frontiers in ICT

CODE
published: 21 November 2016
doi: 10.3389/fict.2016.00028

OpenVX-Based Python Framework for Real-time Cross-Platform Acceleration of Embedded Computer Vision Applications

*Ori Heimlich and Elishai Ezra Tsur**

Neuro-Biomorphic Engineering Lab, Faculty of Engineering, Jerusalem College of Technology, Jerusalem, Israel

Embedded real-time vision applications are being rapidly deployed in a large realm of consumer electronics, ranging from automotive safety to surveillance systems. However, the relatively limited computational power of embedded platforms is considered as a bottleneck for many vision applications, necessitating optimization. OpenVX is a standardized interface, released in late 2014, in an attempt to provide both system and kernel level optimization to vision applications. With OpenVX, Vision processing is modeled with coarse-grained data flow graphs, which can be optimized and accelerated by the platform implementer. Current full implementations of OpenVX are given in the programming language C, which neither supports advanced programming paradigms, such as object-oriented, imperative, and functional programming, nor does it has runtime or type checking. Here, we present a python-based full Implementation of OpenVX, which eliminates much of the discrepancies between the object-oriented paradigm used by many modern applications and the native C implementations. Our open-source implementation can be used for rapid development of OpenVX applications in embedded platforms. Demonstration includes static and real-time image acquisition and processing using a Raspberry Pi and a GoPro camera. Code is given in Supplementary Material. Code project and linked deployable virtual machine are located on GitHub: https://github.com/NBEL-lab/PythonOpenVX.

Keywords: embedded computer vision, code: Python, real-time, OpenVX, object-oriented framework

OPEN ACCESS

Edited by:
Giuseppe Boccignone, University of Milan, Italy

Reviewed by:
Stefano Berretti, University of Florence, Italy
Helder Araujo, University of Coimbra, Portugal

***Correspondence:**
Elishai Ezra Tsur
elishai85@gmail.com

Specialty section:
This article was submitted to Vision Systems Theory, Tools and Applications, a section of the journal Frontiers in ICT

Received: *29 August 2016*
Accepted: *31 October 2016*
Published: *21 November 2016*

Citation:
Heimlich O and Ezra Tsur E (2016) OpenVX-Based Python Framework for Real-time Cross-Platform Acceleration of Embedded Computer Vision Applications. Front. ICT 3:28. doi: 10.3389/fict.2016.00028

INTRODUCTION

In the last two decades, the emergence of powerful, low-cost, and energy-efficient micro-processors enabled computer vision (CV) capabilities to be applied in embedded platforms, allowing real-time machines to visually interpret their environment (Kisacanin and Gelautz, 2014). Embedded vision applications are currently ingrained into many aspects of modern life, from automotive safety (Mandal et al., 2014), optical character recognition (Neumann and Matas, 2012), and gesture interfaces (Rautaray and Agrawal, 2015) to medical instrumentation (Economou and Papaioannou, 2013) and surveillance systems (Lin et al., 2012).

Three main challenges are currently considered as bottlenecks in the embedded vision pipeline: the growing diversity of camera hardware, the relatively limited computational resources of embedded

and real-time platforms, and the execution of vision-based decisions, due to the required integration with other components, such as sensors and actuators (Lee et al., 2015; Dedeoğlu et al., 2011). Obviously, the underlying challenge of each bottleneck is standardization.

Here, we address the second bottleneck: acceleration of CV application for real-time platforms. A general use case of an embedded vision application starts with the application (application layer), which makes use of a CV library (framework layer). The CV library uses an accelerator API (acceleration layer) that was implemented to run on a specific machine (engine layer).

Embedded acceleration of CV application can be performed in both system and kernel level. While system-level optimization issues, such as power consumption and memory bandwidth, are generally dealt within the engine layer, kernel optimization is locally achieved within the framework layer, where specific algorithms are refactored with more efficient implementation. Both approaches have their limitations: system-level optimization has to be addressed at the framework level, and kernel optimization has only a local effect on the efficiency of the entire algorithmic process (Rainey et al., 2014).

