Lane Detection OpenVX* Sample

Developer Guide

Intel® Computer Vision SDK – Samples

Contents

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

Introduction

This Lane Detection OpenVX* sample teaches how to use OpenVX* to implement simple lane detection pipeline. The sample uses extension OpenVX* node in combination with the standard ones to implement a basic algorithm that produces candidates for lane border based on top view (obtained via by perspective transformation); followed by linear filter and Hough transform operations. Please note that this is not production quality implementation but just a sample pipeline used to showcase the OpenVX* technology.

The following topics are covered in the sample

- Creation and initialization vx_matrix object to use it as parameter for vxWarpPerspectiveNode.
- Creation and initialization vx_convolution object to use it with vxConvolveNode.
- Creation and initialization vx_threshold object to use it with vxThresholdNode.
- How to use vxHoughLinesPNode to detect line segments.

If you need a detailed step-by-step introduction to the *basics* of OpenVX* development, see the *Auto Contrast sample*, available in this SDK (<SDK_ROOT>/samples/auto_contrast).

Brief Introduction to OpenVX*

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

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

Lane Detection Pipeline as an OpenVX* Graph

There are many approaches to detect lane bounds on video from a windshield-mounted camera. One, particularly similar approach [3, 4, 6] is to use pipeline based on:

1. Mapping input image into birds-eye view
2. Running a filter to detect pixels that can be part of lane markers
3. Running Hough transform algorithm to detect long lane segments.

This sample implements such pipeline type using 5 standard OpenVX* nodes and one Intel's vendor extension OpenVX* node.

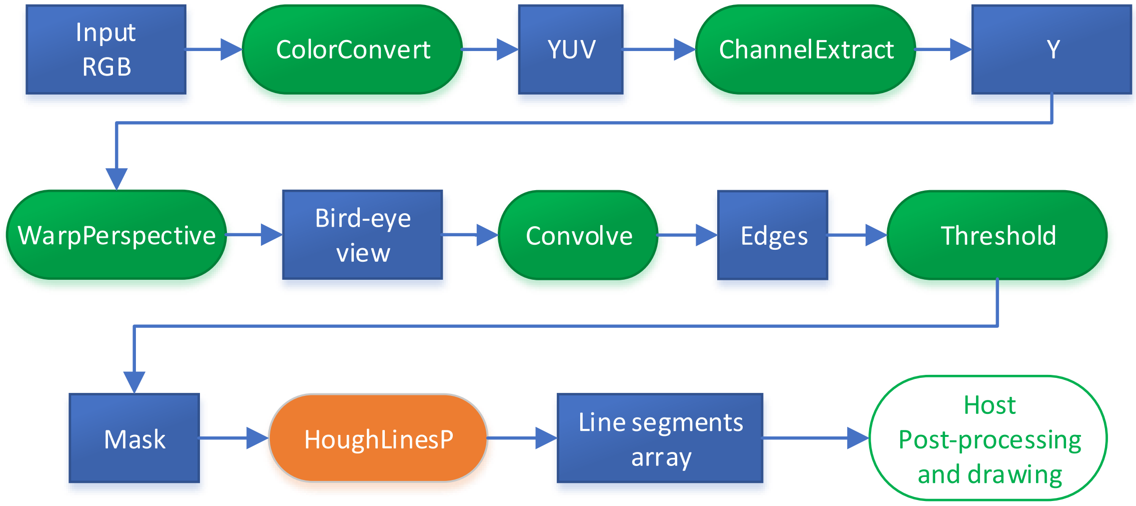


Figure 1: Lane Detection OpenVX* graph implemented in the sample. Legend: Green are standard OpenVX* nodes. Orange is Intel vendor extension node. Blue boxes are data objects in OpenVX* graph.

The complete OpenVX* graph is presented in Figure 1. In addition to OpenVX* nodes there is a final simple post processing step that aggregates line segments from Hough transform into long detected lane marks, maps result back to original image and draws them. The implementation for OpenVX* graph is located in `lane_detection.cpp` file. The additional host post-processing step is implemented in separate files (`collect_lane_marks.hpp` and `collect_lane_marks.cpp`) as `CollectLaneMarks` class.

