Auto-Contrast 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

The Auto-Contrast sample provides step-by-step instructions for OpenVX* code development. Specifically, the sample demonstrates acceleration of image processing tasks with OpenVX, implementing a straightforward intensity normalization via histogram equalization.

The code uses only standard OpenVX nodes and the bare minimum resources. For example, wherever possible, it relies on OpenVX virtual images. The sample also shows basic interoperability with OpenCV*, which is used for loading/saving and displaying the images.

What Is the OpenVX* Standard?

OpenVX is a new standard from Khronos*, offering a set of optimized primitives for low-level image processing and computer vision primitives. The OpenVX specification is applicable across multiple vendors and platforms.

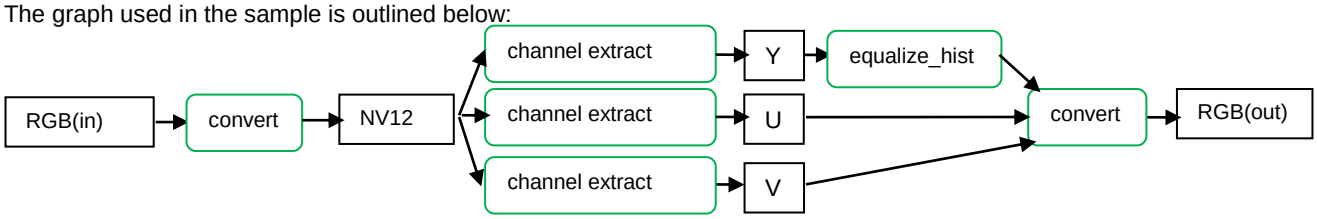
The abstraction of OpenVX notions of resources and execution enables hardware vendors to optimize implementation with a strong focus on a particular platform. At the same time, standardization allows software developers to separate algorithmic innovations from the performance back-ends coming from companies providing accelerators.

For example, the CPU OpenVX implementation from Intel offers graph-level optimizations like tiling and threading, as well as kernel-level efficiency, via intensive use of Intel® Integrated Performance Primitives (Intel® IPP). Similarly, the GPU implementation from Intel relies on the mature OpenCL™ stack to deliver the performance promise.

Expressing the Algorithm as an OpenVX Graph

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 changing only the data inputs.

The graph used in the sample is outlined below:



The first node converts RGB data into the NV12 color image format. Three Channel Extract nodes extract the Y, U and V components respectively. The Y component is equalized using Equalize Histogram, while U and V components are left intact. Then the equalized Y plus the original U and V are combined into another NV12 image, which is finally converted to the output RGB image.

The rest of document describes each step in the sample application in more details.

Creating an OpenVX Context and Graph

All objects in OpenVX exist within a context. Actual data, such as images and graphs also exist within this context. So the first step is to create an OpenVX context:

```
vx_context ctx = vxCreateContext();
vx_status status = vxGetStatus((vx_reference)ctx);
//check the result of context creation:
```

```
if(status != VX_SUCCES)
{
      //report error and exit, refer to the sample code for details
}
```

Notice that OpenVX has no notion of vendors, platforms, or devices.

The next step is to create a graph within the specific context:

```
vx_graph graph = vxCreateGraph(ctx);
status = vxGetStatus((vx_reference)graph);
//check the result of graph creation:
if(status != VX_SUCCES)
{
      //report error and exit, refer to the sample code for details
}
```

Using OpenCV to Create OpenVX Input from Image File

NOTE: Refer to a dedicated chapter in the SDK User Guide for the details of the OpenCV provided with the Intel® Computer Vision SDK. This includes details on environment setup and build instructions for applications using OpenCV. The User Guide also explains details on the samples using CMake* to automatically detect the installed Intel OpenCV package, and setup the paths for include directories, inputs for the linker and so on. It also describes alternative build strategies.

The sample uses the OpenCV to read and write data from/to image files.

In OpenVX, image data objects are defined as opaque. When memory access is required, for example to update the inputs, you must call vxMapImagePatch and then vxUnmapImagePatch the data.