OpenVX is a standardized interface, designed by the Khronos Group and released in late 2014, in an attempt to provide both system and kernel level optimization by modeling the system with graphs, which can be used for optimization and acceleration by the platform implementer (Rainey et al., 2014; Lin et al., 2015; Tagliavini et al., 2014). This model of standardization defers the responsibility for optimization from

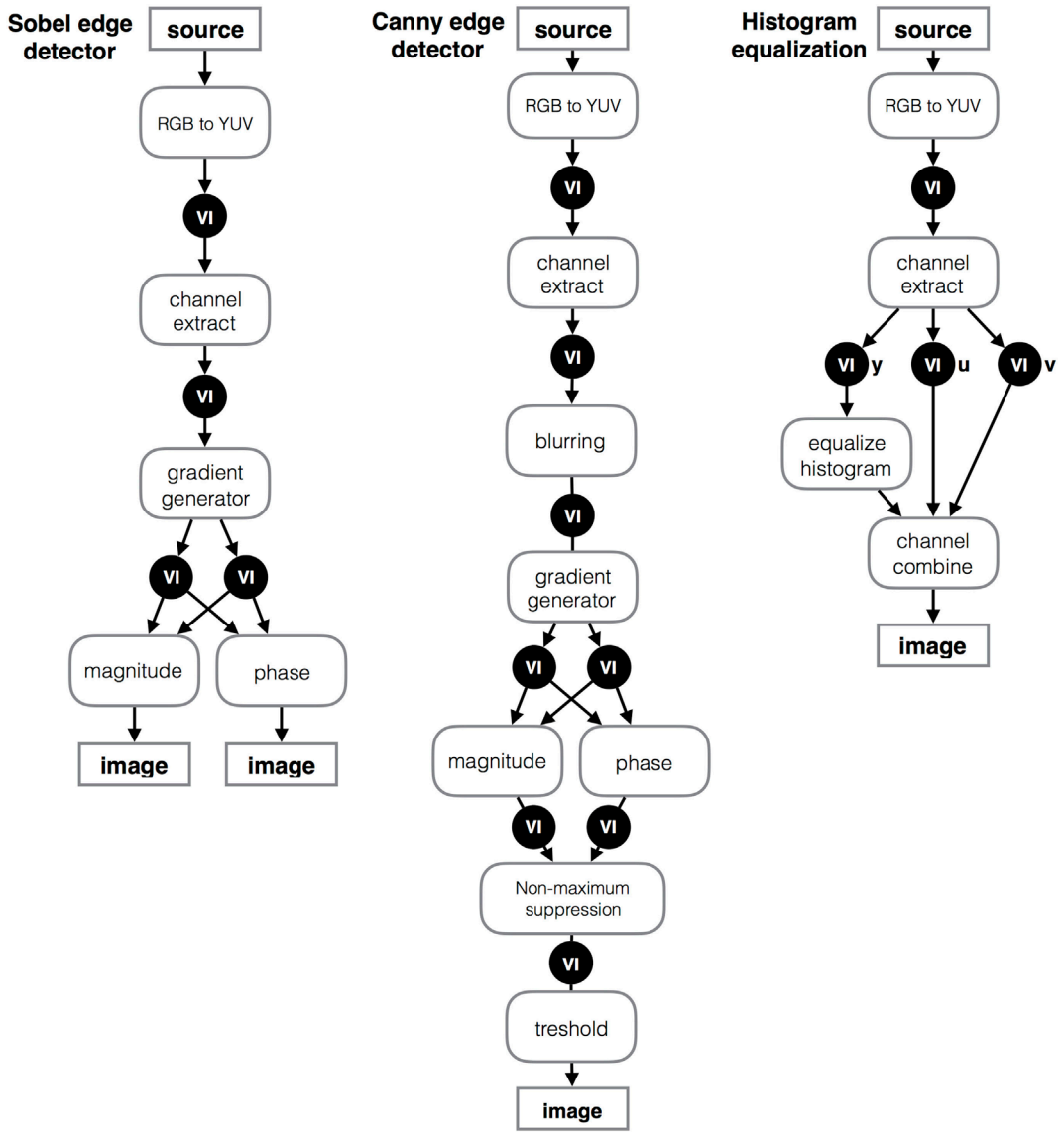

FIGURE 1 | Modeling vision processing with coarse-grained data flow graphs. Nodes represent atomic kernels performed on image data, connected internally *via* virtual images and externally *via* real images.

the user to the platform developer – separating the application and the hardware knowledge domains. For example, NVIDIA Corporation, one of the world's leading companies in the design of graphics processing units (GPUs), developed the VisionWorks toolkit – an implementation of OpenVX for CUDA-capable GPUs and System on Chips (Soc) (Elliott et al., 2015). CUDA is a software layer, also developed by NVIDIA, that encapsulates GPU with a virtual instruction set, which enables a distributed execution of kernels. The development model of OpenVX is similar to the standardization of computer graphics provided by OpenGL (Rainey et al., 2014), which specifies an abstract interface for defining graphical components. OpenGL applications are extensively used in a wide spectrum of fields ranging from virtual reality to computer-aided design.