The sample folder structure is shown below:

```
-- lane_detection
    |-- CMakeLists.txt                (CMake file for the sample)
    |-- collect_lane_marks.cpp/hpp    (C++ class that implements finalization
    |                                  the processing result)
    |-- lane_detection.cpp            (main file with OpenVX pipeline)
    |-- road_lane.mp4                 (default video file for processing)
    |-- lane_detection.graphml        (file with graph for VAD)
    |-- lane_detection_user_nodes_module.cpp
    |                                 (file to create user node for VAD)
    |-- sample_lane_detection_user_guide.pdf
    |                                 (sample documentation)
    `-- README                        (this readme file)
```

Let's consider the pipeline steps in more details

Convert from RGB to Gray image.

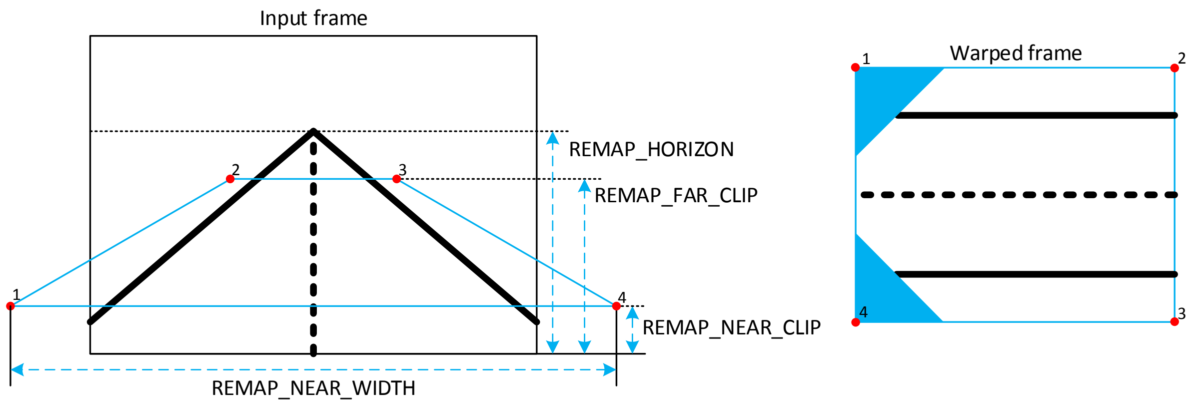
To do this operation OpenVX* provides two standard nodes. The first `vxColorConvertNode` call creates the node to convert input `ovxImgRGB` image from RGB color space into `ovxImgYUV` image in YUV color space. For further processing, only gray Y component is needed. To separate Y channel the `vxChannelExtractNode` call creates node to extract Y component and store it in `ovxImgGray` image. The images and nodes creation code is shown below:

```
vx_image ovxImgRGB = vxCreateImage(ovxContext,width,height,VX_DF_IMAGE_RGB);
vx_image ovxImgYUV = vxCreateVirtualImage(ovxGraph,0,  0,VX_DF_IMAGE_YUV4);
vx_image ovxImgGray = vxCreateVirtualImage(ovxGraph,0,  0,VX_DF_IMAGE_U8);
vxColorConvertNode( ovxGraph, ovxImgRGB, ovxImgYUV );
vxChannelExtractNode( ovxGraph, ovxImgYUV, VX_CHANNEL_Y, ovxImgGray);
```

Note that only input `ovxImgGray` image is created as non-virtual because this image is accessed by host (i.e. outside the graph). The rest of images (`ovxImgYUV` and `ovxImgGray`) are created as virtual since there is no need to access data of these images from the host. Using virtual images allows certain optimizations by a graph compiler. The sizes of virtual images omitted because they are deduced automatically by the graph compiler. In contrast, types of the virtual images (`VX_DF_IMAGE_YUV4` and `VX_DF_IMAGE_U8`) are defined explicitly because they define operations that nodes do on the images (e.g. color conversion type). More example of virtual image using can be found in video_stabilization sample

Remap an Image to get Bird-Eye view.

The OpenVX* has standard WarpPerspective node for this. It requires 3x3 matrix input to define perspective transformation. This matrix can be calculated directly from camera position and orientation relative to the road plane [7]. Another way is to point four corner points of destination and input images and calculate this matrix based on given correspondence. In the code snipped below, we call sample's calcPerspectiveTransform function that returns OpenCV cv::Mat 3x3 matrix for perspective transformation. This function calculates the correct matrix for default sample video (road_lane.mp4) based on some definitions (REMAP_HORIZON, REMAP_FAR_CLIP, REMAP_NEAR_CLIP and REMAP_NEAR_WIDTH) that have to be corrected to process alternative video (see details in lane_detection.cpp).



