Color Copy OpenVX* Sample

Developer Guide

Intel® Distribution of OpenVINO™ Toolkit – Samples

Contents

Legal Information

Introduction

This sample shows the implementation of a “Color Copy” pipeline (specific to the **Printing Imaging** domain) using OpenVX*. The “Color Copy” pipeline is a workload that would typically be used within a multi-function printer (MFP). At the topmost level, it consists of two “subgraphs”. The first, which is referred to as the “Scan Pre-Process” subgraph, accepts raw 10-bit or 12-bit uncalibrated RGB (Red, Green, and Blue) frames as input, and applies Gain/Offset correction as well as Skew Correction (small-angle rotation), producing the 8-bit “calibrated” RG B images. The second subgraph, which is referred to as the “RGB-to-CMYK” subgraph, takes the output from the “Scan Pre-Process” subgraph and produces a “bitonal” CMYK (Cyan, Magenta, Yellow, Black) image to be sent to a printer.

This sample is an example of an OpenVX* graph with sufficiently large number of nodes in order to show the benefits of heterogeneous processing using CPU and GPU.

Brief Introduction to OpenVX*

OpenVX* is a standard from Khronos*, offering a set of optimized primitives for low-level image processing and computer visions primitives. OpenVX* is a specification for writing a code that is portable 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.

Color Copy Pipeline – The “Scan Pre-Process” OpenVX* Graph

Figure 1 presents a high-level diagram for the “Scan Pre-Process” OpenVX* graph implemented in the sample:

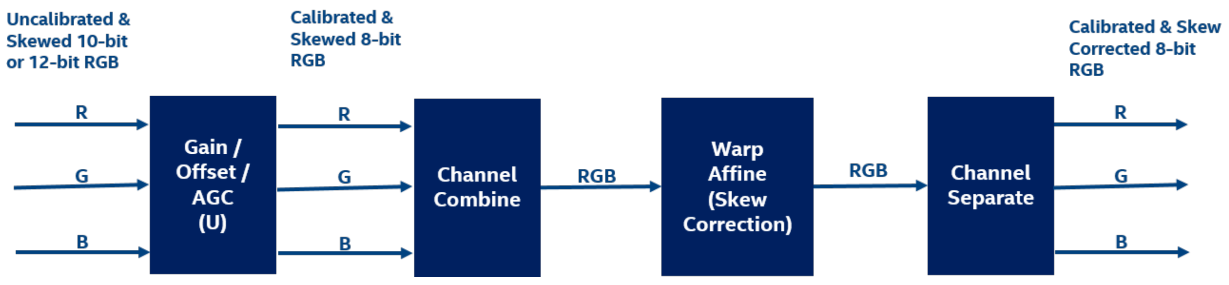

Uncalibrated & Skewed 10-bit or 12-bit RGB
R
G
B
Gain / Offset / AGC (U)
Calibrated & Skewed 8-bit RGB
R
G
B
Channel Combine
RGB
Warp Affine (Skew Correction)
RGB
Channel Separate
Calibrated & Skew Corrected 8-bit RGB
R
G
B

Figure 1: Top-level structure of “Scan Pre-Process” OpenVX* graph implemented in the sample.

The input to the “Scan Pre-Process” OpenVX graph is packed 10-bit or 12-bit uncalibrated and skewed RG B images. Since the input image provided by the user (using command line option `--input` or `-i`) is an 8bpp RGB image, the uncalibrated and skewed 10-bit or 12-bit images, which are used as input to the graph, are synthetically generated using the following process:

1. The input image provided by a user is skewed using the angle specified by `--sppskew <degrees>`. For example, specifying `--sppskew 3.0` will generate the skewed image by rotating the user's image counter-clockwise by 3 degrees, as illustrated below. The rotation operation is done using OpenVX WarpAffine node with BiCubic interpolation.

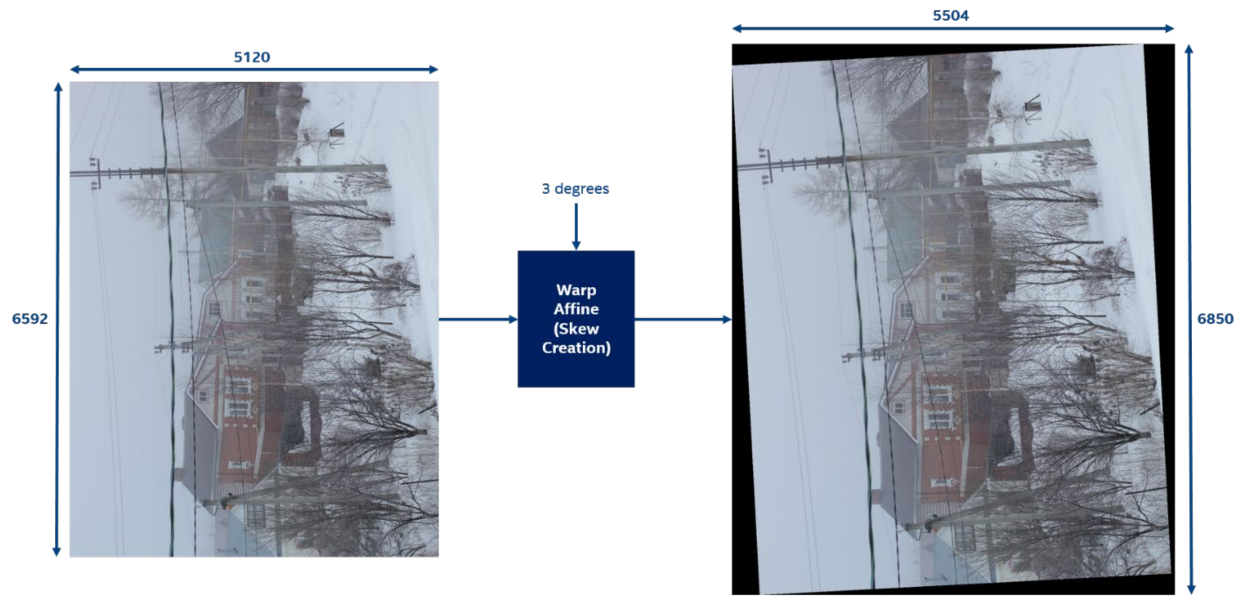
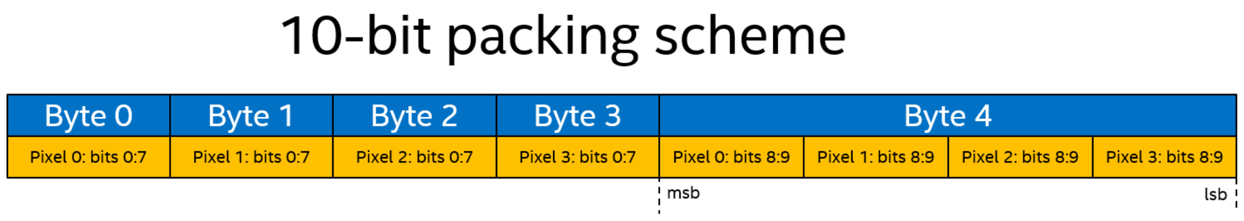


Figure 2: Illustration of applying skew to user image.

2. The skewed image created in step 1 above is converted to a packed 10-bit or 12-bit image. If you specify `--sppbits 10`, a packed 10-bit image is created. If you specify `--sppbits 12`, a packed 12-bit image is created. The packing scheme used for 10-bit and 12-bit is specified in the Figures 3 and 4 below.



10-bit packing scheme

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4			
Pixel 0: bits 0:7	Pixel 1: bits 0:7	Pixel 2: bits 0:7	Pixel 3: bits 0:7	Pixel 0: bits 8:9	Pixel 1: bits 8:9	Pixel 2: bits 8:9	Pixel 3: bits 8:9
				msb			lsb

Figure 3: Illustration of packing scheme used to pack 10 bit/pixel images into the bytestream. Every 4 sequential 10-bit pixels are packed into 5 sequential bytes. The first 4 bytes hold the lower 8 bits of each of the 10-bit pixels. The last byte holds the upper 2 bits of each of the 10-bit pixels.

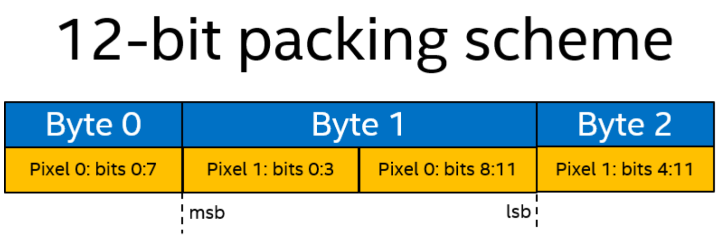


Figure 4: Illustration of packing scheme used to pack 12 bit/pixel images into the bytestream. Every 2 sequential 12-bit pixels are packed into 3 sequential bytes. The first byte holds the lower 8 bits of the first 12-bit pixel. The upper 4 bits of the second byte hold the lower 4 bits of the second 12-bit pixel. The lower 4 bits of the second byte hold the upper 4 bits of the first 12-bit pixel. The third byte holds the upper 8 bits of the second 12-bit pixel.

During the conversion, a random gain and offset arrays are generated to use for the correction process. The size of the gain and offset arrays is equal to the width of the skewed image. When the “Scan Pre-Process” graph is executed, the Gain / Offset advanced tiling node will first unpack the 10-bit or 12-bit packed input data to a floating point temporary result and apply the following formula to produce the 8-bit “calibrated” result:

output_8[x] = (input[x] * gain[x] + offset[x]) * agc

where:

- *input[x]* is an unpacked pixel at position X for the current line
- *gain[x]* is the corresponding gain value for position X
- *offset[x]* is the corresponding offset value for position X
- *agc* (Automatic Gain Correction) is a page constant value

In order to have the Gain / Offset node generate (close to) the original image, the inverse of the above formula is applied to the image during the 8-bit to 10-bit / 12-bit package image conversion process. The formula applies a random gain and offset to column of the image for each R, G, and B plane, synthetically producing an “uncalibrated” image that looks as follows:

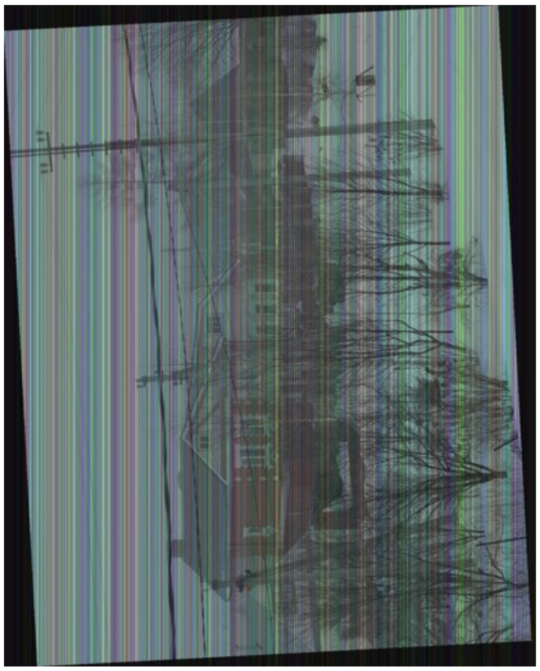


Figure 5: Synthetically generated "uncalibrated" & skewed 10-bit / 12-bit image to be used as input to "Scan Pre-Process" graph.

This synthetically generated "uncalibrated" and skewed 10-bit / 12-bit image is used as an input to the "Scan Pre-Process" graph. The Scan Pre-Process Graph, shown in Figure 1 above, reverses the steps taken to produce the "uncalibrated" input image.

