The **OpenVX™** Neural Network Extension

Version 1.0 (Provisional)

Document Revision: 7505566

Khronos Vision Working Group

Editor: Schwartz Tomer
Editor: Hagog Mostafa

Contents

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 - Samsung
- Kari Pulli - Intel
- Tomer Schwartz - Intel
- Mostafa Hagog - Intel
- Chuck Pilkington - synopsis
- Thierry Lepley - NVIDIA
- Radha Giduthuri - AMD
- Jesse Villarreal - TI
- Victor Eruhimov - Itseez
- Xin Wang - Verisilicon

1.2 Background and Terminology

Deep Learning using Deep Convolutional Networks techniques is being increasingly used to perform vision classification and recognition tasks. Deep Convolutional Networks have significantly improved image recognition capabilities over previous technologies. The Neural Network extension for OpenVX is intended to enable the implementation of Deep Convolution 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 Deep Convolution Network building block. The trained data represents the problem to be addressed. During the training Phase, the parameters (also referred to as weights/biasses 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 Deep Convolution 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 Convolutional Networks in the deep learning domain. We refer to the term Convolutional 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 Convolution Network to refer to the Deep Convolution Network, CNN and ConvNet.

Weights - Will use the term Weights to refer to the parameters or coefficients that are the result of training the Convolution 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 Convolution Network.

Convolution Layer - A type of layer in the Convolution 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 Convolutional Networks in OpenVX processing graphs. The extension introduces the `vx_tensor` object, which is a multidimensional array with an arbitrary number of dimensions. The `vx_tensor` object can represent all varieties of data typically used in a Convolutional Network. It can represent 2-dimensional images, 3-dimensional sequences of images (usually the input and outputs of a Convolutional Network)and 4-dimensional weights.

OpenVX implementations compliant to this extension must support `vx_tensor` objects of at least 4 dimensions, although a vendor can choose to support more dimensions in his implementation. The maximum number of dimensions supported by a given implementation can be queried via the context attribute VX_CONTEXT_TENSOR_MAX_DIMENSION. Implementations must support tensors from one dimension (i.e., vectors) through VX_CONTEXT_TENSOR_MAX_DIMENSION, inclusive. The individual elements of the tensor object may be any numerical data type. A compliant implementations must support at least elements of type `VX_TYPE_INT16`. The `VX_TYPE_INT16` elements can represent fractional values by assigning a non-zero radix point. Compliant implementations must support a radix point of 8, which corresponds to Q7.8 signed fixed-point in “Q” notation. A vendor may choose to support additional values for the radix point in his implementation.

Application can build an OpenVX graph that represents Convolution 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 (Convolution Network). The application can as well build an OpenVX graph that is a mix of Convolution Network layers and Vision nodes. All graphs (including Convolution 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 Convolutional 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 Convolutional 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 Convolutional 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.

Chapter 2

Module Documentation

2.1 Extension: Deep Convolutional Networks API

Convolutional Network Nodes.

Enumerations

- enum vx_convolutional_network_activation_func_e {
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_LOGISTIC = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x0,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_HYPERBOLIC_TAN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x1,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_RELU = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x2,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_BRELU = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x3,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_SOFTRELU = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x4,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_ABS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x5,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_SQUARE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x6,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_SQRT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x7,
  VX_CONVOLUTIONAL_NETWORK_ACTIVATION_LINEAR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ACTIVATION_FUNC << 12)) + 0x8 }

  The Convolutional Network activation functions list.
- enum vx_convolutional_network_norm_type_e {
  VX_CONVOLUTIONAL_NETWORK_NORM_SAME_MAP = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_NORM_TYPE << 12)) + 0x0,
  VX_CONVOLUTIONAL_NETWORK_NORM_ACROSS_MAPS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_NORM_TYPE << 12)) + 0x1 }

  The Convolutional Network normalization type list.
- enum vx_convolutional_network_pooling_type_e {
  VX_CONVOLUTIONAL_NETWORK_POOLING_MAX = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_POOL_TYPE << 12)) + 0x0,
  VX_CONVOLUTIONAL_NETWORK_POOLING_AVG = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_POOL_TYPE << 12)) + 0x1 }

  The Convolutional Network pooling type list.
- enum vx_convolutional_networks_rounding_type_e {
  VX_CONVOLUTIONAL_NETWORK_DS_SIZE_ROUNDING_FLOOR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ROUNDING_TYPE << 12)) + 0x0,
  VX_CONVOLUTIONAL_NETWORK_DS_SIZE_ROUNDING_CEILING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVOLUTIONAL_NETWORK_ROUNDING_TYPE << 12)) + 0x1 }

The Convolutional Network down scaling size rounding type list.