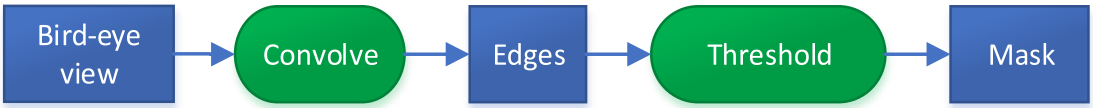
To store 3x3 perspective transformation matrix the OpenVX* `vx_matrix` object has to be created and initialized by the data. Please note that to initialize OpenVX* perspective transform matrix the data entries have to be stored as `vx_float32` and in the transposed way. The `vx_matrix` is initialized by `vxWriteMatrix` and passed into `vxWarpPerspectiveNode` to create node that produces required perspective transformation.

```
// create and init perspective transform OpenVX matrix ovxH
cv::Mat ocvH;
// calc perspective transform matrix for given input image width and height
calcPerspectiveTransform(width,height).convertTo(ocvH,CV_32F);
vx_matrix ovxH = vxCreateMatrix(ovxContext, VX_TYPE_FLOAT32, 3, 3);
vx_float32  data[9] =
{
    ocvH.at<float>(0,0),ocvH.at<float>(1,0),ocvH.at<float>(2,0),
    ocvH.at<float>(0,1),ocvH.at<float>(1,1),ocvH.at<float>(2,1),
    ocvH.at<float>(0,2),ocvH.at<float>(1,2),ocvH.at<float>(2,2)
};
vxWriteMatrix(ovxH, (void*)data);

//create virtual destination image
vx_image ovxImgMapped = vxCreateVirtualImage(ovxGraph,IW,IH,VX_DF_IMAGE_U8);

//create WarpPerspective node
vxWarpPerspectiveNode(
    ovxGraph,           // graph the node to be placed
    ovxImgGray,         // input 1 channel image
    ovxH,               // perspective transformation 3x3 matrix
    VX_INTERPOLATION_BILINEAR, // interpolation type that is used
    ovxImgMapped);      // output image for mapped result
```

Mark Pixels that Potentially Belong to Lane Markers



As result of this operation, binary image is produced where every marked pixel has value 255 and non-marked has value of 0.

There are many ways to detect lane pixels. The sample implements the simplest one based on gray image convolution with specific kernel and further thresholding of the result (by pre-defined threshold value). In a real production application, this step decently will be more complex and could contain adaptive thresholding mechanism, additional noise filtering and many other advanced steps.

Before create convolution node we have to define and create OpenVX* convolution kernel. The convolution kernel form is based on lane marker intensity profile. In the sample the following convolution function is used. It gives maximum response for areas where set of white pixels are surrounded by dark ones.

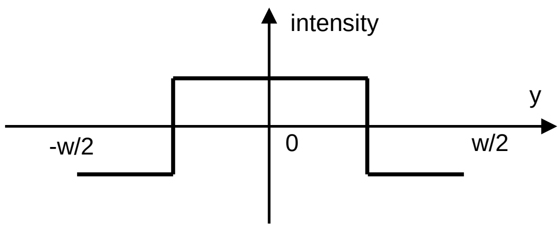


Figure 2: Convolution kernel profile to detect lane mark pixels. w is width of kernel

This lane profile can be defined in `FILTER2D_WxFILTER2D_H` array that is passed to `vxCreateConvolution` function. In addition to kernel values itself it is also needed to set scale factor that is used to scale result after convolution itself. This scale factor is setup as convolution object `VX_CONVOLUTION_ATTRIBUTE_SCALE` attribute by `vxSetConvolutionAttribute` function

```
// define filter sizes and filter kernel values
// we need convolution in y direction only but
// minimal convolution kernel size in OpenVX 1.0 is 3
#define FILTER2D_W  3
#define FILTER2D_H  11
static short gFilter2D_Data[FILTER2D_W*FILTER2D_H] =
{
    -5, -5, -5,  //  |
    -5, -5, -5,  //  |
    -5, -5, -5,  //  |______
     6,  6,  6,  //         |
     6,  6,  6,  //         |
     6,  6,  6,  //         |
     6,  6,  6,  //         |
     6,  6,  6,  //   ______|
    -5, -5, -5,  //  |
    -5, -5, -5,  //  |
    -5, -5, -5   //  |
};
// Scale factor for filter. It has to be power of 2 for OpenVX 1.0
#define FILTER2D_SCALE  16
// Threshold for filtered image to detect lane marker pixels
#define THRESHOLD_VALUE 100

// create filter to get horizontal line response
vx_uint32 filterScale = FILTER2D_SCALE;
vx_convolution  ovxFilter = vxCreateConvolution(ovxContext, FILTER2D_W, FILTER2D_H);
vxCopyConvolutionCoefficients(ovxFilter, &gFilter2D_Data, VX_WRITE_ONLY, VX_MEMORY_TYPE_HOST)
vxSetConvolutionAttribute(
    ovxFilter,
```

```
    VX_CONVOLUTION_ATTRIBUTE_SCALE,
    &filterScale,
    sizeof(filterScale));
```