Briefly, an OpenVX application starts with constructing a directed acyclic graph in which nodes representing atomic kernels performed on image data. Nodes are connected internally *via* virtual images (which can be optimized out) and externally *via* real images. Essentially, image processing stages are arranged in a connected graph. Nodes (or processing primitives) are then distributed to available hardware, as it is implemented by the OpenVX implementation provider. OpenVX defines an extendible library of ~40 primitives ranging from simple arithmetic to key points detection or tracking (Khronos Group, 2016). Basic CV data structure includes images, convolution matrices arrays, and scalars. We note that OpenVX did not invent the concept of harnessing the expression power of graphs to accelerate vision application. This idea was already proposed by Olson et al. (1993), who described vision processing in terms of coarse-grained data flow graphs, constructed using a graphical interface. However, while their implementation was specifically designed for the Datacube MV20, a pipeline image processor that was once widely used for research and robot vision, OpenVX provides a standardization of the process – regardless of the used hardware.

Examples for OpenVX graph model for Canny edge detection (Canny, 1986), Sobel edge detection (Rainey et al., 2014), and histogram equalization (Lepley, 2015) are given in **Figure 1**.

Graph level optimization strategies include remote processing, aggregate function replacements, inner-processor communication aggregation, peer-to-peer topologies, parallelism, and pipelining. All are reviewed in length by Rainey et al. (2014).

Current full implementations of OpenVX are given in the programming language C. While C programming supports low level operations, structure programming, and portability, it does not support advanced programming paradigms, such as object-oriented programming, and has no runtime or type checking. Therefore, current OpenVX applications are cumbersome. Moreover, current OpenVX implementations frequently cause discrepancies between the object-oriented model used by many modern applications and native C implementations. Code for Canny edge detection is given in **Code listing 1**.

Code listing for Sobel edge detection and histogram equalization, which were described above, is given in Code listing S1 and S2 in Supplementary Material, respectively.

CODE LISTING 1 | Implementation of Canny edge detection in the C implementation of OpenVX.

```
vx_image nvidia_graph(vx_image input){
  int width = 100, height = 100;
  vx_status status = VX_SUCCESS;
  vx_ context context = vxCreateContext();
  vx_graph graph = vxCreateGraph(context);
  vx_int32 val = 0;
  vx_scalar shift = vxCreateScalar(context, VX_TYPE_INT32, &val);
  vx_image output = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);

  vx_image iyuv = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_IYUV);
  vx_image y = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image by = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image gx = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image gy = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image ang = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image mag = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT);
  vx_image non = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8);

  vx_threshold t = vxCreateThreshold(context ,VX_THRESHOLD_TYPE_RANGE,
  VX_TYPE_UINT8);
  vx_uint8 upper = 240 , lower = 10;

  vxColorConvertNode (graph, input, iyuv);
  vxChannelExtractNode (graph, iyuv, VX_CHANNEL_Y, y);
  vxGaussian3x3Node (graph, y, by);
  vxSobel3x3Node (graph, by, gx, gy);
  vxPhaseNode  (graph, gx, gy, ang);
  vxMagnitudeNode (graph, gx, gy, mag);
  vxConvertDepthNode (graph, mag, non, VX_CONVERT_POLICY_WRAP, shift);
  vxThresholdNode (graph, non, t, output);
  vxSetThresholdAttribute(t, VX_THRESHOLD_ATTRIBUTE_THRESHOLD_
      UPPER, &upper, sizeof(upper));
  vxSetThresholdAttribute(t, VX_THRESHOLD_ATTRIBUTE_THRESHOLD_
      LOWER, &lower, sizeof(lower));
  status = vxVerifyGraph(graph);
  if (status == VX_SUCCESS){
    status = vxProcessGraph(graph);
    if (status == VX_SUCCESS){
      return output;
    }
  }
  return NULL;
}
```

Modern programming environments, such as C#, Java, Python, and others, offer a more efficient and simple environment relative to C. They feature high-levels of expressions, greater readability and object-oriented architecture. A naïve solution for bridging different programming environments is called language binding. Binding acts as a bridge between one programming language to another, allowing invocation of kernels from different environments. For example, OpenGL, which was built upon the same development model as OpenVX, has many language bindings such as for JavaScript (WebGL), C, and Java. Indeed, the official release of OpenVX includes language binding services to Java and Ruby. Both are high level programming languages that supports modern programming paradigms. Although bindings play important roles in cross-platform applications, they are only a "glue-code" that bridge incompatibilities between languages. They do not contribute

functionality or framework, and therefore, they are not sufficient to simplify or enrich low level implementation with modern approaches.

