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 supports modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems.

[n.n] refers to sections in the OpenVX 1.2 specification available at khronos.org/registry/vx/
Parameter color coding: ***Output***, ***Input***, and ***Input/Output*** parameters.
Parameters marked as [Opt.] are optional. Pass NULL to indicate the parameter is unused.
In descriptions, curly braces {} mean "one of," and square braces [] mean "optional."
- Indicates the overflow policy used is VX_CONVERT_POLICY_SATURATE.
- Refer to list of enumerations on pages 9-10 of this reference guide.

OpenVX Functions and Objects

VX_API_CALL

In every OpenVX function and object, insert VX_API_CALL before the function name, as shown in the example below. It was omitted from the functions on this reference card to save space.

<return_type> **VX_API_CALL vxFunctionName**(T *arg1* , . . . , T *argN*);

Vision Functions

Vision functions in OpenVX may be graph mode or immediate mode.

- **Graph mode functions** have "Node" in the function name. They 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.

In the vision functions, the parameter ***graph*** is the reference to the graph, and the parameter ***context*** is the reference to the overall context.

Vision Functions

Absolute Difference [3.2]

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

vx_node **vxAbsDiffNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuAbsDiff** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Accumulate [3.3]

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

vx_node **vxAccumulateImageNode** (vx_graph *graph*, vx_image *input*, vx_image *accum*);

vx_status **vxuAccumulateImage** (vx_context *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]

- Accumulate a squared input image to an output image.

vx_node **vxAccumulateSquareImageNode** (vx_graph *graph*, vx_image *input*, vx_scalar *shift*, vx_image *accum*);

vx_status **vxuAccumulateSquareImage** (vx_context *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.

Data Object Copy [3.5]

Copy data from one object to another.

vx_node **vxCopyNode** (vx_graph *graph*, vx_reference *input*, vx_reference *output*);

vx_status **vxuCopy** (vx_context *context*, vx_reference *input*, vx_reference *output*);

input, output: The image of VX_DF_IMAGE_U8 format.

Accumulate Weighted [3.6]

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

vx_node **vxAccumulateWeightedImageNode** (vx_graph *graph*, vx_image *input*, vx_scalar *alpha*, vx_image *accum*);

vx_status **vxuAccumulateWeightedImage** (vx_context *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.

Control Flow [3.7]

Define the predicated execution model of OpenVX.

vx_node **vxScalarOperationNode** (vx_graph *graph*, vx_enum *scalar_operation*, vx_scalar *a*, vx_scalar *b*, vx_scalar *output*);

vx_node **vxSelectNode** (vx_graph *graph*, vx_scalar *condition*, vx_reference *true_value*, vx_reference *false_value*, vx_reference *output*);

scalar_operation: • A VX_TYPE_ENUM (vx_scalar_operation_e).
a: The first scalar operand.
b: The second scalar operand.
output: Result of the scalar operation, or output data object.
condition: VX_TYPE_BOOL predicate variable.
[true, false]_value: Data object for [true, false].

Logical, comparison, and arithmetic operations supported

VX_SCALAR_OP_*x* where *x* may be:
AND, OR, XOR, NAND
EQUAL, NOTEQUAL, LESS[EQ], GREATER[EQ]
ADD, SUBTRACT, MULTIPLY, DIVIDE, MODULUS, MIN, MAX

Arithmetic Addition [3.8]

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

vx_node **vxAddNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

vx_status **vxuAdd** (vx_context *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: VX_CONVERT_POLICY_{WRAP, SATURATE}.

Arithmetic Subtraction [3.9]

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

vx_node **vxSubtractNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

vx_status **vxuSubtract** (vx_context *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: VX_CONVERT_POLICY_{WRAP, SATURATE}.

Bitwise AND [3.10]

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

vx_node **vxAndNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuAnd** (vx_context *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 [3.11]

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

vx_node **vxXorNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuXor** (vx_context *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 [3.12]

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

vx_node **vxOrNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuOr** (vx_context *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 [3.13]

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

vx_node **vxNotNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuNot** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Box Filter [3.14]

Compute a box filter over a window of the input image.

vx_node **vxBox3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuBox3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Non-Maxima Suppression [3.15]

Find local maxima, or suppress non-local-maxima pixels in image.

vx_node **vxNonMaxSuppressionNode** (vx_graph *graph*, vx_image *input*, vx_image *mask*, vx_int32 *win_size*, vx_image *output*);

vx_status **vxuNonMaxSuppression** (vx_context *context*, vx_image *input*, vx_image *mask*, vx_int32 *win_size*, vx_image *output*);

mask: [optional] Constrict suppression to an ROI.
win_size: Size of window over which to perform the operation.
input, output: Input; the non-maxima-suppressed output image.

Canny Edge Detector [3.16]

Provide a Canny edge detector kernel.

vx_node **vxCannyEdgeDetectorNode** (vx_graph *graph*, vx_image *input*, vx_threshold *hyst*, vx_int32 *gradient_size*, vx_enum *norm_type*, vx_image *output*);

vx_status **vxuCannyEdgeDetector** (vx_context *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. VX_TYPE_{U8, U16}.
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 [3.17]

Combines multiple image planes.

vx_node **vxChannelCombineNode** (vx_graph *graph*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*);

vx_status **vxuChannelCombine** (vx_context *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 [3.18]

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

vx_node **vxChannelExtractNode** (vx_graph *graph*, vx_image *input*, vx_enum *channel*, vx_image *output*);

vx_status **vxuChannelExtract** (vx_context *context*, vx_image *input*, vx_enum *channel*, vx_image *output*);

input: One of the defined vx_df_image_e multi-channel formats.
channel: Channel to extract. VX_CHANNEL_{0, 1, 2, 3, R, G, B, A, Y, U, V}.
output: The output image of VX_DF_IMAGE_U8 format.

Color Convert [3.19]

Convert the format of an image.

vx_node **vxColorConvertNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuColorConvert** (vx_context *context*, vx_image *input*, vx_image *output*);

• Convert Bit Depth [3.20]

vx_node **vxConvertDepthNode** (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_scalar *shift*);

vx_status **vxuConvertDepth** (vx_context *context*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_int32 *shift*);

input, output: The {input, output} image.
policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.
shift: The shift value of type VX_TYPE_INT32.

(Continued on next page) ▶

◀ Vision Functions (cont.)

Custom Convolution [3.21]

• Convolve an image with a user-defined matrix.

vx_node **vxConvolveNode** (vx_graph *graph*, vx_image *input*, vx_convolution *conv*, vx_image *output*);

vx_status **vxuConvolve** (vx_context *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 [3.22]

Grow the white space in a VX_DF_IMAGE_U8 Boolean image.

vx_node **vxDilate3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuDilate3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Equalize Histogram [3.23]

Normalize brightness and contrast of a grayscale image.

vx_node **vxEqualizeHistNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuEqualizeHist** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: The grayscale image of VX_DF_IMAGE_U8 format.

Erode Image [3.24]

Shrink the white space in a VX_DF_IMAGE_U8 Boolean image.

vx_node **vxErode3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuErode3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Fast Corners [3.25]

Find corners in an image.

vx_node **vxFastCornersNode** (vx_graph *graph*, vx_image *input*, vx_scalar *strength_thresh*, vx_bool *nonmax_suppression*, vx_array *corners*, vx_scalar *num_corners*);

vx_status **vxuFastCorners** (vx_context *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-maximum 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 [3.26]

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

vx_node **vxGaussian3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuGaussian3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Non-linear Filter [3.27]

Compute a non-linear filter over a window of the input image.

vx_node **vxNonLinearFilterNode** (vx_graph *graph*, vx_enum *function*, vx_image *input*, vx_matrix *mask*, vx_image *output*);

vx_status **vxuNonLinearFilter** (vx_context *context*, vx_enum *function*, vx_image *input*, vx_matrix *mask*, vx_image *output*);

function: The non-linear function of type VX_NONLINEAR_FILTER_{MEDIAN, MIN, MAX}.
input: An input image of VX_DF_IMAGE_U8 format.
mask: The mask to apply (See **vxCreateMatrixFromPattern**).
output: The output image of VX_DF_IMAGE_U8 format.

Harris Corners [3.28]

Compute the Harris Corners of an image.

vx_node **vxHarrisCornersNode** (vx_graph *graph*, 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*);

vx_status **vxuHarrisCorners** (vx_context *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.] Num. of detected corners in image. (VX_TYPE_SIZE)

Histogram [3.29]

vx_node **vxHistogramNode** (vx_graph *graph*, vx_image *input*, vx_distribution *distribution*);

vx_status **vxuHistogram** (vx_context *context*, vx_image *input*, vx_distribution *distribution*);

input: The input image of VX_DF_IMAGE_U8 format.
distribution: The output distribution.

Gaussian Image Pyramid [3.30]

Compute a Gaussian image pyramid using a 5x5 kernel.

vx_node **vxGaussianPyramidNode** (vx_graph *graph*, vx_image *input*, vx_pyramid *gaussian*);

vx_status **vxuGaussianPyramid** (vx_context *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.

Laplacian Image Pyramid [3.31]

Compute a Laplacian Image Pyramid from an input image.

vx_node **vxLaplacianPyramidNode** (vx_graph *graph*, vx_image *input*, vx_pyramid *laplacian*, vx_image *output*);

vx_status **vxuLaplacianPyramid** (vx_context *context*, vx_image *input*, vx_pyramid *laplacian*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.
laplacian: The Laplacian pyramid of VX_DF_IMAGE_S16 format.
output: The lowest-resolution image of VX_DF_IMAGE_[S16, U8] necessary to reconstruct the input image from the pyramid.

Reconstruction from Laplacian Image Pyramid [3.32]

Reconstruct the original image from a Laplacian Image Pyramid.

vx_node **vxLaplacianReconstructNode** (vx_graph *graph*, vx_pyramid *laplacian*, vx_image *input*, vx_image *output*);

vx_status **vxLaplacianReconstruct** (vx_context *context*, vx_pyramid *laplacian*, vx_image *input*, vx_image *output*);

laplacian: The Laplacian pyramid of VX_DF_IMAGE_S16 format.
input: The lowest-resolution image of VX_DF_IMAGE_S16 format.
output: The image of VX_DF_IMAGE_U8 with the highest possible resolution reconstructed from the pyramid.

Integral Image [3.33]

Compute the integral image of the input.

vx_node **vxIntegralImageNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuIntegralImage** (vx_context *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 [3.34]

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

vx_node **vxMagnitudeNode** (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *mag*);

vx_status **vxuMagnitude** (vx_context *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 [3.35]

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: [Optional] Average pixel value of type VX_TYPE_FLOAT32.
stddev: [Optional only in vxMeanStdDevNode] Standard deviation of the pixel values of VX_TYPE_FLOAT32.

Median Filter [3.36]

vx_node **vxMedian3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuMedian3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Min, Max Location [3.37]

vx_node **vxMinMaxLocNode** (vx_graph *graph*, vx_image *input*, vx_scalar *minVal*, vx_scalar *maxVal*, vx_array *minLoc*, vx_array *maxLoc*, vx_scalar *minCount*, vx_scalar *maxCount*);

vx_status **vxuMinMaxLoc** (vx_context *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}Val: The {min, max} value in the image.
{min, max}Loc: [Optional] The {min, max} locations of type VX_TYPE_COORDINATES2D.
{min, max}Count: [Optional] The number of detected {mins, maxes} in image. Type VX_TYPE_SIZE.

