OpenVX (Open Computer Vision Acceleration API) is a low-level programming framework domain to access computer vision hardware acceleration with both functional and performance portability. OpenVX has been designed to support modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems.

[n.n] refers to sections in the OpenVX 1.0 specification available at khronos.org/registry/vx/

Parameter color coding: ***Output***, ***Input***, and ***Input/Output*** parameters.

Parameters marked as [Opt.] are optional. If omitted, NULL.

- Indicates the overflow policy used is VX_CONVERT_POLICY_SATURATE.

Graph and Immediate Mode Vision Functions

Functions in OpenVX may be graph mode or immediate mode.

Graph Mode	Immediate Mode
Graph nodes may be created and linked together, verified by the implementation, then executed as often as needed.	Immediate mode functions are executed on a context immediately, as if they were single node graphs, with no leaking side-effects.
vx_node **vx*Name*Node** (vx_graph *graph*, T *arg1* , . . . , T *argN*); ***graph***: The reference to the graph	vx_status **vxu*Name*** (vx_context *context*, T *arg1*, . . . , T *argN*); ***context***: The reference to the overall context

In the Vision functions below, replace *ret* with vx_node or vs_status, modify the function name as appropriate, and replace the placeholder in the first argument with vx_graph *graph* or vx_context *context*.

Vision Functions

Absolute Difference [3.2]

$out(x, y) = |in_1(x, y) - in_2(x, y)|$

ret **vx[u]AbsDiff[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_U8 format.

Accumulate [3.3]

- $accum(x, y) = accum(x, y) + input(x, y)$

ret **vx[u]AccumulateImage[Node]** (*graph*|*context*, vx_image *input*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.

accum: The accumulation image of VX_DF_IMAGE_S16 format.

Accumulate Squared [3.4]

- Accumulates a squared input image to an output image.

ret **vx[u]AccumulateSquareImage[Node]** (*graph*|*context*, vx_image *input*, vx_scalar *shift*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.

shift: Shift amount of type VX_TYPE_UINT32 ($0 \le shift \le 15$)

accum: The accumulation image of VX_DF_IMAGE_S16 format.

Accumulate Weighted [3.5]

$accum(x, y) = (1 - \alpha) * accum(x, y) + \alpha * input(x, y)$

ret **vx[u]AccumulateWeightedImage[Node]** (*graph*|*context*, vx_image *input*, vx_scalar *alpha*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.

alpha: The scale of type VX_TYPE_FLOAT32 ($0.0 \le \alpha \le 1.0$)

accum: The accumulation image of VX_DF_IMAGE_U8 format.

Arithmetic Addition [3.6]

$out(x, y) = in_1(x, y) + in_2(x, y)$

ret **vx[u]Add[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_{U8, S16} format.

policy: A vx_convert_policy_e enumeration.

Arithmetic Subtraction [3.7]

$out(x, y) = in_1(x, y) - in_2(x, y)$

ret **vx[u]Subtract[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_{U8, S16} format.

policy: A vx_convert_policy_e enumeration.

Bitwise AND [3.8]

$out(x, y) = in_1(x, y) \wedge in_2(x, y)$

ret **vx[u]And[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.

out: The output image of VX_DF_IMAGE_U8 format.

Bitwise EXCLUSIVE OR [3.9]

$out(x, y) = in_1(x, y) \oplus in_2(x, y)$

ret **vx[u]Xor[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.

out: The output image of VX_DF_IMAGE_U8 format.

Bitwise INCLUSIVE OR [3.10]

$out(x, y) = in_1(x, y) \vee in_2(x, y)$

ret **vx[u]Or[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.

out: The output image of VX_DF_IMAGE_U8 format.

Bitwise NOT [3.11]

$out(x, y) = \overline{in(x, y)}$

ret **vx[u]Not[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Box Filter [3.12]

Computes a Box filter over a window of the input image.

ret **vx[u]Box3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Canny Edge Detector [3.13]

Provides a Canny edge detector kernel.

ret **vx[u]CannyEdgeDetector[Node]** (*graph*|*context*, vx_image *input*, vx_threshold *hyst*, vx_int32 *gradient_size*, vx_enum *norm_type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

hyst: The double threshold for hysteresis.

gradient_size: Size of Sobel filter window; supports at least 3, 5, 7.

norm_type: Norm used to compute the gradient, VX_NORM_{L1, L2}.

Channel Combine [3.14]

Combines multiple image planes.

ret **vx[u]ChannelCombine[Node]** (*graph*|*context*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*);

plane{0, 1}: Plane forming channel {0, 1}. VX_DF_IMAGE_U8.

plane{2, 3}: [Opt.] Plane that forms channel {2, 3}. VX_DF_IMAGE_U8.

output: The output image.

Channel Extract [3.15]

Extracts a plane from a multi-planar or interleaved image.

ret **vx[u]ChannelExtract[Node]** (*graph*|*context*, vx_image *input*, vx_enum *channel*, vx_image *output*);

input: One of the defined vx_df_image_e multi-planar formats.

channel: The vx_channel_e channel to extract.

output: The output image of VX_DF_IMAGE_U8 format.