Functions

- vx_node vxActivationLayer (vx_graph graph, vx_tensor inputs, vx_enum func, vx_int32 a, vx_int32 b, vx_tensor outputs)

  [Graph] Creates a Convolutional Network Activation Layer Node.
- vx_node vxConvolutionLayer (vx_graph graph, vx_tensor inputs, vx_tensor weights, vx_tensor biases, vx_uint32 pad_x, vx_uint32 pad_y, vx_uint8 accumulator_bits, vx_enum overflow_policy, vx_enum rounding_policy, vx_enum down_scale_size_rounding, vx_tensor outputs)

  [Graph] Creates a Convolutional Network Convolution Layer Node.
- vx_node vxFullyConnectedLayer (vx_graph graph, vx_tensor inputs, vx_tensor weights, vx_tensor biases, vx_uint32 pad, vx_uint8 accumulator_bits, vx_enum overflow_policy, vx_enum rounding_policy, vx_enum down_scale_size_rounding, 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_uint32 norm_size, vx_float32 alpha, vx_float32 beta, vx_tensor outputs)

  [Graph] Creates a Convolutional Network Normalization Layer Node.
- vx_node vxPoolingLayer (vx_graph graph, vx_tensor inputs, vx_enum pool_type, vx_uint32 pool_size_x, vx_uint32 pool_size_y, vx_uint32 pool_pad_x, vx_uint32 pool_pad_y, vx_enum rounding, vx_tensor outputs)

  [Graph] Creates a Convolutional Network Pooling Layer 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 Enumeration Type Documentation

enum vx_convolutional_network_activation_func_e

The Convolutional 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_INT32
relu	$f(x) = max(0,x)$		
bounded relu	$f(x) = min(a,max(0,x))$	a	VX_INT32
soft relu	$f(x) = log(1+e^{x})$		
abs	$f(x) = \mid x \mid$		
square	$f(x) = x^2$		
square root	$f(x) = \sqrt{x}$		
linear	$f(x) = ax+b$	a,b	VX_INT32

Definition at line 161 of file vx_khr_cnn.h.

enum vx_convolutional_network_norm_type_e

The Convolutional Network normalization type list.

Enumerator

VX_CONVOLUTIONAL_NETWORK_NORM_SAME_MAP normalization is done on same IFM

VX_CONVOLUTIONAL_NETWORK_NORM_ACROSS_MAPS Normalization is done across different IFMs.

Definition at line 135 of file vx_khr_cnn.h.

enum vx_convolutional_network_pooling_type_e

The Convolutional Network pooling type list.
kind of pooling done in pooling function

Enumerator

VX_CONVOLUTIONAL_NETWORK_POOLING_MAX	max pooling
VX_CONVOLUTIONAL_NETWORK_POOLING_AVG	average pooling

Definition at line 123 of file vx_khr_cnn.h.

enum vx_convolutional_networks_rounding_type_e

The Convolutional Network down scaling size rounding type list.
rounding done downscaling, In convolution and pooling functions. Relevant when input size is even.

Enumerator

VX_CONVOLUTIONAL_NETWORK_DS_SIZE_ROUNDING_FLOOR	floor rounding
VX_CONVOLUTIONAL_NETWORK_DS_SIZE_ROUNDING_CEILING	ceil rounding

Definition at line 110 of file vx_khr_cnn.h.

2.1.3 Function Documentation

vx_node vxActivationLayer (vx_graph *graph*, vx_tensor *inputs*, vx_enum *func*, vx_int32 *a*, vx_int32 *b*, vx_tensor *outputs*)

[Graph] Creates a Convolutional Network Activation Layer Node.

