Camera Tampering OpenVX* Sample

User's Guide

Intel® Distribution of OpenVINO™ toolkit – OpenVX Samples*

Contents

Tabes

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 sample will illustrates three ideas.

- First is a simple Tampering Detection Algorithm to detect defocus and occlusion tampering types, which demonstrates how user can easily implement an algorithm with OpenVX kernels which CVSDK provides.
- The second is a use case to combine with other target algorithm, in this sample is Motion Detection, as a tiny digital security surveillance system.
- The last one, this sample can also demonstrate performance boost by heterogeneity.

As mentioned, this sample merges with Motion Detection, if you need a detailed introduction to the background knowledge of Motion Detection sample, see the *Motion Detection sample*, available in this SDK (`<SDK_ROOT>/samples/motion_detection`).

Meanwhile, the sample features basic **interopratability with OpenCV** through data sharing/processing. OpenCV is used for calculating the image information from a video file and feature extraction.

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.

Sample Concept Description

The core of this sample utilizes feature extraction of image process. More specific, we use the edge information of each video frame. By using the edge information, we can estimate whether current camera scene is blocked/tampered by others or not. There are a lot of Edge Detection theories in the modern academic community, for instance, Sobel operator and Canny Edge detector [3]. Here, we apply Canny Edge detector in our implementation.

In general, an image applies the edge detection and the result would like Figure 1. As the right figure in Figure 1 shown, we could gather the edge feature image of original input.

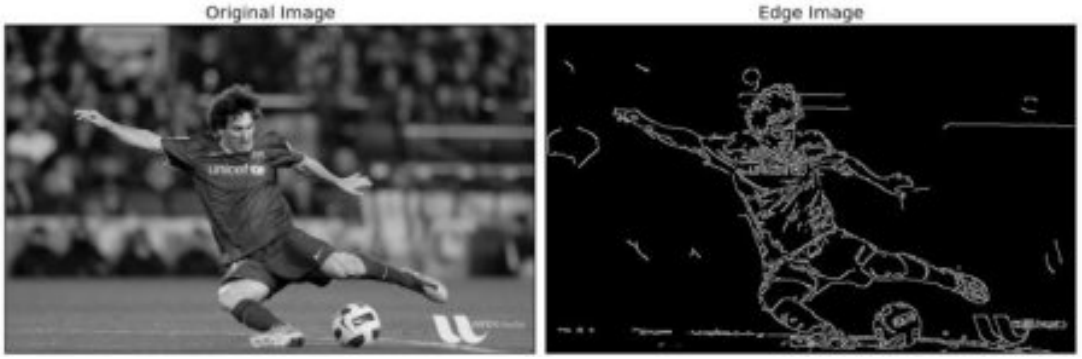

Figure 1 Image Applies Edge Detection

The next step is to quantize the feature image and then decide threshold value to judge whether a tampering event occurred or not. You can image that if something else placed in front of the camera, the corresponding edge image of it is totally different from the original one. The sample flowchart can be illustrated as Figure 2 below.

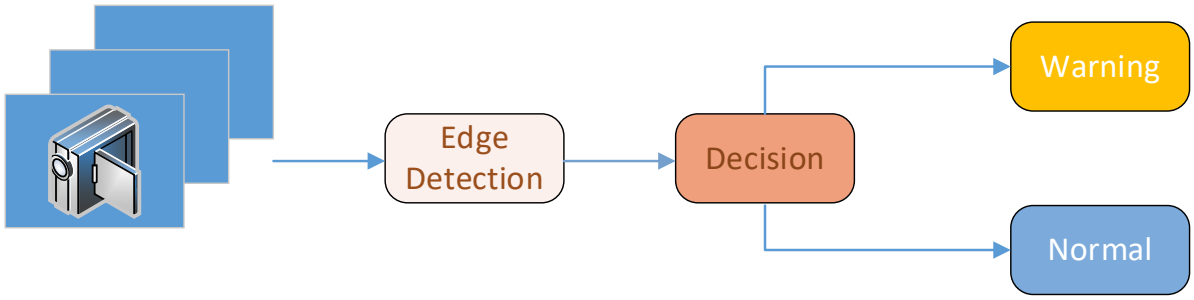

Figure 2 Flowchart of this sample.