Color Convert [3.16]

Converts the format of a vx_df_image_e image to another.

ret **vx[u]ColorConvert[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: The image {to, from} which to convert.

Convert Bit Depth [3.17]

- Converts image bit depth.

ret **vx[u]ConvertDepth[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_scalar *shift*);

input, output: The {input, output} image.

policy: A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.

shift: The shift value of type VX_TYPE_INT32.

Custom Convolution [3.18]

- Convolves an image with a user-defined matrix.

ret **vx[u]Convolve[Node]** (*graph*|*context*, vx_image *input*, vx_convolution *conv*, vx_image *output*);

input: An input image of VX_DF_IMAGE_U8 format.

conv: The vx_int16 convolution matrix.

output: The output image of VX_DF_IMAGE_{S16, U8} format.

Dilate Image [3.19]

Grows the white space in a VX_DF_IMAGE_U8 Boolean image.

ret **vx[u]Dilate3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Equalize Histogram [3.20]

Normalizes brightness and contrast of a grayscale image.

ret **vx[u]EqualizeHist[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: The grayscale image of VX_DF_IMAGE_U8 format.

Erode Image [3.21]

Shrinks the white space in a VX_DF_IMAGE_U8 Boolean image.

ret **vx[u]Erode3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Fast Corners [3.22]

Finds corners in an image.

ret **vx[u]FastCorners[Node]** (*graph*|*context*, vx_image *input*, vx_scalar *strength_thresh*, vx_bool *nonmax_suppression*, vx_array *corners*, vx_scalar *num_corners*);

input: The input image of VX_DF_IMAGE_U8 format.

strength_thresh: (VX_TYPE_FLOAT32) Intensity difference threshold.

nonmax_suppression: Boolean specifying if non-max. suppression is applied to detected corners before being placed in the vx_array.

corners: Output corner vx_array of type VX_TYPE_KEYPOINT.

num_corners: [Opt.] Number of detected corners in image. (VX_TYPE_SIZE)

Gaussian Filter [3.23]

Computes a Gaussian filter over a window of the input image.

ret **vx[u]Gaussian3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Harris Corners [3.24]

Computes the Harris Corners of an image.

ret **vx[u]HarrisCorners[Node]** (*graph*|*context*, vx_image *input*, vx_scalar *strength_thresh*, vx_scalar *min_distance*, vx_scalar *sensitivity*, vx_int32 *gradient_size*, vx_int32 *block_size*, vx_array *corners*, vx_scalar *num_corners*);

input: An input image of VX_DF_IMAGE_U8 format.

strength_thresh: The minimum threshold of type VX_TYPE_FLOAT32 with which to eliminate Harris corner scores.

min_distance: The radial Euclidean distance of type VX_TYPE_FLOAT32 for non-maximum suppression.

sensitivity: The scalar sensitivity threshold *k* of type VX_TYPE_FLOAT32.

gradient_size: The gradient window size to use on the input.

block_size: Block window size used to compute the Harris corner score.

corners: The array of objects of type VX_TYPE_KEYPOINT.

num_corners: [Opt.] Number of detected corners in image. (VX_TYPE_SIZE)

Histogram [3.25]

Generates a distribution from an image.

ret **vx[u]Histogram[Node]** (*graph*|*context*, vx_image *input*, vx_distribution *distribution*);

input: The input image of VX_DF_IMAGE_U8 format.

distribution: The output distribution.

Gaussian Image Pyramid [3.26]

Computes a Gaussian image pyramid using a 5x5 kernel.

ret **vx[u]GaussianPyramid[Node]** (*graph*|*context*, vx_image *input*, vx_pyramid *gaussian*);

input: The input image of VX_DF_IMAGE_U8 format.

gaussian: The Gaussian pyramid of VX_DF_IMAGE_U8 format.

Integral Image [3.27]

Computes the integral image of the input.

ret **vx[u]IntegralImage[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.

output: The output image of VX_DF_IMAGE_U32 format.

Magnitude [3.28]

$mag(x, y) = \sqrt{grad_x(x, y)^2 + grad_y(x, y)^2}$

ret **vx[u]Magnitude[Node]** (*graph*|*context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *mag*);

grad_{x, y}: The input {x, y} image of VX_DF_IMAGE_S16 format.

mag: The magnitude image of VX_DF_IMAGE_S16 format.

Mean and Standard Deviation [3.29]

Computes the mean and standard deviation of the input pixels.

vx_node **vxMeanStdDevNode** (vx_graph *graph*, vx_image *input*, vx_scalar *mean*, vx_scalar *stddev*);

vx_status **vxuMeanStdDev** (vx_context *context*, vx_image *input*, vx_float32 **mean*, vx_float32 **stddev*);

input: The input image. VX_DF_IMAGE_U8 is supported.

mean: Average pixel value of type VX_TYPE_FLOAT32.

stddev: Standard deviation of the pixel values of VX_TYPE_FLOAT32.

(Continued on next page) ▶

Vision Functions (cont.)