Min [3.38]

vx_node **vxMinNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuMin** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1. in2: Input image of type VX_DF_IMAGE_[U8, S16].
out: Output image which holds the result of min.

Max [3.39]

vx_node **vxMaxNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuMax** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1. in2: Input image of type VX_DF_IMAGE_[U8, S16].
out: Output image which holds the result of max.

Optical Flow Pyramid (LK) [3.40]

Compute the optical flow between two pyramid images.

vx_node **vxOpticalFlowPyrLKNode** (vx_graph *graph*, 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*);

vx_status **vxuOpticalFlowPyrLK** (vx_context *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 vx_array of type VX_TYPE_KEYPOINT.
termination: VX_TERM_CRITERIA_{ITERATIONS, EPSILON, 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 [3.41]

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

vx_node **vxPhaseNode** (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *orientation*);

vx_status **vxuPhase** (vx_context *context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *orientation*);

grad_{x, y}: The input {x, y} image of VX_DF_IMAGE_S16 format.
orientation: The phase image of VX_DF_IMAGE_U8 format.

Pixel-wise Multiplication [3.42]

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

vx_node **vxMultiplyNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*);

vx_status **vxuMultiply** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_float32 *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*);

in{1, 2}, out: Image of VX_DF_IMAGE_{U8, S16} format.
rounding_policy: VX_ROUND_POLICY_TO_{ZERO, NEAREST_EVEN}.
overflow_policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.
scale: A non-negative value of type VX_TYPE_FLOAT32.

(Continued on next page) ▶

◀ Vision Functions (cont.)

Remap [3.43]

$output(x, y) = input(map_x(x, y), map_y(x, y))$

vx_node **vxRemapNode** (vx_graph *graph*, vx_image *input*, vx_remap *table*, vx_enum *policy*, vx_image *output*);

vx_status **vxuRemap** (vx_context *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.
VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Scale Image [3.44]

vx_node **vxScaleImageNode** (vx_graph *graph,* vx_image *src*, vx_image *dst*, vx_enum *type*);

vx_node **vxHalfScaleGaussianNode** (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*);

vx_status **vxuScaleImage** (vx_context *context,* vx_image *src*, vx_image *dst*, vx_enum *type*);

vx_status **vxuHalfScaleGaussian** (vx_context *context*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*);

src, dst: The {source, destination} image of type VX_DF_IMAGE_U8.
type: An interpolation type.
VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.
input, output: Image of VX_DF_IMAGE_U8 format.
kernel_size: Gaussian filter input size. Supported values are 1, 3, and 5.

Sobel3x3 [3.45]

vx_node **vxSobel3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output_x*, vx_image *output_y*);

vx_status **vxuSobel3x3** (vx_context *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 [3.46]

Use a LUT to convert pixel values.

vx_node **vxTableLookupNode** (vx_graph *graph*, vx_image *input*, vx_lut *lut*, vx_image *output*);

vx_status **vxuTableLookup** (vx_context *context*, vx_image *input*, vx_lut *lut*, vx_image *output*);

input: Image of VX_DF_IMAGE_{U8, S16} format.
lut: The LUT of type VX_DF_IMAGE_{U8, S16}.
output: Output image of the same type as the input image.

Thresholding [3.47]

vx_node **vxThresholdNode** (vx_graph *graph*, vx_image *input*, vx_threshold *thresh*, vx_image *output*);

vx_status **vxuThreshold** (vx_context *context*, vx_image *input*, vx_threshold *thresh*, vx_image *output*);

input: The input image of VX_DF_IMAGE_[U8, S16] format.
output: The output Boolean image of type VX_DF_IMAGE_U8.
thresh: Thresholding object.

Warp Affine [3.48]

vx_node **vxWarpAffineNode** (vx_graph *graph*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

vx_status **vxuWarpAffine** (vx_context *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.
VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Warp Perspective [3.49]

vx_node **vxWarpPerspectiveNode** (vx_graph *graph*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

vx_status **vxuWarpPerspective** (vx_context *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.
VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Scan Classifier [3.50]

Scan a feature-map and classify each scan-window.

vx_node **vxScanClassifierNode** (vx_graph *graph*, vx_tensor *input_feature_map*, vx_classifier_model *model*, vx_int32 *scanwindow_width*, vx_int32 *scanwindow_height*, vx_int32 *step_x*, vx_int32 *step_y*, vx_array *object_confidences*, vx_array *object_rectangles*, vx_scalar *num_objects*);

vx_status **vxuScanClassifier** (vx_context *context*, vx_tensor *input_feature_map*, vx_classifier_model *model*, vx_int32 *scanwindow_width*, vx_int32 *scanwindow_height*, vx_int32 *step_x*, vx_int32 *step_y*, vx_array *object_confidences*, vx_array *object_rectangles*, vx_scalar *num_objects*);

input_feature_map: For example, the output of **vxHOGFeaturesNode**.
model: The pre-trained model loaded using **vxImportClassifierModel**.
scan_window_width: Width of the scan window
scan_window_height: Height of the scan window
step_x: Horizontal step-size (along x-axis)
step_y: Vertical step-size (along y-axis)
object_confidences: [Optional] An array of confidences measure.
object_rectangles: An array of object positions in VX_TYPE_RECTANGLE.
num_objects: [Optional] Number of object detected in a VX_SIZE scalar.

Bilateral FIlter [3.51]

Apply bilateral filtering to the input tensor.

vx_node **vxBilateralFilterNode** (vx_graph *graph*, vx_tensor *src*, vx_int32 *diameter*, vx_float32 *sigmaSpace*, vx_float32 *sigmaValues*, vx_tensor *dst*);

vx_status **vxuBilateralFilter** (vx_context *context*, vx_tensor *src*, vx_int32 *diameter*, vx_float32 *sigmaSpace*, vx_float32 *sigmaValues*, vx_tensor *dst*);

src: The input data vx_tensor of type VX_TYPE_[UINT8, INT16].
diameter: Diameter of each pixel neighbourhood used during filtering.
sigmaValues: Filter sigma in the radiometric space.
sigmaSpace: Filter sigma in the spatial space.
dst: The output data of same type and size of the input in *src*.

MatchTemplate [3.52]

Compare an image template against overlapped image regions.

vx_node **vxMatchTemplateNote** (vx_graph *graph*, vx_image *src*, vx_image *templateImage*, vx_enum *matchingMethod*, vx_image *output*);

vx_status **vxuMatchTemplateNode** (vx_context *context*, vx_image *src*, vx_image *templateImage*, vx_enum *matchingMethod*, vx_image *output*);

src: The input image of type VX_DF_IMAGE_U8.
templateImage: Searched template of type VX_DF_IMAGE_U8.
matchingMethod: • Specifies comparison method vx_comp_metric_e.
output: Map image of comparison results of type VX_DF_IMAGE_S16

LBP [3.53]

(Local Binary Pattern)
Compute local binary pattern over a window of the input image.

vx_node **vxLBPNode** (vx_graph *graph*, vx_image *in*, x_enum *format*, vx_int8 *kernel_size*, vx_image *out*);

vx_status **vxuLBPNode** (vx_context *context*, vx_image *in*, x_enum *format*, vx_int8 *kernel_size*, vx_image *out*);

in: Input image of type VX_DF_IMAGE_U8. *SrcImg* in the equations.
format: A variation of LBP like original LBP and mLBP from vx_lbp_format_e: VX_{LBP, MLBP, ULBP}
kernel_size: Kernel size. Only size of 3 and 5 are supported
out: Output image of type VX_DF_IMAGE_U8. *DstImg* in equations.

HOG [3.54]

(Histogram of Oriented Gradients)
Extract HOG features from the input grayscale image.

vx_node **vxHOGCellsNode** (vx_graph *graph*, vx_image *input*, vx_int32 *cell_width*, vx_int32 *cell_height*, vx_int32 *num_bins*, vx_tensor *magnitudes*, vx_tensor *bins*);

vx_status **vxuHOGCells** (vx_context *context*, vx_image *input*, vx_int32 *cell_size*, vx_int32 *num_bins*, vx_tensor *magnitudes*, vx_tensor *bins*);

input :The input image of type VX_DF_IMAGE_U8.
cell_{width,height}: Histogram cell width/height; type VX_TYPE_INT32.
num_bins: The histogram size of type VX_TYPE_INT32.
magnitudes: The output gradient magnitudes of type VX_TYPE_FLOAT32.
bins: The output gradient angle bins of of type VX_TYPE_INT8.

vx_node **vxHOGFeaturesNode** (vx_graph *graph*, vx_image *input*, vx_tensor *magnitudes*, vx_tensor *bins*, const vx_hog_t *params*, vx_size *hog_param_size*, vx_tensor *features*);

vx_status **vxuHOGFeatures** (vx_context *context*, vx_image *input*, vx_tensor *magnitudes*, vx_tensor *bins*, const vx_hog_t *params*, vx_size *hog_param_size*, vx_tensor *features*);

input: The input image of type VX_DF_IMAGE_U8.
magnitudes: Gradient magnitudes of vx_tensor, type VX_TYPE_FLOAT32.
bins: The gradient angle bins of vx_tensor of type VX_TYPE_INT8.
params: The parameters of type vx_hog_t.
hog_param_size: Size of vx_hog_t in bytes.
features: Output HOG features of vx_tensor of type VX_TYPE_FLOAT32.

HoughLinesP [3.55]

Find Probabilistic Hough Lines detected in input binary image.

vx_node **vxHoughLinesPNode** (vx_graph *graph*, vx_image *input*, const vx_hough_lines_p_t *params*, vx_array *lines_array*, vx_scalar *num_lines*);

vx_status **vxuHoughLinesPNode** (vx_context *context*, vx_image *input*, const vx_hough_lines_p_t *params*, vx_array *lines_array*, vx_scalar *num_lines*);

input: The 8 bit, single channel binary source image
params: Parameters of the struct vx_hough_lines_p_t
lines_array: Contains array of lines.
num_lines: [Optional] Total number of detected lines in image.

Tensor Multiply [3.56]

vx_node **vxTensorMultiplyNode** (vx_graph *graph*, vx_tensor *input1*, vx_tensor *input2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_tensor *output*);

vx_status **vxuTensorMultiply** (vx_context *context*, vx_tensor *input1*, vx_tensor *input2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_tensor *output*);

context: The reference to the overall context.
input1, ***input2***: Input tensor data. Data types must match.
scale A non-negative VX_TYPE_FLOAT32 multiplied to each product before overflow handling.
overflow_policy: A vx_convert_policy_e enumeration.
rounding_policy: A vx_round_policy_e enumeration.
output: Output tensor data with same dimensions as input tensor data.

Tensor Add [3.57]

vx_node **vxTensorAddNode** (vx_graph *graph*, vx_tensor *input1*, vx_tensor *input2*, vx_enum *policy*, vx_tensor *output*);

vx_status **vxuTensorAdd** (vx_context *context*, vx_tensor *input1*, vx_tensor *input2*, vx_enum *policy*, vx_tensor *output*);

input1, input2: Input tensor data. Data types must match.
policy: A vx_convert_policy_e enumeration.
output: Output tensor data with same dimensions as input tensor data.

Tensor Subtract [3.58]

vx_node **vxTensorSubtractNode** (vx_graph *graph*, vx_tensor *input1*, vx_tensor *input2*, vx_enum *policy*, vx_tensor *output*);

vx_status **vxuTensorSubtract** (vx_context *context*, vx_tensor *input1*, vx_tensor *input2*, vx_enum *policy*, vx_tensor *output*);

input1, input2: Input tensor data. Data types must match.
policy: A vx_convert_policy_e enumeration.
output: Output tensor data with same dimensions as input tensor data.

