



The **OpenVX™** Neural Network Extension

Version 1.2

Document Revision: 1b0bfd8

Generated on Fri Apr 28 2017 02:15:33

Khronos Vision Working Group

Editor: Schwartz Tomer
Editor: Hagog Mostafa

Copyright ©2017 The Khronos Group Inc.

Copyright ©2017 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast, or otherwise exploited in any manner without the express prior written permission of the Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part. Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be reformatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group website should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials. SAMPLE CODE and EXAMPLES, as identified herein, are expressly depicted herein with a "grey" watermark and are included for illustrative purposes only and are expressly outside of the Scope as defined in Attachment A - Khronos Group Intellectual Property (IP) Rights Policy of the Khronos Group Membership Agreement. A Member or Promoter Member shall have no obligation to grant any licenses under any Necessary Patent Claims covering SAMPLE CODE and EXAMPLES.

Contents

1	Neural Network Extension	2
1.1	Acknowledgements	2
1.2	Background and Terminology	2
1.3	Introduction	3
1.4	Weights/Biasses Setting	3
1.5	Kernel names	4
1.6	8-bit extension and 16-bit extension	4
2	Module Documentation	5
2.1	Extension: Deep Convolutional Networks API	5
2.1.1	Detailed Description	7
2.1.2	Data Structure Documentation	7
	struct vx_nn_convolution_params_t	7
	struct vx_nn_deconvolution_params_t	7
	struct vx_nn_roi_pool_params_t	8
2.1.3	Enumeration Type Documentation	9
	vx_kernel_nn_ext_e	9
	vx_nn_activation_function_e	9
	vx_nn_norm_type_e	10
	vx_nn_pooling_type_e	10
	vx_nn_rounding_type_e	10
2.1.4	Function Documentation	10
	vxActivationLayer	10
	vxConvolutionLayer	11
	vxDeconvolutionLayer	12
	vxFullyConnectedLayer	13
	vxNormalizationLayer	13
	vxPoolingLayer	15
	vxROIPoolingLayer	16
	vxSoftmaxLayer	16

Chapter 1

Neural Network Extension

1.1 Acknowledgements

This specification would not be possible without the contributions from this partial list of the following individuals from the Khronos Working Group and the companies that they represented at the time:

- Frank Brill - Cadence Design Systems
- Kari Pulli - Intel
- Tomer Schwartz - Intel
- Mostafa Hagog - Intel
- Chuck Pilkington - synopsis
- Thierry Lepley - Cadence Design Systems
- Radha Giduthuri - AMD
- Jesse Villarreal - TI
- Victor Eruhimov - Itseez3D
- Xin Wang - Verisilicon

1.2 Background and Terminology

Deep Learning using Neural Networks techniques is being increasingly used to perform vision classification and recognition tasks. Deep Neural Networks have significantly improved image recognition capabilities over previous technologies. The Neural Network extension for OpenVX is intended to enable the implementation of Deep Neural Network in the OpenVX framework. It is well known that the Deep learning domain for vision, has two fundamental stages. At first the network topology is designed and trained given a collection of labelled data. The network topology is represented as a graph of several nodes comprising Neural Network building block. The trained data represents the problem to be addressed. During the training Phase, the parameters (also referred to as weights/biases or coefficients) are determined for the given network topology. The network topology solution can then be deployed.

In Deployment the network topology as well as parameters are fixed which allow optimizing in hardware and software. In certain scenarios an additional intermediate step is performed to optimize the parameters to a certain target hardware. As an example, using fixed point calculations. When Deployed, the Neural Network is used for inferences on input data. The main objective of the Neural Network Extension for OpenVX is to enable the deployment phase (in other words inferences).

This section provides the definition of the basic terminology to be used across the document, in an attempt to address the various use and different naming in the academy as well as the industry. Those names refer to the same fundamental concept of Deep Neural Networks in the deep learning domain. We refer to the term Deep Neural Network to the network topology of the deep learning network, that is composed of multiple layers in which one of the main layer is Convolution. Other names used in the academia and industry to refer to the same type

of network topologies are CNN (Convolutional Neural Networks) and ConvNets. Throughout this document we will use the Deep Neural Network to refer to the Neural Network, CNN and ConvNet.

Weights - Will use the term Weights to refer to the parameters or coefficients that are the result of training the Deep Neural Network. Weights can be shared or non shared. Or have local connectivity.

Biasses - Will use the term Biasses to refer to the parameters or coefficients, per output only, that are the result of training the Deep Neural Network.

Convolution Layer - A type of layer in the Deep Neural Network that has local connectivity and shared weights, other naming are Locality connected with shared weights.

Fully Connected Layer - All inputs to the layer affect outputs of the layer, in other words connection from every element of input to every element of output.

Activation Layer - A layer that performs operations on every input data and is inspired by the neuron activation function approximated usually using non-Linear functions.

