OpenVX* Heterogeneous Basic 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

Intel® Computer Vision SDK provides the API to schedule parts of an OpenVX* graph to different compute units (e.g. CPU, GPU or IPU). In the context of OpenVX*, such a compute unit is named *target*. The set of available targets depends on a platform used to run an application.

A developer might want to schedule nodes to a particular target to improve performance or power consumption. For example, for certain parameters (e.g. input image size) some nodes are doing better on the GPU than on CPU and vice versa. Then for a graph that includes such nodes, the developer can experiment with running different parts of the pipeline on the GPU while measuring performance and/or power consumption metrics. Alternatively, if there is enough parallelism in the graph, multiple branches of the graph (like individual RGB channels, below) can execute on the different targets *simultaneously*.

Intel Computer Vision SDK implements OpenVX* Khronos Target API for assigning selected nodes in an OpenVX* graph to a particular target, overriding the default run-time choice. This API enables heterogeneous usages of Intel platforms that is a way to better hardware utilization.

This document provides an introduction to the Target API, illustrating basic usages on an example OpenVX* graph. The graph is doing Canny Edge Detection on individual channels of the input image. Each channel, R, G and B is assigned to different target, CPU, GPU or IPU. This mapping is made for API illustration purposes and may or may not provide any performance difference in real scenario.

NOTE: The Khronos* OpenVX* Specification 1.1 contains description of Khronos* Targets. Please refer to section 2.12 “Targets” of the OpenVX* Specification 1.1.

Brief Introduction to OpenVX*

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

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

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

Sample Description

Algorithm implemented in this sample serves illustrative purposes. The algorithm allows to quickly introduce the Targets API.

The sample consumes an input RGB video file frame by frame. Each frame is processed with Canny Edge Detector, every channel (R, G and B) separately. In the end of the pipeline, the processed images are combined in a single output RGB image. This output image is optionally showed with a pop-up GUI window.

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

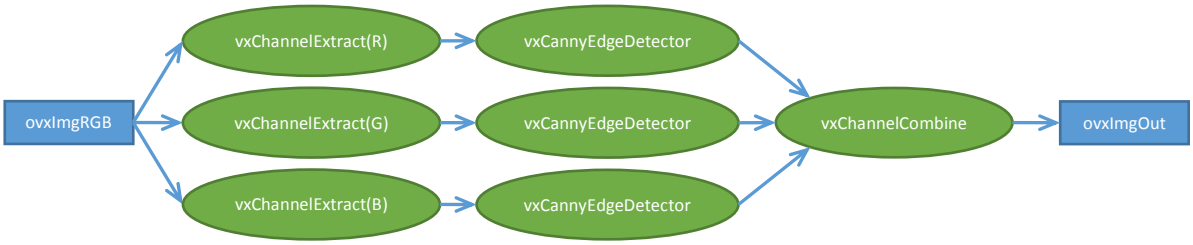


Figure 1: OpenVX* graph implemented in the sample.

Intel OpenVX* Targets

Intel Targets

Targets can be identified by enum or by name. Here is the list of target names and enums associated with specific compute units supported by Intel implementation:

- **`VX_TARGET_CPU_INTEL`** enum and **`"intel.cpu"`** name is for CPU device
- **`VX_TARGET_GPU_INTEL`** enum and **`"intel.gpu"`** name is for GPU device
- **`VX_TARGET_IPU_INTEL`** enum and **`"intel.ipu"`** name is for IPU device

Assigning Targets to Nodes

When a specific target reference is available and node is created with one of the node creation function, the developer should call `vxSetNodeTarget` to assign the node to the target:

```
vx_node node;
vx_enum target;

node = . . .;   // node is created
target = . . .; // target is specified (VX_TARGET_CPU_INTEL,
                // VX_TARGET_GPU_INTEL or VX_TARGET_IPU_INTEL

vx_status status = vxSetNodeTarget(node, target, 0);
if (status != VX_ERROR_NOT_SUPPORTED)
{
    // the node is not supported by the target
    // so the vxSetNodeTarget had no effect
    // implementation falls back to the default affinity
}
```

As not all targets are capable to run a specific node, the developer may check if the result of `vxSetNodeTarget` is successful. If `vxSetNodeTarget` returns `VX_ERROR_NOT_SUPPORTED`, that means the node cannot be run on a particular target. If it happens, the node will be executed on the default target which is chosen by the OpenVX* run-time.

NOTE: Call to the `vxSetNodeTarget` function for a node in a graph should happen before `vxVerifyGraph` for this graph.

In the sample code, each channel of RGB image is processed by nodes assigned to different targets as shown on Figure 2. If target is not available on the platform, the default run-time behavior is used.