Tensor Table Lookup [3.59]

Performs LUT on element values in the input tensor data.

vx_node **vxTensorTableLookupNode** (vx_graph *graph*, vx_tensor *input1*, vx_lut *lut*, vx_tensor *output*);

vx_status **vxuTensorTableLookup** (vx_context *context*, vx_tensor *input1*, vx_lut *lut*, vx_tensor *output*);

input1: Input tensor data. Data types must match.
lut: The lookup table.
output: Output tensor data with same dimensions as input tensor data.

Tensor Transpose [3.60]

vx_node **vxTensorTransposeNode** (vx_graph *graph*, vx_tensor *input*, vx_tensor *output*, vx_size *dimension1*, vx_size *dimension2*);

vx_status **vxuTensorTranspose** (vx_context *context*, vx_tensor *input*, vx_tensor *output*, vx_size *dimension1*, vx_size *dimension2*);

input: Input tensor data.
dimension1: Dimension index that is transposed with dim 2.
dimension2: Dimension index that is transposed with dim 1.

(Continued on next page) ▶

◀ Vision Functions (cont.)

Tensor Convert Bit-Depth [3.61]

Compute median values over a window of the input image.

vx_node **vxTensorConvertDepthNode** (vx_graph *graph*, vx_tensor *input*, vx_enum *policy*, vx_scalar *norm*, vx_scalar *offset*, vx_tensor *output*);

vx_status **vxuTensorConvertDepth** (vx_context *context*, vx_tensor *input*, vx_enum *policy*, vx_scalar *norm*, vx_scalar *offset*, vx_tensor *output*);

input: The input tensor.
policy: A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
norm: Scalar containing a VX_TYPE_FLOAT32 of the normalization value.
offset: A scalar containing a VX_TYPE_FLOAT32 of the offset value subtracted before normalization.
output The output tensor.

Tensor Matrix Multiply [3.62]

Compute median values over a window of the input image.

vx_node **vxTensorMatrixMultiplyNode** (vx_graph *graph*, vx_tensor *input1*, vx_tensor *input2*, vx_tensor *input3*, const vx_matrix_multiply_params_t *matrix_multiply_params*, vx_tensor *output*);

vx_status **vxuMatrixMultiply** (vx_context *context*, vx_tensor *input1*, vx_tensor *input2*, vx_tensor *input3*, const vx_matrix_multiply_params_t *matrix_multiply_params*, vx_tensor *output*);

input1: The first input 2D tensor of type VX_TYPE_{INT16, UINT8, INT8}.
input2: The second 2D tensor. Must be in the same data type as *input1*.
input3: [Optional] The third 2D tensor. Must be same type as *input1*.
output: The output 2D tensor. Must be in the same data type as *input1*.

Array Objects [3.69]

Attribute: enum vx_array_attribute_e:
VX_ARRAY_{ITEMTYPE, NUMITEMS, CAPACITY, ITEMSIZE}

Macro allowing access a specific indexed element in an array.

#define **vxFormatArrayPointer**(*ptr*, *index*, *stride*) \
(&(((vx_uint8)(*ptr*))[(*index*) (*stride*)]))

Macro allowing access to an array item as a typecast pointer de-reference.

#define **vxArrayItem**(*type*, *ptr*, *index*, *stride*) \
((*type*)(vxFormatArrayPointer((*ptr*), (*index*), (*stride*))))

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

count: The total number of elements to insert.
ptr: Location from which to read the input values.
stride: The stride in bytes between elements.

ret **vxCopyArrayRange** (vx_array *array*, vx_size *range_start*, vx_size *range_end*, vx_size *user_stride*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

range_start: The index of the first item of the array object to copy.
range_end: Index of item following last item of array object to copy.
user_stride: Number of bytes between the beginning of two consecutive items in the user memory pointed by *user_ptr*.
user_ptr: Address of memory location to store/get the requested data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

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

item_type: The type of objects to hold.
capacity: The maximal number of items that the array can hold.

Create opaque reference to a virtual array with no direct user access.

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

item_type: The type of objects to hold, or zero to indicate an unspecified item type.
capacity: The maximal number of items that the array can hold, or zero to indicate an unspecified capacity.

Gives the application direct access to a range of an array object.

vx_status **vxMapArrayRange** (vx_array *array*, vx_size *range_start*, vx_size *range_end*, vx_map_id **map_id*, vx_size **stride*, void ***ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_uint32 *flags*);

range_start: The index of the first item of the array object to map.
range_end: The index of the item following the last item of the array object to map.
map_id: The address of a vx_map_id variable where the function returns a map identifier.
stride: The address of a vx_size variable where the function returns the memory layout of the mapped array range.
ptr: Pointer to the address to access the requested data.
usage: VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: VX_NOGAP_X.

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

attribute: An attribute from vx_array_attribute_e.
ptr: Location at which to store the result.
size: The size in bytes of the container to which *ptr* points.

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*);

new_num_items: The new number of items for the Array.

Unmap and commit potential changes to array object range.

vx_status **vxUnmapArrayRange** (vx_array *array*, vx_map_id *map_id*);

map_id: Unique map identifier returned by **vxMapArrayRange**.

Object: Array (Advanced) [3.84]

Register user-defined structures to the context.

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

size: The size of user struct in bytes.

Classifier Model Objects [3.81]

Create an opaque reference to a classifier model.

vx_classifier_model **vxImportClassifierModel** (vx_context *context*, vx_enum *format*, const vx_uint8 * *ptr*, vx_size *length*);

format: The binary format which contains the classifier model. Must be VX_CLASSIFIER_MODEL_UNDEFINED
ptr: A memory pointer to the binary format.
length: Size in bytes of binary format data.

vx_status **vxReleaseClassifierModel** (vx_classifier_model * *model*);

Convolution Objects [3.70]

Attribute: enum vx_convolution_attribute_e:
VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}

Copy coefficients from/into a convolution object.

vx_status **vxCopyConvolutionCoefficients** (vx_convolution *conv*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

user_ptr: The address of the memory location at which to store or get the coefficient data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to a [virtual] convolution matrix object. Virtual convolution matrix object has no direct user access.

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

vx_convolution **vxCreateVirtualConvolution** (vx_graph *graph*, vx_size *columns*, vx_size *rows*);

columns, rows: Must be odd and >= 3 and less than the value returned from VX_CONTEXT_CONVOLUTION_MAX_DIMENSION.

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

attribute: VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container or data to which *ptr* points.

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

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

attribute: VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}.
ptr: The pointer to the value to which to set the attribute.
size: The size in bytes of the container or data to which *ptr* points.

Get Status [3.63.6]

Return status values from Object constructors if they fail.

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

Context Objects [3.66]

Attribute: enum vx_context_attribute_e: VX_CONTEXT_*x* where *x* may be VENDOR[_ID], UNIQUE_KERNELS, MODULES, REFERENCES, IMPLEMENTATION, EXTENSIONS[_SIZE], UNIQUE_KERNEL_TABLE, IMMEDIATE_BORDER[_POLICY], OPTICAL_FLOW_MAX_WINDOW_DIMENSION, {CONVOLUTION, NONLINEAR}_MAX_DIMENSION, MAX_TENSOR_DIMS, VERSION

vx_context **vxCreateContext** (void);

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

reference: The reference from which to extract the context.

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

attribute: Attribute to query from vx_context_attribute_e.
ptr: Pointer to where to store the result.
size: Size in bytes of the container to which *ptr* points.

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

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

attribute: Attribute to set from vx_context_attribute_e.
ptr: Pointer to the data to which to set the attribute.
size: Size in bytes of the container to which *ptr* points.

Set the default target of the immediate mode.

vx_status **vxSetImmediateModeTarget** (vx_context *context*, vx_enum *target_enum*, const char **target_string*);

target_enum: Default immediate mode target enum to be set to vx_context object. VX_TARGET_{ANY, STRING, VENDOR_BEGIN}.
target_string: The target name ASCII string.

Delay Objects [3.87]

Attribute: enum vx_delay_attribute_e:
VX_DELAY_{TYPE, SLOTS}

Shift the internal delay ring by one.

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

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

exemplar: The exemplar object. Supported types are VX_TYPE_*x* where *x* may be ARRAY, CONVOLUTION, DISTRIBUTION, IMAGE, LUT, MATRIX, OBJECT_ARRAY, PYRAMID, REMAP, SCALAR, THRESHOLD, TENSOR
num_slots: The number of objects in the delay.

Retrieve a reference from a delay object.

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

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

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

attribute: The attribute to query from vx_delay_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

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

Distribution Objects [3.71]

Attribute: enum vx_distribution_attribute_e:
VX_DISTRIBUTION_*x* where *x* is DIMENSIONS, OFFSET, RANGE, BINS, WINDOW, SIZE

Allow the application to copy from/into a distribution object.

vx_status **vxCopyDistribution** (vx_distribution *distribution*, void * *user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

user_ptr: The address of memory location to store or get the data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to a [virtual] 1D distribution. Virtual distribution object has no direct user access.

vx_distribution **vxCreateDistribution** (vx_context *context*, vx_size *numBins*, vx_int32 *offset*, vx_int32 *range*);

(Continued on next page) ▶

◀ Distribution Objects (cont.)

vx_distribution **vxCreateVirtualDistribution** (vx_graph *graph*, vx_size *numBins*, vx_int32 *offset*, vx_int32 *range*);

numBins: The number of bins in the distribution.
offset: The start offset into the range value.
range: Total number of consecutive values of the distribution interval.

Allow application direct access to the distribution object.

vx_status **vxMapDistribution** (vx_distribution *distribution*, vx_map_id **map_id*, void ***ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_bitfield *flags*);

map_id: Address of variable where function returns a map identifier.
ptr: Address of a pointer that the function sets to the address where the requested data can be accessed.
usage: VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: Must be 0.

vx_status **vxQueryDistribution** (vx_distribution *distribution*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: Attribute to query from vx_distribution_attribute_e.
ptr: The location at which to store the result.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseDistribution** (vx_distribution **distribution*);

Set the distribution back to the memory.

vx_status **vxUnmapDistribution** (vx_distribution *distribution*, vx_map_id *map_id*);

map_id: The unique map identifier that was returned when calling **vxMapDistribution**.

Graph Objects [3.67]

Attribute: enum vx_graph_attribute_e: VX_GRAPH_{ NUMNODES, PERFORMANCE, NUMPARAMETERS, STATE}

Create an empty graph.

vx_graph **vxCreateGraph** (vx_context *context*);

Return 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 the attributes of a graph.

vx_status **vxQueryGraph** (vx_graph *graph*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: An attribute from vx_graph_attribute_e.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

Register a delay for auto-aging.

vx_status **vxRegisterAutoAging** (vx_graph *graph*, vx_delay *delay*);

delay: The delay to automatically age.

Release a reference to a graph.

vx_status **vxReleaseGraph** (vx_graph **graph*);

Schedule a graph for future execution.

vx_status **vxScheduleGraph** (vx_graph *graph*);

Allow the user to set attributes on the graph.

vx_status **vxSetGraphAttribute** (vx_graph *graph*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: An attribute from vx_graph_attribute_e.
ptr: The location from which to read the value.
size: The size in bytes of the container to which *ptr* points.