Parameters

`in`	*graph*	The handle to the graph.
`in`	*inputs*	The input tensor data.
`in`	*func*	Non-linear function (see `vx_convolutional_network_activation_func_e`).
`in`	*a*	Function parameters a. (see `vx_convolutional_network_activation_func_e`).
`in`	*b*	Function parameters b. (see `vx_convolutional_network_activation_func_e`).
`out`	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

vx_node vxConvolutionLayer (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *weights*, vx_tensor *biases*, vx_uint32 *pad_x*, vx_uint32 *pad_y*, vx_uint8 *accumulator_bits*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_enum *down_scale_size_rounding*, vx_tensor *outputs*)

[Graph] Creates a Convolutional Network Convolution Layer Node.

This function implement Convolutional Network Convolution layer. In case the input and output `vx_tensor` are signed 16. A fixed point calculation is performed with round and saturate according to the number of accumulator bits.

round: rounding according the `vx_round_policy_e` enumeration.

saturate: A saturation according the `vx_convert_policy_e` enumeration. The saturation is done based on the accumulator_bits parameter. According the accumulator_bits, the saturation might not be performed every operation. But every a specified amount of operations, that are suspected to saturate the accumulation bits

The following equation is implemented:

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

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 first 2 dimensions of the `vx_tensor`. Therefore, we use here the term x for the first dimension and y for the second.

before the Convolution is done, a padding of the first 2D with zeros is performed. Then 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 relation between input to output is as follows:

$width_{output} = round(\frac{(width+2*pad_x-kernel_x)}{skip_x}+1)$

and

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

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

Parameters

`in`	*graph*	The handle to the graph.
`in`	*inputs*	The input tensor data. 3 lower dims represent a single input, and an optional 4th dimension for batch of inputs.
`in`	*weights*	Weights are 4d tensor with dimensions [kernel_x, kernel_y, #IFM, #OFM].
`in`	*biases*	The biases, which may be shared (one per ofm) or unshared (one per ofm $*$ output location).
`in`	*pad_x*	Number of elements added at each side in the x dimension of the input.
`in`	*pad_y*	Number of elements added at each side in the y dimension of the input. In fully connected layers this input is ignored.
`in`	*accumulator_bits*	Is the total number of bits used during intermediate accumulation.
`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.
`in`	*down_scale_size-_rounding*	Rounding method for calculating output dimensions. See `vx_-convolutional_network_rounding_type_e`
`out`	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
$*$	Node handle.

vx_node vxFullyConnectedLayer (vx_graph *graph*, vx_tensor *inputs*, vx_tensor *weights*, vx_tensor *biases*, vx_uint32 *pad*, vx_uint8 *accumulator_bits*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_enum *down_scale_size_rounding*, vx_tensor *outputs*)

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

This function implement Fully connected Convolutional Network layers. In case the input and output `vx_-tensor` are signed 16. A fixed point calculation is performed with round and saturate according to the number of accumulator bits.

round: rounding according the `vx_round_policy_e` enumeration.

saturate: A saturation according the `vx_convert_policy_e` enumeration. The saturation is done based on the accumulator_bits parameter. According the accumulator_bits, the saturation might not be performed every operation. But every a specified amount of operations, that are suspected to saturate the accumulation bits

The equation for Fully connected layer:

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

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: $size_{output} = round(\frac{(size_{input}+2*pad)}{skip}+1)$

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_convolutional_network_rounding_type_e`.

Parameters

in	*graph*	The handle to the graph.
in	*inputs*	The input tensor data. 1-3 lower dims represent a single input, and all dims above dim(weights)-1 are optional for batch of inputs. Note that batch may be multidimensional.
in	*weights*	Number of dimensions equals dim(single input)+1. Single input dims are [width, height, #IFM], with height and #IFM being optional.
in	*biases*	The biases, which may be shared (one per ofm) or unshared (one per ofm $*$ output location).
in	*pad*	Number of elements added at each side in the input.
in	*accumulator_bits*	Is the total number of bits used during intermediate accumulation.
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.
in	*down_scale_size-_rounding*	Rounding method for calculating output dimensions. See `vx_convolutional_network_rounding_type_e`
out	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
$*$	Node handle.

vx_node vxNormalizationLayer (vx_graph *graph,* vx_tensor *inputs,* vx_enum *type,* vx_uint32 *norm_size,* vx_float32 *alpha,* vx_float32 *beta,* vx_tensor *outputs*)

[Graph] Creates a Convolutional Network Normalization Layer Node.