1. The "uncalibrated" and skewed 10-bit / 12-bit packed input R, G, and B planes are passed as input to the Gain / Offset node(s), producing a calibrated (but still skewed) RGB image.
2. The skewed image is passed to the OpenVX Warp Affine node to correct the given skew using BiCubic interpolation.

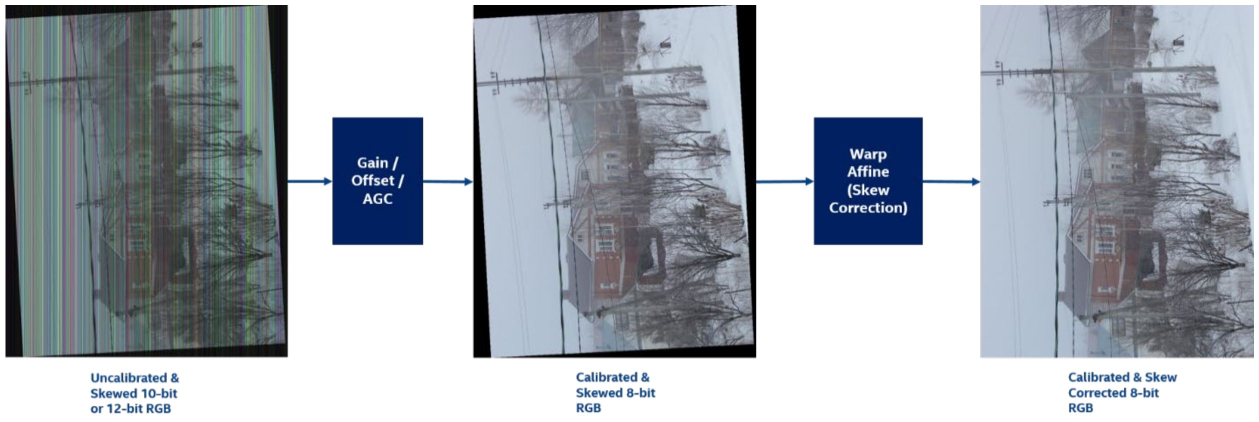

Uncalibrated & Skewed 10-bit or 12-bit RGB
Gain / Offset / AGC
Calibrated & Skewed 8-bit RGB
Warp Affine (Skew Correction)
Calibrated & Skew Corrected 8-bit RGB

Figure 6: "Scan Pre-Process" flow from input uncalibrated / skewed image, to calibrated / skew corrected output image.

Color Copy Pipeline – The "RGB-to-CMYK" OpenVX* Graph

Figure 7 presents a high-level diagram for the "RGB-to-CMYK" OpenVX* graph implemented in the sample:

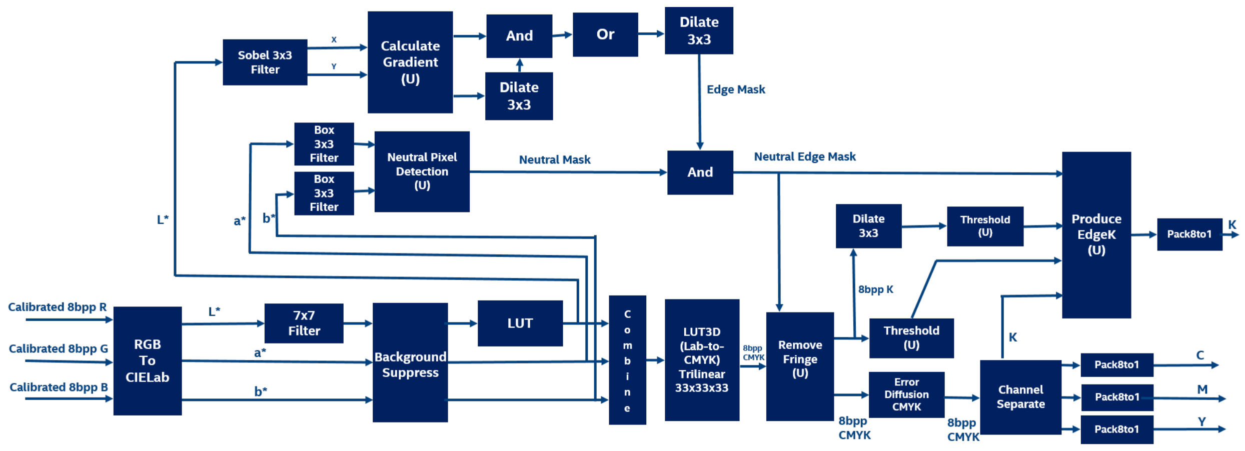


Figure 7: Top-level structure of "RGB-to-CMYK" OpenVX* graph implemented in the sample.

The input to the "RGB-to-CMYK" OpenVX graph is the "raw" calibrated 8bpp device-dependent RGB images. Such images are generated as follows:

1. A device-dependent RGB image is converted to a device-independent color space, CIELab. For this, `vxLUT3DNodeIntel` is used with the interpolation type set to `VX_INTERPLATION_TYPE_TETRAHEDRAL_INTEL`. CIELab is preferable to use as an intermediary between RGB and CMYK because this color space is divided into a luminance (L*) and chrominance (a*/b*) components. Many operations, such as filtering, can be applied only to the L* channel, thus we can save processing time as compared to applying a filter directly to RGB or CMYK. The other advantage is that you can easily partition device-dependent portions of the pipeline from other device dependent portions of the pipeline. For example, if you want to switch to a new scanner, you only need to generate a new table to convert RGB of the new scanner to CIELab. Or if you want to switch to a different printer, you would only need to set a new table to convert CIELab to the CMYK of the new printer.
2. Once the image is converted to CIELab, the L* component is passed through `vxSymmetrical7x7FilterNodeIntel` mostly to boost sharpness, but it has other general image quality (IQ) enhancing properties.
3. The filtered L* and the chrominance components (a* / b*) are passed through `vxBackgroundSuppressNode`. This kernel compares the CIELab image with a series of thresholds and determines for each pixel if its value is "pure white" or "pure black", and if so, adjusts the image accordingly.
4. The L* is passed through `vxTableLookupNode` to apply the desired lightness / darkness / contrast transform.
5. The L* output of `vxTableLookupNode`, along with the a* and b* outputs from `vxBackgroundSuppressNode` are passed into the *Neutral Edge Mask Generation* subgraph. The following diagram shows the expanded subgraph:

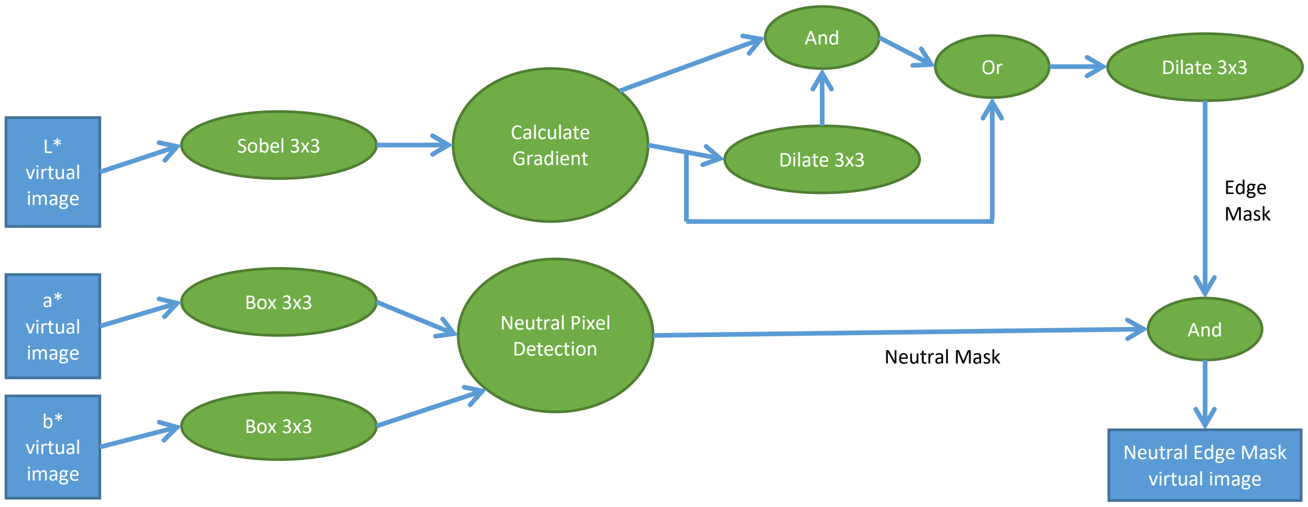


Figure 8: Neutral Edge Generation subgraph.

The Neutral Edge Generation subgraph consists of two components:

- The first component generates an *Edge Mask*. This image is comprised of pixels that are either 0 or 255. Pixels that are 255 indicate that the corresponding pixel in the CIELab image represents an 'edge'. It accomplishes this with a "lightweight" version of canny edge detection using `vxSobel3x3Node`, `vxCalculateGradientNode`, `vxDilate3x3Node`, `vxAndNode`, and `vxOrNode`.
- The second component generates a *Neutral Mask*. This image is comprised of pixels that are either 0 or 255. Pixels that are 255 indicate that the corresponding CIELab image is "neutral" (for example, lack of color, "grey"). To accomplish this, it passes the a* / b* components through `vxBox3x3Node`, and then through `vxNeutralPixelDetectionNode`. This node sets the output Neutral Mask pixel to 255 if a* and b* are within a certain distance from "pure neutral", and 0 otherwise.

These two components are combined using `vxAndNode` to generate the Neutral Edge Mask.

6. The CIELab image (the L* output of `vxTableLookupNode` and a*/b* outputs of `vxBackgroundSuppressNode`) is passed through `vxLUT3DNode` with the interpolation type set to `VX_INTERPOLATION_TYPE_TETRAHEDRAL` to convert CIELab to device-dependent CMYK (cyan, magenta, yellow, black).
7. This CMYK image, along with the L* input to `vxLUT3DNodeIntel`, as well as the Neutral Edge Mask, are passed to `vxRemoveFringeNode,` which produces a "cleaned-up" version of CMYK. When printing in CMYK, neutral (grey) pixels can either be represented by (somewhat) equal levels of CMY (cyan, magenta, yellow) ink, or they can be represented by "pure K", or purely black ink. For neutral edges such as black text or line art, using "pure K" for representation is usually more preferable than using CMY from an image quality perspective, because with CMY, the edges will appear to be "fuzzy". This fuzziness is called "Color Fringing". The purpose of this node is to "clean" pixels that are marked as Neutral Edge and to make sure that the output CMYK values are "pure K" (0,0,0,K). This module also produces another output 'K', which is the result of passing L* through an 'L-to-K' LUT.

8. The CMYK output of `vxRemoveFringeNode` is passed to `vxErrorDiffusionCMYKNodeIntel`. Error diffusion is an algorithm for converting an 8 bit/pixel continuous tone image (an image that has gray levels in the range 0 to 255), to a 'bitonal' image (an image that has only 2 pixel values, 0 and 1). This is required because the printing engine that is targeted in this case can either print a dot (1) or not (0). The algorithm accomplishes this by performing thresholding, and propagating the "error" (desired – printed) to neighboring pixels.

9. The C, M, and Y (cyan, magenta, and yellow) outputs of `vxErrorDiffusionCMYKNode` are then passed to `vxPack8to1Node`, which simply converts the 8 bit/pixel image whose pixel values are either 0 or 255, to a 1 bit/pixel image whose pixel values are either 0 or 1.