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*);

Image Objects [3.72]

Attribute: enum vx_image_attribute_e: VX_IMAGE_{WIDTH, HEIGHT, FORMAT, PLANES, SPACE, RANGE, MEMORY_TYPE, IS_UNIFORM, UNIFORM_VALUE}

Addressing image patch structure: The addressing image patch structure vx_imagepatch_addressing_t is used by the host to address pixels in an image patch. The fields of the structure are:

dim: The dimensions of image in logical pixel units in x & y direction.
stride: The physical byte distance from a logical pixel to the next logically adjacent pixel in the positive x or y direction.
scale: The relationship of scaling from primary plane to this plane.
step: The number of logical pixel units to skip to arrive at the next physically unique pixel.

Copy a rectangular patch from/into an image object plane.

vx_status **vxCopyImagePatch** (vx_image *image*, const vx_rectangle_t **image_rect*, vx_uint32 *image_plane_index*, const vx_imagepatch_addressing_t **user_addr*, void * *user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

image_rect: The coordinates of the image patch.
image_plane_index: The plane index of the image object.
user_addr: The address of a structure describing the layout of the user memory location pointed by *user_ptr*.
user_ptr: The address of the memory location to store or get the data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

vx_image **vxCreateImage** (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*);

width, height: The image {width, height} in pixels.
color: • The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.

Create a sub-image from a single plane channel of another image.

vx_image **vxCreateImageFromChannel** (vx_image *img*, vx_enum *channel*);

img: The reference to the parent image.
channel: VX_CHANNEL_{0, 1, 2, 3, R, G, B, A, Y, U, V}.

Create a reference to an externally allocated image object.

vx_image **vxCreateImageFromHandle** (vx_context *context*, vx_df_image *color*, const vx_imagepatch_addressing_t *addrs*[ ], void *const *ptrs*[ ], vx_enum *memory_type*);

color: • The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
***addrs*[ ]:** The array of image patch addressing structures that define the dimension and stride of the array of pointers.
***ptrs*[ ]:** The array of platform-defined references to each plane.
memory_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create an image from another image given a rectangle.

vx_image **vxCreateImageFromROI** (vx_image *img*, const vx_rectangle_t **rect*);

img: The reference to the parent image.
rect: The region of interest rectangle.

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*, const vx_pixel_value_t **value*);

width, height: The image {width, height} in pixels.
color: • The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
value: The pointer to the pixel value to which to set all pixels.

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*);

width, height: The image {width, height} in pixels.
color: • The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and color space.

Access a specific indexed pixel in an image patch.

void * **vxFormatImagePatchAddress1d** (void **ptr*, vx_uint32 *index*, const vx_imagepatch_addressing_t **addr*);

ptr: The base pointer.
index: The 0-based index of the pixel count in the patch.
addr: Pointer to addressing mode information returned from **vxMapImagePatch**.

Access a specific pixel at a 2d coordinate in an image patch.

void * **vxFormatImagePatchAddress2d** (void **ptr*, vx_uint32 *x*, vx_uint32 *y*, const vx_imagepatch_addressing_t **addr*);

ptr: The base pointer.
addr: Pointer to addressing mode information returned from **vxMapImagePatch**.

Retrieve the valid region of the image as a rectangle.

vx_status **vxGetValidRegionImage** (vx_image *image*, vx_rectangle_t **rect*);

image: The image from which to retrieve the valid region.
rect: The destination rectangle.

Give direct access to a rectangular patch of image object plane.

vx_status **vxMapImagePatch** (vx_image *image*, const vx_rectangle_t * *rect*, vx_uint32 *plane_index*, vx_map_id * *map_id*, vx_imagepatch_addressing_t * *addr*, void **ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_uint32 *flags*);

rect: The coordinates of the image patch.
plane_index: The plane index of the image object to be accessed.
map_id: Address of a vx_map_id variable where function returns a map identifier.
addr: Address of a structure describing memory layout of image patch.
ptr: The address of a pointer that the function sets to the address where the requested data can be accessed.
usage: VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: VX_NOGAP_X.

vx_status **vxQueryImage** (vx_image *image*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query from vx_image_attribute_e.
ptr: The pointer to the location at which to store the result.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseImage** (vx_image **image*);

image: The reference to the image to release.

vx_status **vxSetImageAttribute** (vx_image *image*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: The attribute to set from vx_image_attribute_e.
ptr: The pointer to the location from which to read the value.
size: The size in bytes of the container to which *ptr* points.

Initialize an image with constant pixel value.

vx_status **vxSetImagePixelValues** (vx_image *image*, const vx_pixel_value_t * *pixel_value*);

pixel_value: Pointer to the constant pixel value.

Set valid rectangle for image according to a supplied rectangle.

vx_status **vxSetImageValidRectangle** (vx_image *image*, const vx_rectangle_t **rect*);

rect: The value to be set to the image valid rectangle.

Swaps image handle of an image previously created from handle.

vx_status **vxSwapImageHandle** (vx_image *image*, void *const *new_ptrs*[ ], void **prev_ptrs*[ ], vx_size *num_planes*);

***new_ptrs*[ ]:** The pointer to a caller-owned array that contains the new image handle.
***prev_ptrs*[ ]:** The pointer to a caller-owned array in which the application returns the previous image handle.
num_planes: Number of planes in the image.

Unmap and commit potential changes to an image object patch.

vx_status **vxUnmapImagePatch** (vx_image *image*, vx_map_id *map_id*);

map_id: The unique map identifier returned by **vxMapImagePatch**.

Kernel Objects [3.88]

Attribute: enum vx_kernel_attribute_e:
VX_KERNEL_{PARAMETERS, NAME, ENUM, LOCAL_DATA_SIZE}

Obtain a reference to kernel using vx_kernel_e enumeration.

vx_kernel **vxGetKernelByEnum** (vx_context *context*, vx_enum *kernel*);

kernel: • Value from vx_kernel_e or vendor or client-defined value.

Obtain a reference to a kernel using a string to specify name.

vx_kernel **vxGetKernelByName** (vx_context *context*, const vx_char **name*);

name: The string of the name of the kernel to get. See the list of allowable string names.

vx_status **vxQueryKernel** (vx_kernel *kernel*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query from vx_kernel_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

vx_status **vxReleaseKernel** (vx_kernel **kernel*);

Allowable strings for **vxGetKernelByName** *name*

The allowable string names are org.khronos.openvx.*x*, where *x* may be one of the strings in the following list:

absdiff	convertdepth	histogram	non_linear_filter	tensor_add
accumulate	custom_convolution	integral_image	not	tensor_convert_depth
accumulate_square	dilate_3x3	laplacian_pyramid	optical_flow_pyr_lk	tensor_multiply
accumulate_weighted	equalize_histogram	laplacian_reconstruct	or	tensor_subtract
add	erode_3x3	magnitude	phase	tensor_table_lookup
and	fast_corners	matrix_multiply	remap	tensor_transpose
box_3x3	gaussian_3x3	mean_stddev	scale_image	threshold
canny_edge_detector	gaussian_pyramid	median_3x3	sobel_3x3	warp_affine
channel_combine	halfscale_gaussian	minmaxloc	subtract	warp_perspective
channel_extract	harris_corners	multiply	table_lookup	xor
color_convert				

Also see Neural Network Extension on page 11 of this reference guide for additional strings.

LUT Objects [3.73]

Attribute: enum vx_lut_attribute_e:
VX_LUT_{TYPE, COUNT, SIZE, OFFSET}

Allow the application to copy from/into a LUT object.

vx_status **vxCopyLUT** (vx_lut *lut*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

user_ptr: The address of the memory location to store or get data.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create [virtual] LUT object of a given type. Virtual LUT has no direct user access.

vx_lut **vxCreateLUT** (vx_context *context*, vx_enum *data_type*, vx_size *count*);

]vx_lut **vxVirtualCreateLUT** (vx_graph *graph*, vx_enum *data_type*, vx_size *count*);

data_type: VX_TYPE_UINT8 or VX_TYPE_INT16
count: The number of entries desired.

Allow the application to directly access a LUT object.

vx_status **vxMapLUT** (vx_lut *lut*, vx_map_id **map_id*, void ***ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_bitfield *flags*);

map_id: Address where the function returns a map identifier.
ptr: The address of a pointer that the function sets to the address where the requested data can be accessed.
usage: VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: Must be 0.

vx_status **vxQueryLUT** (vx_lut *lut*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: Attribute to query. From vx_lut_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

vx_status **vxReleaseLUT** (vx_lut **lut*);

Allow the application to copy from/into a LUT object.

vx_status **vxUnmapLUT** (vx_lut *lut*, vx_map_id *map_id*);

map_id: The unique map identifier returned by **vxMapLUT**.

Matrix Objects [3.74]

Attribute: enum vx_matrix_attribute_e:
VX_MATRIX_{TYPE, ROWS, COLUMNS, SIZE, ORIGIN, PATTERN}

Copy from or to a matrix object.

vx_status **vxCopyMatrix** (vx_matrix *matrix*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

user_ptr: The address of the memory location for storage or retrieval.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a [virtual] reference to a matrix object.

vx_matrix **vxCreateMatrix** (vx_context *c*, vx_enum *data_type*, vx_size *columns*, vx_size *rows*);

vx_matrix **vxCreateVirtualMatrix** (vx_graph *graph*, vx_enum *data_type*, vx_size *columns*, vx_size *rows*);

data_type: Unit format of matrix. VX_TYPE_{UINT8, INT32, FLOAT32}.
columns: The first dimension.
rows: The second dimension.

Create a reference to a matrix object from a Boolean pattern.

vx_matrix **vxCreateMatrixFromPattern** (vx_context *c*, vx_enum *pattern*, vx_size *columns*, vx_size *rows*);

vx_matrix **vxCreateMatrixFromPatternAndOrigin** (vx_context *c*, vx_enum *pattern*, vx_size *columns*, vx_size *rows*, vx_size *origin_col*, vx_size *origin_row*);

c: The reference to the overall context.
pattern: Matrix pattern. See VX_MATRIX_PATTERN and vx_pattern_e.
columns: The first dimension.
rows: The second dimension.

vx_status **vxQueryMatrix** (vx_matrix *mat*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query from vx_matrix_attribute_e.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseMatrix** (vx_matrix **mat*);

Node Objects [3.68]

Attribute: enum vx_node_attribute_e: VX_NODE_{ STATUS, PERFORMANCE, BORDER, LOCAL_DATA_{SIZE, PTR}, PARAMETERS, IS_REPLICATED, REPLICATE_FLAGS, VALID_RECT_RESET}

vx_status **vxQueryNode** (vx_node *node*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The vx_node_attribute_e value to query.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseNode** (vx_node **node*);

Remove a node from its parent graph and release it.

void **vxRemoveNode** (vx_node **node*);

Create replicas of the same node to process a set of objects.

vx_status **vxReplicateNode** (vx_graph *graph*, vx_node *first_node*, vx_bool *replicate*[ ], vx_uint32 *number_of_parameters*);

graph: The reference to the graph.
first_node: The reference to the graph node to replicate.
***replicate*[ ]:** An array indicating the parameters to replicate.
number_of_parameters: Number of elements in the replicate array.

Set attributes of a node before graph validation.

vx_status **vxSetNodeAttribute** (vx_node *node*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: The vx_node_attribute_e value to set.
ptr: Pointer to desired value of the attribute.
size: The size in bytes of the objects to which *ptr* points.

Set the node target to the provided value.

vx_status **vxSetNodeTarget** (vx_node *node*, vx_enum *target_enum*, const char **target_string*);

target_enum: The target enum to be set to the vx_node object. VX_TARGET_{ANY, STRING, VENDOR_BEGIN}.
target_string: The target name ASCII string.