The further work is to combine this concept with OpenVX. Since the concept is not complex, beyond implementation by OpenVX. We also associate with the other sample, Motion Detection to demonstrate the multiple functionality.

Camera Tampering Algorithm as an OpenVX* Graph

Figure 3 presents a Camera Tampering Detection algorithm based on the other Motion Detection sample to adopt more image processing algorithm for implementation.

The graph consists of Image scaling (optional), ConvertColor, CannyEdge, Background subtraction MOG2, Dilate, Erode, Dilate and Connected component labeling nodes. The graph consumes an image with RGB format and produces a label image and an array of bounding rectangles of detected moving objects.

Image scaling is optional, it can scale down the size of input image to save computations for subsequent processing. It can greatly improve performance without obvious loss in accuracy.

CannyEdge is general image process method for edge detection in grey level image. This is why we need to convert source images color space to grey level to do further processing. After we extract the edge information of corresponding images, we could quantize and utilize it as an evaluation reference. With these evaluation reference, the next step is to fine-tune the threshold to decide the tampering behavior happened or not. Finally, the program itself would send out the alarm. The illustration would like below:

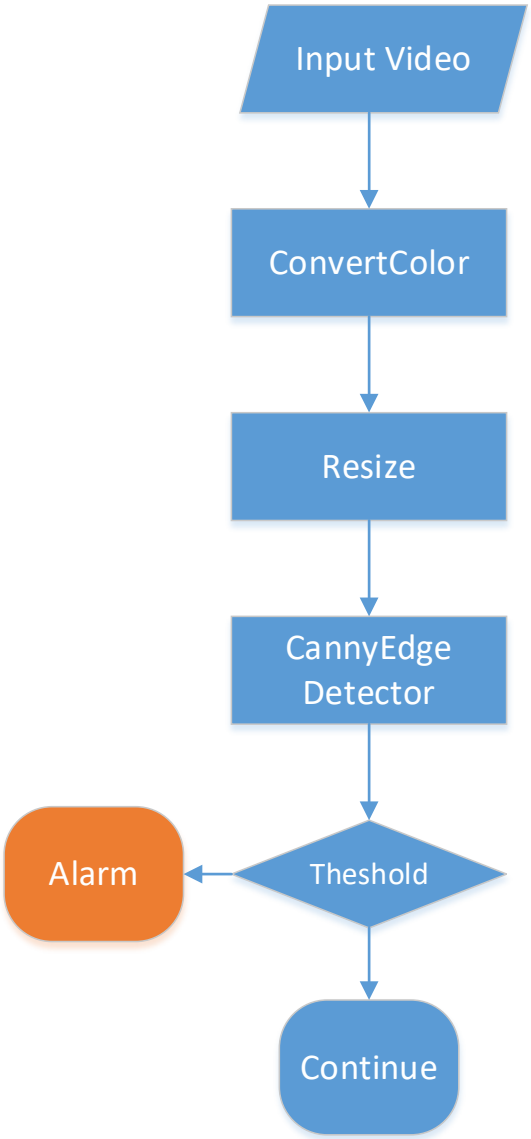

Figure 3: Illustration of the algorithm of Camera Tampering Detection

OpenCV may have functions which have similar functionality. This algorithm is adopted as the code is simple, and it's easier for GPU implementation in the future.

The complete OpenVX* graph is presented in Figure 4. See `camera_tampering.cpp` where the following code can be found:

- creation of `vx_context` and `vx_graph` instances;
- creation of all non-virtual data objects;
- the main loop to process an input video file, frame-by-frame;
- creation of all virtual data objects;
- Population of the graph with nodes to build the pipeline presented on Figure 4.