The documentation below uses the abbreviations IFM and OFM, which stand for “Input Feature Maps” and “Output Feature Maps,” respectively. Each feature map is a 2 dimensional image. A CNN input or output tensor will typically have 3 dimensions, where the first two are the width and height of the images, and the third is the number of feature maps. For inputs, the third dimension is the number of IFMs, and for outputs, the third dimension is the number of OFMs.

1.3 Introduction

The Neural Networks extension enables execution and integration of Deep Neural Networks in OpenVX processing graphs. The extension is dependent on a `vx_tensor` object which is introduced in OpenVX 1.2. Therefore this extension is extending OpenVX 1.2 and not previous OpenVX specifications. The `vx_tensor` object is a multidimensional array with an arbitrary number of dimensions. The `vx_tensor` object can represent all varieties of data typically used in a Deep Neural Network. It can represent 2-dimensional images, 3-dimensional sequences of images (usually the input and outputs of a Deep Neural Network) and 4-dimensional weights.

Application can build an OpenVX graph that represents Deep Neural Network topologies where the layers are represented as OpenVX nodes (`vx_node`) and the `vx_tensor` as the data objects connecting the nodes (layers) of the OpenVX graph (Deep Neural Network). The application can as well build an OpenVX graph that is a mix of Deep Neural Network layers and Vision nodes. All graphs (including Deep Neural Networks) are treated as any OpenVX graph, and must comply with the graph concepts as specified in section 2.8 of OpenVX 1.1, especially but not limit to the graph formalisms in section 2.8.6. Additionally, this extension defines several auxiliary functions to create, release, and copy `vx_tensor` objects. Moreover, the extension introduces the concept of “view” for `vx_tensor` objects, which is similar to the ROI of a `vx_image`. The use of “view” enables splitting and merging `vx_tensor` objects, which are common operations in Convolutional Networks. The layers of the Deep Neural Network (represented by `vx_node` objects) perform the computations on the tensor data objects and form a dataflow graph of computations. The extension defines the following layer types: convolution, activation, pooling, fully-connected, and soft-max.

1.4 Weights/Biasses Setting

It is assumed that the Deep Neural Networks are trained in framework external to OpenVX and imported. This requires the application to allocate a memory area for the weights/biasses, read the weight values from a file into this memory area, and then use the `vxCopyTensorPatch` API to copy the weights/biasses from the memory area into the appropriate OpenVX Tensor object. The `vxCopyTensorPatch` function will convert the application memory to the implementation-specific format before putting it into the Tensor object. While effective, this method has the drawback that an intermediate memory area needs to be allocated and a copy and conversion needs to be done.

A separate “import/export” extension defines a `vxImportBinary` function that can be implemented more efficiently. Implementations of `vxImportBinary` could read a weight file or perhaps an entire graph description directly without the need for an intermediate copy. The format of this binary will be implementation-dependent. OpenVX implementations that support both the Neural Network extension and the binary import/export extension can use this more efficient method to set the Deep Neural Networks weights/biasses. The `vxImportBinary` function will return a handle to an object that can be queried to get handles for the individual objects within it via the `vxGetImportReferenceByName` or `vxGetImportReferenceByIndex` functions. Further details and alternate usages of the `vxImportBinary` function are provided in the specification of the “import/export” extension.

1.5 Kernel names

When using `vxGetKernelByName` the following are strings specifying 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`

1.6 8-bit extension and 16-bit extension

The Neural Network Extension is actually two different extensions. Neural Network 16-bit extension and Neural Network 8-bit extension. The 8-bit extension is required. The 16-bit extension is optional. For 8-bit extension, `VX_TYPE_UINT8` and `VX_TYPE_INT8`, with `fixed_point_position` 0, must be supported for all functions. For 16-bit extension, `VX_TYPE_INT16` with `fixed_point_position` 8, must be supported for all functions. The users can query `VX_CONTEXT_EXTENSIONS`, the extension strings are returned to identify two extensions. Implementations must return the 8-bit extension string, and may return the 16-bit extension string. If implementations return the 16-bit extension string, the 8-bit extension string must be returned as well. The 8-bit extension string is "KHR_NN_8" and the 16-bit extension string is "KHR_NN_16". The legal string combinations are "KHR_NN_8" or "KHR_NN_8 KHR_NN_16".

Chapter 2

Module Documentation

2.1 Extension: Deep Convolutional Networks API

Convolutional Network Nodes.

Data Structures

- struct [vx_nn_convolution_params_t](#)
Input parameters for a convolution operation. [More...](#)
- struct [vx_nn_deconvolution_params_t](#)
Input parameters for a deconvolution operation. [More...](#)
- struct [vx_nn_roi_pool_params_t](#)
Input parameters for ROI pooling operation. [More...](#)

Macros

- #define [VX_LIBRARY_KHR_NN_EXTENSION](#) (0x1)
The Neural Network Extension Library Set.