Python is a cross-platform, widely spread programming language, which is also interpreted, dynamic, and modern, that supports multiple programming paradigms, such as object-oriented, imperative, and functional programming. Many examples for Python-binded implementations exists; among them are PyGtk, which binds the Gtk Toolkit – a cross-platform toolkit for the generation of graphical user interfaces, originally written in C, and rpy2, which is used to bridge the language R to Python.

Here, we utilized a binding implementation for Python and C, to provide a Python-based full object-oriented framework for OpenVX.

IMPLEMENTATION AND ARCHITECTURE

Description

Our OpenVX Python platform is encapsulated within few main packages: OpenVX implementation in C, our implementation of OpenVX in Python, and a binding package, which encapsulates the native C implementation and bridges it to our implementation. Package view of our design is shown in Figure S1 in Supplementary Material. This design uncouples the original OpenVX C implementation from our Python framework. Therefore, our framework can be used for any vendor-specific implementation of OpenVX. Here, the OpenVX interface and the native C sample implementation of OpenVX (Khronos Group, 2016) was provided by Khronos group, and the Python binding for OpenVX was adopted from Ardo Hakan's work (termed PyVX), which was distributed *via* PyPI, the official repository of Python (Ardo, 2014). Class diagram of our python implementation of OpenVX is shown in Figure S2 in Supplementary Material. This hierarchical class architecture provides a full object-oriented framework for OpenVX, enabling the developer to invoke each of the specified kernels, with the appropriate parameters. Our implementation includes the entire OpenVX library of ~40 kernels in a single static class (not shown in Figure S2 in Supplementary Material). Full specification of our implementation of OpenVX library is given in Supplementary Material.

Application, Example of Use, and Methods

OpenVX provides a definition of a series of image processing kernels, which can be combinatorially interconnected to form graphs for the implementation of complicated vision tasks. Here, we utilized various kernels to implement edge detections and histogram equalization graph models as they were described above.

Our Python implementation of OpenVX dramatically simplifies the exemplified Code listing S1–S3 (given in Supplementary Material). Code example of Canny edge detection is given in **Code listing 2**. Additional code examples for Sobel edge detection and histogram equalization using our Python implementation are given in Code listing S3 and S4 in Supplementary Material, respectively.

CODE LISTING 2 | Implementation of Canny edge detection in our Python implementation of OpenVX.

```
def cannyEdgeGraph():
    width = 100
    height = 100
    with Graph(verify=True) as g:
        rgb = Image(g.context, width, height, Color.VX_DF_IMAGE_RGB)
        out = Image(g.context, width, height, Color.VX_DF_IMAGE_U8)
        threshold = Threshold(g.context, thresh_type=Threshold.
        THRESHOLD_TYPE_RANGE)
        threshold.set_upper(240)
        threshold.set_lower(10)
        iyuv = ColorConvertNode(g, rgb, color=Color.VX_DF_IMAGE_IYUV)
        y = ChannelExtractNode(g, iyuv, Channel.VX_CHANNEL_Y)
        by = Gaussian3x3Node(g, y)
        gx, gy = Sobel3x3Node(g, by)
        mag = MagnitudeNode(g, gx, gy)
        converted = ConvertDepthNode(g, mag, Policy.
            VX_CONVERT_POLICY_WRAP,
            color=Data_type.VX_TYPE_UINT8)
        ThresholdNode(g, converted, threshold, out)
    g.vxProcessGraph()
```

We utilized Python 2.7.6, the compiled sample implementation of OpenVX provided by Khronos, PyVX, and our implementations of OpenVX for edge detections and histogram equalization on a Raspberry PI 3 Model B board. Results are shown in **Figure 2A**. Files are provided in our code repository as stated in the *Code repository section*. We used the PyCharm Python development environment by JetBrains.

Our implementation of edge detection can be iteratively executed in real-time for processing of camera-acquired images. However, image acquisition is not a part of OpenVX, and therefore, our framework has to be embedded within an existing image acquisition module. In fact, standardization of image acquisition is an important bottleneck in embedded CV, as was mentioned earlier.