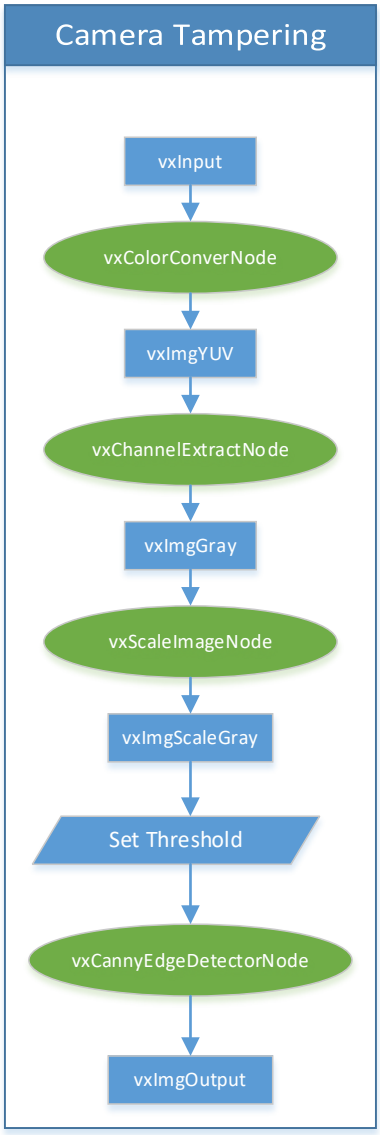

Figure 4: Complete OpenVX* graph of Camera Tampering Detection implemented in the sample.

Aforementioned, this sample is on top of Motion Detection. It could detect tampering event, in the meanwhile, there is no conflicted to perform Motion Detection as well. Therefore, it proves that there is no problem to combine 2 different algorithms/samples in order to achieve variety purposes. Below Figure 5 illustrates how to associate these 2 samples into 1.