To simplify the code, the sample operates on a single, still image as input. As the input image is not changed, the straightforward way is to create the vx_image, wrapping actual data in system memory:

```
cv::Mat src = cv::imread(file_path,CV_LOAD_IMAGE_UNCHANGED);
if(!src.data)
{
      //report error and exit, refer to the sample code for details
}
//now we need to populate the OpenVX structure to specify image layout
vx_imagepatch_addressing_t format;
//dimension in x, in pixels
format.dim_x = src.cols;
//dimension in y, in pixels
format.dim_y = src.rows;
//the rest of image format parameters (refer to the sample code for details)
…
//pointer to the Mat data
void* data = src.data;
```

```
//now we can create a vx_image (of the specified format) around Mat’s data
vx_image imageOrig =
vxCreateImageFromHandle(ctx,VX_DF_IMAGE_RGB,&format,&data,VX_MEMORY_TYPE_HOST);
status = vxGetStatus((vx_reference)imageOrig);
if(status != VX_SUCCES)
{
      //report error and exit, refer to the sample code for details
}
//creating output image, which size and type match the input
vx_image imageRes = vxCreateImage(ctx, width, height, VX_DF_IMAGE_RGB);
```

Creating Intermediate OpenVX Images as Virtual

To enable certain optimizations, in OpenVX programs the intermediate images can be declared as virtual. Such images cannot be accessed for read or write outside the graph, as there is no associated accessible memory. Basically, virtual images are stubs to define a dependency on OpenVX nodes (just like temporary variables), and these can be eliminated by the OpenVX graph compiler.

The following is an example how to create a virtual image holding a data in the NV12 format:

```
vx_image imageNV12=vxCreateVirtualImage(graph, width, height,VX_DF_IMAGE_NV12);
status = vxGetStatus((vx_reference)imageNV12);
if(status != VX_SUCCES)
{
      //report error and exit, refer to the sample code for details
}
//similarly the rest of intermediate images are defined
//virtual image to store the results of image processing (in the NV12 format)
vx_image imageNV12Eq= vxCreateVirtualImage(graph, width, height,
VX_DF_IMAGE_NV12);
//a separate virtual image to store extracted Y plane, for processing
vx_image imageChannelY  = vxCreateVirtualImage(graph, width, height,
VX_DF_IMAGE_U8);
//a separate virtual image to store results of processing the Y plane
vx_image imageChannelYEq = vxCreateVirtualImage(graph, width, height,
VX_DF_IMAGE_U8);
//a separate virtual image to store extracted U plane, left intact
vx_image imageU = vxCreateVirtualImage(graph, width / 2, height / 2,
VX_DF_IMAGE_U8);
//a separate virtual image to store extracted V plane, left intact
vx_image imageV = vxCreateVirtualImage(graph, width / 2, height / 2,
VX_DF_IMAGE_U8);
//error handing is omitted, refer to the sample for more details
```

Guidelines for Virtual and non-Vrtual Images

Notice that graph input/output images cannot be virtual. Also notice that virtual images are created for the particular graph and cannot be shared between graphs, unlike regular images created for the OpenVX context.

Finally the image metadata (size and pixel format) can be left empty in the creation but must be known in verification step, so the graph compiler should be able to deduce it from associated nodes.

Populating the Graph with Nodes

Graphs are composed of nodes. Each of these nodes is an instance of an OpenVX kernel with its associated input and output parameters. Below, the virtual images are marked as green, and non-virtual as red:

```
//the following call creates a node and plugs it to the specified graph
//notice that you should store the object to release properly (upon exit)
//refer to the dedicated chapter on releasing resources and the sample's code
vx_node node= vxColorConvertNode  (graph, imageOrig /* input RGB image */, imageNV12);
//the rest of the graph:
vxChannelExtractNode(graph, imageNV12, VX_CHANNEL_Y, imageChannelY);
vxChannelExtractNode(graph, imageNV12, VX_CHANNEL_U, imageU);
vxChannelExtractNode(graph, imageNV12, VX_CHANNEL_V, imageV);
vxEqualizeHistNode  (graph, imageChannelY, imageChannelYEq);
vxChannelCombineNode(graph, imageChannelYEq, imageU, imageV, 0, imageNV12Eq);
vxColorConvertNode  (graph, imageNV12Eq, imageRes /*output RGB image */);
```

This sequence of nodes matches the graph depicted in the earlier section.

The order of the nodes creation does not matter. Rather, the data dependencies (via images) implicitly create the edges of the graph.