Median Filter [3.30]

Computes median values over a window of the input image.

ret **vx[u]Median3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Min, Max Location [3.31]

Locates the minima and maxima in an image.

ret **vx[u]MinMaxLoc[Node]** (*graph*|*context*vx_image *input*, vx_scalar *minVal*, vx_scalar *maxVal*, vx_array *minLoc*, vx_array *maxLoc*, vx_scalar *minCount*, vx_scalar *maxCount*);

input: The input image of VX_DF_IMAGE_{U8, S16} format.

{min, max}Count: [Opt.] The number of detected {minimums, maximums} in image.

{min, max}Loc: [Opt.] The {minimum, maximum} locations of type VX_TYPE_COORDINATES2D.

{min, max}Val: The {minimum, maximum} value in the image.

Optical Flow Pyramid (LK) [3.32]

Computes the optical flow between two pyramid images.

ret **vx[u]OpticalFlowPyrLK[Node]** (*graph*|*context*, vx_pyramid *old_images*, vx_pyramid *new_images*, vx_array *old_points*, vx_array *new_points_estimates*, vx_array *new_points*, vx_enum *termination*, vx_scalar *epsilon*, vx_scalar *num_iterations*, vx_scalar *use_initial_estimate*, vx_size *window_dimension*);

{old, new}_images: The {old, new} image pyramid of VX_DF_IMAGE_U8.

old_points: Array of key points in a vx_array of VX_TYPE_KEYPOINT.

new_points_estimates: An array of estimation on what is the output key points in a vx_array of type VX_TYPE_KEYPOINT.

new_points: Array of key points in a vx_array of type VX_TYPE_KEYPOINT.

termination: Termination can be VX_TERM_CRITERIA_ITERATIONS, VX_TERM_CRITERIA_EPSILON, or VX_TERM_CRITERIA_BOTH.

epsilon: The vx_float32 error for terminating the algorithm.

num_iterations: The number of iterations of type VX_TYPE_UINT32.

use_initial_estimate: VX_TYPE_BOOL set to vx_false_e or vx_true_e.

window_dimension: The size of the window.

Phase [3.33]

$\varphi = \tan^{-1}(grad_y(x,y) / grad_x(x,y)), 0 \le \varphi \le 255$

vx_status **vx[u]Phase[Node]** (*graph*|*context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *output*);

grad_{x, y}: The input {x, y} image of VX_DF_IMAGE_S16 format.

output: The phase image of VX_DF_IMAGE_U8 format.

Pixel-wise Multiplication [3.34]

$out(x,y) = in_1(x,y)\ in_2(x,y)\ scale$

ret **vx[u]Multiply[Node]** (*graph*|*context*, vx_image *in1*, vx_image *in2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*);

in{1, 2}, out: Image of VX_DF_IMAGE_{U8, S16} format.

{overflow, rounding}_policy: A VX_TYPE_ENUM of the vx_{convert, round}_policy_e enumeration.

scale: Scale value of type VX_TYPE_FLOAT32.

Remap [3.35]

$output(x1, y1) = input(map_x(x0, y0), map_y(x0, y0))$

ret **vx[u]Remap[Node]** (*graph*|*context*, vx_image *input*, vx_remap *table*, vx_enum *policy*, vx_image *output*);

input, output: Image of type VX_DF_IMAGE_U8.

table: The remap table object.

policy: An interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.

Scale Image [3.36]

Performs a Gaussian blur on an image then half-scales it.

ret **vx[u]ScaleImage[Node]** (*graph*|*context*, vx_image *src*, vx_image *dst*, vx_enum *type*);

src, dst: The {source, destination} image of type VX_DF_IMAGE_U8.

type: The interpolation type to use.

Half Scale Gaussian [3.36]

Performs a Gaussian blur on an image then half-scales it.

ret **vx[u]HalfScaleGaussian[Node]** (*graph*|*context*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*);

input, output: Image of VX_DF_IMAGE_U8 format.

kernel_size: Gaussian filter input size. Supported values are 3 and 5.

Sobel3x3 [3.37]

Sobel filter, produces *X* and *Y* output planes.

ret **vx[u]Sobel3x3[Node]** (*graph*|*context*, vx_image *input*, vx_image *output_x*, vx_image *output_y*);

input: The input of VX_DF_IMAGE_U8 format.

output_{x, y}: [Opt.] The output gradient in the {x, y} direction of VX_DF_IMAGE_S16 format.

Table Lookup [3.38]

Uses a LUT to convert pixel values.

ret **vx[u]TableLookup[Node]** (*graph*|*context*, vx_image *input*, vx_lut *lut*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

lut: The LUT of type VX_TYPE_UINT8.

Thresholding [3.39]

Produces a Boolean image by thresholding the input image.

ret **vx[u]Threshold[Node]** (*graph*|*context*, vx_image *input*, vx_threshold *thresh*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.

output: The output Boolean image: either 0 or 255. (VX_DF_IMAGE_U8)

thresh: Thresholding object.

