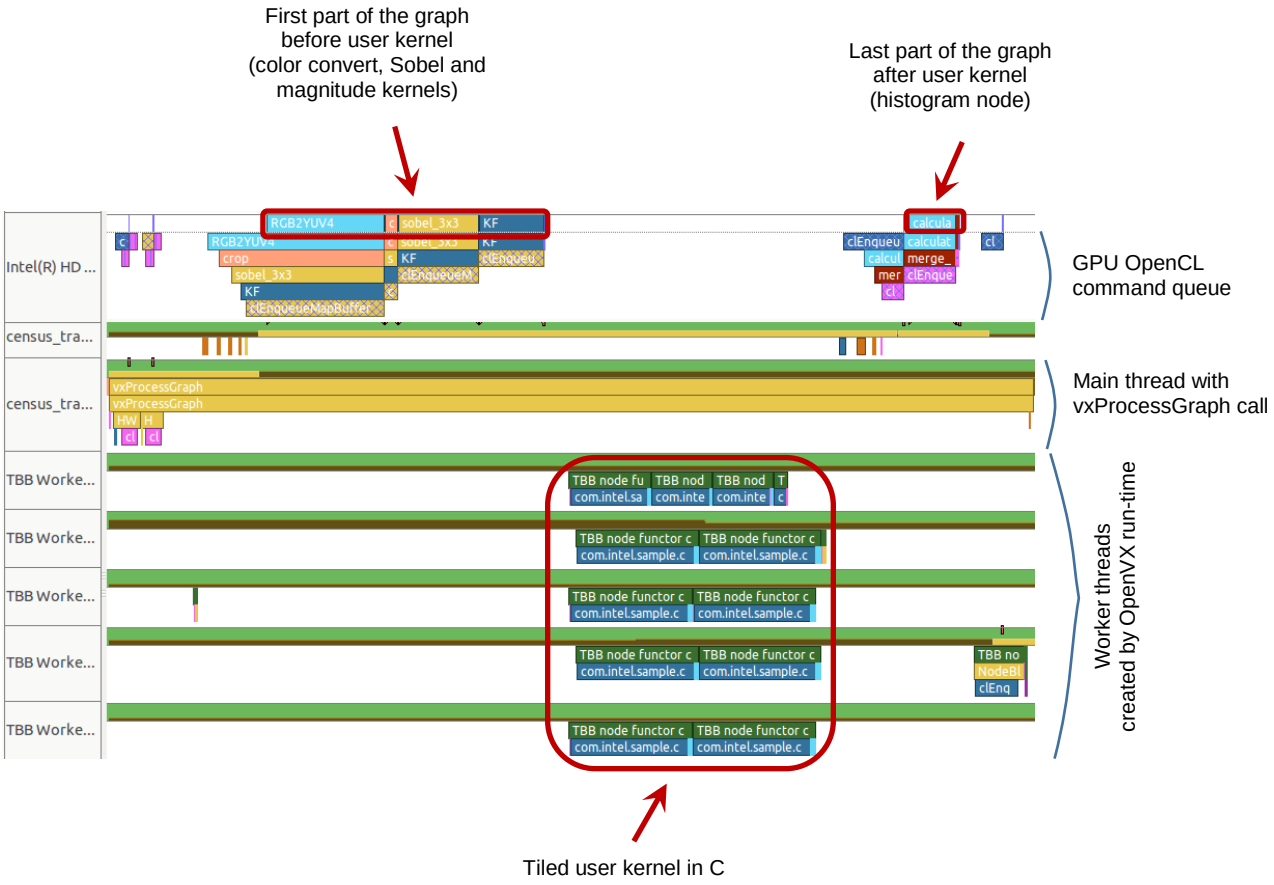


Let's see that having OpenCL user kernel is critical to achieve the best GPU results in our case. Offload all the nodes in the graph except user node to the GPU. For the user node, the best tiled version of CPU kernel is used. As it is seen from timing results below, such distribution of the nodes between CPU and GPU leads to much poorer performance due to higher overhead from extra synchronization between CPU and GPU, and less efficient CPU kernel in comparison to GPU kernel. Let's see what's happening in this case:

```
export VX_INTEL_ALLOWED_TARGETS=gpu

./census_transform --no-show

[ INFO ] OpenCL build options: -DWORK_ITEM_XSIZE=4 -DWORK_ITEM_YSIZE=1
Current color space is VX_COLOR_SPACE_BT709 == VX_COLOR_SPACE_DEFAULT
Switched to VX_COLOR_SPACE_BT601_625
Frames are over
Number of processed frames is : 166
13.35 ms by vxProcessGraph averaged by 166 samples
```



Conclusion

- Using tiled CPU user kernel unlocks multi-threaded execution and increase CPU utilization together with better cache-locality of execution.
- Running OpenCL user kernel on GPU frees CPU resources for execution of other CPU-kernels in parallel. Consider `<SDK_ROOT>/samples/video_stabilization` sample, where this effect is really exploited.
- Expressing of user functionality as OpenCL user kernel unlocks opportunity to run the complete graph on GPU that gives the best performance result in our case.

References

1. OpenVX* 1.1 Specification
2. Intel® Computer Vision SDK Developer Guide
3. Vision Algorithm Designer Developer Guide
4. The OpenVX™ User Kernel Tiling Extension
5. “C4: A Real-Time Object Detection Framework”, Jianxin Wu, Nini Liu, Christopher Geyer, and James M. Rehg, IEEE Transactions on Image Processing, VOL. 22, NO. 10, OCTOBER 2013
6. “A non-parametric approach to visual correspondence”, R. Zabih and J. Woodfill, IEEE Transactions on Pattern Analysis and Machine intelligence, 1996.
7. Instrumentation and Tracing Technology APIs