The threshold object `ovxThreshold` has to be created in advance by `vxCreateThreshold()` call before using it as threshold node input argument. `vxSetThresholdAttribute` is then used to setup the threshold value.

```
//create threshold for threshold node
vx_threshold ovxThreshold = vxCreateThreshold(
    ovxContext,
    VX_THRESHOLD_TYPE_BINARY,
    VX_TYPE_UINT8);
vx_int32 threshValue = THRESHOLD_VALUE;
vxSetThresholdAttribute(
    ovxThreshold,
    VX_THRESHOLD_THRESHOLD_VALUE,
    &threshValue,
    sizeof(threshValue));
```

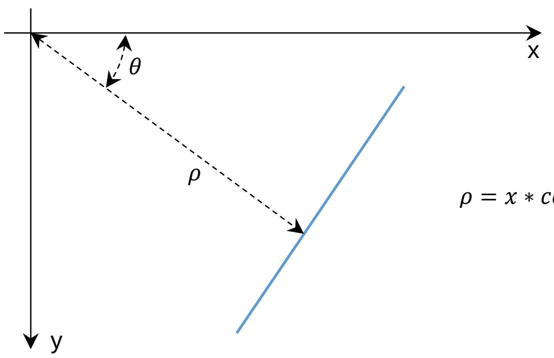
Then after these steps it is possible to create filter and threshold nodes that produce binary result in `ovxImgBin` image.

```
vxConvolveNode(ovxGraph, ovxImgMapped, ovxFilter, ovxImgEdges);
vxThresholdNode(ovxGraph, ovxImgEdges, ovxThreshold, ovxImgBin);
```

HoughLinesP

The final OpenVX* node in the pipeline is probabilistic Hough transform [5] to detect line segments from binary image.

Hough lines transform works with parametric line representation by a ρ and θ [5]. Here, ρ is distance from the coordinate origin and θ is the line angle (in radians).



$$\rho = x * cos\,(\theta) + y * sin(\theta)$$

The main idea of line detection algorithm then is to create (ρ, θ) parametric space array accumulators and accumulate votes for (ρ, θ) from each pixel. The size of each accumulator cell is defined by `HOUGH_RHO_RESOLUTION` and `HOUGH_THETA_RESOLUTION` values. Every non zero pixel from input image can belong to many lines defined by (ρ, θ) pair. For each this pair the pixel adds votes to the related ρ, θ accumulators according to the line equation. When some accumulator exceeds the given `HOUGH_THRESHOLD` value then the corresponding (ρ, θ) pair is used as parameters for a new line. To detect ends of the line segment the algorithm scans input pixels inside some area around of detected line. The final segment has to be covered by input non-zero pixels except some small gaps defined by `HOUGH_MAX_LINE_GAP`. The mode detailed description can be found in [5]

To get result from the node the `ovxLineArray` array of line segments has to be created. Each array element contains segment start point (`r.start_x,r.start_y`) and segment end point (`r.end_x,end_y`).

```
// create array for lines detected by Hough transform node
vx_array  ovxLineArray = vxCreateArray(
    ovxContext,
    VX_TYPE_RECTANGLE, // Type of each element of created array
    HOUGH_MAX_LINES);  // The maximal number of items that the array can hold

vx_scalar ovxLineCount = vxCreateScalar(
    ovxContext,
    VX_TYPE_INT32,     // Scalar type
    NULL);             // NULL is passed because we have not initial value

// create probabilistic Hough transform node to detect line segments
vxHoughLinesPNodeIntel(
    ovxGraph,                  // OpenVX graph to add node
    ovxImgBin,                 // Input 8U binary image
    HOUGH_RHO_RESOLUTION,      // Rho cell size (in pixels)
    HOUGH_THETA_RESOLUTION,    // Theta cell size (in radians)
    HOUGH_THRESHOLD,           // Threshold for accumulator
    HOUGH_MIN_LINE_LENGHT,     // Minimal line segments length that has to be detected
    HOUGH_MAX_LINE_GAP,        // Minimal gap inside detected line segment
    HOUGH_MAX_LINES,           // Maximal number of detected line segments
    ovxLineArray,              // Detected lines stored in vx_array of vx_rectangle_t
    ovxLineCount);             // number of detected lines, vx_int32
```

Host Postprocessing Step

Hough transform algorithm might produces multiple line segments for one lane mark. In addition to OpenVX* pipeline the final host post-processing step is used to stich segments in a single solid lane mark, and to draw final result over the input image. Because this step is not OpenVX* related then it is not described in this OpenVX* sample document.