Enumerations

- enum [vx_kernel_nn_ext_e](#) {
[VX_KERNEL_CONVOLUTION_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x0,
[VX_KERNEL_FULLYCONNECTED_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x1,
[VX_KERNEL_POOLING_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x2,
[VX_KERNEL_SOFTMAX_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x3,
[VX_KERNEL_NORMALIZATION_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x4,
[VX_KERNEL_ACTIVATION_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x5,
[VX_KERNEL_ROIPOOLING_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x6,
[VX_KERNEL_DECONVOLUTION_LAYER](#) = (((VX_ID_KHRONOS) << 20) | ((0x1) << 12)) + 0x7 }
The list of Neural Network Extension Kernels.
- enum [vx_nn_activation_function_e](#) {
[VX_NN_ACTIVATION_LOGISTIC](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE << 12)) + 0x0,
[VX_NN_ACTIVATION_HYPERBOLIC_TAN](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE << 12)) + 0x1,
[VX_NN_ACTIVATION_RELU](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE << 12)) + 0x2,
[VX_NN_ACTIVATION_BRELU](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE << 12)) + 0x3,
[VX_NN_ACTIVATION_SOFTRELU](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE << 12)) + 0x4,
[VX_NN_ACTIVATION_ABS](#) = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ACTIVATION_FUNCTION-

```

TYPE << 12)) + 0x5,
VX_NN_ACTIVATION_SQUARE = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_NN_ACTIVATION_FUNC-
TION_TYPE << 12)) + 0x6,
VX_NN_ACTIVATION_SQRT = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_NN_ACTIVATION_FUNC-
TION_TYPE << 12)) + 0x7,
VX_NN_ACTIVATION_LINEAR = ((( VX_ID_KHRONOS ) << 20) | ( VX_ENUM_NN_ACTIVATION_FUNC-
TION_TYPE << 12)) + 0x8 }

```

The Neural Network activation functions list.

- enum **vx_nn_enum_e** {
 VX_ENUM_NN_ROUNDING_TYPE = 0x18,
 VX_ENUM_NN_POOLING_TYPE = 0x19,
 VX_ENUM_NN_NORMALIZATION_TYPE = 0x1A,
 VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE = 0x1B }

NN extension type enums.

- enum **vx_nn_norm_type_e** {
 VX_NN_NORMALIZATION_SAME_MAP = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_NORMALIZATI-
 ON_TYPE << 12)) + 0x0,
 VX_NN_NORMALIZATION_ACROSS_MAPS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_NORMALI-
 ZATION_TYPE << 12)) + 0x1 }

The Neural Network normalization type list.

- enum **vx_nn_pooling_type_e** {
 VX_NN_POOLING_MAX = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_POOLING_TYPE << 12)) + 0x0,
 VX_NN_POOLING_AVG = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_POOLING_TYPE << 12)) + 0x1
 }

The Neural Network pooling type list.

- enum **vx_nn_rounding_type_e** {
 VX_NN_DS_SIZE_ROUNDING_FLOOR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ROUNDING_TY-
 PE << 12)) + 0x0,
 VX_NN_DS_SIZE_ROUNDING_CEILING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NN_ROUNDING_T-
 YPE << 12)) + 0x1 }

down scale rounding.

Functions

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

*[Graph] Creates a Convolutional Network Activation Layer Node. The function operate a specific function (Specified in **vx_nn_activation_function_e**), On the input data. the equation for the layer is: $outputs(i, j, k, l) = function(inputs(i, j, k, l), a, b)$ for all i, j, k, l .*
- vx_node **vxConvolutionLayer** (vx_graph graph, vx_tensor inputs, vx_tensor weights, vx_tensor biases, const **vx_nn_convolution_params_t** *convolution_params, vx_size size_of_convolution_params, vx_tensor outputs)

[Graph] Creates a Convolutional Network Convolution Layer Node.
- vx_node **vxDeconvolutionLayer** (vx_graph graph, vx_tensor inputs, vx_tensor weights, vx_tensor biases, const **vx_nn_deconvolution_params_t** *deconvolution_params, vx_size size_of_deconv_params, vx_tensor outputs)

[Graph] Creates a Convolutional Network Deconvolution 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)

[Graph] Creates a Fully connected Convolutional Network 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)

[Graph] Creates a Convolutional Network Normalization Layer Node. This function is optional for 8-bit extension with the extension string 'KHR.NN.8'.
- 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)

[Graph] Creates a Convolutional Network Pooling Layer Node.