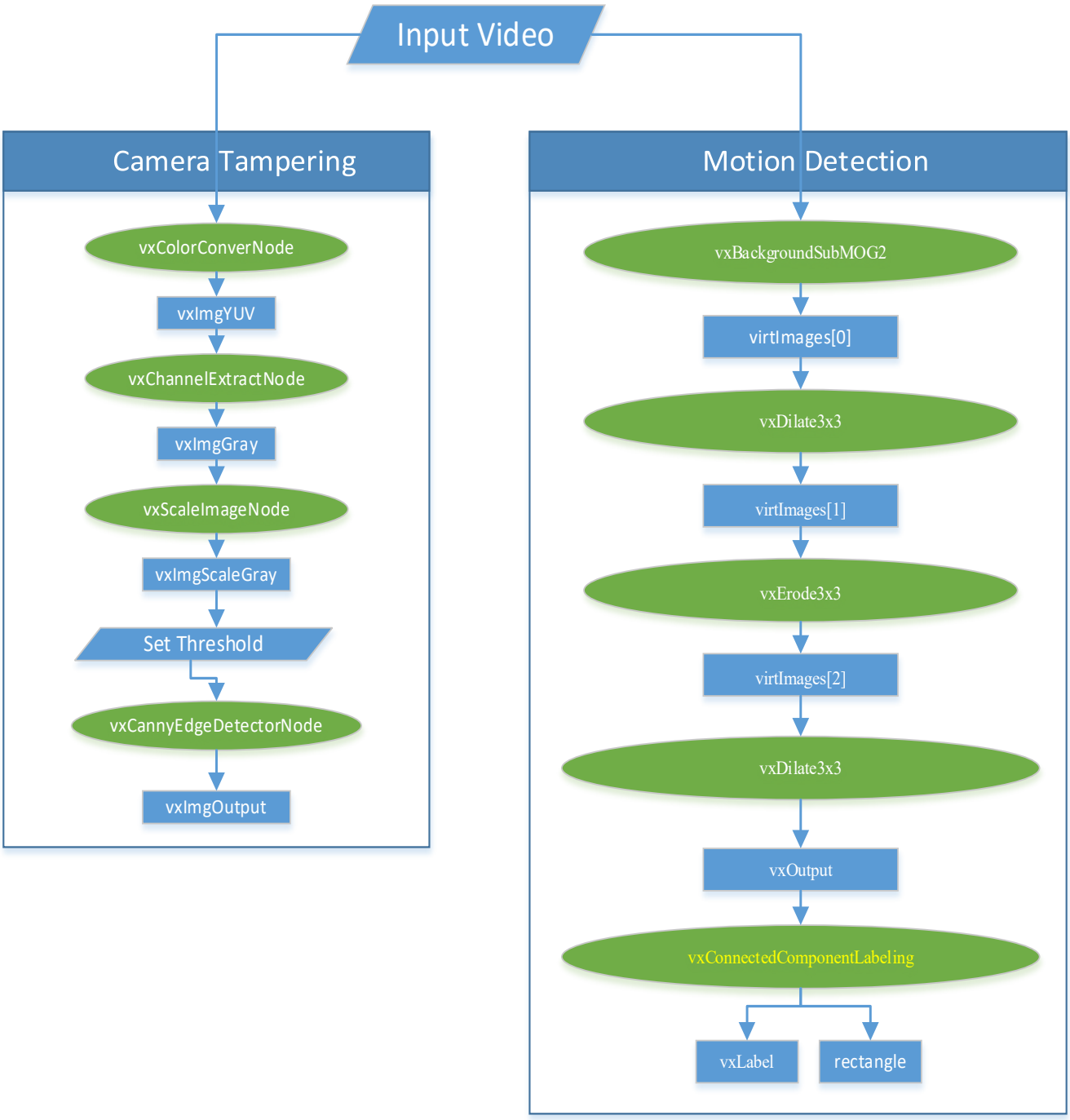

Figure 5 Combination Illustration of 2 samples.

How to Change Heterogenity Configuration in This Sample

In respond to "Introduction" paragraph, the heterogeneity topic could be described as following, since this sample is combined with 2 different purposes of algorithm. 1 is for camera tampering and the other is for Motion Detection (Illustrated as Figure 5).

To increase the Hardware-wise resources manipulation, we have tried to offload Motion Detection part to GPU and compare to all assign to CPU. Under the same input video, the process time of each frame, namely FPS result could be observed as Table 4.

User could refer below paragraph, “Running the Sample”, by change the sample launch parameter `hetero-config` to switch the offload assignment. If user is not assigned, it would default load `hetero.config.default.txt` in which it would not do optimize. 2 algorithms are all running in CPU portion. On the other hand, if assigned `hetero-config` as `hetero.config.gpu-offload.txt`. We could observe the different FPS number between 2 heterogeneity configurations.

Building the Sample

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

Running the Sample

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

```
$ ./camera_tampering --help
```

The complete command line arguments are:

```
$ ./camera_tampering --input inputFileName --output outputFileName --ct_enable
enableFlag --ct_ratio_threshold thresholdValue --ct_scale processImageSizeRatio --
hetero-config configFileName
```

Here `inputFileName` specifies input file name, and `outputFileName` specifies output file name. `ct_enable` specifies the flag of Camera Tampering detection process, default is turn on, namely 1. `ct_ratio_threshold` stands for the value to decide the tampering behavior happened or not. `ct_scale` means the proceed image size to scale down for performance improvement. `hetero-config` specifies the configuration of each function to execute on different targets(CPU or GPU), default is using `hetero.config.default.txt` if user is not assigned while execution and all nodes in the sample are running in CPU. On the other hand, if assigned to `hetero.config.gpu-offload.txt`, Motion Detection related nodes are offload to GPU and Camera Tampering related one keep in CPU.