In order to exemplified the embodiment of OpenVX in an image acquisition module, we deployed our framework on a Raspbery-Pi 3, connected it to a GoPro camera (with fish-eye lens) and performed phase and edge detection processing. Setup and results from an image taken statically from the camera is shown in **Figures 2B,C**, and real-time processing is shown in **Figure 2D**. Code is given in Code listing S5 in Supplementary Material.

DISCUSSION

Embedded vision applications are currently constrained by the growing diversity of camera hardware, the relatively limited computational resources of embedded and real-time platforms and the execution of vision-based decisions. An attempt to relax the second constraint of limited computational resources in embedded platforms was made in late 2014 by Khronos Group, which released their OpenVX standardization for vision optimization. OpenVX provides definitions of 41 image processing kernels, which can be combinatorially interconnected as graphs for the implementation of complicated vision tasks.

Current full implementations of OpenVX are given in C, a fact that limits the usability, readability and adaptability of OpenVX

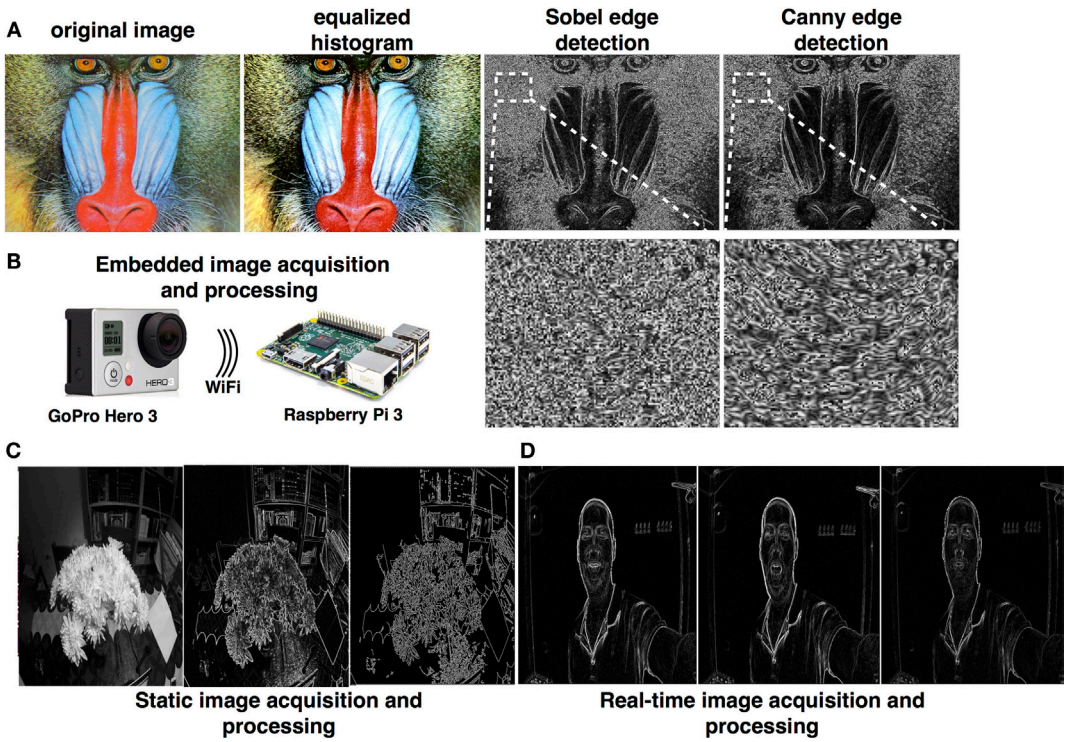

FIGURE 2 | **(A)** Sample results of executing histogram equalization, Soble edge detection, and Canny edge detection using our Python implementation of OpenVX. **(B)** Schematics of our embedded framework, which is comprised of a Raspberry Pi and a GoPro camera. **(C,D)** Static and real-time image acquisition and processing.

application due to the language's lack of support of modern programming paradigms and advanced error handling. Previous attempts to provide implementation of OpenVX in Python, including PyVX (Ardo, 2014). PyVX neither includes implementation of kernels nor a full object-oriented framework for OpenVX. On the other hand, PyVX provides a bridge, or a "glue-code," between the C implementation and Python, providing us with an important stepping stone toward our end-goal of providing a Pythonic object-oriented framework for OpenVX. We used PyVX as a binding service and provided a full object-oriented framework to OpenVX. Our library can be used to implement OpenVX applications on open-source embedded platforms that support Python, such as Raspberry pi and Arduino. Other platforms, such as iOS-based devices, may need to use an additional interpreter, such as Pythonista, to use this framework. We note that since image acquisition is not a part of OpenVX, our framework has to be embedded within an existing image acquisition module as was exemplified above.