Warp Affine [3.40]

Performs an affine transform on an image.

ret **vx[u]WarpAffine [Node]**(*graph*|*context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

matrix: The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.

type: The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.

Warp Perspective [3.41]

Performs a perspective transform on an image.

ret **vx[u]WarpPerspective[Node]** (*graph*|*context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

matrix: Perspective matrix. Must be 3x3 of type VX_TYPE_FLOAT32.

type: The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.

Objects

Get Status [3.42.6]

Return status values from Object constructors if they fail.

vx_status **vxGetStatus** (vx_reference *reference*);

reference: The reference to check for construction errors.

Object: Array [3.48]

Grant access to a sub-range of an Array.

vx_status **vxAccessArrayRange** (vx_array *arr*, vx_size *start*, vx_size *end*, vx_size **stride*, void ***ptr*, vx_enum *usage*);

Add items to the Array.

vx_status **vxAddArrayItems** (vx_array *arr*, vx_size *count*, void **ptr*, vx_size *stride*);

Commit data back to the Array object.

vx_status **vxCommitArrayRange** (vx_array *arr*, vx_size *start*, vx_size *end*, void **ptr*);

Create a reference to an Array object.

vx_array **vxCreateArray** (vx_context *context*, vx_enum *item_type*, vx_size *capacity*);

Create opaque reference to virtual Array; no direct user access.

vx_array **vxCreateVirtualArray** (vx_graph *graph*, vx_enum *item_type*, vx_size *capacity*);

Query the Array for some specific information.

vx_status **vxQueryArray** (vx_array *arr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference of an Array object.

vx_status **vxReleaseArray** (vx_array **arr*);

Truncate an Array (remove items from the end).

vx_status **vxTruncateArray** (vx_array *arr*, vx_size *new_num_items*);

arr: The Array to release. For **vxTruncateArray**: (in, out)

attribute: The vx_array_attribute_e attribute to query.

capacity: The maximal number of items that the array can hold.

context: The reference to the overall Context.

count: The total number of elements to insert.

graph: The reference to the parent graph.

item_type: The type of objects to hold.

new_num_items: The new number of items for the Array.

size: The size of the container to which *ptr* points.

start, end: The {start, end} index.

stride: The stride in bytes between elements.

usage: Declares the intended usage of the pointer. From vx_accessor_e.

ptr: Location at which to store the result.

Object: Array (Advanced) [3.60]

Registers user-defined structures to the context.

vx_enum **vxRegisterUserStruct** (vx_context *context*, vx_size *size*);

size: The size of user struct in bytes.

Object: Context [3.45]

Create a vx_context.

vx_context **vxCreateContext** ();

Retrieve the context from any reference from within a context.

vx_context **vxGetContext** (vx_reference *reference*);

Query the context for some specific information.

vx_status **vxQueryContext** (vx_context *context*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release the OpenVX object context.

vx_status **vxReleaseContext** (vx_context **context*);

Set an attribute on the context.

vx_status **vxSetContextAttribute** (vx_context *context*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: Attribute to query of type vx_context_attribute_e.

reference: The reference from which to extract the context.

size: Size of the container to which *ptr* points.

ptr: Pointer to where to store the result.

Object: Convolution [3.49]

Get the convolution data (copy).

vx_status **vxAccessConvolutionCoefficients** (vx_convolution *conv*, vx_int16 **array*);

Set the convolution data (copy).

vx_status **vxCommitConvolutionCoefficients** (vx_convolution *conv*, vx_int16 **array*);

Create a reference to a convolution matrix object.

vx_convolution **vxCreateConvolution** (vx_context *context*, vx_size *columns*, vx_size *rows*);

Queries an attribute on the convolution matrix object.

vx_status **vxQueryConvolution** (vx_convolution *conv*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release the reference to a convolution matrix.

vx_status **vxReleaseConvolution** (vx_convolution **conv*);

Set attributes on the convolution object.

vx_status **vxSetConvolutionAttribute** (vx_convolution *conv*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query or modify.

columns, rows: The {columns, rows} dimension of the convolution.

conv: The pointer to the convolution matrix to release; or convolution matrix object to set; or the coordinates object to set.

size: The size of the container or data to which *ptr* points.

array: The array to place or read the convolution.

ptr: The location at which to store the resulting value; or pointer to the value to which to set the attribute.

Object: Delay (Advanced) [3.63]

Age the internal delay ring by one.

vx_status **vxAgeDelay** (vx_delay *delay*);

Create a Delay object.

vx_delay **vxCreateDelay** (vx_context *context*, vx_reference *exemplar*, vx_size *count*);

Retrieve a reference from a delay object.

vx_reference **vxGetReferenceFromDelay** (vx_delay *delay*, vx_int32 *index*);

Query a vx_delay object attribute.

vx_status **vxQueryDelay** (vx_delay *delay*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a delay object.

vx_status **vxReleaseDelay** (vx_delay **delay*);

attribute: The attribute to query from vx_delay_attribute_e.

context: The reference to the system context.

count: The number of reference in the delay.

delay: The pointer to the delay object".

exemplar: The exemplar object.

index: The index into the delay from which to extract the reference.

size: The size of the container to which *ptr* points.

ptr: The location at which to store the resulting value.