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 the region centred at that value (zero padding is added where necessary).

Parameters

in	*graph*	The handle to the graph.
in	*inputs*	The input tensor data. 3 lower dims represent a single input with dimensions [width, height, IFM], and an optional 4th dimension for batch of inputs.

in	*type*	Either same map or across maps (see vx_convolutional_network_norm_type_e).
in	*norm_size*	Number of elements to normalize across.
in	*alpha*	Alpha parameter in the normalization equation.
in	*beta*	Beta parameter in the normalization equation.
out	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

vx_node vxPoolingLayer (vx_graph *graph,* vx_tensor *inputs,* vx_enum *pool_type,* vx_uint32 *pool_size_x,* vx_uint32 *pool_size_y,* vx_uint32 *pool_pad_x,* vx_uint32 *pool_pad_y,* vx_enum *rounding,* vx_tensor *outputs*)

[Graph] Creates a Convolutional Network Pooling Layer Node.

Pooling is done on the first 2 dimensions or the `vx_tensor`. Therefore, we use here the term x for the first dimension and y for the second.

Pooling operation is a function operation over a rectangle size and then a nearest neighbour down scale. Here we use pool_size_x and pool_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 in the first 2D with zeros. 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.

Parameters

in	*graph*	The handle to the graph.
in	*inputs*	The input tensor data. 3 lower dims represent a single input with dimensions [width, height, IFM], and an optional 4th dimension for batch of inputs.
in	*pool_type*	Either max pooling or average pooling (see `vx_convolutional_network_pooling_type_e`).
in	*pool_size_x*	Size of the pooling region in the x dimension
in	*pool_size_y*	Size of the pooling region in the y dimension.
in	*pool_pad_x*	Padding size in the x dimension.
in	*pool_pad_y*	Padding size in the y dimension.
in	*rounding,Rounding*	method for calculating output dimensions. See `vx_convolutional_network_rounding_type_e`
out	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

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

[Graph] Creates a Convolutional Network Softmax Layer Node.

Parameters

in	*graph*	The handle to the graph.
in	*inputs*	The input tensor data, with number of dimensions equals dim(input batch) + 1. Softmax will be calculated per IFM.
out	*outputs*	The output tensor data. Output will have the same number of dimensions as input.

Returns

`vx_node`.

Return values

0	Node could not be created.
*	Node handle.

2.2 Tensor API

The Tensor API for Deep Convolutional Networks Functions.

Typedefs

- typedef struct _vx_tensor_t ∗ vx_tensor

  The multidimensional data object (Tensor).
- typedef struct _vx_tensor_addressing_t ∗ vx_tensor_addressing

  The addressing of a tensor view patch structure is used by the Host only to address elements in a tensor view patch.
- typedef struct _vx_tensor_view_t ∗ vx_tensor_view

  The multi dimensional view data structure.

Enumerations

- enum vx_context_attribute_e { VX_CONTEXT_MAX_TENSOR_DIMENSIONS }

  A list of context attributes.
- enum vx_tensor_attribute_e {
  VX_TENSOR_NUM_OF_DIMS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_TENSOR << 8)) + 0x0,
  VX_TENSOR_DIMS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_TENSOR << 8)) + 0x1,
  VX_TENSOR_DATA_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_TENSOR << 8)) + 0x2,
  VX_TENSOR_FIXED_POINT_POS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_TENSOR << 8)) + 0x4 }

  tensor Data attributes.

Functions

- vx_status vxCopyTensorPatch (vx_tensor tensor, vx_tensor_view view, vx_tensor_addressing user_addr, void ∗user_ptr, vx_enum usage, vx_enum user_mem_type)

  Allows the application to copy a view patch from/into an tensor object .
- vx_object_array vxCreateImageObjectArrayFromTensor (vx_tensor tensor, vx_rectangle_t rect, vx_uint32 array_size, vx_uint32 stride, vx_df_image image_format)