Validating and Executing the Graph

Upon validation you cannot change the graph or its parameters, such as adding or removing nodes.But you can alter the actual data, for example, the data referred by the input parameters.

```
status = vxVerifyGraph(graph);
if(status != VX_SUCCES)
{
      //report error and exit, refer to the sample code for details
}
status = vxProcessGraph(graph);
//check for execution errors, refer to the sample code for details
//use data from imageRes, see next section
```

Using OpenCV to Save/Display the OpenVX Output

Any regular (non-virtual) vx_image content can be accessed on the host. To simplify debugging, this data can be wrapped with cv::Mat, and then displayed or stored to the disk using a regular OpenCV routine. Below, we access the image that was updated with a result of the graph execution:

```
//this function below is implemented in the samples for illustrative purposes
//it accesses data of the specified image, and returns the result as cv::Mat
//refer to the sample code for details
vx_map_id map_id;
cv::Mat res = IntelVXSample::mapAsMat(imageRes, VX_READ_ONLY, &map_id);
//store the results
cv::imwrite(file_name, res);
//display the results
cv::imshow("Processed image:", res);
cv::waitKey(0);
//unmap the vx_image
IntelVXSample::unmapAsMat(imageRes,res,map_id);
```

The IntelVXSample::mapAsMat() is a function implemented in the samples merely for illustrative purposes.

Carefully Releasing the OpenVX Resources

According to the OpenVX specification, it is your responsibility to release resources properly. This includes nodes, images, graphs and context.

Notice that releasing a graph, doesn't mean releasing associated virtual images.OpenVX relies on the reference counting for tracking the resources. In contrast, after releasing a context, all of it's reference counted objects are garbage-collected.

The rule of thumb is that for any resource type, it is very important to decrease the reference counter when you don't need the resource anymore. This is done by calling the appropriate release routine. Here is an example of releasing various resources:

```
//releasing an example node, created in the earlier section
status = vxReleaseNode(node);
//check status and report any error, refer to the sample code for details
//releasing the rest of nodes
…
status = vxReleaseImage(imageOrig);
//check status and report any error, refer to the sample code for details
//releasing the rest of images
…
//release graph
status = vxReleaseGraph(&graph);
//check status and report any error, refer to the sample code for details
...
```

```
//release context
status = vxReleaseContext(&ctx);
//check status and report any error, refer to the sample code for details
```

Notice that the sample keeps std::vector for nodes and images to release them upon exit. An alternative strategy would be to release a node or an image, once they are added (i.e. get referenced) to the graph.

Sample Directory

The sample directory has the following structure:

+-- **auto_contrast** (a separate directory per sample)

  +-- **auto_contrast.cpp** (main sample file with code)

  +-- **CMakeLists.txt** (CMake file for the sample)

  +-- **low_contrast_vga.jpg** (default input to the sample, 640x480 RGB image)

  +-- **sample_auto_contrast_user_guide.pdf** (this User's Guide)

  +-- **README** (short readme file for the sample)

Building and Running the Sample

You need to compile the sample code on a machine running Ubuntu* 14.04. You need the following software to compile the sample:

- CMake* 2.8.12 or higher
- GCC* 4.8.4 or higher

Refer to the *README* file in the sample root directory for details about compilation of the samples. The sample targets systems with Intel microarchitecture code name Broxton running Yocto*. Refer to the *README* file in the samples root directory for details about installing and running the samples and system requirements for target platforms.

During execution the sample logs every step by sending messages into console, and outputs execution statistics upon exit. It also renders the input and resulting images (when running in the GUI-enabled environment) using OpenCV .

Basic Command-Line Options for the Sample

To get detailed list of command-line parameters, run the sample with '-h' ('--help'):

```
$ ./auto_contrast  --help
```

By default the sample loads and processes the low_contrast_vga.jpg, available with the sample directory.

- To change the input file, use '-i' (or '--input')
- To store the output to a file, use '-o' ('--output')
- To process input as a gray-scale image, use '-g'

Example command-line:

```
$ ./auto_contrast  -i ../images/test.jpg -g -o result.jpg
```

The sample also supports a simple loop around graph execution, which provides more stable performance numbers. A corresponding command-line option is '-l or '--loops':

```
$ ./auto_contrast  -l 10
```

An example output with basic performance statistics from the actual target platform (Broxton-P based system running Yocto) is shown below. Notice how execution statistics of the graph are averaged by 10 iterations:

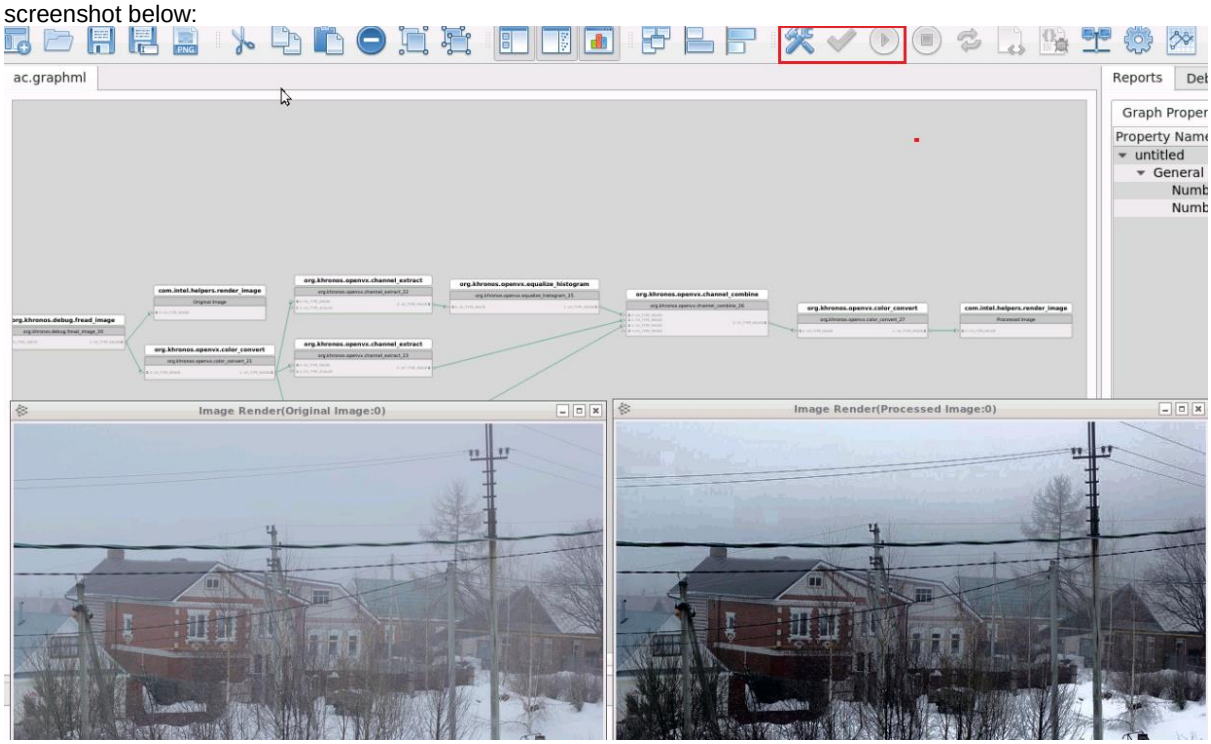
```
$ ./auto_contrast  -l 10
[ INFO ] Reading input file using OpenCV file I/O: ./low_contrast_vga.jpg
[ INFO ] Input file ok: 640x480 and 3 channel(s)
[ INFO ] Verifying the graph: OK 0x25847a0
[ INFO ] Graph execution is OK 0x25847a0
End of execution
4.322585 ms by vxProcessGraph averaged by 10 samples
```

To skip rendering the input/output images, use '-n' (or '--no-show'). This option is useful when you are not running in the GUI environment.

Alternative Way to Play with OpenVX: Vision Algorithm Designer (VAD)

Intel® Computer Vision SDK comes with a handy tool for visual graph creation, execution and analysis. It is relatively straightforward to implement the sample's same graph in the Vision Algorithm Designer (VAD).

The tool comes with several pre-defined graph examples (available in the Graphs tab), including the graph for Auto-Contrast. So you can load, inspect and play with the auto_contrast.xml. The execution workflow with VAD is to go thru the sequence of **Build > Verify > Execute** for the graph, by pressing buttons marked in the screenshot below:



See the Vision Algorithm Designer User Guide for more details on the workflow.

VAD can also generate the code for the graph in a form of .h/.cpp anc cmake files with graph "factory" – a function that creates and returns the desired graph. You can plug this code to your application then.

However, even with the tool-generated code, you need to correctly setup the include and library paths. You can re-use the samples CMake* infrastructure for that. See the samples root *README* file for details.