Understanding the Output of this Sample

If user is setup everything correctly in their environment, this sample output result in console would like below Figure 6. As highlight in Figure 6, please read below description for these 4 sections to understand more and Table 1 shows the corresponding description of it.

```
maxnuc … vx_build bin camera_tampering ./camera_tampering
[input setting] imgNumber = 0 threshold = 0 merge = 0 scale = 0 ct_ratio_threshold = 0.01 ct_sc
ale = 1
Video frame size: 1280x720
OpenCV: FFMPEG: tag 0x31637661/'avc1' is not supported with codec id 28 and format 'mp4 / MP4 (
MPEG-4 Part 14)'
OpenCV: FFMPEG: fallback to use tag 0x00000021/'!???'
[ INFO ] hetero.config.default.txt      (a)
[ INFO ] Number of supported targets: 3
[ INFO ]     Target[0] name: intel.cpu
[ INFO ]     Target[1] name: intel.gpu
[ INFO ]     Target[2] name: intel.ipu  (b)
[ INFO ] ct_enable :1
[ INFO ] ct_scale :1
[ INFO ] w2 :1280,Original :1280
[ INFO ] h2 :720,Original :720
[ INFO ] Node CT_vxColorConvertNode is assigned to intel.cpu
[ INFO ] Node CT_vxChannelExtractNode is assigned to intel.cpu
[ INFO ] Node CT_vxScaleImageNode is assigned to intel.cpu
[ INFO ] Node CT_vxCannyEdgeDetectorNode is assigned to intel.cpu
[ INFO ] Node vxBackgroundSubMOG2Node is assigned to intel.cpu
[ INFO ] Node vxDilate3x3Node(1) is assigned to intel.cpu
[ INFO ] Node vxErode3x3Node is assigned to intel.cpu
[ INFO ] Node vxDilate3x3Node(2) is assigned to intel.cpu      (c)
Reached end of video file
28.718 ms by ProcessFrame averaged by 3019 samples    (d)
2.089 ms by ReadFrame averaged by 3020 samples
55.764 ms by Frame averaged by 3019 samples
28.322 ms by vxProcessGraph averaged by 3019 samples
30.872 ms by vxVerifyGraph averaged by 1 samples
```

Figure 6 Console Output of this sample

Index	Description
(a)	Offload config. Default is `hetero.config.default.txt`.
(b)	Support Targets of your physical device.
(c)	Assign Node to corresponding targets
(d)	Performance Profile Statistic.

Table 1 Description of Figure 6

Section (a), if user doesn't attempt to assign different configuration of this sample, it would load default configuration file in his SDK (`<SDK_ROOT>/samples/camera_tampering`). Section (b) stands the device under test with what kind of capability to support, namely, extra GPU or IPU. Section (c) means the configuration file we assigned in the `hetero.config.default.txt`. The last one, section (d) is the performance profiles we collect in this sample.

Here `ReadFrame` means the time which reads 1 frame from a video file and transfers to sample input; `vxProcessGraph` represents the core algorithm of Camera Tampering and Motion Detection process time within openVX; `ProcessFrame` stands for the time of `vxProcessGraph` output and plus the drawing rectangle time for Motion Detection. The last, `vxVerifyGraph` is using for verifying the entire Graph is with any problem or not.

This sample expected output result would like Figure 7 shown. (a) is the original camera snapshot, (b) is partial tampering behavior, and (c) is full 100% occlusion the camera. The alarm form pops up a warning message in the bottom of scene. In general, any kind of warning/alarm can be defined as any user like. Here, we utilize a common and easy way for everyone to understand.

Figure 7 Camera Tampering Result.

If there is any other object trying to shelter the camera scene, the sample's algorithm would compute the coresponding image feature and judge whether the tampering event is occurred or not.

The other use case is defocus of the camera. This sample is able to detect this kind of abnormal Camera Tampering behavior. You can refer below Figure 8. (b) is out of focus of (a).