Object: Distribution [3.50]

Get direct access to a Distribution in memory.

vx_status **vxAccessDistribution** (vx_distribution *distribution*, void ***ptr*, vx_enum *usage*);

Set the Distribution back to the memory.

vx_status **vxCommitDistribution** (vx_distribution *distribution*, void **ptr*)

Create a reference to a 1D Distribution.

vx_distribution **vxCreateDistribution** (vx_context *context*, vx_size *numBins*, vx_size *offset*, vx_size *range*);

(Continued on next page) ▶

Objects (cont.)

Query a Distribution object.
vx_status **vxQueryDistribution** (vx_distribution *distribution*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a distribution object.
vx_status **vxReleaseDistribution** (vx_distribution **distribution*);

attribute: Attribute to query, from vx_distribution_attribute_e.
context: The reference to the overall context.
distribution: The reference to the distribution to release, query, modify, or access.
numBins: The number of bins in the distribution.
offset: The offset into the range value.
range: The total range of the values.
size: The size of the container to which *ptr* points.
usage: The vx_accessor_e value to describe the access of the object.
ptr: The location at which to store the resulting value or store the pointer to the Distribution memory.
For **vxCommitDistribution**: The pointer returned from (or not modified by) **vxAccessDistribution**.

Object: Graph [3.46]

Create an empty graph.
vx_graph **vxCreateGraph** (vx_context *context*);

Returns a Boolean to indicate the state of graph verification.
vx_bool **vxIsGraphVerified** (vx_graph *graph*);

Cause the synchronous processing of a graph.
vx_status **vxProcessGraph** (vx_graph *graph*);

Query attributes of the Graph.
vx_status **vxQueryGraph** (vx_graph *graph*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a graph.
vx_status **vxReleaseGraph** (vx_graph **graph*);

Schedule a graph for future execution.
vx_status **vxScheduleGraph** (vx_graph *graph*);

Set to attributes on the Graph.
vx_status **vxSetGraphAttribute** (vx_graph *graph*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Verify the state of the graph before it is executed.
vx_status **vxVerifyGraph** (vx_graph *graph*);

Wait for a specific graph to complete.
vx_status **vxWaitGraph** (vx_graph *graph*);

attribute: The vx_graph_attribute_e type needed.
context: The reference to the implementation context.
graph: The graph to execute, schedule, or wait on; or the pointer to the graph to release or verify.
size: The size of the container to which *ptr* points.
ptr: The location at which to store the resulting value.

Object: Image [3.51]

Allow the user to extract a rectangular patch (subset) of an image from a single plane.
vx_status **vxAccessImagePatch** (vx_image *image*, vx_rectangle_t **rect*, vx_uint32 *plane_index*, vx_imagepatch_addressing_t **addr*, void ***ptr*, vx_enum *usage*);

Commit a rectangular patch (subset) of image from single plane.
vx_status **vxCommitImagePatch** (vx_image *image*, vx_rectangle_t **rect*, vx_uint32 *plane_index*, vx_imagepatch_addressing_t **addr*, void **ptr*);

Compute the size needed to retrieve an image patch.
vx_size **vxComputeImagePatchSize** (vx_image *image*, vx_rectangle_t **rect*, vx_uint32 *plane_index*);

Create an opaque reference to an image buffer.
vx_image **vxCreateImage** (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*);

Create a reference to an externally allocated image object.
vx_image **vxCreateImageFromHandle** (vx_context *context*, vx_df_image *color*, vx_imagepatch_addressing_t *addrs*[ ], void **ptrs*[ ], vx_enum *import_type*);

Create an image from another image given a rectangle.
vx_image **vxCreateImageFromROI** (vx_image *img*, vx_rectangle_t **rect*);

Create a reference to an image object that has a singular, uniform value in all pixels.
vx_image **vxCreateUniformImage** (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*, void **value*);

Create opaque reference to image buffer with no direct user access.
vx_image **vxCreateVirtualImage** (vx_graph *graph*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*);

Access a specific indexed pixel in an image patch.
void * **vxFormatImagePatchAddress1d** (void **ptr*, vx_uint32 *index*, vx_imagepatch_addressing_t **addr*);

Access a specific pixel at a 2d coordinate in an image patch.
void * **vxFormatImagePatchAddress2d** (void **ptr*, vx_uint32 *x*, vx_uint32 *y*, vx_imagepatch_addressing_t **addr*);

Retrieve the valid region of the image as a rectangle.
vx_status **vxGetValidRegionImage** (vx_image *image*, vx_rectangle_t **rect*);

Retrieve various attributes of an image.
vx_status **vxQueryImage** (vx_image *image*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to an image object.
vx_status **vxReleaseImage** (vx_image **image*);