Understanding the Graph Performance with Vision Algorithm Designer (VAD)

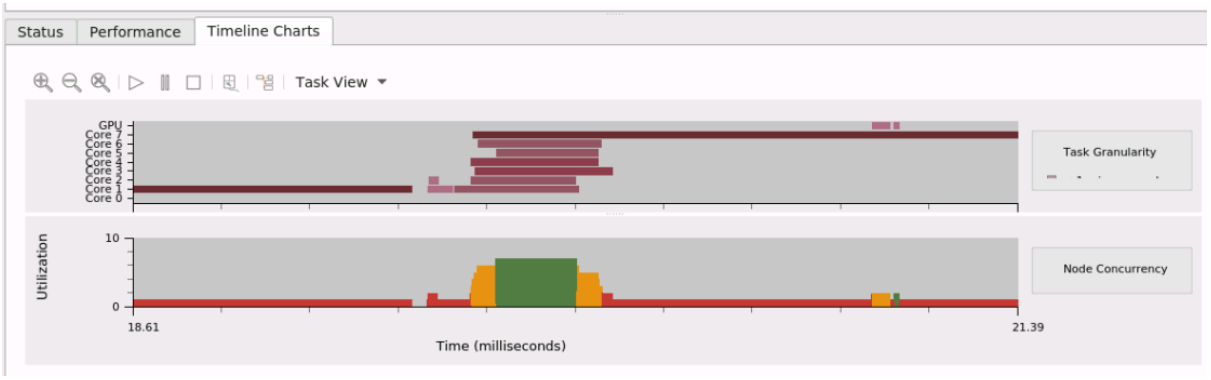
One of the particularly useful features of VAD is performance breakdown for a graph.

Below is example of the execution flow for the same graph in the timeline view (notice that this is “Task View” that correlates execution to the compute resources, like CPU cores in the example below).

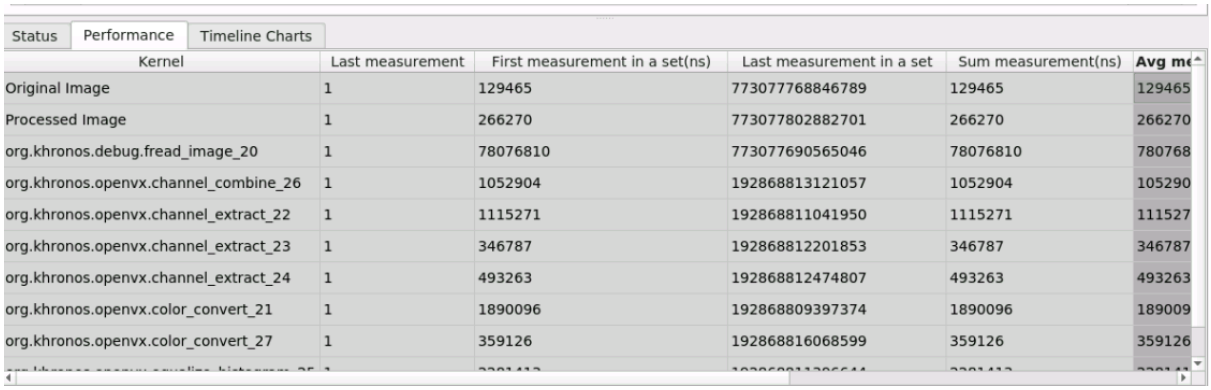
The graph starts with reading the image from the file (dark red bar, attributed just to one core, as the code is the file i/o code is serial), followed by actual graph nodes (red bars) that spans all the cores (thanks to the tiling). Together, the nodes saturate the CPU device really well, refer to the Utilization timeline.

Finally, the image display routines pop up as another instance of the render_image heleper node (again dark red), which displays the result when the graph completes.

Total execution (excluding file_read and render_image helper nodes) is few milliseconds (on a development system with the Intel® microarchitecture code named Broadwell, running Ubuntu* OS). This correlates well with execution statistics of the human-produced code (this sample).



Another view in the Performance tab allows to see various execution statistics for individual nodes:



Kernel	Last measurement	First measurement in a set(ns)	Last measurement in a set	Sum measurement(ns)	Avg me
Original Image	1	129465	773077768846789	129465	129465
Processed Image	1	266270	773077802882701	266270	266270
org.khronos.debug.fread_image_20	1	78076810	773077690565046	78076810	780768
org.khronos.openvx.channel_combine_26	1	1052904	192868813121057	1052904	105290
org.khronos.openvx.channel_extract_22	1	1115271	192868811041950	1115271	111527
org.khronos.openvx.channel_extract_23	1	346787	192868812201853	346787	346787
org.khronos.openvx.channel_extract_24	1	493263	192868812474807	493263	493263
org.khronos.openvx.color_convert_21	1	1890096	192868809397374	1890096	189009
org.khronos.openvx.color_convert_27	1	359126	192868816068599	359126	359126

Refer to the SDK User Guide for details on the Performance analysis with Intel ® VTune.

References

- OpenVX 1.1 Specification
- Intel Computer Vision SDK User Guide
- Vision Algorithm Designer User Guide