- vx_node [vxROIPoolingLayer](#) (vx_graph graph, vx_tensor input_data, vx_tensor input_rois, const [vx_nn_roi_pool_params_t](#) *roi_pool_params, vx_size size_of_roi_params, vx_tensor output_arr)

[Graph] Creates a Convolutional Network ROI pooling node

- vx_node [vxSoftmaxLayer](#) (vx_graph graph, vx_tensor inputs, vx_tensor outputs)

[Graph] Creates a Convolutional Network Softmax Layer Node.

2.1.1 Detailed Description

Convolutional Network Nodes.

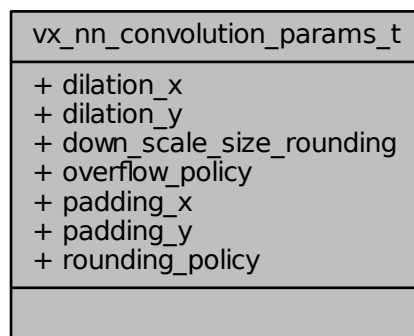
2.1.2 Data Structure Documentation

struct vx_nn_convolution_params_t

Input parameters for a convolution operation.

Definition at line 182 of file [vx_khr_nn.h](#).

Collaboration diagram for vx_nn_convolution_params_t:



Data Fields

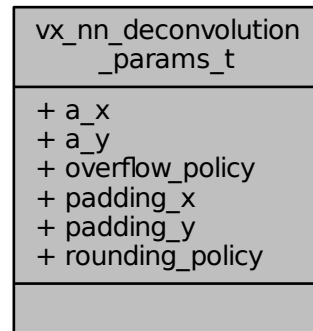
vx_size	dilation_x	"inflate" the kernel by inserting zeros between the kernel elements in the x direction. The value is the number of zeros to insert.
vx_size	dilation_y	"inflate" the kernel by inserting zeros between the kernel elements in the y direction. The value is the number of zeros to insert.
vx_enum	down_scale_size_rounding	Rounding method for calculating output dimensions. See vx_nn_rounding_type_e
vx_enum	overflow_policy	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
vx_size	padding_x	Number of elements added at each side in the x dimension of the input.
vx_size	padding_y	Number of elements added at each side in the y dimension of the input.
vx_enum	rounding_policy	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.

struct vx_nn_deconvolution_params_t

Input parameters for a deconvolution operation.

Definition at line 197 of file [vx_khr_nn.h](#).

Collaboration diagram for vx_nn_deconvolution_params_t:



Data Fields

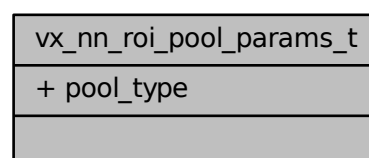
vx_size	a_x	user-specified quantity used to distinguish between the <i>upscale_x</i> different possible output sizes.
vx_size	a_y	user-specified quantity used to distinguish between the <i>upscale_y</i> different possible output sizes.
vx_enum	overflow_policy	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
vx_size	padding_x	Number of elements subtracted at each side in the x dimension of the input.
vx_size	padding_y	Number of elements subtracted at each side in the y dimension of the input.
vx_enum	rounding_policy	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.

struct vx_nn_roi_pool_params_t

Input parameters for ROI pooling operation.

Definition at line 210 of file vx_khr_nn.h.

Collaboration diagram for vx_nn_roi_pool_params_t:



Data Fields

vx_enum	pool_type	Of type vx_nn_pooling_type_e . Only VX_NN_POOLING_MAX pooling is supported.
---------	-----------	-------------------------------------------------------------------------------------------------------------

2.1.3 Enumeration Type Documentation

enum vx_kernel_nn_ext_e

The list of Neural Network Extension Kernels.

Enumerator

VX_KERNEL_CONVOLUTION_LAYER The Neural Network Extension convolution Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_FULLYCONNECTED_LAYER The Neural Network Extension fully connected Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_POOLING_LAYER The Neural Network Extension pooling Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_SOFTMAX_LAYER The Neural Network Extension softmax Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_NORMALIZATION_LAYER The Neural Network Extension normalization Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_ACTIVATION_LAYER The Neural Network Extension activation Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_ROIPOOLING_LAYER The Neural Network ROI Pooling Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

VX_KERNEL_DECONVOLUTION_LAYER The Neural Network Extension Deconvolution Kernel.

See Also

[Extension: Deep Convolutional Networks API](#)

Definition at line 63 of file [vx_khr_nn.h](#).

enum vx_nn_activation_function_e

The Neural Network activation functions list.

Function name	Mathematical definition	Parameters	Parameters type
logistic	$f(x) = 1 / (1 + e^{-x})$		
hyperbolic tangent	$f(x) = a \cdot \tanh(b \cdot x)$	a,b	VX.FLOAT32
relu	$f(x) = \max(0, x)$		

bounded relu	$f(x) = \min(a, \max(0, x))$	a	VX_FLOAT32
soft relu	$f(x) = \log(1 + e^x)$		
abs	$f(x) = x $		
square	$f(x) = x^2$		
square root	$f(x) = \sqrt{x}$		
linear	$f(x) = ax + b$	a,b	VX_FLOAT32

Definition at line 166 of file [vx_khr_nn.h](#).

enum vx_nn_norm_type_e

The Neural Network normalization type list.