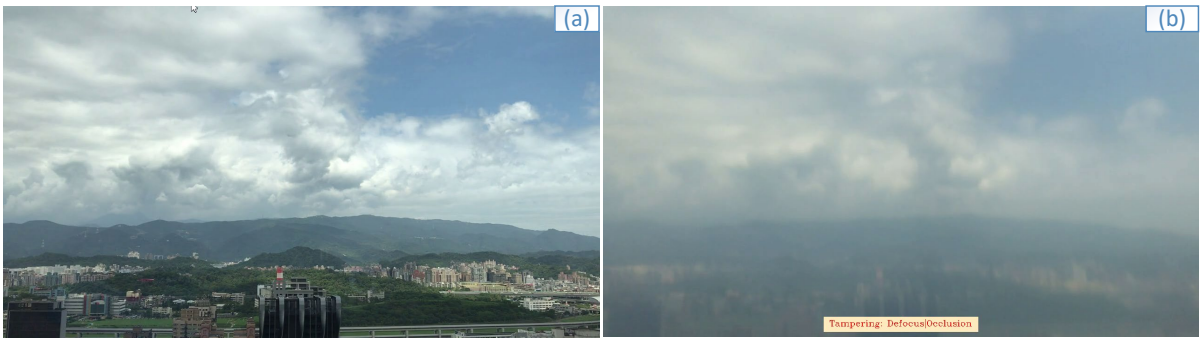

Figure 8 Defocus Case of Camera Tampering

Meanwhile, as mentioned previously, this sample merges the Motion Detection as well. Users should observe the detection result of Motion Detection. In the video we provided in the SDK, the main purpose is used for Camera Tampering, however, user would also observe some while rectangle highlight in the video. The white rectangle stands for the Motion Detection result in the corresponding frame. Figure 9 shows some frames with white rectangles. Figure 9(a) is caused by the moving cars in the road and the moving cars meet Motion Detection criteria and identified them. Figure 9(b) is triggered by the quick scene change and Motion Detection algorithm is regarded as motion in different segments and it's a kind of false alarm. Figure 9(c) is following frame of Figure 9(b) and they are false alarm of the Motion Detection.

Figure 9 Motion Detection Results in The Video.

Performance Comparison

For the heterogeneity-wise and performance-wise evaluation, the sample was tested in two different hardware environments: Intel® Pentium® Processor N4200/5, N3350/5, N3450/5 (formerly Apollo Lake) setup and 6th Generation Intel® Core™ Processors (formerly Skylake) setup in accordance with the CV SDK target system. The table below shows details of the target system and the device test unit:

Platform	CPU	Memory	OS (64-bit)
6th Generation Intel® Core™ Processors (formerly Skylake) with Intel® Iris® Pro Graphics and HD Graphics	i7-6770HQ	16 GB	Ubuntu* 16.04.2 LTS
Intel® Pentium® Processor N4200/5, N3350/5, N3450/5 (formerly Apollo Lake) with Intel® HD Graphics	Celeron® J3455	1.8 GB	Yocto* MR3

To observe the offload benefits in this sample, you need to install OpenCL driver of the corresponding OS environment for both setups. Any error messages appearing during the sample launch might be related to the driver problem.

The following table compares processing time results of two approaches: assigning algorithms to different targets and to CPU only to process the test video. The FPS, frame per second, process time uses vxProcessGraph as the main profile aim to evaluate in two different hardware system. As Motion Detection and Camera Tampering algorithms are assigned to CPU and CPU respectively, the average performance time is better than the CPU-only case.

Platform	Test Condition	Performance (ms)	Performance (fps)
6th Generation Intel® Core™ Processors (formerly Skylake) with Intel® Iris® Pro Graphics and HD Graphics	Heterogeneity (MD* in GPU, CT* in CPU)	7.07	141
	CPU Only	28.627	35
Intel® Pentium® Processor N4200/5, N3350/5, N3450/5 (formerly Apollo Lake) with Intel® HD Graphics	Heterogeneity (MD* in GPU, CT* in CPU)	26.162	38
	CPU Only	77.588	13

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 section teaches how to analyze an application that uses OpenVX to find and exploit performance opportunities provided by OpenVX.

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 systems.