OpenCV Reference Implementation

Besides OpenVX* implementation of the lane detection pipeline, there is a complete OpenCV code that implements almost the same algorithm. It is defined in the lane_detection.cpp source file as simple `OCVPipeline` class that has `Init()` function to initialize data and `Process()` function to process input image.

Building the Sample

See the common `README` file for all samples in the root sample directory for complete instructions about how to build the samples.

Running the Sample and Understanding the Output

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

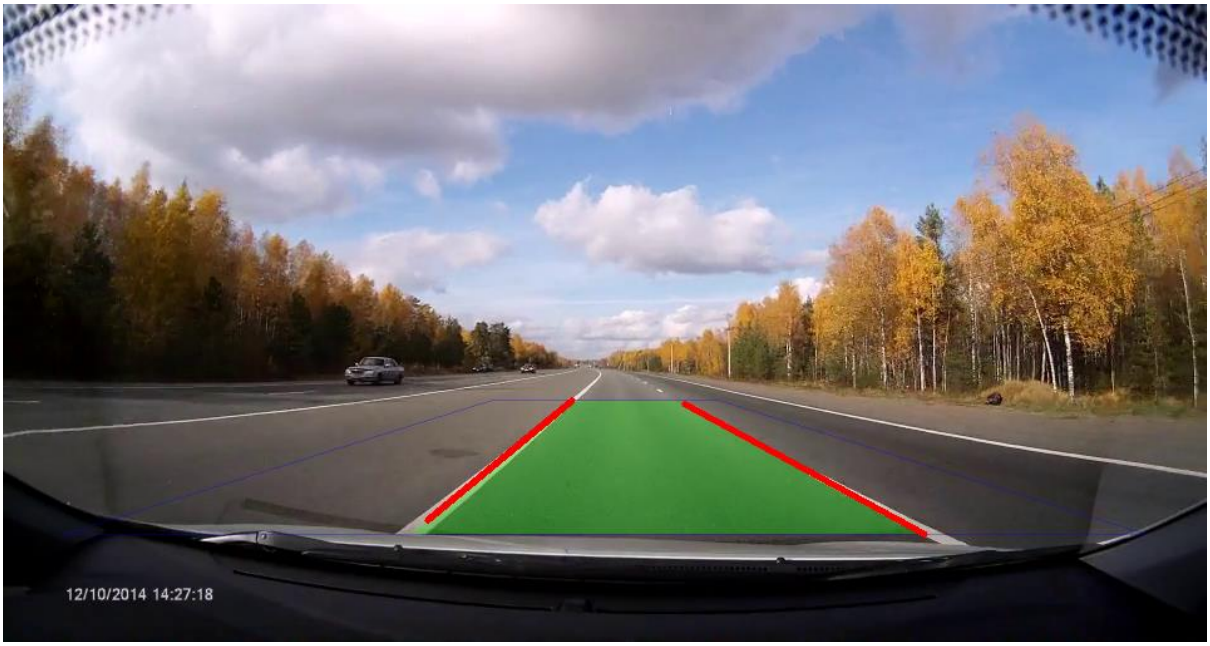
```
$ ./lane_detection --help
```

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

By the default, the sample reads the video file `road_lane.mp4` that should be located in the same directory where lane_detection executable is. So, calling sample without parameters:

```
$ ./lane_detection
```

Will open road_lane.mp4 and create two pop-up windows rendering input with lane marks detected by OpenVX* and OpenCV pipeline respectively. One of windows is shown below:



The red line segments show detected lane marks. The area bounded by blue rectangle is a road part mapped into internal bird-eye view image by perspective transformation for processing.

You can exit from the sample by pressing 'Esc' key when one of the GUI windows is in focus. Also you can stop pipeline by pressing 'Space' and run it frame by frame using 'Enter' key.