10. Using the K output of `vxRemoveFringeNode`, the Neutral Edge Mask, and the K output of `vxErrorDiffusionCMYKNode`, the final K output is generated using `vxSimpleThresholdNode`, `vxDilate3x3Node`, and `vxProduceEdgeKNode`. The following flow chart represents how the final K output is generated:

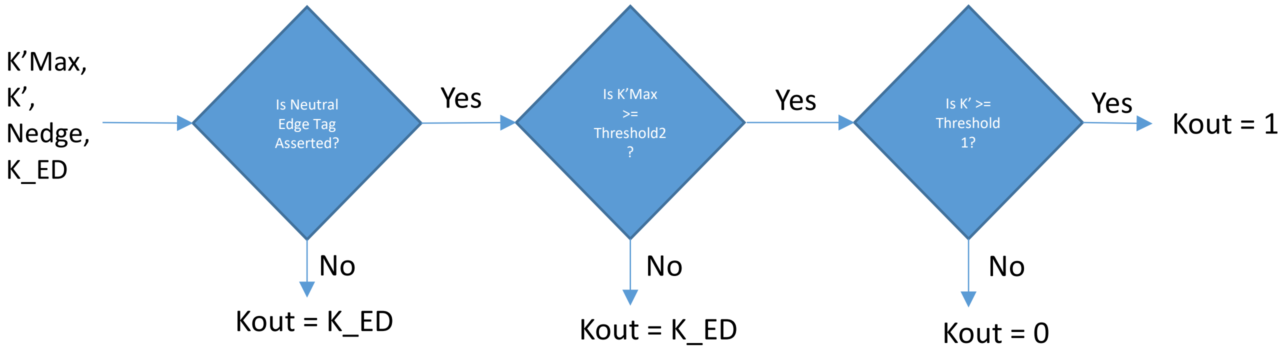


Figure 9: K channel generation.

In this flow chart, **K'Max** is the output from `vxDilate3x3Node`, **K'** is the output from `vxRemoveFringeNode`, **Nedge** is the output from the Neutral Edge Mask Generation subgraph, and **K_ED** is the K output from `vxErrorDiffusionCMYKNodeIntel`.

Kout, the output of the described 'K-selection logic' is then passed to `vxPack8to1Node`, which simply converts the 8 bit/pixel image whose pixel values are either 0 or 255, to a 1 bit/pixel image whose pixel values are either 0 or 1.

Extending OpenVX* with User Kernels

It is a common situation when 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 an 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 nine user kernels in the sample. Eight are implemented using the Intel® Advanced Tiling Extension and one of the kernels is written in OpenCL™ C. If you need a detailed step-by-step introduction to the *basics* of OpenVX tiling extensions, see the samples available in the Intel® Distribution of OpenVINO™ tookit: Census Transform sample (`<OPENVINO_ROOT>/openvx/samples/samples/census_transform`) and Custom OpenCL Kernel sample (`<OPENVINO_ROOT>/openvx/samples//samples/ocl_custom_kernel`).

Intel® Advanced Tiling Extension Kernels

- **Background Suppress Node**: Takes as input three images of type `VX_DF_IMAGE_U8` and outputs three images of type `VX_DF_IMAGE_U8`. The input/outputes are a 'planar' representation of CIELab images (L* image, a* image, b*image).

  The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel L_pixel, a_pixel, b_pixel &
// output pixel out_L, out_a, out_b
 cond1 = false;
 if( L_pixel >= 230 )
 {
   tmp = 255;
  cond1 = true;
 }
 else
 if( L_pixel <= 70 )
 {
   tmp = 0;
   cond1 = true;
 }

 if(    cond1   &&
  (a_pixel <= 138) &&
  (a_pixel >= 118) &&
  (b_pixel <= 138) &&
  (b_pixel >= 118) )
 {
    //set CIELab output to "pure white" or "pure black"
    out_L = tmp;
    out_a = 128;
    out_b = 128;
 }
 else
 {
   //set output to input
    out_L = L_pixel;
    out_a = a_pixel;
    out_b = b_pixel;
 }
```

The algorithm simply checks if the input CIELab value is either “close to white” or “close to black”, and if so, it sets the output CIELab value to “pure white” or “pure black”. This is a critical step for cleaning up the white background on a scanned image, called “background suppression”.

- **Calculate Gradient Node**: Takes as input two images of type `VX_DF_IMAGE_S16`, and produces two images of type `VX_DF_IMAGE_U8`. The input images are expected to be the output from the Sobel filter, so SobelX and SobelY, respectively. This kernel implements a “lightweight” gradient calculation, and produces two output images. The first output image is the gradient passed through threshold 1, and the second output image is the output image passed through threshold 2. While the output images are U8, each pixel will be either 0 or 255.

  The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel input_X, input_Y & output pixel output_0,
// output_1
gradient = abs(input_X )+ abs(input_Y);

if( gradient > thresh1 ) output_1 = 255;
else output_1 = 0;

if( gradient > thresh2 ) output_2 = 255;
else output_2 = 0;
```

  This kernel is a crucial part of the Edge Mask generation subgraph explained above.

- **Neutral Pixel Detection:** Takes as input two images of type `VX_DF_IMAGE_U8`, and produces one image of type `VX_DF_IMAGE_U8`. The input images are expected to be the a* and b* components of a CIELab image. This kernel will check if a* and b* are within a range to be considered “neutral”, or in other words, "grey". For a* and b*, "pure neutral" has pixel values of 128. So this kernel checks if both a* and b* are within 10 gray levels of 128. If the pixel is considered "neutral", the output will be set to 255, otherwise it will be set to 0.

The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel 'input_a', 'input_b' & output pixel 'output'
if( ((input_a > 116) && (input_a < 138)) &&
    (input_b > 116) && (input_b < 138))
{
  output = 255;
}
else
{
  output = 0;
}
```

- **Remove Fringe:** Takes as input three images: two of type `VX_DF_IMAGE_U8` and one of type `VX_DF_IMAGE_RGBX`, as well as a 16-element array of element type vx_uint8. This kernel produces two output images: one of type `VX_DF_IMAGE_RGBX` and another of type `VX_DF_IMAGE_U8`. The following is a brief description of the input and outputs:
  - **Input 0, vx_image(VX_DF_IMAGE_RGBX):** The contone CMYK output from the Lab→CMYK color space conversion.
  - **Input 1, vx_image(VX_DF_IMAGE_U8):** The L* image that was passed into the Lab→CMYK color space conversion.
  - **Input 2, vx_image(VX_DF_IMAGE_U8):** The Neutral Edge Mask (values 0 or 255), indicating which pixels are "neutral edges". 255 means that the pixel is a neutral edge, 0 means that it is not. This image is the output of the Neutral Edge Generation subgraph.
  - **Input 3, vx_array(vx_uint8):** 16 subsampled points of a larger 256-entry LUT, which is intended to convert L* to K (black) based on the characteristics of the printer in use. This 16-entry LUT is used as opposed to a full 256-entry LUT, as you can use SIMD instructions to interpolate K = L_to_K[L] much faster than serially passing a given pixel value through a full LUT.
  - **Output 0, vx_image(VX_DF_IMAGE_RGBX):** CMYK output where the "Color Fringe" around "neutral edges" has been removed.
  - **Output 1, vx_image(VX_DF_IMAGE_U8):** K output. Simply the result of K = L_to_K[L].

When printing in CMYK, neutral (grey) pixels can either be represented as somewhat equal levels of CMY (cyan, magenta, and yellow) ink, or they can be represented by "pure K", meaning purely black ink. For neutral edges such as black text or line art, using "pure K" for representation is much more desirable than using CMY, because with CMY, the edges will appear to be "fuzzy". This fuzziness is called "Color Fringing".

The purpose of this node is to “clean” pixels marked as “Neutral Edge” and to make sure that the output CMYK values are “Pure K” (0,0,0,K).

The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel ‘input_CMYK’, ‘input_L’, ‘input_nedge_mask’
// and output pixel output_CMYK, output_L2K

output_L2K = L_to_K[inputL];
if( input_nedge_mask )
{
  output_CMYK.c = 0;
  output_CMYK.m = 0;
  output_CMYK.y = 0;
  output_CMYK.k = output_L2K;
}
else
{
  output_cmyk = input_cmyk;
}
```

- **Simple Threshold:** This kernel takes as input an image of type `VX_DF_IMAGE_U8`, as well as a threshold value (scalar of type uint8), and outputs an image of type `VX_DF_IMAGE_U8`. This kernel performs a very simple threshold operation.

  The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel ‘input’, and output pixel ‘output’
if( input >= thresh )
  output = 255;
else
  output = 0;
```

- **Produce Edge K:** This kernel takes as input four images and produces a single output image, all of type `VX_DF_IMAGE_U8`. The following is a description of the expected inputs and outputs. Note that all inputs and outputs are 8-bits, but they are “bitonal”, meaning that their pixel values will either be 0 or 255.
  - **Input 0:** The “Neutral Edge Mask” (values 0 or 255), indicating which pixels are “neutral edges”. 255 means that the pixel is a neutral edge, 0 means that it is not. This image is the output of the “Neutral Edge Generation” subgraph.

   - **Input 1:** Output of 'Remove Fringe' "L-to-K" operation passed through 'Simple Threshold'. We can call this image thresh1.
   - **Input 2:** Output of 'Remove Fringe' "L-to-K" operation, passed through a max filter, and then passed through 'Simple Threshold'. We can call this image thresh2.
   - **Input 3:** K-extracted output of CMYKErrorDiffusion.
   - **Output 0:** The final K output.

  The purpose of this kernel is to combine the K output from error diffusion with the K outputs of a series of thresholds into the "final" K output, which will be sent to the printer.

  The pixel processing loop implemented for this kernel is an SSE optimized representation of the following pseudo-code:

```
//For each input pixel nedge_mask, thresh_1, thresh_2, k_ed
// & each output pixel k_out

if( thresh2 && nedge_mask )
{
  k_out = thresh1;
}
else
{
  k_out = k_ed
}
```

- **Pack 8-to-1:** This kernel takes as input an image of VX_DF_IMAGE_U8 or VX_DF_IMAGE_RGBX type and produces an output image of the same type as the input. The input image is expected to be "bitonal", meaning that every pixel is explicitly 0 or 255. The purpose of this kernel is to convert this "bitonal" 8 bit/pixel input image to a "packed" 1 bit/pixel output. Basically, if the input pixel is 0, the output pixel is 0. If the input pixel is 255, the output pixel is 1.

  The Intel Advanced Tiling Extension was chosen to implement this kernel because currently there is no OpenVX image format which maps to a 1bpp image. So, `VX_DF_IMAGE_U8` is chosen, but the output width is set to 1/8 of the input width. This cannot be achieved with the standard "OpenVX User Kernel Tiling Extension", as the output image must be equal to the input image in that case.

Intel® Custom OpenCL™ Kernels

Symmetrical 7x7 Filter: This kernel takes as input an image of `VX_DF_IMAGE_U8` type and produces an output image of the same type as input. The following spatial filtering patterns are implemented:

- **JIHGHIJ**
- **IFEDEFI**
- **HECBCEH**
- **GDBABGD**
- **HECBCEH**
- **IFEDEFI**
- **JIHGHIJ**

Border mode processing is identical to `VX_BORDER_CONSTANT` with the constant value 255. This implementation functionality is similar to `vxSymmetrical7x7FilterNodeIntel`, which is used in the sample for processing on CPU/IPU. The new device kernels `vxSymm7x7OpenCLNode` and `vxSymm7x7OpenCLTiledNode` allows offloading this functionality on GPU with bit-exact identical result.

All the kernels above are compiled in the separate `libcolor_copy_pipeline_lib.so` file and loaded to the main application by the `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 loaded 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 `vxAddAdvancedTilingKernelIntel`, `vxAddDeviceKernelIntel`, passing parameter validation, and processing callbacks and some other parameters. Refer to the `vxPublishKernels` function defined in `vxpublishkernels.cpp` and corresponding `Publish*Kernel` functions implementation in user kernels `vx*.cpp` files for details.

Besides the module, you should provide `vx` functions to be used during graph construction, or 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` functions can be found in the `vx_user_pipeline_nodes.h` file and they are defined in corresponding `vx_*_lib.cpp` files.

Intel® Custom OpenCL™ Tiled Kernel Specific.

Additionally to the non-tiled `vxSymm7x7OpenCLNode` user node, this sample implements tiled version `vxSymm7x7OpenCLTiledNode`, which produces exactly the same output result but allows for achieving additional performance gain due to parallel data processing. Non-tiled kernels implementation are described in details in the Custom OpenCL Kernel sample (`<OPENVINO_ROOT>/openvx/samples/samples/ocl_custom_kernel`). This chapter describes specifics of the tiled kernel implementation. For implementation details of tiled and non-tiled kernels, refer to `vxsymm7x7_opencl_impl.cl`. Tiled kernel signature contains additional arguments related to tiles.