Offload Motion Detectoin to GPU

As previous explanation, to gain the hardware-wise benefit, user could assign different `hetero-config` while execution. If user used the 2nd configuration (`hetero.config.gpu-offload.txt`) in the camera tampering sample folder, OpenVX would arrange the corresponding nodes to CPU or GPU according to the configuration file.

On the other hand, Intel VTune is a good tool to observe the offload effect. Here we use it to demonstrate offload effect. Figure 11 presents the hardware information of SkyLake we mentioned in Table 3 and the offload target GPU is the Integrated Graphics of it.

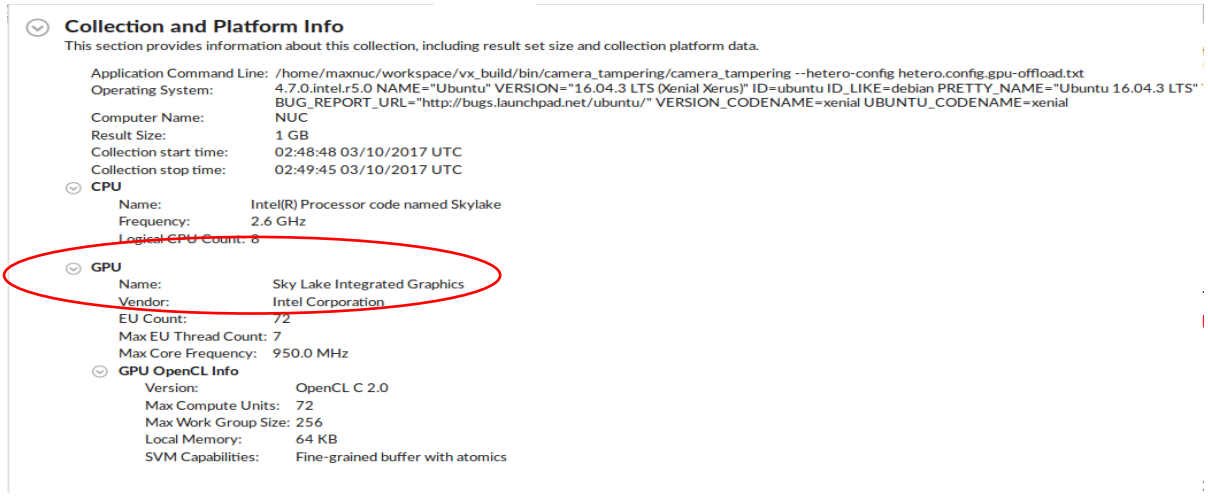

Figure 10 Device information in Vtune screenshot

Figure 12 shows the Motion Detection nodes in OpenVX are assigned in the GPU command queue. The orange part highlight indicates the zoom in of Vtune screenshot. User could compare below figure with hetero-config file to see the node is really offload to GPU or not. Moreover, user could try to compare the different output of Vtune while assign different hetero-config.

As you can see in Figure 12, the zoom-in (A) could observe the Motion Detection Nodes offload to GPU in our sample, and the corresponding CPU portion is running Camera Tampering Nodes, including Canny_Edge_Detector in (B) and Connected Component Labeling in (C).