Set attributes on the image.
vx_status **vxSetImageAttribute** (vx_image *image*, vx_enum *attribute*, void **out*, vx_size *size*);

addrs[]: The array of image patch addressing structures that define the dimension and stride of the array of pointers.
attribute: The attribute to query or set. From vx_image_attribute_e.
color: The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
context: The reference to the implementation context.
image: The reference to the image to query, to release, from which to extract the patch, or on which to set the attribute.
img, graph: The reference to the parent {image, graph}.
import_type: vx_import_type_e.
index: The 0 based index of the pixel count in the patch.
out: The pointer to the location from which to read the value.
plane_index: The plane index from which to get or set the data.
ptrs[]: The array of platform-defined references to each plane.
rect: The region of interest rectangle.
size: The size of the container to which *ptr* points.
usage: Intended usage, from vx_accessor_e.
value: The pointer to the pixel value to which to set all pixels.
width, height: The image {width, height} in pixels.
addr: The addressing information for the image patch.
ptr: The pointer to the location at which to store or read the data.

Object: Kernel (Advanced) [3.64]

Obtain a reference to kernel using vx_kernel_e enumeration.
vx_kernel **vxGetKernelByEnum** (vx_context *context*, vx_enum *kernel*);

Obtain a reference to a kernel using a string to specify the name.
vx_kernel **vxGetKernelByName** (vx_context *context*, vx_char **name*);

Query the kernel to get information about the number of parameters, enum values, etc.
vx_status **vxQueryKernel** (vx_kernel *kernel*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release the reference to the kernel.
vx_status **vxReleaseKernel** (vx_kernel **kernel*);

attribute: The attribute to query. From vx_kernel_attribute_e.
context: The reference to the implementation context.
kernel: A value from vx_kernel_e or a vendor or client-defined value, or the pointer to the kernel reference to release.
name: The string of the name of the kernel to get.
size: The size of the container to which *ptr* points.
ptr: The location at which to store the resulting value.

Object: LUT [3.52]

Get direct access to the LUT table data.
vx_status **vxAccessLUT** (vx_lut *lut*, void ***ptr*, vx_enum *usage*);

Commit the Lookup Table.
vx_status **vxCommitLUT** (vx_lut *lut*, void **ptr*);

Create LUT object of a given type.
vx_lut **vxCreateLUT** (vx_context *context*, vx_enum *data_type*, vx_size *count*);

Query attributes from a LUT.
vx_status **vxQueryLUT** (vx_lut *lut*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a LUT object.
vx_status **vxReleaseLUT** (vx_lut **lut*);

attribute: The attribute to query. From vx_lut_attribute_e.
context: The reference to the context.
count: The number of entries desired.
data_type: The type of data stored in the LUT.
lut: The LUT to query or modify; or the pointer to the LUT to release.
size: The size of the container to which *ptr* points.
usage: Declare the intended usage of the pointer. From vx_accessor_e.
ptr: The location at which to store the resulting value.
For **vxAccessLUT**: (in, out)

Object: Matrix [3.53]

{Get, Set} the matrix data (copy).
vx_status **vx{Access, Commit}Matrix** (vx_matrix *mat*, void **array*);

Create a reference to a matrix object.
vx_matrix **vxCreateMatrix** (vx_context *c*, vx_enum *data_type*, vx_size *columns*, vx_size *rows*);

Query an attribute on the matrix object.
vx_status **vxQueryMatrix** (vx_matrix *mat*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a matrix object.
vx_status **vxReleaseMatrix** (vx_matrix **mat*);

array: Array to read the matrix, or in which to place the matrix.
attribute: The attribute to query. From vx_matrix_attribute_e.
c: The reference to the overall context.
columns: The first dimension.
data_type: Unit format of the matrix of type VX_TYPE_{INT32, FLOAT32}.
mat: The matrix reference to release, or matrix object to set.
rows: The second dimension.
size: The size of the container to which *ptr* points.
ptr: The location at which to store the resulting value.

Object: Node [3.47]

Query information out of a node.
vx_status **vxQueryNode** (vx_node *node*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a Node object.
vx_status **vxReleaseNode** (vx_node **node*);

Remove a Node from its parent Graph and release it.
void **vxRemoveNode** (vx_node **node*);

Set attributes of a node before Graph Validation.
vx_status **vxSetNodeAttribute** (vx_node *node*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The vx_node_attribute_e value to query for information.
node: The reference to the node to query, to the node to set, or the pointer to the reference of the node to release.
size: The size of the container to which *ptr* points.
ptr: The location at which to store the resulting value.

Object: Node (Advanced) [3.61]

Defines the advanced features of the Node Interface.
vx_node **vxCreateGenericNode** (vx_graph *graph*, vx_kernel *kernel*);

graph: The reference to the graph in which this node exists.
kernel: The kernel reference to associate with this new node.