Use 'visualization' option 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:

```
$ ./lane_detection --visualization 0
```

When the sample finishes execution, it prints performance statistics to the console (time is in milliseconds):

```
$ ./lane_detection –-input ./road_lane.mp4 --visualization 0

Input frame size: 960x508
Frame: 300
Release data...
1.23 ms by ReadFrame averaged by 301 samples
2.55 ms by ProcessOpenCVReference averaged by 300 samples
1.50 ms by vxProcessGraph averaged by 300 samples
0.24 ms by CollectLaneMarks averaged by 600 samples
```

The main metrics are:

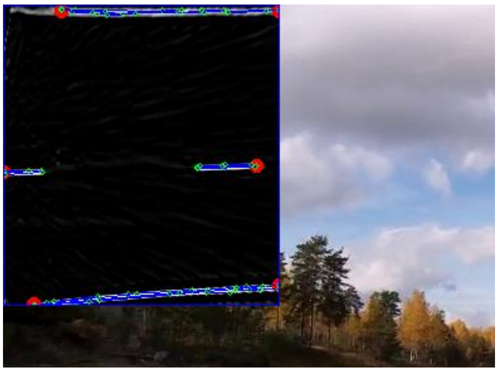
- **ReadFrame** averages time of next frame preparation and putting it into vx_image for further processing.
- **ProcessOpenCVReference** averages time of frame processing by OpenCV reference implementation (frame reading time and frame visualization time are not included).
- **vxProcessGraph** is the pure `vxProcessGraph` API call time (averaged).
- **CollectLaneMarks** is the final step time (averaged).

Debug Visualization

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

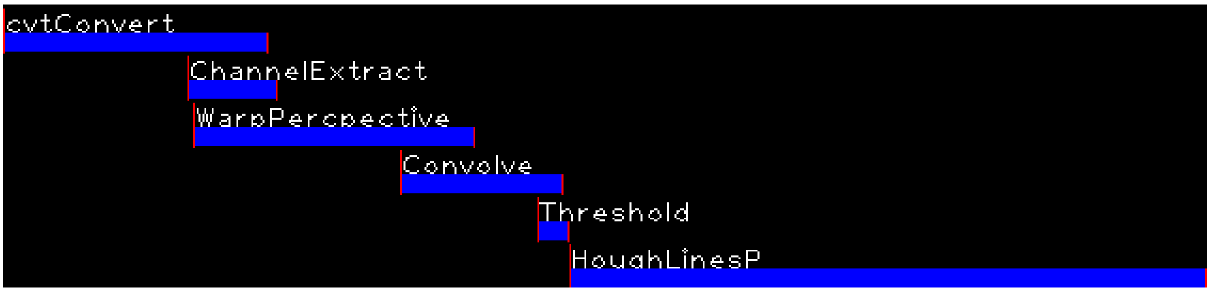
```
$ ./lane_detection --input ./road_lane.mp4 --visualization 2
```

It draws sub window in the top left corner that display results of the OpenCV or OpenVX pipelines respectively. You can see an example of the debug output on the following screen shot:



The blue lines are Hough transform result. Each line segment is drawn as blue line ended by 2 green points. The post-processing step takes these blue segments as input and forms final line segment (per lane) that are marked by pair of red points.

This mode also creates 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. The left edge of the window is where the graph starts execution. The right edge is where the graph completes execution. For more details, please refer to the implementation of `IntelVXSample::drawNodesAtTimeline` function in samples common infrastructure file `samples/common/src/perfprof.cpp`.

References

1. OpenVX* 1.1 Specification
2. Intel® Computer Vision SDK Developer Guide

3. Borkar, A.: Multi-viewpoint lane detection with applications in driver safety systems. PhD thesis, Georgia Institute of Technology (2012)
4. A. Takahashi, Y. Ninomiya, M. Ohta, M. Nishida, and M. Takayama, “Rear view lane detection by wide angle camera,” in IEEE Intelligent Vehicle Symposium, vol. 1, 2002, pp. 148–153
5. J. Matas, C. Galambos, J. Kittler, “Robust detection of lines using the progressive probabilistic hough transform”, Comput. Vis. Image Underst., 78 (1) (2000), pp. 119–137
6. B. Fardi and G. Wanielik, "Hough Transformation Based Approach For Road Border Detection in Infrared Images", 2004 IEEE intelligent vehicles symposium, university of parma, parma, italy, june 14-17, 2004.
7. M. Aly, "Real time detection of lane markers in urban streets", Proc. IEEE Intell. Vehicles Symp., pp. 7-12, 2008