Enumerator

VX_NN_NORMALIZATION_SAME_MAP normalization is done on same IFM

VX_NN_NORMALIZATION_ACROSS_MAPS Normalization is done across different IFMs.

Definition at line 140 of file [vx_khr_nn.h](#).

enum vx_nn_pooling_type_e

The Neural Network pooling type list.

kind of pooling done in pooling function

Enumerator

VX_NN_POOLING_MAX max pooling

VX_NN_POOLING_AVG average pooling

Definition at line 128 of file [vx_khr_nn.h](#).

enum vx_nn_rounding_type_e

down scale rounding.

Due to different scheme of downscale size calculation in the various training frameworks. Implementation must support 2 rounding methods for down scale calculation. The floor and the ceiling. In convolution and pooling functions. Relevant when input size is even.

Enumerator

VX_NN_DS_SIZE_ROUNDING_FLOOR floor rounding

VX_NN_DS_SIZE_ROUNDING_CEILING ceil rounding

Definition at line 115 of file [vx_khr_nn.h](#).

2.1.4 Function Documentation

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

[Graph] Creates a Convolutional Network Activation Layer Node. The function operate a specific function (Specified in [vx_nn_activation_function_e](#)), On the input data. the equation for the layer is: $outputs(i, j, k, l) = function(inputs(i, j, k, l), a, b)$ for all i,j,k,l.

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor data. Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
in	<i>function</i>	Non-linear function (see vx_nn_activation_function_e). Implementations must support VX_NN_ACTIVATION_LOGISTIC, VX_NN_ACTIVATION_HYPERBOLIC_TAN and VX_NN_ACTIVATION_RELU
in	<i>a</i>	Function parameters a. must be positive.
in	<i>b</i>	Function parameters b. must be positive.
out	<i>outputs</i>	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

[Graph] Creates a Convolutional Network Convolution Layer Node.

This function implement Convolutional Network Convolution layer. For fixed-point data types, a fixed point calculation is performed with round and saturate according to the number of accumulator bits. The number of the accumulator bits are implementation defined, and should be at least 16.

round: rounding according the `vx_round_policy_e` enumeration.

saturate: A saturation according the `vx_convert_policy_e` enumeration. The following equation is implemented:

$$outputs[j, k, i] = saturate(round(\sum_l (\sum_{m, n} inputs[j - m, k - n, l] \times weights[m, n, l, i]) + biases[j, k, i]))$$

Where m, n are indexes on the convolution matrices. l is an index on all the convolutions per input. i is an index per output. j, k are the inputs/outputs spatial indexes. Convolution is done on the width and height dimensions of the `vx_tensor`. Therefore, we use here the term x for index along the width dimension and y for index along the height dimension.

before the Convolution is done, a padding with zeros of the width and height input dimensions is performed. Then down scale is done by picking the results according to a skip jump. The skip in the x and y is determined by the output size dimensions. The relation between input to output is as follows:

$$width_{output} = round(\frac{(width_{input} + 2 * padding_x - kernel_x - (kernel_x - 1) * dilation_x)}{skip_x} + 1)$$

and

$$height_{output} = round(\frac{(height + 2 * padding_y - kernel_y - (kernel_y - 1) * dilation_y)}{skip_y} + 1)$$

where $width$ is the size of the input width dimension. $height$ is the size of the input height dimension. $width_{output}$ is the size of the output width dimension. $height_{output}$ is the size of the output height dimension. $kernel_x$ and $kernel_y$ are the convolution sizes in width and height dimensions. skip is calculated by the relation between input and output. rounding is done according to `vx_nn_rounding_type_e`.

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor data. 3 lower dimensions represent a single input, all following dimensions represent number of batches, possibly nested. The dimension order is [width, height, #IFM, #batches] . Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
in	<i>weights</i>	Weights are 4d tensor with dimensions [kernel_x, kernel_y, #IFM, #OFM]. see <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> Weights data type must match the data type of the inputs.
in	<i>biases</i>	Optional, ignored if NULL. The biases, which may be shared (one per ofm) or unshared (one per ofm * output location). The possible layouts are either [#OFM] or [width, height, #OFM]. Biases data type must match the data type of the inputs.
in	<i>convolution_params</i>	Pointer to parameters of type <code>vx_nn_convolution_params_t</code>
in	<i>size_of_convolution_params</i>	Size in bytes of convolution_params.
out	<i>outputs</i>	The output tensor data. Output will have the same number and structure of dimensions as input. Output tensor data type must be same as the inputs.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

[Graph] Creates a Convolutional Network Deconvolution Layer Node.