Object: Parameter (Advanced) [3.65]

Retrieve a vx_parameter from a vx_kernel.
vx_parameter **vxGetKernelParameterByIndex** (vx_kernel *kernel*, vx_uint32 *index*);

Retrieve a vx_parameter from a vx_node.
vx_parameter **vxGetParameterByIndex** (vx_node *node*, vx_uint32 *index*);

Query a parameter to determine its meta-information.
vx_status **vxQueryParameter** (vx_parameter *param*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a parameter object.
vx_status **vxReleaseParameter** (vx_parameter **param*);

Set the specified parameter data for a kernel on the node.
vx_status **vxSetParameterByIndex** (vx_node *node*, vx_uint32 *index*, vx_reference *value*);

Associate a parameter reference and a data reference with a kernel on a node.
vx_status **vxSetParameterByReference** (vx_parameter *parameter*, vx_reference *value*);

attribute: The attribute to query from vx_parameter_attribute_e.
index: The index of the parameter.
kernel: The reference to the kernel.
node: The node that contains the kernel.
param: The pointer to the parameter.

(Continued on next page) ▶

Objects (cont.)

parameter: The reference to the kernel parameter.
size: The size of the container to which *ptr* points.
value: The reference to the parameter.
ptr: The location at which to store the resulting value.

Object: Pyramid [3.54]

Create a reference to a pyramid object.
vx_pyramid **vxCreatePyramid** (vx_context *context*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

Create a reference to a virtual pyramid object.
vx_pyramid **vxCreateVirtualPyramid** (vx_graph *graph*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

Retrieve a level of the pyramid as a vx_image, which can be used elsewhere in OpenVX.
vx_image **vxGetPyramidLevel** (vx_pyramid *pyr*, vx_uint32 *index*);

Query an attribute from an image pyramid.
vx_status **vxQueryPyramid** (vx_pyramid *pyr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a pyramid object.
vx_status **vxReleasePyramid** (vx_pyramid **pyr*);

attribute: The attribute to query from vx_pyramid_attribute_e.
context: The reference to the overall context.
format: The format of all images in the pyramid.
graph: The reference to the parent graph.
height: The height of the 0th level image in pixels.
index: The index of the level, such that *index* is less than *levels*.
levels: The number of levels desired.
pyr: The pointer to the pyramid to release or query.
scale: Indicates the scale between pyramid levels.
size: The size of the container to which *ptr* points.
width: The width of the 0th level image in pixels.
ptr: The location at which to store the resulting value.

Object: Reference [3.44]

Query any reference type for basic information (count, type).
vx_status **vxQueryReference** (vx_reference *ref*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The value to query, from vx_reference_attribute_e.
ref: Reference to the object to query.
size: The size of the container to which *ptr* points.
ptr: The location at which to store the resulting value.

Object: Remap [3.55]

Create a remap table object.
vx_remap **vxCreateRemap** (vx_context *context*, vx_uint32 *src_width*, vx_uint32 *src_height*, vx_uint32 *dst_width*, vx_uint32 *dst_height*);

Retrieve the source pixel point from a destination pixel.
vx_status **vxGetRemapPoint** (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 **src_x*, vx_float32 **src_y*);

Query attributes from a remap table.
vx_status **vxQueryRemap** (vx_remap *r*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a remap table object.
vx_status **vxReleaseRemap** (vx_remap **table*);

Assign a destination pixel mapping to the source pixel.
vx_status **vxSetRemapPoint** (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 *src_x*, vx_float32 *src_y*);

attribute: The attribute to query from vx_remap_attribute_e.
context: The reference to the overall context.
dst_{x, y}: The destination {x, y} coordinate.
dst_{width, height}: The {width, height} of the destination image in pixels.
r: The remap to query.
size: The size of the container to which *ptr* points.
src_{width, height}: The {width, height} of the source image in pixels.
src_x: The source {x, y} coordinate in float representation to allow interpolation.
table: Remap table reference, or a pointer to the remap table to release.
ptr: The location at which to store the resulting value.

Object: Scalar [3.56]

Get the scalar value out of a reference.
vx_status **vxAccessScalarValue** (vx_scalar *ref*, void **ptr*);

Set the scalar value in a reference.
vx_status **vxCommitScalarValue** (vx_scalar *ref*, void **ptr*);

Create a reference to a scalar object.
vx_scalar **vxCreateScalar** (vx_context *context*, vx_enum *data_type*, void **ptr*);

Query attributes from a scalar.
vx_status **vxQueryScalar** (vx_scalar *scalar*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a scalar object.
vx_status **vxReleaseScalar** (vx_scalar **scalar*);

attribute: The enumeration to query from vx_scalar_attribute_e.
context: The reference to the system context.
data_type: The vx_type_e of the scalar.
ref: The reference from which to get the scalar value.
scalar: The scalar object, or the pointer to the scalar to release.
size: The size of the container to which *ptr* points.
ptr: Location of the scalar.

Object: Threshold [3.57]

Create a reference to a threshold object of a given type.
vx_threshold **vxCreateThreshold** (vx_context *c*, vx_enum *thresh_type*, vx_enum *data_type*);

Query an attribute on the threshold object.
vx_status **vxQueryThreshold** (vx_threshold *thresh*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Release a reference to a threshold object.
vx_status **vxReleaseThreshold** (vx_threshold **thresh*);

Set attributes on the threshold object.
vx_status **vxSetThresholdAttribute** (vx_threshold *thresh*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to modify from vx_threshold_attribute_e.
c: The reference to the overall context.
data_type: The data type of the threshold's value(s).
size: The size of the data pointed to by *ptr*.
thresh: Threshold object to set or query, or a pointer to a threshold object to release.
thresh_type: The type of threshold to create.
ptr: The pointer to the value to which to set the attribute. The qualifier is "in" for all except **vxQueryThreshold**.