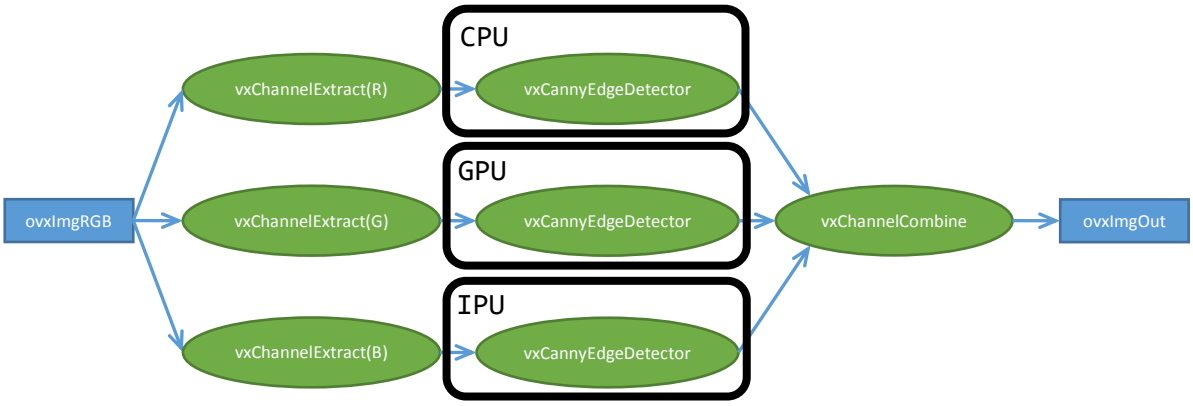


Figure 2: Nodes assignment to targets.

Here is the code that makes necessary assignments. Look into the source code of the sample to see nodes creation that are used in this snippet.

```
if(vxSetNodeTarget(cannyRNode, VX_TARGET_CPU_INTEL, 0) == VX_SUCCESS)
{
   std::cout << "[ INFO ] Target intel.cpu is set for channel R.\n";
}
else
{
   std::cout << "[ WARNING ] Target intel.cpu is NOT available. ";
   std::cout << "Nodes related to channel R won't be assigned to any specific target.\n";
}

if(vxSetNodeTarget(cannyGNode, VX_TARGET_GPU_INTEL, 0) == VX_SUCCESS )
{
   std::cout << "[ INFO ] Target intel.gpu is set for channel G.\n";
}
else
{
   std::cout << "[ WARNING ] Target intel.gpu is NOT available. ";
   std::cout << "Nodes related to channel G won't be assigned to any specific target.\n";
}

if(vxSetNodeTarget(cannyBNode, VX_TARGET_IPU_INTEL, 0) == VX_SUCCESS )
{
   std::cout << "[ INFO ] Target intel.ipu is set for channel B.\n";
}
else
{
   std::cout << "[ WARNING ] Target intel.ipu is NOT available. ";
   std::cout << "Nodes related to channel B won't be assigned to any specific target.\n";
}
```

Building the Sample

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

Running the Sample and Understanding the Output

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

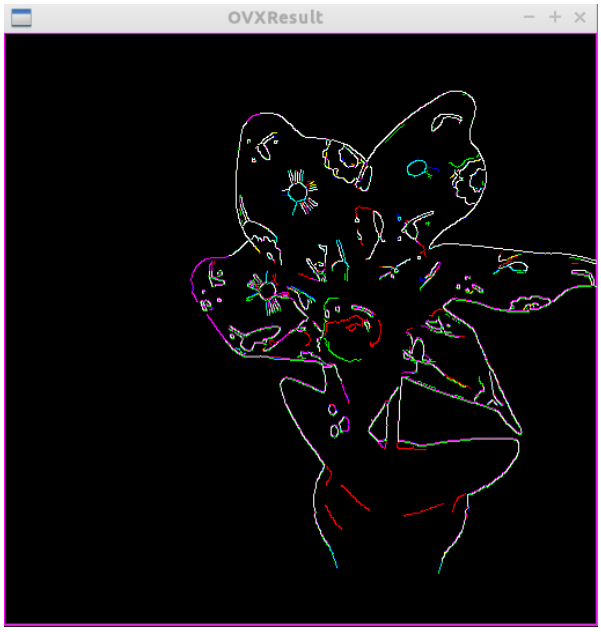
```
$ ./hetero_basic --help
```

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

By the default, the sample reads the video file `toy_flower_512x512.mp4` that is located in the directory with sample executable. So calling sample without parameters:

```
$ ./hetero_basic
```

Will open `toy_flower_512x512.mp4` and create a pop-up window with processed frame.



You can exit from the sample by pressing **Esc** when one of the GUI windows is in focus.

To run the sample in pure command line mode without a pop-up window (for example when using a remote terminal or for benchmarking purposes) the visualization mode can be disabled:

```
$ ./hetero_basic --visualization 0
```

When the sample finishes execution, it prints performance statistics out to the console in milliseconds (in the next example `--max-frames` is used to limit the number of processed frames):

```
$ ./hetero_basic --visualization 0
```

```
toy_flower_512x512.mp4 is opened
Input frame size: 512x512
[ INFO ] Target intel.cpu is set for channel R.
[ INFO ] Target intel.gpu is set for channel G.
[ INFO ] Target intel.ipu is set for channel B.

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

NOTE: The output above is produced on the system that has support for IPU, which is represented in OpenVX* as `intel.ipu` target. If this target is not available, the output will be different: "`[ WARNING ] Target intel.ipu is NOT available. Nodes related to channel B won't be assigned to any specific target.`"

Performance metrics reported by the sample output are:

- **vxProcessGraph** is the pure `vxProcessGraph` API call average time. Here is where heterogeneous execution happens. It is the main metrics that characterizes the efficiency of a chosen targets-to-nodes mapping. When trying to assign targets for the nodes differently, the developer should control this metric as a feedback.
- **ReadFrame** captures the average time of frame preparation. It includes reading a frame content from `VideoCapture`, time spent in `vxMapImagePatch`/`vxUnmapImagePatch` and some other necessary operations.

References

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