Empowering open-source embedded platforms with a straight forward API in a powerful programming language can open a broad spectrum of new possibilities.

CODE REPOSITORY

Code repository
Name: GitHub
Identifier: https://github.com/NBEL-lab/PythonOpenVX
Licence: GNU General Public License V3
Date published: 29/04/2016
Language of repository: Python 2.7x.

Supporting files: OpenVX sample OpenVX implementation and the PyVX package – both are freely available. In order to ease framework installation, we have created a virtual machine that contains the operating system, the compiled environment, and examples, and can be deployed any VM framework, such as Oracle VirtualBox. Download links and detailed instructions for use are given in the README file, located at the repository.

AUTHOR CONTRIBUTIONS

ET and OH conceptualized and designed the framework. OH wrote the code, and ET wrote the manuscript.

FUNDING

This work was supported by JCT research grant.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at http://journal.frontiersin.org/article/10.3389/fict.2016.00028/full#supplementary-material.

REFERENCES

Ardo, H. (2014). *PyVX*. Available at: https://pypi.python.org/pypi/PyVX/0.2.7.

Canny, J. (1986). A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 679–698. doi:10.1109/TPAMI.1986.4767851

Dedeoğlu, G., Kisačanin, B., Moore, D., Sharma, V., and Miller, A. (2011). "An optimized vision library approach for embedded systems," in *Computer Vision and Pattern Recognition* (Colorado Springs).

Economou, G.-P. K., and Papaioannou, V. (2013). "Medical decision making via artificial neural networks: a smart phone-embedded application addressing pulmonary diseases' diagnosis," in *Engineering Applications of Neural Networks* (Berlin, Heidelberg: Springer), 156–163.

Elliott, G. A., Yang, K., and Anderson, J. H. (2015). "Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms," in *Real-Time Systems Symposium* (San Antonio).

Khronos Group. (2016). *Khronos OpenVX Registry*. Available at: https://www.khronos.org/registry/vx/

Kisacanin, B., and Gelautz, M. (2014). *Advances in Embedded Computer Vision*. Cham: Springer.

Lee, H., Trevett, N., Olson, T., Erukhimov, V., and Oh-bach, A. (2015). "How mobile devices are revolutionizing user interaction," in *Extended Abstracts on Human Factors in Computing Systems* (New York).

Lepley, T. (2015). "Tegra X1 and VisionWorks/OpenVX: computer vision on a chip," in *Signal Images: Architecture and Programming GPU* (France).

Lin, F., Dong, X., Chen, B. M., Lum, K.-Y., and Lee, T. H. (2012). A robust real-time embedded vision system on an unmanned rotorcraft for ground target following. *IEEE Trans. Ind. Electron.* 59, 1038–1049. doi:10.1109/TIE.2011.2161248

Lin, T.-H., Lin, C.-Y., and Lee, J.-K. (2015). "Scheduling methods for OpenVX programs on heterogeneous multi-core systems," in *Parallel and Distributed Processing Techniques and Applications* (Las Vegas).

Mandal, D. K., Sankaran, J., Gupta, A., Castille, K., Gondkar, S., Kamath, S., et al. (2014). *An Embedded Vision Engine (EVE) for Automotive Vision Processing*. Melbourne: IEEE.

Neumann, L., and Matas, J. (2012). "Real-time scene text localization and recognition," in *Computer Vision and Pattern Recognition* (Providence).

Olson, T. J., Lockwood, R. J., and Taylor, J. R. (1993). "Programming a pipelined image processor," in *Computer Architectures for Machine Perception* (New Orleans).

Rainey, E., Villarreal, J., Dedeoglu, G., Pulli, K., Lepley, T., and Brill, F. (2014). "Addressing system-level optimization with OpenVX graphs," in *Computer Vision and Pattern Recognition Workshops* (Columbus).

Rautaray, S. S., and Agrawal, A. (2015). Vision based hand gesture recognition for human computer interaction: a survey. *Artif. Intell. Rev.* 1, 1–54. doi:10.1007/s10462-012-9356-9

Tagliavini, G., Haugou, G., and Benini, L. (2014). "Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators," in *Design and Architectures for Signal and Image Processing* (Madrid).

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.