Object: Node (Advanced) [3.85]

Define 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.

ObjectArray Objects [3.79]

Attribute: enum vx_object_array_attribute_e:
VX_OBJECT_ARRAY_{ITEMTYPE, NUMITEMS}

Create a reference to an ObjectArray of count objects.

vx_object_array **vxCreateObjectArray** (vx_context *context*, vx_reference *exemplar*, vx_size *count*);

exemplar: The exemplar object that defines the metadata of the created objects in the ObjectArray.
count: The number of objects to create in the ObjectArray.

Create an opaque reference to a virtual ObjectArray.

vx_object_array **vxCreateVirtualObjectArray** (vx_graph *graph*, vx_reference *exemplar*, vx_size *count*);

graph: The reference to the graph in which to create the ObjectArray.
exemplar: The exemplar object that defines the type of object in the ObjectArray.
count: The number of objects to create in the ObjectArray.

Retrieve a reference to an object in the ObjectArray.

vx_reference **vxGetObjectArrayItem** (vx_object_array *arr*, vx_uint32 *index*);

index: The index of the object in the ObjectArray.

vx_status **vxQueryObjectArray** (vx_object_array *arr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: A value from the attribute enum vx_object_array_attribute_e
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseObjectArray** (vx_object_array **arr*);

Parameter Objects [3.89]

Attribute: enum vx_parameter_attribute_e: VX_PARAMETER_{INDEX, DIRECTION, TYPE, STATE, REF}

Retrieve a vx_parameter from a vx_kernel.

vx_parameter **vxGetKernelParameterByIndex** (vx_kernel *kernel*, vx_uint32 *index*);

index: The index of the parameter.

Retrieve a vx_parameter from a vx_node.

vx_parameter **vxGetParameterByIndex** (vx_node *node*, vx_uint32 *index*);

index: The index of the parameter.

vx_status **vxQueryParameter** (vx_parameter *param*, vx_enum *attribute*, void **ptr*, vx_size *size*);

ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

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*);

index: The index of the parameter.
value: The desired value of the parameter.

Associate parameter and data references with a kernel on a node.

vx_status **vxSetParameterByReference** (vx_parameter *parameter*, vx_reference *value*);

value: The value to associate with the parameter.

Reference Objects [3.65]

Attribute: enum vx_reference_attribute_e: VX_REFERENCE_{COUNT, TYPE, NAME}

Query the attributes of a reference object.

vx_status **vxQueryReference** (vx_reference *ref*, vx_enum *attribute*, void **ptr*, vx_size *size*);

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

Release a reference.

vx_status **vxReleaseReference** (vx_reference **ref_ptr*);

Increment the reference counter of an object.

vx_status **vxRetainReference** (vx_reference *ref*);

Name a reference.

vx_status **vxSetReferenceName** (vx_reference *ref*, const vx_char **name*);

name: NULL if not named, or a pointer to NUL-terminated name.

Remap Objects [3.76]

Attribute: enum vx_remap_attribute_e: VX_REMAP_{ SOURCE_{WIDTH, HEIGHT}, DESTINATION_{WIDTH, HEIGHT}}

Copy a rectangular patch from/into a remap object.

vx_status **vxCopyRemapPatch** (vx_remap *remap*, const vx_rectangle_t * *rect*, vx_size *user_stride_y*, void * *user_ptr*, vx_enum *user_coordinate_type*, vx_enum *usage*, vx_enum *user_mem_type*);

rect: The coordinates of remap patch.
user_stride_y: The difference between the address of the first element of two successive lines of the remap patch in user memory.
user_ptr: Address of user memory for the remap data.
user_coordinate_type: Declares the type of the source coordinate remap data. It must be VX_TYPE_COORDINATES2DF.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}

Create a [virtual] 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*);

vx_remap **vxCreateVirtualRemap** (vx_graph *graph*, vx_uint32 *src_width*, vx_uint32 *src_height*, vx_uint32 *dst_width*, vx_uint32 *dst_height*);

src_{width, height}: {Width, Height} of source image in pixels.
dst_{width, height}: {Width, Height} of destination image in pixels.

vx_status **vxQueryRemap** (vx_remap *r*, vx_enum *attribute*, void **ptr*, vx_size *size*);

r: The remap to query.
attribute: An attribute from vx_remap_attribute_e.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleaseRemap** (vx_remap **table*);

table: A pointer to the remap table to release.

Get direct access to a rectangular patch of a remap object.

vx_status **vxMapRemapPatch** (vx_remap *remap*, const vx_rectangle_t *rect*, vx_map_id *map_id*, vx_size *stride_y*, void ***ptr*, vx_enum *coordinate_type*, vx_enum *usage*, vx_enum *mem_type*);

rect: The coordinates of remap patch.
map_id: Address where map identifier is returned.
stride_y: Address where the function returns the difference between the address of the first element of two successive lines in the mapped remap patch.
ptr: Address of pointer where remap patch data can be accessed.
coordinate_type: Must be VX_TYPE_COORDINATES2DF.
usage: VX_READ_AND_WRITE or VX_{READ, WRITE}_ONLY.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Unmap and commit potential changes to a remap object patch.

vx_status **vxUnmapRemapPatch** (vx_remap *remap*, vx_map_id *map_id*);

map_id: Unique map identifier returned by **vxMapRemapPatch**.

Scalar Objects [3.77]

Attribute: enum vx_scalar_attribute_e: VX_SCALAR_TYPE

Allow application to copy from/into a scalar object [with size].

vx_status **vxCopyScalar**(vx_scalar *scalar*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

vx_status **vxCopyScalarWithSize**(vx_scalar *scalar*, vx_size *size*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

size: The size in bytes of the container to which *user_ptr* points.
user_ptr: Address of the memory location where to store the requested data, or from where to get the data to store into the scalar object.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a [virtual] reference to a scalar object [with size].

vx_scalar **vxCreateScalar** (vx_context *context*, vx_enum *data_type*, const void **ptr*);

vx_scalar **vxCreateVirtualScalar**(vx_graph *graph*, vx_enum *data_type*);

vx_scalar **vxCreateScalarWithSize**(vx_context *context*, vx_enum *data_type*, const void **ptr*, vx_size *size*)

data_type: The type of data to hold. Must be > VX_TYPE_INVALID and ≤ VX_TYPE_VENDOR_STRUCT_END, or must be vx_enum returned from **vxRegisterUserStruct**.
ptr: Pointer to the initial value of the scalar.
size: The size in bytes of the container to which *user_ptr* points.

vx_status **vxQueryScalar** (vx_scalar *scalar*, vx_enum *attribute*, void **ptr*, vx_size *size*);

ptr: Location at which to store the result.
size: The size of the container to which *ptr* points.

vx_status **vxReleaseScalar** (vx_scalar **scalar*);

Pyramid Objects [3.75]

Attribute: enum vx_pyramid_attribute_e: VX_PYRAMID_{LEVELS, SCALE, WIDTH, HEIGHT, FORMAT}

vx_pyramid **vxCreatePyramid** (vx_context *context*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

levels: The number of levels desired.
scale: This must be a non-zero positive value.
width: The width of the 0th-level image in pixels.
height: The height of the 0th-level image in pixels.
format: Format of all images in the pyramid or VX_DF_IMAGE_VIRT.

vx_pyramid **vxCreateVirtualPyramid** (vx_graph *graph*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

levels: The number of levels desired.
scale: This must be a non-zero positive value.
width: The width of the 0th-level image in pixels.
height: The height of the 0th-level image in pixels.
format: Format of all images in the pyramid.

vx_image **vxGetPyramidLevel** (vx_pyramid *pyr*, vx_uint32 *index*);

index: The index of the level, such that *index* is less than *levels*.

vx_status **vxQueryPyramid** (vx_pyramid *pyr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to query from vx_pyramid_attribute_e.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

vx_status **vxReleasePyramid** (vx_pyramid **pyr*);

Tensor Objects [3.80]

Attribute: enum vx_tensor_attribute_e: VX_TENSOR_*x* where *x* may be NUMBER_OF_DIMS, DIMS, DATA_TYPE, or FIXED_POINT_POSITION.

Copy a view patch from/into an tensor object.

vx_status **vxCopyTensorPatch**(vx_tensor *tensor*, vx_size *number_of_dims*, const vx_size * *view_start*, const vx_size * *view_end*, const vx_size * *user_stride*, void * *user_ptr*, vx_enum *usage*, vx_enum *user_memory_type*);

number_of_dims: Number of patch dimensions.
view_start: Array of patch start points in each dimension.
view_end: Array of patch end points in each dimension.
user_stride: Array of user memory strides in each dimension.
user_ptr: Memory location address where to store or get the data.
usage: VX_{READ, WRITE}_ONLY.
user_memory_type: VX_MEMORY_TYPE_{NONE, HOST}

Create an array of images into the multi-dimension data.

x_object_array **vxCreateImageObjectArrayFromTensor**(vx_tensor *tensor*, const vx_rectangle_t * *rect*, vx_size *array_size*, vx_size *jump*, vx_df_image *image_format*);

tensor: Must be a 3D tensor.
rect: Image coordinates within tensor data.
array_size: Number of images to extract.
stride: Delta between two images in the array.
image_format: The requested image format.

Create an opaque reference to a tensor object buffer.

vx_tensor **vxCreateTensor**(vx_context *context*, vx_size *number_of_dims*, const vx_size *dims*, vx_enum *data_type*, vx_int8 *fixed_point_position*);

vx_tensor **vxCreateVirtualTensor**(vx_graph *graph*, vx_size *number_of_dims*, vx_size **dims*, vx_enum *data_type*, vx_int8 *fixed_point_position*);

dims: Dimensions sizes in elements.
data_type: The vx_type_t that represents the data type of the tensor data elements.
fixed_point_position: Specifies the fixed point position when the input element type is integer. If 0, calculations are performed in integer math.

(Continued on next page) ▶

◀ Tensor Objects (cont.)

Create a tensor object from another given a view.

vx_tensor **vxCreateTensorFromView**(vx_tensor *tensor*, vx_size *number_of_dims*, const vx_size * *view_start*, const vx_size * *view_end*);

number_of_dimensions: Number of dimensions in the view.
view_start: View start coordinates.
view_end: View end coordinates.

vx_status **vxQueryTensor**(vx_tensor *tensor*, vx_enum *attribute*, void * *ptr*, vx_size *size*);

attribute: The attribute to query from vx_tensor_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

vx_status **vxReleaseTensor**(vx_tensor * *tensor*);

Advanced Framework

Node Callbacks [3.91]

Assign a callback to a node.

vx_status **vxAssignNodeCallback**(vx_node *node*, vx_nodecomplete_f *callback*);

callback: Callback function pointer.

The following callback is used by **vxAssignNodeCallback**.

typedef vxAction(***vx_nodecomplete_f**)(vx_node *node*)

Retrieve the current node callback function pointer.

vx_nodecomplete_f **vxRetrieveNodeCallback**(vx_node *node*);

Log [3.93]

Add a line to the log.

void **vxAddLogEntry**(vx_reference *ref*, vx_status *status*, const char **message*, ...);

ref: The reference to add the log entry against.
status: The status code. VX_SUCCESS status entries are ignored.
message: The human readable message to add to the log.

Register a callback facility to receive error logs.

void **vxRegisterLogCallback**(vx_context *context*, vx_log_callback_f *callback*, vx_bool *reentrant*);

callback: The callback function or NULL.
reentrant: Boolean reentrancy flag indicating whether the callback may be entered from multiple simultaneous tasks or threads.