Deconvolution denote a sort of reverse convolution, which importantly and confusingly is not actually a proper mathematical deconvolution. Convolutional Network Deconvolution is up-sampling of an image by learned Deconvolution coefficients. The operation is similar to convolution but can be implemented by up-sampling the inputs with zeros insertions between the inputs, and convolving the Deconvolution kernels on the up-sampled result. For fixed-point data types, a fixed point calculation is performed with round and saturate according to the number of accumulator bits. The number of the accumulator bits are implementation defined, and should be at least 16.

round: rounding according the `vx_round_policy_e` enumeration.

saturate: A saturation according the `vx_convert_policy_e` enumeration. The following equation is implemented:

$$outputs[j, k, i] = saturate(round(\sum_l \sum_{m, n} (inputs_{upscaled}[j - m, k - n, l] \times weights[m, n, l, i]) + biases[j, k, i]))$$

Where m, n are indexes on the convolution matrices. l is an index on all the convolutions per input. i is an index per output. j, k are the inputs/outputs spatial indexes. Deconvolution is done on the width and height dimensions of the `vx_tensor`. Therefore, we use here the term x for the width dimension and y for the height dimension.

before the Deconvolution is done, up-scaling the width and height dimensions with zeros is performed. The relation between input to output is as follows:

$$width_{output} = round((width_{input} - 1) * upscale_x - 2 * padding_x + kernel_x + a_x)$$

and

$$height_{output} = round((height_{input} - 1) * upscale_y - 2 * padding_y + kernel_y + a_y)$$

where $width_{input}$ is the size of the input width dimension. $height_{input}$ is the size of the input height dimension. $width_{output}$ is the size of the output width dimension. $height_{output}$ is the size of the output height dimension. $kernel_x$ and $kernel_y$ are the convolution sizes in width and height. a_x and a_y are user-specified quantity used to distinguish between the $upscale_x$ and $upscale_y$ different possible output sizes $upscale_x$ and $upscale_y$ are calculated by the relation between input and output. rounding is done according to `vx_nn_rounding_type_e`.

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor. 3 lower dimensions represent a single input, and an optional 4th dimension for batch of inputs. Dimension layout is [width, height, #IFM, #batches]. See <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> . Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
in	<i>weights</i>	The 4d weights with dimensions [width, height, #IFM, #OFM]. See <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> .
in	<i>biases</i>	Optional, ignored if NULL. The biases have one dimension [#OFM]. Implementations must support input tensor data type same as the inputs.
in	<i>deconvolution_params</i>	Pointer to parameters of type <code>vx_nn_deconvolution_params_t</code>
in	<i>size_of_deconv_params</i>	Size in bytes of <code>deconvolution_params</code> .
out	<i>outputs</i>	The output tensor. The output has the same number of dimensions as the input.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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)

[Graph] Creates a Fully connected Convolutional Network Layer Node.

This function implement Fully connected Convolutional Network layers. For fixed-point data types, a fixed point calculation is performed with round and saturate according to the number of accumulator bits. The number of the accumulator bits are implementation defined, and should be at least 16.

round: rounding according the vx_round_policy_e enumeration.

saturate: A saturation according the vx_convert_policy_e enumeration. The equation for Fully connected layer:

$$outputs[i] = saturate(round(\sum_j (inputs[j] \times weights[j, i]) + biases[i]))$$

Where j is a index on the input feature and i is a index on the output. before the fully connected is done, a padding of the input is performed. Then down scale is done by picking the results according to a skip jump. The skip is determined by the output size dimensions. The relation between input to output is as follows: $size_{output} = round(\frac{(size_{input} + 2 * padding)}{skip} + 1)$

where $size_{input}$ is the size of the input dimension. $size_{output}$ is the size of the output dimension. skip is calculated by the relation between input and output. rounding is done according to vx_nn_rounding_type_e.

Parameters

in	graph	The handle to the graph.
in	inputs	The input tensor data. There two possible input layouts: 1. [#IFM, #batches]. See vxCreateTensor and vxCreateVirtualTensor. 2. [width, height, #IFM, #batches]. See vxCreateTensor and vxCreateVirtualTensor In both cases number of batches are optional and may be multidimensional. The second option is a special case to deal with convolution layer followed by fully connected. The dimension order is [#IFM, #batches]. See vxCreateTensor and vxCreateVirtualTensor. Note that batch may be multidimensional. Implementations must support input tensor data types indicated by the extension strings 'KHR.NN.8' or 'KHR.NN.8 KHR.NN.16'.
in	weights	Number of dimensions is 2. Dimensions are [#IFM, #OFM]. See vxCreateTensor and vxCreateVirtualTensor. Implementations must support input tensor data type same as the inputs.
in	biases	Optional, ignored if NULL. The biases have one dimension [#OFM]. Implementations must support input tensor data type same as the inputs.
in	overflow_policy	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
in	rounding_policy	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.
out	outputs	The output tensor data. Output dimension layout is [#OFM, #batches]. See vxCreateTensor and vxCreateVirtualTensor, where #batches may be multidimensional. Output tensor data type must be same as the inputs.