Advanced Framework

Node Callbacks [3.67]

Assign a callback to a node.
vx_status **vxAssignNodeCallback** (vx_node *node*, vx_nodecomplete_f *callback*);

Retrieve the current node callback function pointer.
vx_nodecomplete_f **vxRetrieveNodeCallback** (vx_node *node*);

Callback prototype.
typedef vxAction (***vx_nodecomplete_f**)(vx_node *node*);

callback: Callback function pointer.
node: The reference to the vx_node object.

Log [3.69]

Add a line to the log.
void **vxAddLogEntry** (vx_reference *ref*, vx_status *status*, const char **message*, ...);

Register a callback facility to receive error logs.
void **vxRegisterLogCallback** (vx_context *context*, vx_log_callback_f *callback*, vx_bool *reentrant*);

Callback prototype for vxRegisterLogCallback.
typedef void (***vx_log_callback_f**) (vx_context *context*, vx_reference *ref*, vx_status *status*, vx_char *string*[]);

callback: The callback function or NULL.
context: The overall context to OpenVX.
message: The human readable message to add to the log.
reentrant: Boolean reentrancy flag indicating whether the callback may be entered from multiple simultaneous tasks or threads.
ref: The reference to add the log entry against.
status: The status code.

Hints [3.70]

Provide a generic API to give platform-specific hints.
vx_status **vxHint** (vx_context *context*, vx_reference *reference*, vx_enum *hint*);

context: The reference to the implementation context.
reference: The reference to the object to hint at.
hint: A vx_hint_e hint to give the OpenVX context.

Directives [3.71]

Provide a generic API to give platform-specific directives.
vx_status **vxDirective** (vx_context *context*, vx_reference *reference*, vx_enum *directive*);

context: The reference to the implementation context.
directive: The directive to set.
reference: The reference to the object to set the directive on.

User Kernels [3.72]

Add custom kernels at run-time.
vx_kernel **vxAddKernel** (vx_context *context*, vx_char *name*[VX_MAX_KERNEL_NAME], vx_enum *enumeration*, vx_kernel_f *func_ptr*, vx_uint32 *numParams*, vx_kernel_input_validate_f *input*, vx_kernel_output_validate_f *output*, vx_kernel_initialize_f *init*, vx_kernel_deinitialize_f *deinit*);

Set the signatures of the custom kernel.
vx_status **vxAddParameterToKernel** (vx_kernel *kernel*, vx_uint32 *index*, vx_enum *dir*, vx_enum *data_type*, vx_enum *state*);

Called after parameters have been added and kernel is ready.
vx_status **vxFinalizeKernel** (vx_kernel *kernel*);

Load one or more kernels into the OpenVX context.
vx_status **vxLoadKernels** (vx_context *context*, vx_char **module*);

Remove a non-finalized vx_kernel from the vx_context.
vx_status **vxRemoveKernel** (vx_kernel *kernel*);

Set kernel attributes.
vx_status **vxSetKernelAttribute** (vx_kernel *kernel*, vx_enum *attribute*, void **ptr*, vx_size *size*);

Set the attributes of a vx_meta_format object.
vx_status **vxSetMetaFormatAttribute** (vx_meta_format *meta*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query, from vx_kernel_attribute_e.
context: The reference to the implementation context.
data_type: Type of parameter, from vx_type_e.
dir: Direction of the parameter, from vx_direction_e.
enumeration: Enumerated value of the kernel to be used by clients.
func_ptr: The process-local function pointer to be invoked.
index: The index of the parameter to add.
init, deinit: The kernel {initialization, deinitialization} function.
input, output: The pointer to vx_kernel_input_validate_f, which validates the {input, output} parameters to this kernel.
kernel: Reference to the loaded kernel from vxAddKernel.
meta: The reference to the vx_meta_format object to set.
module: The short name of the module to load.
name: The string to use to match the kernel.
numParams: The number of parameters for this kernel.
ptr: Pointer to the attribute.
size: The size of the container to which *ptr* points.
state: State of the parameter, from vx_parameter_state_e.

Graph Parameters [3.73]

Add the given parameter extracted from a vx_node to the graph.
vx_status **vxAddParameterToGraph** (vx_graph *graph*, vx_parameter *parameter*);

Retrieve a vx_parameter from a vx_graph.
vx_parameter **vxGetGraphParameterByIndex** (vx_graph *graph*, vx_uint32 *index*);

Set a reference to the parameter on the graph.
vx_status **vxSetGraphParameterByIndex** (vx_graph *graph*, vx_uint32 *index*, vx_reference *value*);

graph: The graph reference.
index: The parameter index.
parameter: Parameter reference to add to the graph from the node.
value: The reference to set to the parameter.