The following callback is used by **vxRegisterLogCallback**.

typedef void(***vx_log_callback_f**)(vx_context *context*, vx_reference *ref*, vx_status *status*, vx_char *string*[ ]);

ref: The reference to add the log entry against.
status: The status code.

Hints [3.94]

Provide a generic API to give platform-specific hints.

vx_status **vxHint**(vx_reference *reference*, vx_enum *hint*, const void **data*, vx_size *data_size*);

reference: The reference to the object to hint at.
hint: • See vx_hint_e.
data: Optional vendor specific data.
data_size: Size of the data structure *data*.

Directives [3.95]

A generic API to give platform-specific directives.

vx_status **vxDirective**(vx_reference *reference*, vx_enum *directive*);

reference: The reference to the object to set the directive on.
directive: • The directive to set from vx_directive_e.

User Kernels [3.96]

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*);

index: The index of the parameter to add.
dir: VX_INPUT or VX_OUTPUT.
data_type: • Type of parameter from vx_type_e.
state: • Parameter state from vx_parameter_state_e.

Threshold Objects [3.78]

Attributes: enum vx_threshold_attribute_e: VX_THRESHOLD_{TYPE, TYPE_BINARY, TYPE_RANGE}, VX_THRESHOLD_{INPUT, OUTPUT}_FORMAT

Copy the true and false output values from/into threshold object.

vx_status **vxCopyThresholdOutput**(vx_threshold *thresh*, vx_pixel_value_t * *true_value_ptr*, vx_pixel_value_t * *false_value_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

true_value_ptr: Address of memory location for the true output value.
false_value_ptr: Address of memory location for the false output value.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}

Copy type VX_THRESHOLD_TYPE_RANGE thresholding values.

vx_status **vxCopyThresholdRange**(vx_threshold *thresh*, vx_pixel_value_t * lower_value_ptr, vx_pixel_value_t * *upper_value_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

lower_value_ptr: Memory location address for lower thresh value.
upper_value_ptr: Memory location address for upper thresh value.
usage: VX_{READ, WRITE}_ONLY.

Allows application to copy the thresholding value from or into a threshold object with type VX_THRESHOLD_TYPE_BINARY.

vx_status **vxCopyThresholdValue**(vx_threshold *thresh*, vx_pixel_value_t * *value_ptr*, vx_enum usage, vx_enum *user_mem_type*);

value_ptr: Address of memory location for the thresholding value.
usage: VX_{READ, WRITE}_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}

Create a [virtual] threshold object and returns a reference to it.

vx_threshold **vxCreateThresholdForImage**(vx_context *context*, vx_enum *thresh_type*, vx_df_image *input_format*, vx_df_image *output_format*);

vx_threshold **vxCreateVirtualThresholdForImage**(*vx_graph* graph, vx_enum *thresh_type*, vx_df_image *input_format*, vx_df_image *output_format*);

thresh_type: The type of thresholding operation, VX_THRESHOLD_TYPE_{BINARY, RANGE}
input_format: The format of images that will be used as input of the thresholding operation.
output_format: The format of images that will be generated by the thresholding operation.

vx_status **vxQueryThreshold** (vx_threshold *thresh*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: The attribute to modify from vx_threshold_attribute_e.
ptr: The location at which to store the result.
size: The size of the container pointed to by *ptr*.

vx_status **vxReleaseThreshold** (vx_threshold **thresh*);

vx_status **vxSetThresholdAttribute** (vx_threshold *thresh*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: The attribute to modify from vx_threshold_attribute_e.
ptr: The pointer to the value to which to set the attribute.
size: The size of the data pointed to by *ptr*.

Add custom kernels to the known kernel database.

vx_kernel **vxAddUserKernel**(vx_context *context*, const vx_char *name*[VX_MAX_KERNEL_NAME], vx_enum *enumeration*, vx_kernel_f *func_ptr*, vx_uint32 *numParams*, vx_kernel_validate_f *validate*, vx_kernel_initialize_f *init*, vx_kernel_deinitialize_f *deinit*);

name: The string to use to match the kernel.
enumeration: Enumerated value of the kernel to be used by clients.
func_ptr: The process-local function pointer to be invoked.
numParams: The number of parameters for this kernel.
validate: The pointer to vx_kernel_validate_f, which validates parameters to this kernel.
init, deinit: The kernel {initialization, deinitialization} function.

The following types are used by **vxAddUserKernel**.

typedef vx_status(***vx_kernel_f**)(vx_node *node*, const vx_reference * *parameters*, vx_uint32 *num*)

typedef vx_status(***vx_kernel_initialize_f**)(vx_node *node*, const vx_reference * *parameters*, vx_uint32 *num*)

typedef vx_status(***vx_kernel_deinitialize_f**)(vx_node *node*, const vx_reference * *parameters*, vx_uint32 *num*)

typedef vx_status(***vx_kernel_validate_f**)(vx_node *node*, const vx_reference * *parameters*, vx_uint32 *num*, vx_meta_format *metas*[])

parameters: The array of parameter references.
num: The number of parameters.
metas: A pointer to a pre-allocated array of structure references that the system holds.

Allocate/register user-defined kernel enumeration to a context.

vx_status **vxAllocateUserKernelId**(vx_context *context*, vx_enum **pKernelEnumId*);

pKernelEnumId: The pointer to return vx_enum for user-defined kernel.

Allocate/register user-defined kernel library ID to a context.

vx_status **vxAllocateUserKernelLibraryId**(vx_context *context*, vx_enum **pLibraryId*);

pLibraryId: The pointer to vx_enum for user-kernel libraryId.

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*, const vx_char **module*);

module: The short name of the module to load.

Remove a vx_kernel from the vx_context.

vx_status **vxRemoveKernel**(vx_kernel *kernel*);

Set kernel attributes.

vx_status **vxSetKernelAttribute**(vx_kernel *kernel*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: The attribute to set, from vx_kernel_attribute_e. (VX_KERNEL_{PARAMETERS, NAME, ENUM, LOCAL_DATA_SIZE})
ptr: Pointer to the attribute.
size: The size in bytes of the container to which *ptr* points.

Set the attributes of a vx_meta_format object.

vx_status **vxSetMetaFormatAttribute**(vx_meta_format *meta*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

meta: The reference to the vx_meta_format struct to set.
attribute: Use the subset of attributes that define the meta data of this object or attributes from vx_meta_format.
ptr: The input pointer of the value to set on the meta format object.
size: The size in bytes of the container to which *ptr* points.

Set a meta format object from an exemplar data object reference.

vx_status **vxSetMetaFormatFromReference**(vx_meta_format *meta*, vx_reference *exemplar*);

meta: The meta format object to set.
exemplar: The exemplar data object.

Unload one or more kernels from the module.

vx_status **vxUnloadKernels**(vx_context *context*, const vx_char **module*);

module: The short name of the module to unload.

Graph Parameters [3.97]

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*);

value: The reference to set to the parameter.

Macros [3.63.3]

Define calling convention for OpenVX API.
#define VX_API_CALL

Define the manner to combine the Vendor and Object IDs to get the base value of enumeration.
#define VX_ATTRIBUTE_BASE(vendor, object) (((vendor) << 20) j (object << 8))

An object's attribute ID is within the range of [0,2^8 — 1] (inclusive).
#define VX_ATTRIBUTE_ID_MASK (0x000000FF)

Define calling convention for user callbacks.
#define VX_CALLBACK

Convert a set of four chars into a uint32_t container of a VX_DF_IMAGE code.
#define VX_DF_IMAGE(a, b, c, d) ((a) | (b << 8) | (c << 16) | (d << 24))

Define the manner to combine the Vendor and Object IDs to get the base value of enumeration.
#define VX_ENUM_BASE(vendor, id) (((vendor) << 20) | (id << 12))

A generic enumeration list can have values between [0,2^{12} — 1] (inclusive).
#define VX_ENUM_MASK (0x00000FFF)

A macro to extract the enum type from an enumerated value.
#define VX_ENUM_TYPE(e) (((vx_uint32)e & VX_ENUM_TYPE_MASK) >> 12)

A type of enumeration. The valid range is between [0,2^8 — 1] (inclusive).
#define VX_ENUM_TYPE_MASK (0x000FF000)

Use to aid in debugging values in OpenVX.
#define VX_FMT_REF "%p"

Use to aid in debugging values in OpenVX.
#define VX_FMT_SIZE "%zu"

Define the manner to combine the Vendor and Library IDs to get the base value of enumeration.
#define VX_KERNEL_BASE(vendor, lib) (((vendor) << 20) j (lib << 12))

An individual kernel in a library has its own unique ID within [0,2^{12} — 1] (inclusive).
#define VX_KERNEL_MASK (0x00000FFF)

A macro to extract the kernel library enumeration from a enumerated kernel value.
#define VX_LIBRARY(e) (((vx_uint32)e & VX_LIBRARY_MASK) >> 12)

A set of vision kernels with its own ID supplied by a vendor. The ID range is [0,2^8 — 1] (inclusive).
#define VX_LIBRARY_MASK (0x000FF000)

Define the length of a message buffer to copy from the log, including the trailing zero.
#define VX_MAX_LOG_MESSAGE_LEN (1024)

Use to indicate the 1:1 ratio in Q22.10 format.
#define VX_SCALE_UNITY (1024u)

A macro to extract the type from an enumerated attribute value.
#define VX_TYPE(e) (((vx_uint32)e & VX_TYPE_MASK) >> 8)

Remove scalar/object type from attribute. It is 3 nibbles in size, contained between bytes 2 and 3.
#define VX_TYPE_MASK (0x000FFF00)

A macro to extract the vendor ID from the enumerated value.
#define VX_VENDOR(e) (((vx_uint32)e & VX_VENDOR_MASK) >> 20)

Vendor IDs are 2 nibbles in size and are located in the upper byte of the 4 bytes of an enumeration.
#define VX_VENDOR_MASK (0xFFF00000)

Define the predefined version number for 1.0.
#define VX_VERSION_1_0 (VX_VERSION_MAJOR(1) | VX_VERSION_MINOR(0))

Define the predefined version number for 1.1.
#define VX_VERSION_1_1 (VX_VERSION_MAJOR(1) | VX_VERSION_MINOR(1))

Define the major version number macro.
#define VX_VERSION_MAJOR(x) ((x & 0xFF) << 8)

Define the minor version number macro.
#define VX_VERSION_MINOR(x) ((x & 0xFF) << 0)