Returns

vx_node.

A node reference vx_node. Any possible errors preventing a successful creation should be checked using vxGetStatus.

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)

[Graph] Creates a Convolutional Network Normalization Layer Node. This function is optional for 8-bit extension with the extension string 'KHR.NN.8'.

Normalizing over local input regions. Each input value is divided by $(1 + \frac{\alpha}{n} \sum_i x_i^2)^\beta$, where n is the number of elements to normalize across. and the sum is taken over a rectangle region centred at that value (zero padding is added where necessary).

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor data. 3 lower dimensions represent a single input, 4th dimension for batch of inputs is optional. Dimension layout is [width, height, IFM, #batches]. See <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> . Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8 KHR_NN_16'. Since this function is optional for 'KHR_NN_8', so implementations only must support <code>VX_TYPE_INT16</code> with <code>fixed_point_position</code> 8.
in	<i>type</i>	Either same map or across maps (see vx_nn_norm_type_e).
in	<i>normalization-size</i>	Number of elements to normalize across. Must be a positive odd number with maximum size of 7 and minimum of 3.
in	<i>alpha</i>	Alpha parameter in the normalization equation. must be positive.
in	<i>beta</i>	Beta parameter in the normalization equation. must be positive.
out	<i>outputs</i>	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

`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)`

[Graph] Creates a Convolutional Network Pooling Layer Node.

Pooling is done on the width and height dimensions of the `vx_tensor`. Therefore, we use here the term x for the width dimension and y for the height dimension.

Pooling operation is a function operation over a rectangle size and then a nearest neighbour down scale. Here we use `pooling_size_x` and `pooling_size_y` to specify the rectangle size on which the operation is performed.

before the operation is done (average or maximum value). the data is padded with zeros in width and height dimensions . The down scale is done by picking the results according to a skip jump. The skip in the x and y dimension is determined by the output size dimensions. The first pixel of the down scale output is the first pixel in the input.

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor data. 3 lower dimensions represent a single input, 4th dimension for batch of inputs is optional. Dimension layout is [width, height, #IFM, #batches]. See <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
in	<i>pooling_type</i>	Either max pooling or average pooling (see vx_nn_pooling_type_e).
in	<i>pooling_size_x</i>	Size of the pooling region in the x dimension
in	<i>pooling_size_y</i>	Size of the pooling region in the y dimension.
in	<i>pooling_padding_x</i>	Padding size in the x dimension.
in	<i>pooling_padding_y</i>	Padding size in the y dimension.

in	<i>rounding, Rounding</i>	method for calculating output dimensions. See vx_nn_rounding_type_e
out	<i>outputs</i>	The output tensor data. Output will have the same number of dimensions as input. Output tensor data type must be same as the inputs.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

`vx_node vxROIPoolingLayer (vx_graph graph, vx_tensor input_data, vx_tensor input_rois, const vx_nn.roi_pool_params_t * roi_pool_params, vx_size size_of_roi_params, vx_tensor output_arr)`

[Graph] Creates a Convolutional Network ROI pooling node

Pooling is done on the width and height dimensions of the `vx_tensor`. The ROI Pooling get an array of roi rectangles, and an input tensor. The kernel crop the width and height dimensions of the input tensor with the ROI rectangles and down scale the result to the size of the output tensor. The output tensor width and height are the pooled width and pooled height. The down scale method is determined by the `pool_type`.

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor data. 3 lower dimensions represent a single input, 4th dimension for batch of inputs is optional. Dimension layout is [width, height, #IFM, #batches]. See <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> . Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
in	<i>inputs_rois</i>	The roi array tensor. ROI array with dimensions [4, roi.count, #batches] where the first dimension represents 4 coordinates of the top left and bottom right corners of the roi rectangles, based on the input tensor width and height. #batches is optional and must be the same as in inputs. roi.count is the number of ROI rectangles.
in	<i>pool_type</i>	Of type vx_nn_pooling_type_e . Only <code>VX_NN_POOLING_MAX</code> pooling is supported.
in	<i>size_of_roi_params</i>	Size in bytes of <code>roi_pool_params</code> .
out	<i>output_arr</i>	The output tensor. Output will have [output_width, output_height, #IFM, #batches] dimensions. #batches is optional and must be the same as in inputs.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

`vx_node vxSoftmaxLayer (vx_graph graph, vx_tensor inputs, vx_tensor outputs)`

[Graph] Creates a Convolutional Network Softmax Layer Node.