```
kernel void symm7x7tiled_opt(

       // The kernel signature is completely defined by param_types array passed to
       // vxAddDeviceKernelIntel 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 VX_DF_IMAGE_U8 image tile
       global const uchar* inImgPtr,


       int in_tile_x,         // x coordinate of the tile
       int in_tile_y,         // y coordinate of the tile
       int in_tile_width,     // width of tile
       int in_tile_height,    // height of tile


       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

       // The first parameter of OpenVX kernel has type vx_image, it is mapped to these 5
       OpenCL // kernel arguments
       // This is output VX_DF_IMAGE_U8 image tile


       global uchar*       outImgPtr,


       int out_tile_x,         // x coordinate of the tile
       int out_tile_y,         // y coordinate of the tile
       int out_tile_width,     // width of tile
       int out_tile_height,    // height of tile


       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
       )
{
    . . .
}
```

Please notice that data pointers correspond to the starting location of the tile, not to the starting location of the full image. Pixels that reside outside of the given tile should not be accessed. Both tiled and non-tiled kernels contain the following code, which checks image boundary:

```
int x = get_global_id(0);   // 0..(widthInImg/VX_OPENCL_WORK_ITEM_XSIZE_INTEL-1)
int y = get_global_id(1);   // 0..(heightInImg/VX_OPENCL_WORK_ITEM_YSIZE_INTEL-1)


// coordinates of a top-left pixel in output image area for processing by one WI
// some of the pixels from WORK_ITEM_XSIZE x WORK_ITEM_YSIZE area are not valid on
// the border
// they will not be processed (look at if's in the code below)
x *= WORK_ITEM_XSIZE;
y *= WORK_ITEM_YSIZE;

…

if (
       ((out_tile_x + x) > 2) &&
       ((out_tile_x + x) < widthOutImg - 3) &&
       ((out_tile_y + y) > 2) &&
       ((out_tile_y + y) < heightOutImg - 3)
       )
{

       //non border pixels processing
       …
}
```

The kernel implements a 7x7 spatial filter. As a result, you need to access three neighbor pixels to the left, right, top, and bottom of each processed pixel. To avoid accessing unutilized data, you need to inform OpenVX runtime that such neighbor pixel access is required. The runtime determines the required size and location of the input tile given an output tile. So input tiles will contain actual data behind tile borders. Please refer to `PublishSymm7x7OpenCLKernel` function in `vxsymm7x7_opencl_module.cpp`. Here is the code snippet demonstrating such attribute set up:

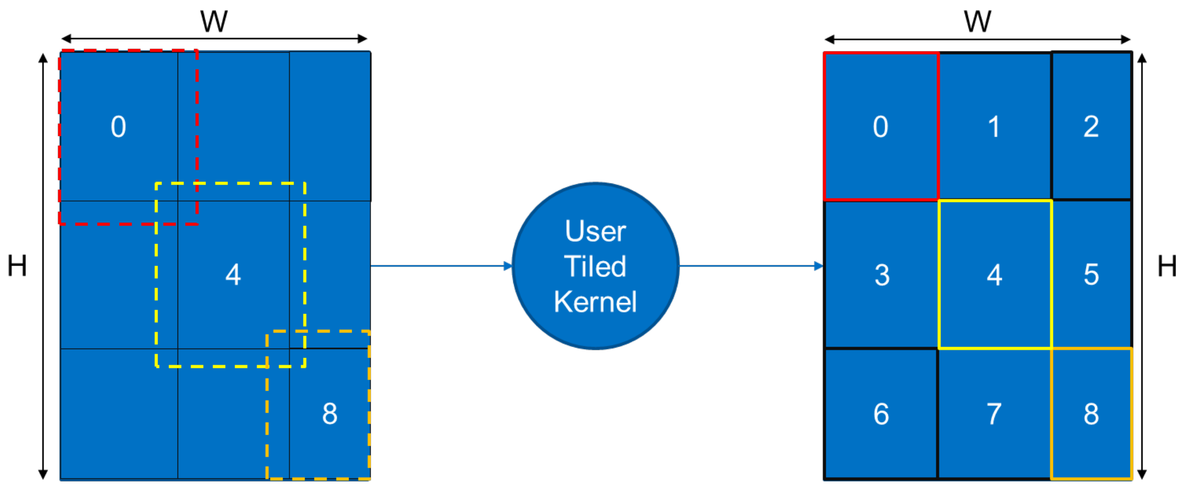
```
vx_neighborhood_size_intel_t n;
n.top = -3; //requires 3 additional input pixel above
n.left = -3; //requires 3 additional input pixels to the left
n.right = 3; //requires 3 additional input pixels to the right
n.bottom = 3; //requires 3 additional input pixels below

vxSetKernelAttribute(oclTiledKernel,
    VX_KERNEL_INPUT_NEIGHBORHOOD_INTEL,
    &n,
    sizeof(vx_neighborhood_size_intel_t)
);
```

To be more specific, given an output tile (x,y,width,height), the required input tile is calculated from the vx_neighborhood_size_intel_t struct in the following way:

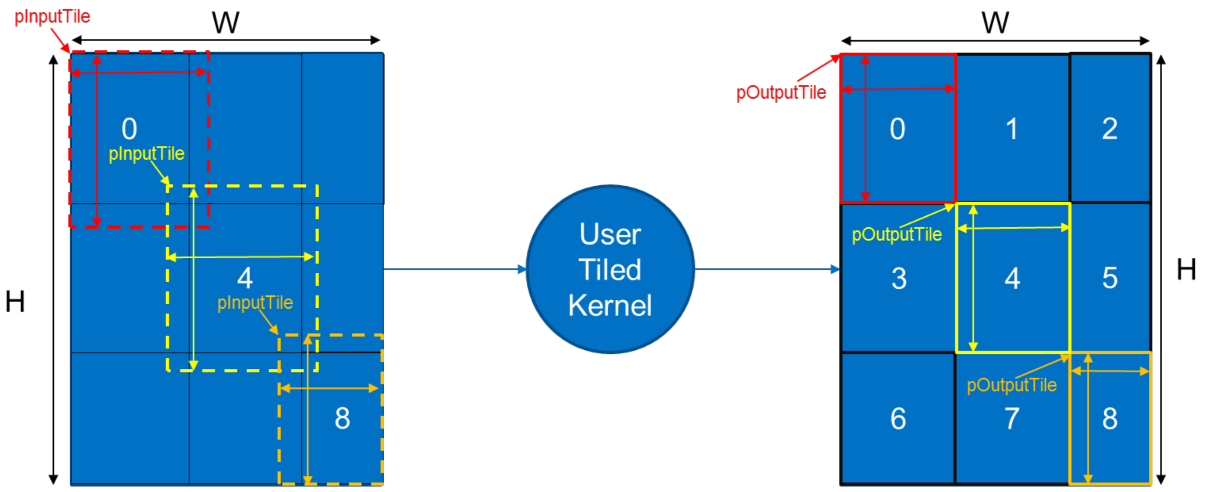
```
input_tile.x = max(output_tile.x + neighborhood.left, 0);
input_tile.y = max(output_tile.y + neighborhood.top, 0);
input_tile.width = min(output_tile.width + (output_tile.x – input_tile.x) +
neighborhood.right + input_tile.x, input_image.width) - input_tile.x;
input_tile.height= min(output_tile.height + (output_tile.y – input_tile.y) +
neighborhood.bottom + input_tile.y, input_image.height) - input_tile.y;
```

For example, to implement a 7x7 filter, the user would set {left=-3, right=3, top=-3, bottom=3}. The input tiles (for output tiles 0, 4, and 8) would be defined as follows. Note of the clipping along the border of the input image.



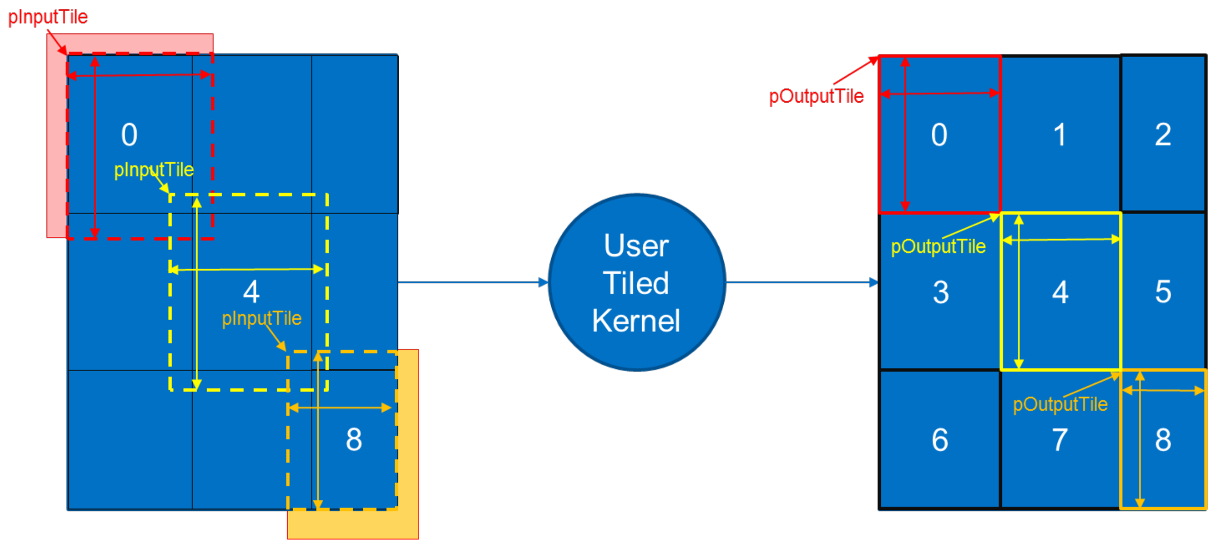
At execution time, the kernel function is called to produce each output tile given the previously defined input tile. In the specific case of our OpenCL plugin, this means that `enqueueNDRange` is called to produce each output tile. For each call to the kernel function, the following parameters will be passed:

```
output tile attributes (x, y, width, height)
input tile attributes (x, y, width, height)
output tile starting pointer (and stride)
input tile starting pointer (and stride)
input and output image dimensions (width, height)
```



For many tiles, there is not enough of “valid” input to actually produce the requested output tile. For example, this is true for all output tiles in the picture below, except for #4. This means that the kernel is responsible for doing some “special” depending on the input / output tile locations (in other words, the kernel becomes “spatially” dependent). You can decide how to handle such special, but it can be tricky as in many cases, it will involve branching as well as calculating a specific kind of border (replicate, constant, etc.).

A common approach is to detect when a pixel that needs to be accessed resides outside of the given input tile, and to replace that access with some border pixels (see shaded regions along the boundary of input tile #0 and #8 below). Also, notice that for input tile #1 below, `pInputTile`, the starting location of the tile, corresponds to (x,y) of (0,0) and matches to the location of the output tile (0,0), which `pOutputTile` will be pointing to. But for the tile #4, `pInputTile` and `pOutputTile` point to different (x,y) coordinates. For this reason, some pointer adjustment will be required depending on input / output tile location.



The following code snippet illustrates such pointer adjustment implemented as a part of `symm7x7tiled_opt` tiled kernel:

```
…
int y_shift = out_tile_y - in_tile_y;
```

```
…

for (int yi = y; yi < y + ySize; ++yi)
{
        for (int xi = x; xi < x + xSize; ++xi)
        {
                //Start src offset
               int srcOffset0 = (yi-3+y_shift)*rowPitchInImg+(xi-3)*pixelStrideInImg;
               int srcOffset1 = (yi-3+y_shift)*rowPitchInImg+(xi-2)*pixelStrideInImg;
               int srcOffset2 = (yi-3+y_shift)*rowPitchInImg+(xi-1)*pixelStrideInImg;
               int srcOffset3 = (yi-3+y_shift)*rowPitchInImg+(xi)*pixelStrideInImg;
               int srcOffset4 = (yi-3+y_shift)*rowPitchInImg+(xi+1)*pixelStrideInImg;
               int srcOffset5 = (yi-3+y_shift)*rowPitchInImg+(xi+2)*pixelStrideInImg;
               int srcOffset6 = (yi-3+y_shift)*rowPitchInImg+(xi+3)*pixelStrideInImg;

                int dstOffset = yi*rowPitchOutImg+xi*pixelStrideOutImg;
…
      }
}
```

Using Targets API for Heterogeneous Computing

Intel® Distribution of OpenVINO™ toolkit provides the API to schedule parts of an OpenVX* graph to different compute units (for example, CPU, GPU or IPU). In the context of the OpenVX*, such compute unit is named *target*. The set of available targets depends on a platform used to run the application.

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

Intel Distribution of OpenVINO toolkit extends OpenVX* with a provisional Target API for assigning selected nodes in an OpenVX* graph to a particular target, overwriting the default run-time choice. This API enables heterogeneous usages of Intel® platforms, which is a way to better hardware utilization.

For better understanding of the Targets API experimental feature, see OpenVX* Heterogeneous Basic sample (`<OPENVINO_ROOT>/openvx/samples/samples/hetero_basic`).

NOTE: In contrast to the Heterogeneous Basic sample, which shows you how to use the Targets API in source code, the Color Copy sample provides a handy tool that allows switching targets for nodes through command line options. It enables you to easily experiment with different mappings of nodes to targets by just supplying different command line options. It does not require code re-compilation.

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

The following list introduces heterogeneous command line options and mapping between

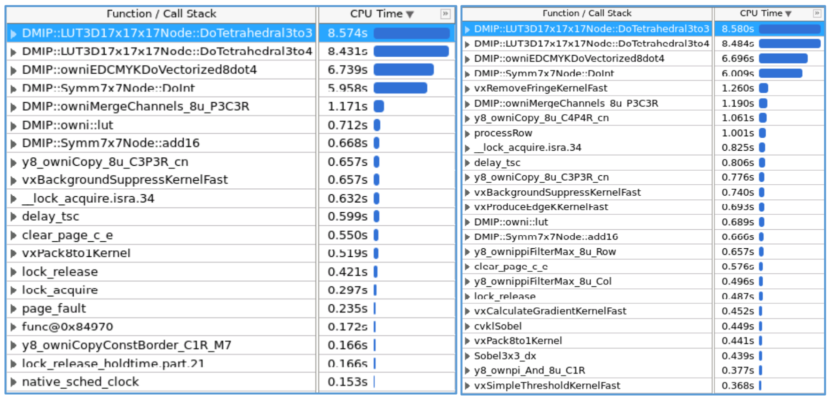
nodes and targets:

```
--gpurgb2lab for vxLUT3DNodeIntel (performing RGB2Lab conversion) offloading on intel.gpu
--gpurgb2labchcombine for vxChannelCombineNode offloading on intel.gpu
--gpuremovefringe for vxRemoveFringeOpenCLTiledNode offloading on intel.gpu
--ipurgb2lab for vxRgbToLabNodeIntel (performing RGB2Lab conversion) offloading on
intel.ipu
--gpulab2cmyk for vxLUT3DNodeIntel (performing Lab2CMYK conversion) offloading on
intel.gpu
--gpulab2cmykchcombine for vxChannelCombineNode offloading on intel.gpu
--ipusymm7x7 for vxSymmetrical7x7FilterNodeIntel offloading on intel.ipu
--gpuboxfilter for vxBox3x3Node (part of Edge Detection (ED)) offloading on intel.gpu
--gpusobelfilter for vxSobel3x3Node (part of ED) offloading on intel.gpu
--gpudilateandor for vxDilate3x3Node, vxAndNode and vxOrNode (part of ED) offloading on
intel.gpu
--gpusymm7x7 for vxSymm7x7OpenCLTiledNode or vxSymm7x7OpenCLNode offloading on intel.gpu
(alternative to vxSymmetrical7x7FilterNodeIntel available on CPU and IPU)
--gpulut for vxTableLookupNode offloading on intel.gpu
--gpuskew for vxWarpAffine (used for skew correction) offloading on intel.gpu
```

NOTE: The sample reports detailed error string if a node is not supported on the target device or the target device is absent on a platform.

Basic Performance Analysis and Heterogeneous Execution Options

Intel® VTune™ Amplifier performance profiler can be employed for color copy pipeline hotspots analysis. The following screenshots illustrate execution time breakdown collected with VTune™ on the Intel® Pentium® processor N4200/5, N3350/5, N3450/5 with Intel® HD Graphics for the simplest color copy pipeline implementation (high1) and for the most advanced and complicated (high6).



Function / Call Stack	CPU Time ▼
DMIP::LUT3D17x17x17Node::DoTetrahedral3to3	8.574s
DMIP::LUT3D17x17x17Node::DoTetrahedral3to4	8.431s
DMIP::owniEDCMYKDoVectorized8dot4	6.739s
DMIP::Symm7x7Node::DoInt	5.958s
DMIP::owniMergeChannels_8u_P3C3R	1.171s
DMIP::owni::lut	0.712s
DMIP::Symm7x7Node::add16	0.668s
y8_owniCopy_8u_C3P3R_cn	0.657s
vxBackgroundSuppressKernelFast	0.657s
__lock_acquire.isra.34	0.632s
delay_tsc	0.599s
clear_page_c_e	0.550s
vxPack8to1Kernel	0.519s
lock_release	0.421s
lock_acquire	0.297s
page_fault	0.235s
func@0x84970	0.172s
y8_owniCopyConstBorder_C1R_M7	0.166s
lock_release_holdtime.part.21	0.166s
native_sched_clock	0.153s

Function / Call Stack	CPU Time ▼
DMIP::LUT3D17x17x17Node::DoTetrahedral3to3	8.580s
DMIP::LUT3D17x17x17Node::DoTetrahedral3to4	8.484s
DMIP::owniEDCMYKDoVectorized8dot4	6.696s
DMIP::Symm7x7Node::DoInt	6.009s
vxRemoveFringeKernelFast	1.260s
DMIP::owniMergeChannels_8u_P3C3R	1.190s
y8_owniCopy_8u_C4P4R_cn	1.061s
processRow	1.001s
__lock_acquire.isra.34	0.825s
delay_tsc	0.806s
y8_owniCopy_8u_C3P3R_cn	0.776s
vxBackgroundSuppressKernelFast	0.740s
vxProduceEdgeKKernelFast	0.693s
DMIP::owni::lut	0.689s
DMIP::Symm7x7Node::add16	0.666s
y8_ownippiFilterMax_8u_Row	0.657s
clear_page_c_e	0.576s
y8_ownippiFilterMax_8u_Col	0.496s
lock_release	0.487s
vxCalculateGradientKernelFast	0.452s
cvklSobel	0.449s
vxPack8to1Kernel	0.441s
Sobel3x3_dx	0.439s
y8_ownpi_And_8u_C1R	0.377s
vxSimpleThresholdKernelFast	0.368s

Figure 3: high1 (left) and high6 (right) color copy pipeline execution time breakdown.

The tables below provide reference for standard nodes, extension nodes, and custom user nodes used in the Color Copy Pipeline sample:

Node Name	Originated	CPU	GPU	IPU
vxAndNode	Khronos	x	x	
vxBox3x3Node	Khronos	x	x	x
vxChannelCombineNode	Khronos	x	x	
vxChannelSeparateNodeIntel	Intel Ext.	x		
vxDilate3x3Node	Khronos	x	x	x
vxErrorDiffusionCMYKNodeIntel	Intel Ext.	x		
vxLUT3DNodeIntel	Intel Ext.	x	x	
vxOrNode	Khronos	x	x	
vxSobel3x3Node	Khronos	x	x	x
vxSymmetrical7x7FilterNodeIntel	Intel Ext.	x		x
vxTableLookupNode	Khronos	x	x	

Table 1: Standard and extension nodes and their availability on CPU, GPU and IPU targets. Green highlight means possible nodes offload implemented as a part of the current version of the sample.

Custom Node Name	Intel Tiling
vxBackgroundSuppressNode	x
vxCalculateGradientNode	x
vxNeutralPixelDetectionNode	x
vxPack8to1Node	x
vxProduceEdgeKNode	x
vxRemoveFringeNode	x

vxSimpleThresholdNode	x
vxUnpack1to8Node	x

Table 2: User nodes profile for tiling.

`vxLUT3DNode`, `vxErrorDiffusionCMYKNodeIntel`, and `vxSymmetrical7x7FilterNodeIntel` are the main hotspots according to VTune analysis (Figure3). User nodes like `vxPack8to1Node`, `vxBackgroundSuppressNode`, and `vxRemoveFringeNode` introduce noticeable impact as well. However, it is not so big in comparison with the previously described hotspots.

In the current implementation, all user nodes are available on CPU only. Hotspot `vxErrorDiffusionCMYKNodeIntel` extension node is supported on CPU only (table 1). At the same time, the top hotspot `vxLUT3DNodeIntel` can be offloaded on GPU. The quite noticeable `vxSymmetrical7x7FilterNodeIntel` can be offloaded on IPU or on GPU as `vxSymm7x7OpenCLTiledNode` user node. After high-level performance analysis using VTune, the following heterogeneous scenarios (which take into account nodes availability on target devices) have emerged:

- Offloading the `vxLUT3DNodeIntel` to the GPU with the `--gpurgb2lab --gpulab2cmyk` command line options
- Offloading the `vxSymmetrical7x7FilterNodeIntel` to the IPU with the `--ipusymm7x7` command line option
- Combination of the above two with the `--gpurgb2lab --gpulab2cmyk --ipusymm7x7` command line options
- Offloading the `vxSymm7x7OpenCLTiledNode` to the GPU with the `--gpusymm7x7` command line option
- Combination of scenario #1 and scenario #4 with the `--gpurgb2lab --gpulab2cmyk --gpusymm7x7` command line options

The current version of the sample provides possibility of other heterogeneous scenarios like `vxSobel3x3Node` offloading on GPU (`--gpusobelfilter`) as well as some other standard nodes (refer to the previous chapter for the full list of options). However, taking into account that such nodes are not among top hotspots, offloading on GPU would be hardly beneficial.

The sample functionality can be easily extended with remaining nodes offloading on GPU and IPU. For example, `vxChannelCombineNode` can be offloaded on GPU as well as `vxBox3x3Node`, which can be offloaded on IPU. The existing nodes offloading code can be used as a reference. For simplicity reasons, all these possible options were not implemented as part of this sample.

Building the Sample

See the root `README` file for all samples and another `README` file located in the `color_copy_pipeline` 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. Provide command line parameters to control its behavior. To get the complete list of command line parameters, run:

```
$ ./color_copy_pipeline --help
```

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

The sample requires explicit choice of underlying algorithm implementation. To choose the algorithm, use one of the following command line options: `--high1`, `--high3`, or `--high6`. These options refer to the complexity of the used "RGB-to-CMYK" graph. --high6 is recommended to generate adequate image quality for neutral lines & text.

By default, the sample reads the image file `low_contrast_5120x6592_I444.raw` located in the current sample directory. Calling sample without optional parameters:

```
$ ./color_copy_pipeline --high1
```

opens `low_contrast_5120x6592_I444.raw` and performs processing. Output performance statistics will be provided as console output. Here is an output example for the Intel® Pentium® processor N4200/5, N3350/5, N3450/5 with Intel® HD Graphics:

```
$ ./color_copy_pipeline --high1
Internal Buffers Allocated = 713564467 bytes.
PPM = 98
196.38 ms by vxVerifyGraph averaged by 1 samples
611.51 ms by vxProcessGraph averaged by 10 samples
```

It provides information about allocated memory in bytes (graph info queried from the OpenVX* runtime) and time spent on OpenVX graph verification and graph processing in milliseconds. The specific image printing metric PPM (Pages per Minute) is provided as well.

Specifying `--output` command line instructs the sample to write output 1bpp CMYK binary image on a disk:

```
 $ ./color_copy_pipeline --high1 --output out.cmyk
```

Specifying `--input` command line instructs sample to use non-default input image. The input image format must be 5120x6592 RAW RGB (8bpp):

```
 $ ./color_copy_pipeline --high1 --input rgb.raw
```

You can try various heterogeneous scenarios using the command line options described in the Using Targets API for Heterogeneous Computing chapter. For example, the following command line instructs the sample to offload `vxLUT3DNodeIntel` nodes functionality on GPU and `vxSymmetrical7x7FilterNodeIntel` node functionality on IPU:

```
$ ./color_copy_pipeline --high1 --gpurgb2lab --gpulab2cmyk --ipusymm7x7
```

The following command line options adjust lightness and contrast parameters of the underlying color copy pipeline algorithm:

```
$ ./color_copy_pipeline --high1 --lightness 1 --contrast -1
```

You can tweak a number of CPU working threads to compare performance or estimate scalability:

```
$ ./color_copy_pipeline --high1 --nthreads 1
```

The tile height (number of scanlines) for tiling extension will impact performance as well:

```
$ ./color_copy_pipeline --high1 --tileheight 32
```

You can change a number of algorithm iterations using `--max-frames` that specifies a number of loops over the color copy pipeline algorithm applied to a same input image. This option may be useful for estimating color copy algorithm performance stability on the current platform:

```
$ ./color_copy_pipeline --high1 --max-frames 100
```

You can disable tiled execution on GPU by specifying the following command line option `--clnontiled`. There is no performance benefits from tiling disabling for this particular sample. But you can estimate overall performance gain due to tiled execution in various heterogeneous scenarios.

```
$ ./color_copy_pipeline --high1 --clnontiled
```

Scan Pre-Process Command Line Options

Unless you add SPP (Scan Pre-Process) command line options, the “Scan Pre-Process” content will not be executed. In this case, the input image will be passed directly to the input of the “RGB-to-CMYK” graph. The SPP options are as follows:

```
    --sppseparate
            Run SPP(Scan Pre-Process), but separate from main copy graph

    --sppconnected
            Run SPP(Scan Pre-Process) as part of the main copy graph

    --sppskew  Skew angle. The input image will be skewed (rotated) using
            this, SPP will correct it. (Default value: 0)

    --gpuskew  Offload skew correction (vxWarpAffine) to GPU

     --sppbits <integer>
            Number of bits to create uncalibrated R, G. B with. 10 or 12
            are supported. (Default value: 0)
```

To run the “Scan Pre-Process” content, specify `--sppseparate` or `--sppconnected` option:

- If you use `--sppseparate`, a distinct `vx_graph` is created for the “Scan Pre-Process” stage and executed separately from the main “RGB-to-CMYK” graph. This allows you to profile and experiment with SPP separately.

- If you use `--sppconnected`, the "Scan Pre-Process" and "RGB-to-CMYK" subgraphs will be added to a single `vx_graph`, connected by virtual `vx_image`, and the entire graph will be executed with each call to `vxProcessGraph`.

If you specify one of these options, you must also set at least one of the two options: `--sppbits` or `--sppskew`:

- Use `--sppbits` to specify the use 10-bit or 12-bit packed "uncalibrated" video as input to SPP.
- Use `--sppskew` to specify the angle to use to skew the input image, which then is corrected as part of SPP. If you set this option, you can also optionally set `--gpuskew` to run the WarpAffine transform operation using the GPU.

SPP Examples

Run SPP separately from the main "RGB-to-CMYK" graph using a skew angle of 3.0 and 10-bit packed "uncalibrated" input:

```
$ ./color_copy_pipeline --input low_contrast_5120x6592_I444.raw --high1 --sppseparate --
sppskew 3.0 --sppbits 10
```

Run the same test, but offload warp affine transform (skew correction) to the GPU:

```
$ ./color_copy_pipeline --input low_contrast_5120x6592_I444.raw --high1 --sppseparate --
sppskew 3.0 --sppbits 10 --gpuskew
```

Run SPP using 12-bit packed "uncalibrated" video and connect it to the main "RGB-to-CMYK" graph:

```
$ ./color_copy_pipeline --input low_contrast_5120x6592_I444.raw --high6 --sppconnected --
sppskew 3.0 --sppbits 12 --gpuskew
```

You can also add *only* `--sppbits` or `--sppskew` options to run / profile these operations by themselves.

Run using SPP graph, which is populated with *only* Gain/Offset nodes:

```
$ ./color_copy_pipeline --input low_contrast_5120x6592_I444.raw --high6 --sppseparate --
sppbits 12
```

Run SPP graph, which is populated with *only* warp affine (skew correction) nodes:

```
  $ ./color_copy_pipeline --input low_contrast_5120x6592_I444.raw --high6 --sppseparate -
-sppskew 3.0 --gpuskew
```

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 teaches how to analyze an application that uses OpenVX to find and exploit performance opportunities provided by OpenVX nodes and OpenCL kernels in various heterogeneous scenarios. Understanding of how `vxProcessGraph` works with different nodes on hetero platforms is a key to achieve better application performance.

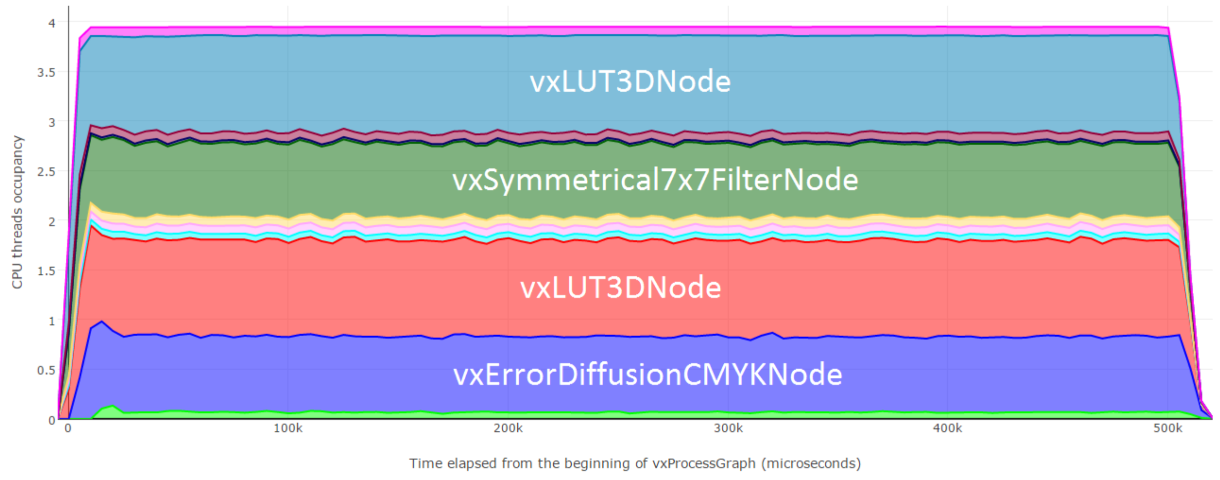
NOTE: Screenshots captured by graph profiling tool presented in this section are provided for illustrative purposes. They express brief performance picture that you may expect to see when looking at your application in Intel® VTune™ Amplifier. The screen shots are gathered on exemplar Intel® Pentium® processor N4200/5, N3350/5, N3450/5 with Intel® HD Graphics system.

Running on CPU

The following graph represents CPU occupancy by OVX nodes (averaged over multiple `vxProcessGraph` calls and stacked for all nodes) for the simplest high1 color copy pipeline graph flavor in case of non-heterogeneous graph (all nodes are executed on CPU). As in VTune clockticks breakdown in the previous chapter, you can clearly see the main hotspots:

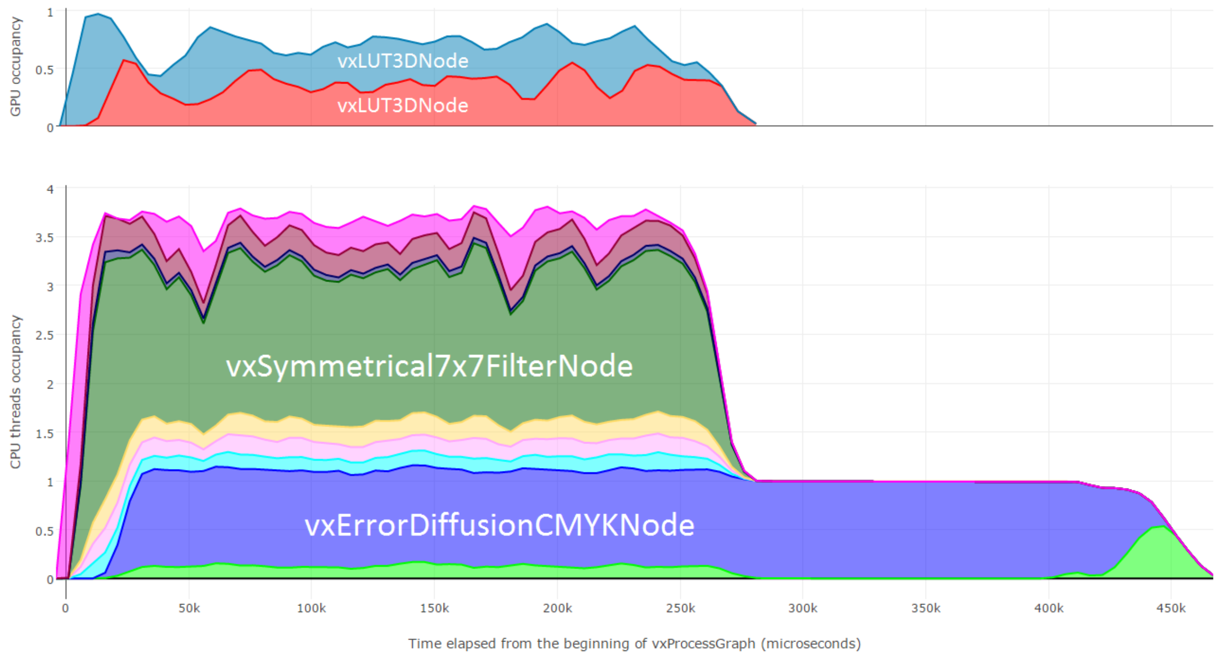
- Two instances of `vxLUT3DNode`: one for RGB2LAB conversion and another one for LAB2CMYK
- Sharp boost filter `vxSymmetrical7x7FilterNode` applied to luma channel
- Error diffusion quasi serial filter `vxErrorDiffusionCMYKNodeIntel` to convert continuous tone image to bitonal

As you can see, all these nodes, as well as other minor hotspot nodes, almost ideally occupies all four cores of Intel® Pentium® processor N4200/5, N3350/5, N3450/5 with Intel® HD Graphics system. Parallel node tiles processing is performed in balanced manner and all nodes finalize almost simultaneously as result., even quasi serial error diffusion, which has tile data dependencies and needs special processing order (from left to right, from top to bottom).

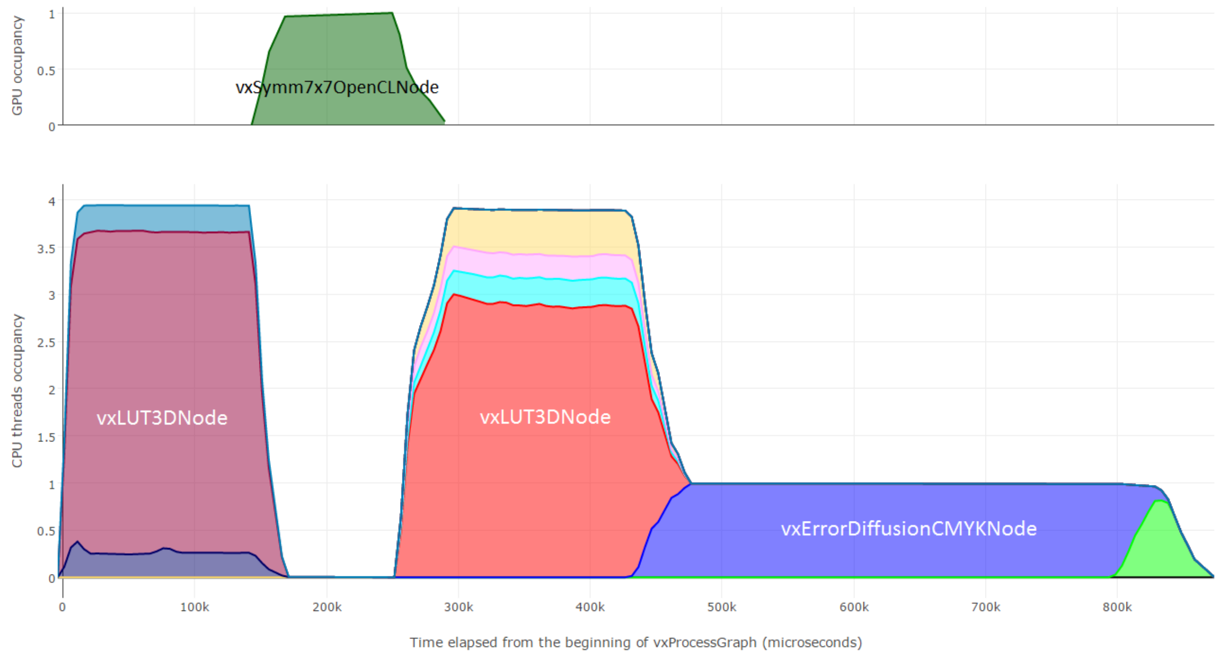


Running on GPU

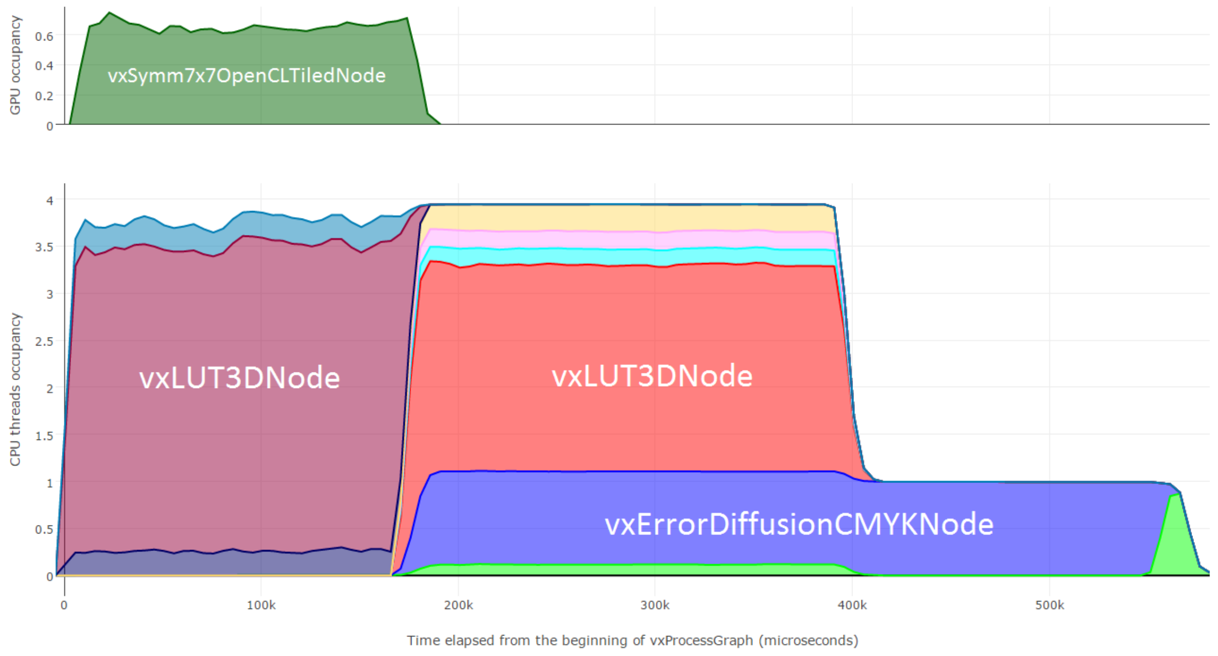
Now consider heterogeneous scenario with vxLUT3DNodeIntel offloading on GPU (`--gpurgb2lab --gpulab2cmyk`). Please notice GPU occupancy graph at the top of screenshot. As you can see we reduced CPU occupancy and added work for GPU (>50% occupancy during first 250-300 milliseconds). Again, almost all nodes perform parallel processing and finish much faster comparing to CPU-only execution. But `vxErrorDiffusionCMYKNodeIntel` depends on the second vxLUT3DNodeIntel results availability from GPU. It starts later and finishes later due to data transfer overheads and non-deterministic processed tiles order received from GPU. Anyway, the performance gain is ~10-20% depending on graph flavor due to CPU+GPU heterogeneous execution.



The next two timelines represent tiling performance effect for OpenCL device custom kernel. `--gpusymm7x7` command line option instructs the sample to use `vxSymm7x7OpenCLTiledNode` (available on GPU) instead of `vxSymmetrical7x7FilterNodeIntel` (available on CPU and IPU). Additional `--clnontiled` command line option replaces tiled node with non-tiled node implementation (`vxSymm7x7OpenCLNode`). The first graph represent non-tiled node case usage. Non-tiled implementation of OpenCL™ device kernel is slightly simpler and more straightforward comparing to the tiled version. This is why such non-tiled implementation was the first choice. The next graph demonstrates CPU/GPU occupancy timeline when serial version of OpenCL device kernel is used.