Define the OpenVX Version Number.
#define VX_VERSION VX_VERSION_1_1

Enumerators

vx_enum_e

VX_ENUM_ACCESSOR	vx_accessor_e
VX_ENUM_ACTION	vx_action_e
VX_ENUM_BORDER	vx_border_e
VX_ENUM_BORDER_POLICY	vx_border_policy_e
VX_ENUM_CHANNEL	vx_channel_e
VX_ENUM_CLASSIFIER_MODEL	vx_classifier_model_e
VX_ENUM_COLOR_RANGE	vx_channel_range_e
VX_ENUM_COLOR_SPACE	vx_color_space_e
VX_ENUM_COMP_METRIC	vx_comp_metric_e
VX_ENUM_COMPARISON	vx_comparison_e
VX_ENUM_CONVERT_POLICY	vx_convert_policy_e
VX_ENUM_DIRECTION	vx_direction_e
VX_ENUM_DIRECTIVE	vx_directive_e
VX_ENUM_GRAPH_STATE	vx_graph_state_e
VX_ENUM_HINT	vx_hint_e
VX_ENUM_INTERPOLATION	vx_interpolation_type_e
VX_ENUM_LBP_FORMAT	vx_lbp_format_e
VX_ENUM_MEMORY_TYPE	vx_memory_type_e
VX_ENUM_NONLINEAR	vx_non_linear_filter_e
VX_ENUM_NORM_TYPE	vx_norm_type_e
VX_ENUM_OVERFLOW	
VX_ENUM_PARAMETER_STATE	vx_parameter_state_e
VX_ENUM_PATTERN	vx_pattern_e
VX_ENUM_ROUND_POLICY	vx_round_policy_e
VX_ENUM_TARGET	vx_target_e
VX_ENUM_TERM_CRITERIA	vx_termination_criteria_e
VX_ENUM_THRESHOLD_TYPE	vx_threshold_type_e

vx_accessor_e
VX_READ_ONLY
VX_WRITE_ONLY
VX_READ_AND_WRITE

vx_action_e
VX_ACTION_CONTINUE
VX_ACTION_ABANDON

vx_border_e
VX_BORDER_UNDEFINED
VX_BORDER_ CONSTANT
VX_BORDER_ REPLICATE

vx_border_policy_e
VX_BORDER_POLICY_DEFAULT_TO_UNDEFINED
VX_BORDER_POLICY_RETURN_ERROR

vx_channel_e
VX_CHANNEL_{0, 1, 2, 3}
VX_CHANNEL_{R, G, B, A}
VX_CHANNEL_{Y, U , V}

vx_channel_range_e
VX_CHANNEL_RANGE_FULL
VX_CHANNEL_RANGE_ RESTRICTED

vx_color_space_e
VX_COLOR_SPACE_NONE
VX_COLOR_SPACE_ DEFAULT
VX_COLOR_SPACE_BT601_{525, 625}
VX_COLOR_SPACE_BT709

vx_comp_metric_e
VX_COMPARE_HAMMING
VX_COMPARE_L1
VX_COMPARE_L2
VX_COMPARE_CCORR
VX_COMPARE_L2_NORM
VX_COMPARE_CCORR_NORM

vx_convert_policy_e
VX_CONVERT_POLICY_WRAP
VX_CONVERT_POLICY_SATURATE

vx_df_image_e
VX_DF_IMAGE_VIRT
VX_DF_IMAGE_{RGB, RGBX}
VX_DF_IMAGE_{NV12, NV21}
VX_DF_IMAGE_{UYVY, YUYV, IYUV, YUV4}
VX_DF_IMAGE_{U8, U16, S16, U32, S32}

vx_direction_e
VX_{INPUT, OUTPUT}
VX_BIDIRECTIONAL

vx_directive_e
VX_DIRECTIVE_{DISABLE, ENABLE}_LOGGING
VX_DIRECTIVE_{DISABLE, ENABLE}_PERFORMANCE

vx_graph_state_e
VX_GRAPH_STATE_{UNVERIFIED, VERIFIED}
VX_GRAPH_STATE_RUNNING
VX_GRAPH_STATE_ABANDONED
VX_GRAPH_STATE_COMPLETED

vx_hint_e
VX_HINT_PERFORMANCE_DEFAULT
VX_HINT_PERFORMANCE_LOW_POWER
VX_HINT_PERFORMANCE_HIGH_SPEED

vx_image_attribute_e
VX_IMAGE_WIDTH
VX_IMAGE_HEIGHT
VX_IMAGE_FORMAT
VX_IMAGE_PLANES
VX_IMAGE_SPACE
VX_IMAGE_RANGE
VX_IMAGE_MEMORY_TYPE
VX_IMAGE_IS_UNIFORM
VX_IMAGE_UNIFORM_VALUE

vx_interpolation_type_e
VX_INTERPOLATION_NEAREST_NEIGHBOR
VX_INTERPOLATION_BILINEAR
VX_INTERPOLATION_AREA

vx_kernel_e
VX_INPUT
VX_OUTPUT
VX_BIDIRECTIONAL

vx_map_flag_e
VX_NOGAP_X

vx_memory_type_e
VX_MEMORY_TYPE_NONE
VX_MEMORY_TYPE_HOST

vx_non_linear_filter_e
VX_NONLINEAR_FILTER_MEDIAN
VX_NONLINEAR_FILTER_MIN
VX_NONLINEAR_FILTER_MAX

(Continued on next page) ▶

◀ Enumerators (cont.)

vx_norm_type_e
VX_NORM_{L1, L2}

vx_parameter_state_e
VX_PARAMETER_STATE_REQUIRED
VX_PARAMETER_STATE_OPTIONAL

vx_pattern_e
VX_PATTERN_BOX
VX_PATTERN_CROSS
VX_PATTERN_DISK
VX_PATTERN_OTHER

vx_round_policy_e
VX_ROUND_POLICY_TO_ZERO
VX_ROUND_POLICY_TO_NEAREST_EVEN

vx_scalar_operation_e
VX_SCALAR_OP_AND
VX_SCALAR_OP_OR
VX_SCALAR_OP_XOR
VX_SCALAR_OP_NAND
VX_SCALAR_OP_EQUAL
VX_SCALAR_OP_NOTEQUAL
VX_SCALAR_OP_LESS
VX_SCALAR_OP_LESSEQ
VX_SCALAR_OP_GREATER
VX_SCALAR_OP_GREATEREQ
VX_SCALAR_OP_ADD
VX_SCALAR_OP_SUBTRACT
VX_SCALAR_OP_MULTIPLY
VX_SCALAR_OP_DIVIDE
VX_SCALAR_OP_MODULUS
VX_SCALAR_OP_{MIN, MAX}

vx_status_e
VX_STATUS_MIN
VX_ERROR_REFERENCE_NONZERO
VX_ERROR_MULTIPLE_WRITERS
VX_ERROR_GRAPH_ABANDONED
VX_ERROR_GRAPH_SCHEDULED
VX_ERROR_INVALID_SCOPE
VX_ERROR_INVALID_NODE
VX_ERROR_INVALID_GRAPH
VX_ERROR_INVALID_TYPE
VX_ERROR_INVALID_VALUE
VX_ERROR_INVALID_DIMENSION
VX_ERROR_INVALID_FORMAT
VX_ERROR_INVALID_LINK

vx_status_e (continued)
VX_ERROR_INVALID_REFERENCE
VX_ERROR_INVALID_MODULE
VX_ERROR_INVALID_PARAMETERS
VX_ERROR_OPTIMIZED_AWAY
VX_ERROR_NO_MEMORY
VX_ERROR_NO_RESOURCES
VX_ERROR_NOT_COMPATIBLE
VX_ERROR_NOT_ALLOCATED
VX_ERROR_NOT_SUFFICIENT
VX_ERROR_NOT_SUPPORTED
VX_ERROR_NOT_IMPLEMENTED
VX_FAILURE
VX_SUCCESS

vx_target_e
VX_TARGET_ANY
VX_TARGET_ STRING
VX_TARGET_ VENDOR_BEGIN

vx_termination_criteria_e
VX_TERM_CRITERIA_ITERATIONS
VX_TERM_CRITERIA_EPSILON
VX_TERM_CRITERIA_BOTH

vx_threshold_type_e
VX_THRESHOLD_TYPE_BINARY
VX_THRESHOLD_TYPE_RANGE

vx_type_e

VX_TYPE_ARRAY	vx_array
VX_TYPE_BOOL	vx_bool
VX_TYPE_CHAR	vx_char
VX_TYPE_CONTEXT	vx_context
VX_TYPE_CONVOLUTION	vx_convolution
VX_TYPE_COORDINATES2D	vx_coordinates2d_t
VX_TYPE_COORDINATES2DF	vx_coordinates2df_t
VX_TYPE_COORDINATES3D	vx_coordinates3d_t
VX_TYPE_DELAY	vx_delay
VX_TYPE_DF_IMAGE	vx_df_image
VX_TYPE_DISTRIBUTION	vx_distribution
VX_TYPE_ENUM	vx_enum
VX_TYPE_ERROR	Error object with no type
VX_TYPE_FLOAT32	vx_float32
VX_TYPE_FLOAT64	vx_float64
VX_TYPE_GRAPH	vx_graph
VX_TYPE_IMAGE	vx_image
VX_TYPE_INT16	vx_int16
VX_TYPE_INT32	vx_int32
VX_TYPE_INT64	vx_int64
VX_TYPE_INT8	vx_int8
VX_TYPE_INVALID	Invalid type value

(more ↗)

vx_type_e (continued)

VX_TYPE_KERNEL	vx_kernel
VX_TYPE_KEYPOINT	vx_keypoint_t
VX_TYPE_LUT	vx_lut
VX_TYPE_MATRIX	vx_matrix
VX_TYPE_META_FORMAT	vx_meta_format
VX_TYPE_NODE	vx_node
VX_TYPE_OBJECT_ARRAY	vx_object_array
VX_TYPE_PARAMETER	vx_parameter
VX_TYPE_PYRAMID	vx_pyramid
VX_TYPE_RECTANGLE	vx_rectangle_t
VX_TYPE_REFERENCE	vx_reference
VX_TYPE_REMAP	vx_remap
VX_TYPE_SCALAR	vx_scalar
VX_TYPE_SIZE	vx_size
VX_TYPE_TENSOR	vx_tensor
VX_TYPE_THRESHOLD	vx_threshold
VX_TYPE_UINT16	vx_uint16
VX_TYPE_UINT32	vx_uint32
VX_TYPE_UINT64	vx_uint64
VX_TYPE_UINT8	vx_uint8

vx_vendor_id_e
VX_ID_DEFAULT
VX_ID_USER
VX_ID_AMD
VX_ID_ARM
VX_ID_AXIS, BDTI
VX_ID_BROADCOM
VX_ID_CADENCE
VX_ID_CEVA
VX_ID_FREESCALE
VX_ID_HUAWEI
VX_ID_IMAGINATION
VX_ID_INTEL
VX_ID_ITSEEZ
VX_ID_KHRONOS
VX_ID_MARVELL
VX_ID_MEDIATEK
VX_ID_MOVIDIUS
VX_ID_NVIDIA
VX_ID_NXP
VX_ID_QUALCOMM
VX_ID_RENESAS
VX_ID_SAMSUNG
VX_ID_ST
VX_ID_SYNOPSYS
VX_ID_TI
VX_ID_VIDEANTIS
VX_ID_VIVANTE
VX_ID_XILINX

Notes

Import and Export Extension

The import and export extension provides a way of importing and exporting pre-verified graphs or other objects in vendor-specific formats. For more about this extension see khronos.org/xxxxxx

Macros

The application will create the object before import.

```
#define VX_IX_USE_APPLICATION_CREATE \
    (VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_IX_USE)
    + 0x0)
```

Data values are exported and restored upon import.

```
#define VX_IX_USE_EXPORT_VALUES \
    (VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_IX_USE)
    + 0x1)
```

Data values are not exported.

```
#define VX_IX_USE_NO_EXPORT_VALUES \
    (VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_IX_USE)
    + 0x2)
```

The Object Type Enumeration for import.

```
#define VX_TYPE_IMPORT_0x814
```

Functions

Exports selected objects to memory in a vendor-specific format.

vx_status **vxExportObjectsToMemory**(vx_context *context*, vx_size *numrefs*, const vx_reference **refs*, const vx_enum **uses*, const vx_uint8 ***ptr*, vx_size **length*);

numrefs: Number of references to export.
refs: Array of length *numrefs* populated with the references to export.
uses: Array of length *numrefs* containing values as described above.
ptr: Returns pointer to binary buffer.
length: Number of bytes at **ptr*.