the softmax function, is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector $\sigma(z)$ of real values in the range (0, 1) that add up to 1. The function is given by: $\sigma(z) = \frac{\exp^z}{\sum_i \exp^{z_i}}$

Parameters

in	<i>graph</i>	The handle to the graph.
in	<i>inputs</i>	The input tensor, with the number of dimensions according to the following scheme. In case IFM dimension is 1. Softmax is be calculated on that dimension. In case IFM dimension is 2. Softmax is be calculated on the first dimension. The second dimension is batching. In case IFM dimension is 3. Dimensions are [Width, Height, Classes]. And Softmax is calculated on the third dimension. In case IFM dimension is 4. Dimensions are [Width, Height, Classes, batching]. Softmax is calculated on the third dimension. Regarding the layout specification, see <code>vxCreateTensor</code> and <code>vxCreateVirtualTensor</code> . In all cases Implementations must support input tensor data types indicated by the extension strings 'KHR_NN_8' or 'KHR_NN_8 KHR_NN_16'.
out	<i>outputs</i>	The output tensor. Output will have the same number of dimensions as input. Output tensor data type must be same as the inputs.

Returns

`vx_node`.

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Index

Extension: Deep Convolutional Networks API

VX_KERNEL_ACTIVATION_LAYER, 9
VX_KERNEL_CONVOLUTION_LAYER, 9
VX_KERNEL_DECONVOLUTION_LAYER, 9
VX_KERNEL_FULLYCONNECTED_LAYER, 9
VX_KERNEL_NORMALIZATION_LAYER, 9
VX_KERNEL_POOLING_LAYER, 9
VX_KERNEL_ROIPOOLING_LAYER, 9
VX_KERNEL_SOFTMAX_LAYER, 9
VX_NN_DS_SIZE_ROUNDING_CEILING, 10
VX_NN_DS_SIZE_ROUNDING_FLOOR, 10
VX_NN_NORMALIZATION_ACROSS_MAPS, 10
VX_NN_NORMALIZATION_SAME_MAP, 10
VX_NN_POOLING_AVG, 10
VX_NN_POOLING_MAX, 10

Extension: Deep Convolutional Networks API, 5

vx_kernel_nn_ext_e, 9
vx_nn_activation_function_e, 9
vx_nn_norm_type_e, 9
vx_nn_pooling_type_e, 10
vx_nn_rounding_type_e, 10
vxActivationLayer, 10
vxConvolutionLayer, 10
vxDeconvolutionLayer, 12
vxFullyConnectedLayer, 12
vxNormalizationLayer, 13
vxPoolingLayer, 15
vxROIpoolingLayer, 16
vxSoftmaxLayer, 16

VX_KERNEL_ACTIVATION_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_CONVOLUTION_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_DECONVOLUTION_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_FULLYCONNECTED_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_NORMALIZATION_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_POOLING_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_ROIPOOLING_LAYER

Extension: Deep Convolutional Networks API, 9

VX_KERNEL_SOFTMAX_LAYER

Extension: Deep Convolutional Networks API, 9

VX_NN_DS_SIZE_ROUNDING_CEILING

Extension: Deep Convolutional Networks API, 10

VX_NN_DS_SIZE_ROUNDING_FLOOR

Extension: Deep Convolutional Networks API, 10

VX_NN_NORMALIZATION_ACROSS_MAPS

Extension: Deep Convolutional Networks API, 10

VX_NN_NORMALIZATION_SAME_MAP

Extension: Deep Convolutional Networks API, 10

VX_NN_POOLING_AVG

Extension: Deep Convolutional Networks API, 10

VX_NN_POOLING_MAX

Extension: Deep Convolutional Networks API, 10

vx_nn_convolution_params_t, 7

vx_nn_deconvolution_params_t, 7

vx_nn_roi_pool_params_t, 8

vx_kernel_nn_ext_e

Extension: Deep Convolutional Networks API, 9

vx_nn_activation_function_e

Extension: Deep Convolutional Networks API, 9

vx_nn_norm_type_e

Extension: Deep Convolutional Networks API, 9

vx_nn_pooling_type_e

Extension: Deep Convolutional Networks API, 10

vx_nn_rounding_type_e

Extension: Deep Convolutional Networks API, 10

vxActivationLayer

Extension: Deep Convolutional Networks API, 10

vxConvolutionLayer

Extension: Deep Convolutional Networks API, 10

vxDeconvolutionLayer

Extension: Deep Convolutional Networks API, 12

vxFullyConnectedLayer

Extension: Deep Convolutional Networks API, 12

vxNormalizationLayer

Extension: Deep Convolutional Networks API, 13

vxPoolingLayer

Extension: Deep Convolutional Networks API, 15

vxROIpoolingLayer

Extension: Deep Convolutional Networks API, 16

vxSoftmaxLayer

Extension: Deep Convolutional Networks API, 16