As you can see, non-tiled node stalls the whole execution pipeline. CPU becomes idle in this case. Sequential CPU nodes in OpenVX graph wait for the `vxSymm7x7OpenCLNode` finalization and output data availability on CPU to continue graph execution. The second graph demonstrates CPU/GPU occupancy timeline when tiled version of OpenCL device kernel used instead of non-tiled one:



Tiled node works in parallel with the first `vxLUT3DNodeIntel` (which performs RGB2Lab conversion) and other CPU nodes. Even though that tiled and non-tiled kernel execution is similar according to timeline, you can see significant overall performance gap (~30-40%) when comparing non-tiled and tiled cases. Tiled execution is crucial for color copy pipeline sample performance.

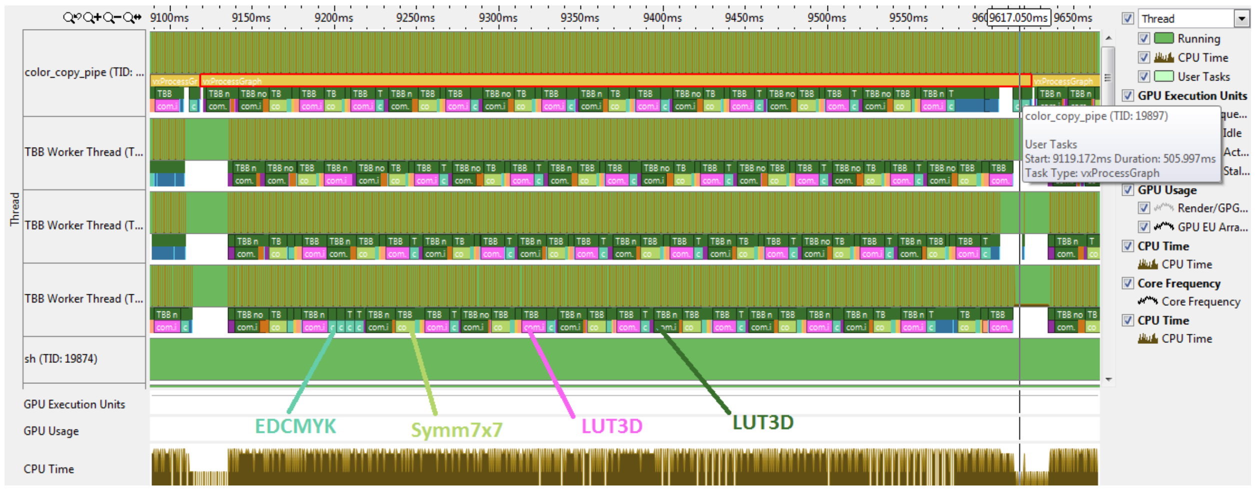
GPU+CPU Heterogenous Scenarios Analysis with Intel® VTune™ Amplifier

This section considers heterogeneous scenarios described in the Basic Performance Analysis and Heterogeneous Execution Options section and visual data that can be collected using Intel® VTune™ Amplifier. Please notice that performance analysis with VTune can impact overall performance and hotspots distribution. As a result, simple sample runs can demonstrate different performance results.

NOTE: In this section, the data collected with VTune is provided for illustrative purposes only.

The following screenshots illustrate CPU and GPU timelines with main hotspots tasks distribution focusing on the top hotspots: LUT3D, Symm7x7, and EDCMYK:

- CPU-only scenario when LUT3D, Symm7x7 and EDCMYK occupy almost 100% of CPU resources:

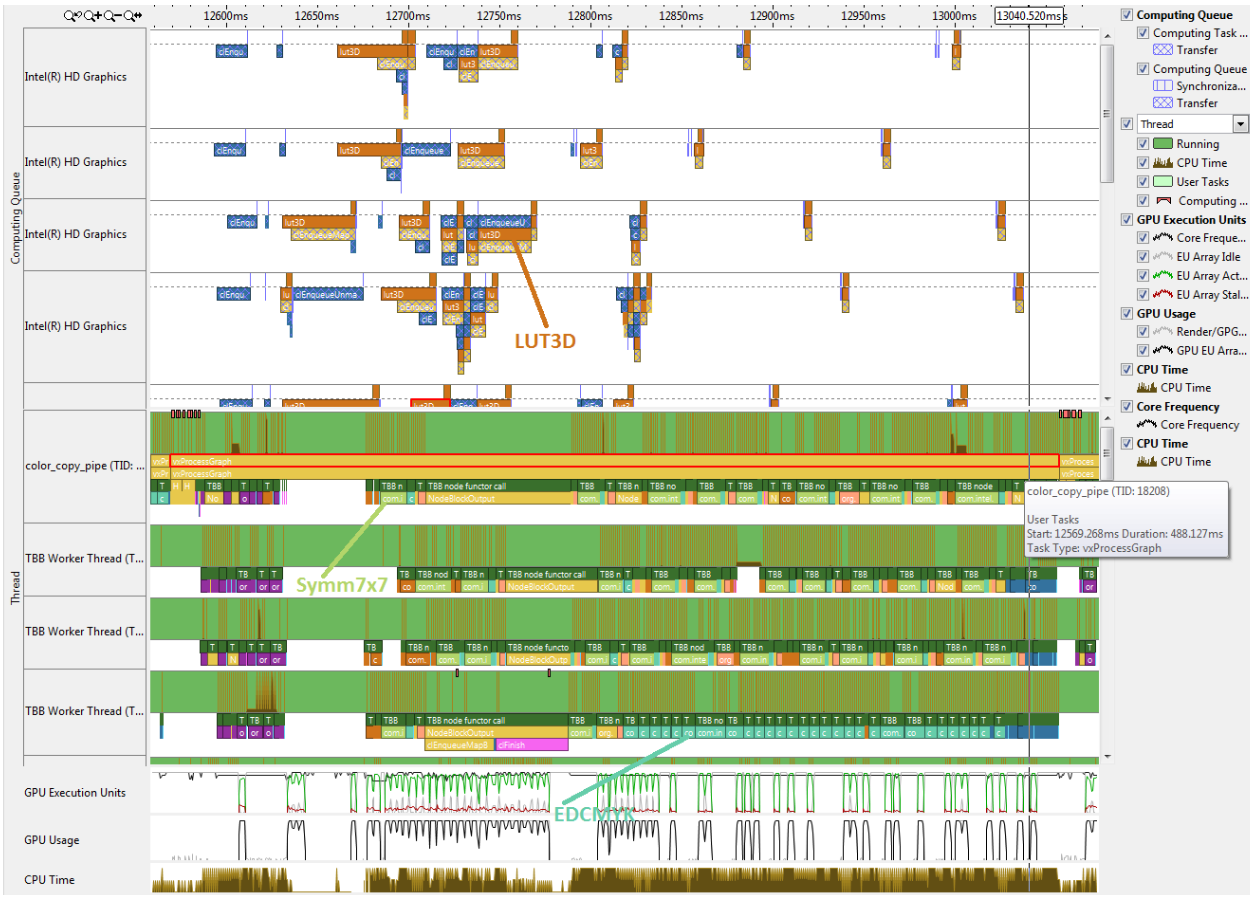


color_copy_pipe (TID: 19897)

User Tasks
Start: 9119.172ms Duration: 505.997ms
Task Type: vxProcessGraph

Here VTune reports ~506 ms taken by the vxProcessGraph call.

- LUT3D offloading on GPU. This is the scenario #1 enabled by the `--gpurgb2lab --gpulab2cmyk` command line options.

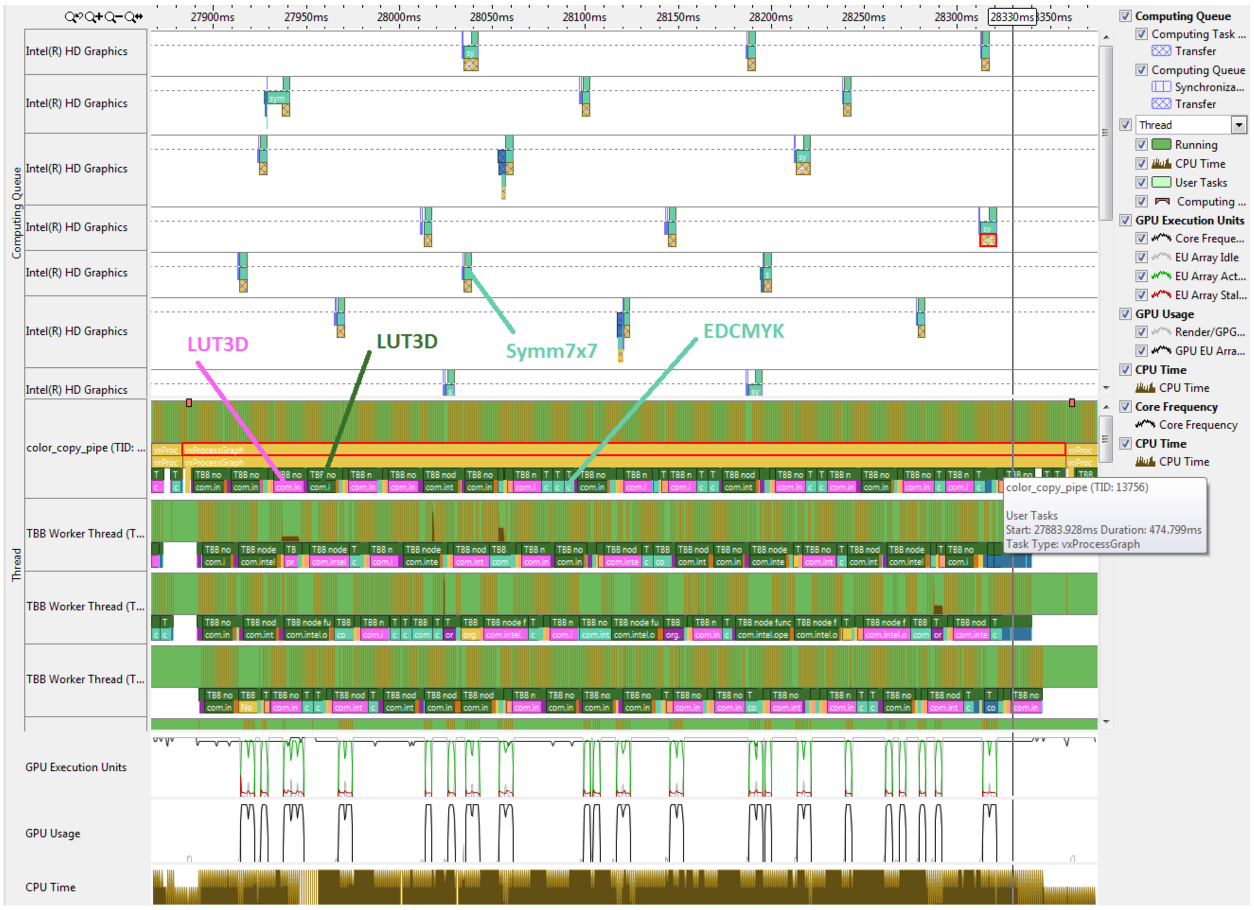


color_copy_pipe (TID: 18208)

User Tasks
Start: 12569.268ms Duration: 488.127ms
Task Type: vxProcessGraph

Here you can still see significant CPU load, but GPU execution units are loaded as well as doing LUT3D calculations in parallel with Symm7x7, EDCMYK, and other tasks performed on CPU. vxProcessGraph takes ~488ms.

- Symm7x7 offloading on GPU. Here the `vxSymm7x7OpenCLTiledNode` user node is used. This is scenario #4 enabled by the `--gpusymm7x7` command line option.

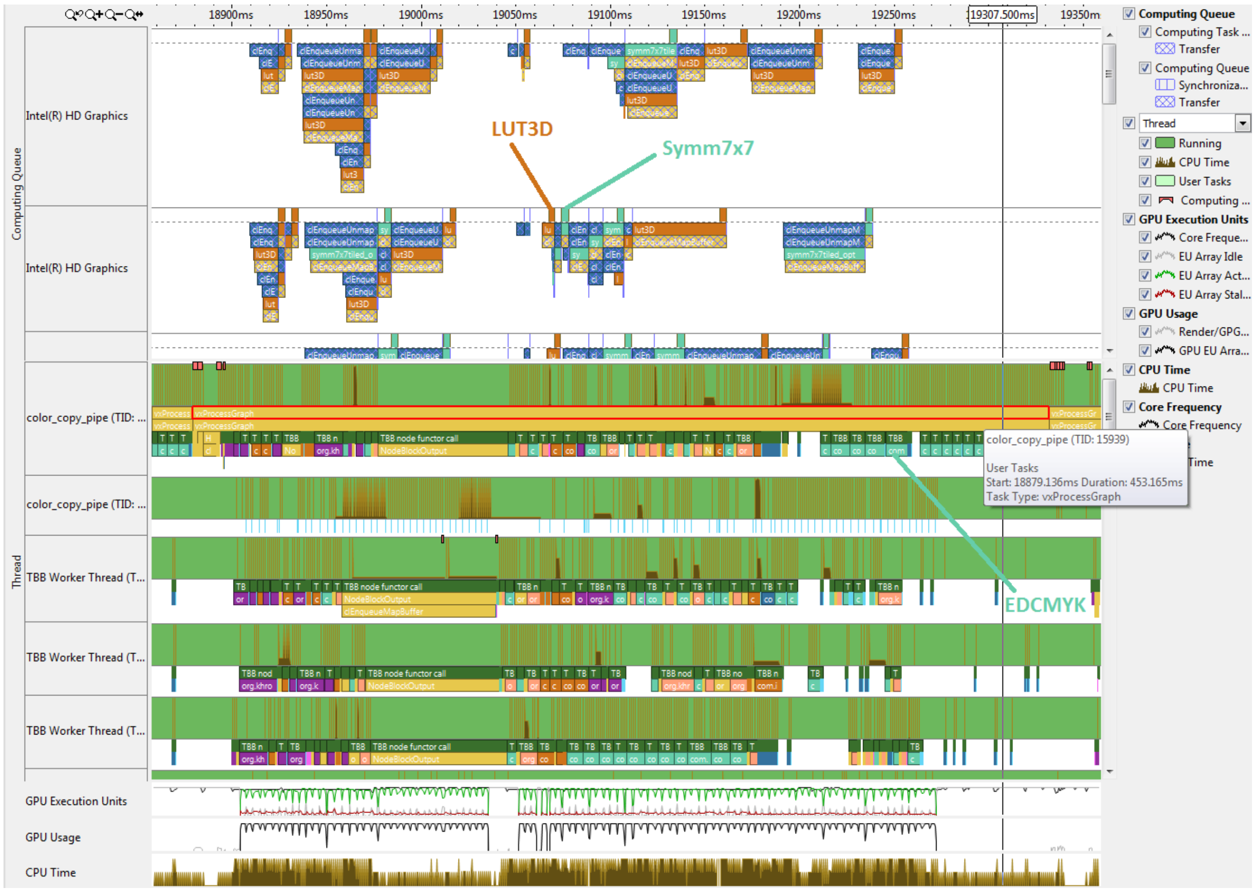


color_copy_pipe (TID: 13756)

User Tasks
Start: 27883.928ms Duration: 474.799ms
Task Type: vxProcessGraph

`vxProcessGraph` shows slightly better performance compared with the previous scenario and takes ~475 ms.

- Very high GPU occupation: EDCMYK hotspot is left on CPU and LUT3D and Symm7x7 are offloaded together on GPU. This is the scenario #5 enabled by the `--gpurgb2lab --gpulab2cmyk --gpusymm7x7` command line options.



color_copy_pipe (TID: 15939)
User Tasks
Start: 18879.136ms Duration: 453.165ms
Task Type: vxProcessGraph

Now GPU is occupied during almost all the processing time. `vxProcessGraph` takes ~453 ms, and this is the best result compared with the previously considered scenarios.

References

- OpenVX* 1.1 Specification
- Intel® Distribution if OpenVINO™ Toolkit
- OpenVX User Kernel Tiling Extension