Get a reference from the import object by name.

vx_reference **vxGetImportReferenceByName**(vx_import *import*, const vx_char * *name*);

import: The import object in which to find the name.
name: Points to a string with the name to find.

Imports objects into a context from a vendor-specific format in memory.

vx_import **vxImportObjectsFromMemory**(vx_context *context*, vx_size *numrefs*, vx_reference **refs*, const vx_enum **uses*, const vx_uint8 **ptr*, vx_size *length*);

numrefs: Number of references to import. Must match export.
refs: References imported or application-created data.
uses: How to import the references, must match export values.
ptr: Pointer to binary buffer containing a valid binary export.
length: Number of bytes at **ptr* describing the length of the export.

Releases memory allocated for a binary export.

vx_status **vxReleaseExportedMemory**(vx_context *context*, const vx_uint8 ***ptr*);

ptr: A pointer previously set by calling **vxExportObjectsToMemory**.

Releases an import object when no longer required.

vx_status **vxReleaseImport**(vx_import **import*);

import: A pointer to the reference to the import object.

Neural Networks Extension

The neural networks extension enables execution and integration of deep neural networks in OpenVX processing graphs.
The extension works with the vx tensor object, which is a multidimensional array with an arbitrary number of dimensions and can represent all varieties of data typically used in a deep neural network.

Kernel Names

When using **vxGetKernelByName** with this extension, the following strings are used in the *name* parameter to specify the neural networks extension kernel names.

org.khronos.nn extension.convolution layer
org.khronos.nn extension.fully connected layer
org.khronos.nn extension.pooling layer
org.khronos.nn extension.softmax layer
org.khronos.nn extension.normalization layer
org.khronos.nn extension.activation layer
org.khronos.nn extension.roi pooling layer
org.khronos.nn extension.deconvolution layer

Data Structures

Input parameters for a convolution operation.

struct vx_nn_convolution_params_t

(vx_uint32) ***dilation_x/y:*** Inflate the kernel by inserting zeros between the kernel elements in the x/y direction.
(vx_enum) ***down_scale_size_rounding:*** Rounding method for calculating output dimensions from vx_nn_rounding_type_e.
(vx_enum) ***overflow_policy:*** A VX_TYPE_ENUM from vx_convert_policy_e.
(vx_uint32) ***pad_x/y:*** The number of elements added at each side in the x/y dimension of the input.
(vx_enum) ***rounding_policy:*** A VX_TYPE_ENUM from vx_round_policy_e.

Input parameters for a deconvolution operation.

struct vx_nn_deconvolution_params_t

(vx_uint32) ***a_x/y:*** A user-specified quantity used to distinguish between the $upscale_{x/y}$ different possible output sizes.
(vx_enum) ***overflow_policy:*** A VX_TYPE_ENUM from vx_convert_policy_e.
(vx_uint32) ***pad_x/y:*** The number of elements added at each side in the x/y dimension of the input.
(vx_enum) ***rounding_policy:*** A VX_TYPE_ENUM from vx_round_policy_e.

Enumerators

The following enumerators are enabled with this extension.

vx_kernel_nn_ext_e
VX_KERNEL_KHR_CONVOLUTION_LAYER

vx_nn_activation_function_e
VX_NN_ACTIVATION_LOGISTIC
VX_NN_ACTIVATION_HYPERBOLIC_TAN
VX_NN_ACTIVATION_{RELU, BRELU, SOFTRELU}
VX_NN_ACTIVATION_{ABS, SQUARE, SQRT, LINEAR}

vx_nn_enum_e_f
VX_ENUM_NN_ROUNDING_TYPE
VX_ENUM_NN_POOLING_TYPE
VX_ENUM_NN_NORMALIZATION_TYPE
VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE

vx_nn_norm_type_e_f
VX_NN_NORMALIZATION_SAME_MAP
VX_NN_NORMALIZATION_ACROSS_MAPS

vx_nn_pooling_type_e_f
VX_NN_POOLING_MAX
VX_NN_POOLING_AVG

vx_nn_rounding_type_e_f
VX_NN_DS_SIZE_ROUNDING_FLOOR
VX_NN_DS_SIZE_ROUNDING_CEILING

Functions

Creates a convolutional network activation layer node.

vx_node **vxActivationLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_enum *function*, vx_float32 *a*, vx_float32 *b*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.
function: Non-linear function from vx.nn.activation.function.e.
a, b: Function parameters a/b. Must be positive.

Creates a convolutional network [de]convolution layer node.

vx_node **vxConvolutionLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *weights*, vx_tensor *biases*, cconst vx_nn_convolution_params_t **convolution_params*, vx_size *size_of_convolution_params*, vx_tensor *outputs*);

vx_node **vxDeconvolutionLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *weights*, vx_tensor *biases*, const vx_nn_convolution_params_t **deconvolution_params*, vx_size *size_of_deconv_params*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.
weights: A 4d tensor with dimensions:
convolution: [kernel x, kernel y, #IFM, #OFM]
deconvolution: [width, height, OFM, IFM]
biases: Optional, ignored if NULL.
[de]convolution_params: Pointer to parameters of type vx_nn_[de]convolution_params_t.
size_of_{convolution, deconv}_params: Size in bytes of convolution/deconvolution params.

Creates a fully connected convolutional network layer node.

vx_node **vxFullyConnectedLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *weights*, vx_tensor *biases*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.
weights: 2d tensor with dimensions [#IFM, #OFM].
biases: Optional, ignored if NULL.
overflow_policy: VX_TYPE_ENUM of vx_convert_policy_e_ enumeration.
rounding_policy: VX_TYPE_ENUM of vx_convert_policy_e_enumeration.

Creates a convolutional network normalization layer node.

vx_node **vxNormalizationLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_enum *type*, vx_size *normalization_size*, vx float32 *alpha*, vx float32 *beta*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.
normalization_size: Number of elements to normalize across.
alpha: Alpha parameter in normalization equation. Must be positive.
beta: Beta parameter in normalization equation. Must be positive.

Creates a convolutional network pooling layer node.

vx_node **vxPoolingLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_enum *pooling_type*, vx_size *pooling_size_x*, vx_size *pooling_size_y*, vx_size *pooling_padding_x*, vx_size *pooling_padding_y*, vx_enum *rounding*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.
pooling_type: Either max or average pooling from vx_nn_pooling_type_e.
pooling_size_{x,y}: Size of the pooling region.
pooling_padding_{x,y}: Padding size.

Creates a convolutional network ROI pooling node.

vx_node **vxROIPoolingLayer** (vx_graph *graph*, vx_tensor *input_data*, vx_tensor *input_rois*, vx_enum *pool_type*, vx_tensor *output_arr*);

input_data: The input tensor data.
inputs_rois: The roi array tensor with dims: [4, roi count, #batches].
pool_type: VX_NN_POOLING_MAX
output_arr: The output tensor.

Creates a convolutional network softmax layer node.

vx_node **vxSoftmaxLayer** (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *outputs*);

inputs, outputs: The input/output tensor data.

Host Memory Data Object Access Patterns: Examples

Matrix Access Example [2.15.1]

```
const vx_size columns = 3;
const vx_size rows = 4;
vx_matrix matrix = vxCreateMatrix(context, VX_TYPE_FLOAT32, columns, rows);
vx_status status = vxGetStatus((vx_reference)matrix);
if (status == VX_SUCCESS) {
    vx_int32 j, i;
    vx_float32 *mat = (vx_float32 *)malloc(rows*columns*sizeof(vx_float32));
    if (vxReadMatrix(matrix, mat) == VX_SUCCESS) {
        for (j = 0; j < (vx_int32)rows; j++) {
            for (i = 0; i < (vx_int32)columns; i++) {
                mat[j*columns + i] = (vx_float32)rand()/(vx_float32)RAND_MAX;
            }
        }
        vxCopyMatrix(matrix, mat, VX_WRITE_ONLY, VX_MEMORY_TYPE_HOST);
    }
    free(mat);
}
```

Array Access Examples [2.15.3]

Arrays require a single value, the stride, instead of the entire addressing structure images need.

```
vx_size i, stride = 0;
void *base = NULL;
vx_map_id map_id;

    /* access entire array at once */
    vxMapArrayRange(array, 0, num_items, &stride, &base, VX_READ_AND_WRITE,
                    VX_MEMORY_TYPE_HOST, 0);
    for (i = 0; i < num_items; i++) {
        vxArrayItem(mystruct, base, i, stride).some_uint += i;
        vxArrayItem(mystruct, base, i, stride).some_double = 3.14f;
    }
    vxUnmapArrayRange(array, map_id);
```

Map/Unmap pairs can also be called on a range of items which can be addressed directly:

```
vx_size start_index = 5;  /* included */
vx_size end_index = 15;  /* excluded */
vx_uint8 *ptr = NULL;

/* access an array item range */
vxMapArrayRange(array, start_index, end_index, &map_id, &stride, (void **)&ptr,
                VX_READ_AND_WRITE, VX_MEMORY_TYPE_HOST, 0);
for (i = start_index; i < end_index; i++) {
    mystruct *myptr = (mystruct *)ptr;
    myptr->some_uint += 1;
    myptr->some_double = 3.14f;
    ptr += stride;
}
vxUnmapArrayRange(array, map_id);
```

Image Access Example [2.15.2]

```
vx_status status = VX_SUCCESS;
void *base_ptr = NULL;
vx_uint32 width = 640, height = 480, plane = 0;
vx_image image = vxCreateImage(context, width, height, VX_DF_IMAGE_U8);
vx_rectangle_t rect;
vx_imagepatch_addressing_t addr;
vx_map_id map_id;

rect.start_x = rect.start_y = 0;
rect.end_x = rect.end_y = PATCH_DIM;

status = vxMapImagePatch(image, &rect, plane, &addr, &base_ptr, VX_READ_AND_WRITE,
                         VX_MEMORY_TYPE_HOST, 0);
if (status == VX_SUCCESS) {
    vx_uint32 x,y,i;
    vx_uint8 pixel = 0;

    /* a couple addressing options */

    /* use linear addressing function/macro */
    for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
        vx_uint8 *ptr2 = (vx_uint8 *)vxFormatImagePatchAddress1d(base_ptr, i, &addr);
        *ptr2 = pixel;
    }
    /* 2d addressing option for subsampled planes, step is > 1 */
    for (y = 0; y < addr.dim_y; y+=addr.step_y) {
        for (x = 0; x < addr.dim_x; x+=addr.step_x) {
            vx_uint8 *ptr2 = (vx_uint8 *)vxFormatImagePatchAddress2d(base_ptr, x, y,
                    &addr);
            *ptr2 = pixel;
        }
    }
    /* more efficient direct addressing for subsampled planes, step is > 1 */
    vx_uint8 *line_ptr = (vx_uint8 *)base_ptr;
    for (y = 0; y < addr.dim_y; y+=addr.step_y) {
        vx_uint8 *pixel_ptr = line_ptr;
        for (x = 0; x < addr.dim_x; x+=addr.step_x) {
            *pixel_ptr = pixel;
            pixel_ptr += addr.stride_x;
        }
        line_ptr += addr.stride_y;
    }
    /* this commits the data back to the image */
    status = vxUnmapImagePatch(image, map_id);
}
vxReleaseImage(&image);
```

Notes