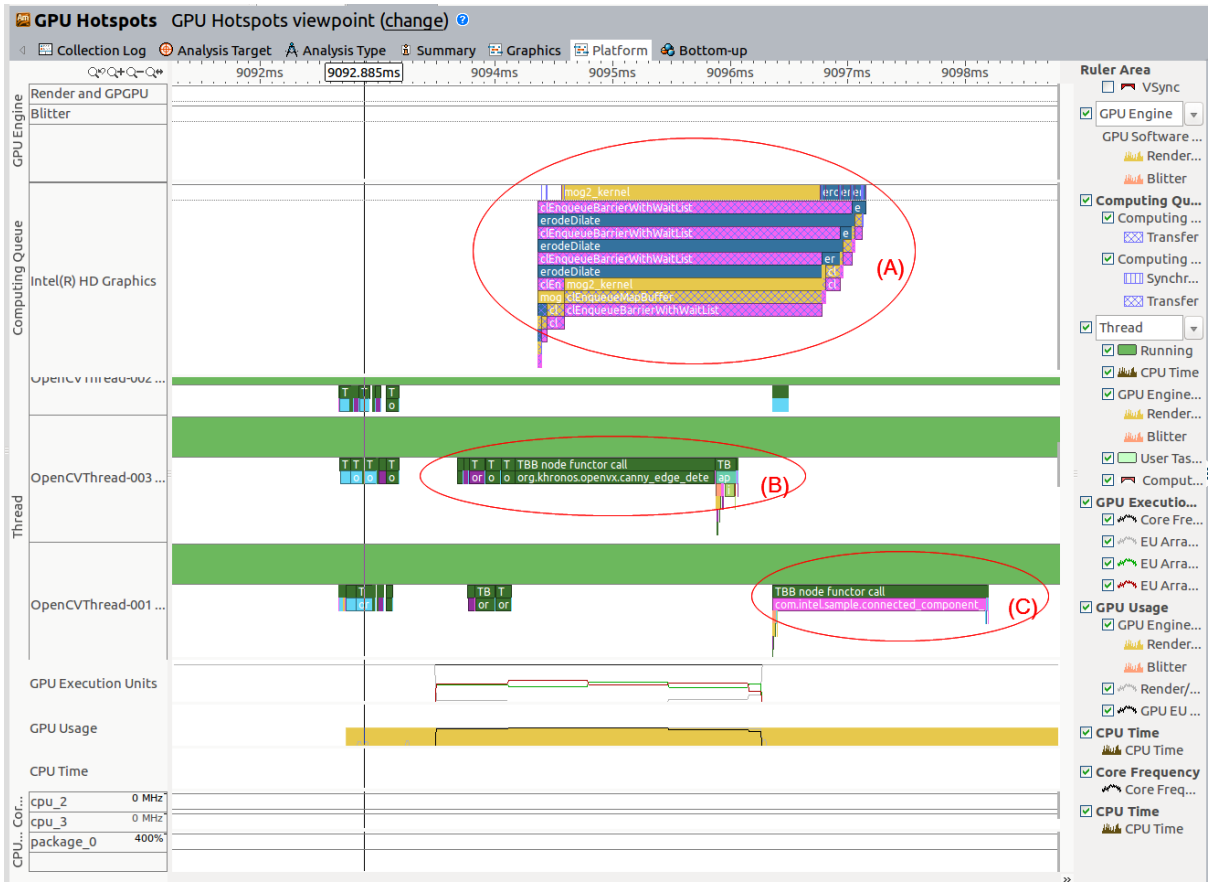

Figure 11 Vtune screenshot with offload to GPU/CPU of this sample.

Figure 13 illustrates the details of Motion Detection offload in GPU of certain period while running our test videos, namely zoom-in Figure 12(A).

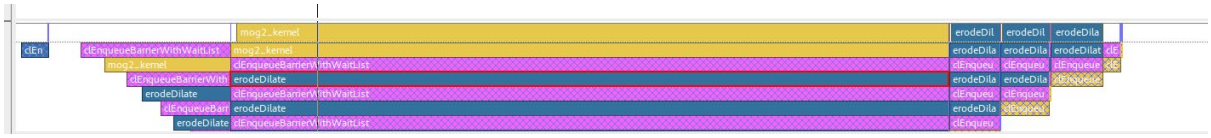

Figure 12 Specific Nodes offload to GPU

References

1. OpenVX* 1.1 Specificationhttp://software.intel.com/en-us/node/505966
2. Intel's OpenVX Developer Guide
3. http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html
4. https://en.wikipedia.org/wiki/Background_subtraction

5. https://en.wikipedia.org/wiki/Connected-component_labeling
6. https://software.intel.com/en-us/articles/opencl-drivers
7. https://software.intel.com/en-us/cvsdk-quickstartguide-installing-intel-computer-vision-sdk