  Creates an array of images into the multi-dimension data, this can be a adjacent 2D images or not depending on the stride value. The stride value is representing bytes in the third dimension. The OpenVX image object that points to a three dimension data and access it as an array of images. This has to be portion of the third lowest dimension, and the stride correspond to that third dimension. The returned Object array is an array of images. Where the image data is pointing to a specific memory in the input tensor.
- vx_tensor vxCreateTensor (vx_context context, vx_uint32 num_of_dims, vx_uint32 ∗sizes, vx_enum data_format, vx_uint8 fixed_point_pos)

  Creates an opaque reference to a tensor data buffer.
- vx_tensor_addressing vxCreateTensorAddressing (vx_context context, vx_uint32 ∗addressing_array_dimension, vx_uint32 ∗addressing_array_stride, vx_uint8 numViewDimensions)

  Create an opaque reference to a tensor addressing object.
- vx_tensor vxCreateTensorFromView (vx_tensor tensor, vx_tensor_view view)

  Creates a tensor data from another tensor data given a view. This second reference refers to the data in the original tensor data. Updates to this tensor data updates the parent tensor data. The view must be defined within the dimensions of the parent tensor data.
- vx_tensor_view vxCreateTensorView (vx_context context, vx_uint32 ∗view_array_start, vx_uint32 ∗view_array_end, vx_uint8 numViewDimensions)

  Create an opaque reference to a tensor view object.
- vx_tensor vxCreateVirtualTensor (vx_graph graph, vx_uint32 num_of_dims, vx_uint32 ∗sizes, vx_enum data_format, vx_uint8 fixed_point_pos)

  Creates an opaque reference to a tensor data buffer with no direct user access. This function allows setting the tensor data dimensions or data format.
- vx_status vxQueryTensor (vx_tensor tensor, vx_enum attribute, void ∗ptr, vx_size size)

Retrieves various attributes of a tensor data.

- vx_status vxReleaseTensor (vx_tensor ∗tensor)

  Releases a reference to a tensor data object. The object may not be garbage collected until its total reference count is zero.
- vx_status vxReleaseTensorAddressing (vx_tensor_addressing ∗tensor_addr)

  Releases a reference to a tensor data addressing object. The object may not be garbage collected until its total reference count is zero.
- vx_status vxReleaseTensorView (vx_tensor_view ∗tensor_view)

  Releases a reference to a tensor data view object. The object may not be garbage collected until its total reference count is zero.
- vx_node vxTensorAddNode (vx_graph graph, vx_tensor in1, vx_tensor in2, vx_enum policy, vx_tensor out)

  [Graph] Performs arithmetic addition on element values in the input tensor data's.
- vx_node vxTensorMultiplyNode (vx_graph graph, vx_tensor in1, vx_tensor in2, vx_scalar scale, vx_enum overflow_policy, vx_enum rounding_policy, vx_tensor out)

  [Graph] Performs element wise multiplications on element values in the input tensor data's with a scale.
- vx_node vxTensorSubtractNode (vx_graph graph, vx_tensor in1, vx_tensor in2, vx_enum policy, vx_tensor out)

  [Graph] Performs arithmetic subtraction on element values in the input tensor data's.
- vx_node vxTensorTableLookupNode (vx_graph graph, vx_tensor in1, vx_lut lut, vx_tensor out)

  [Graph] Performs LUT on element values in the input tensor data's.
- vx_node vxTensorTransposeNode (vx_graph graph, vx_tensor in, vx_tensor out, vx_uint32 dim1, vx_uint32 dim2)

  [Graph] Performs transpose on the input tensor. The node transpose the tensor according to a specified 2 indexes in the tensor (0-based indexing)

2.2.1 Detailed Description

The Tensor API for Deep Convolutional Networks Functions. The tensor is a multidimensional opaque object.Since the object have no visibility to the programmer. Vendors can introduce many optimization possibilities. An example of such optimization can be found in the following article.http://arxiv.org/abs/1510.00149

2.2.2 Typedef Documentation

typedef struct _vx_tensor_t∗ vx_tensor

The multidimensional data object (Tensor).

See Also

vxCreateTensor

Definition at line 89 of file vx_khr_cnn.h.

typedef struct _vx_tensor_addressing_t∗ vx_tensor_addressing

The addressing of a tensor view patch structure is used by the Host only to address elements in a tensor view patch.

See Also

`vxCopyTensorPatch`

Definition at line 103 of file vx_khr_cnn.h.

typedef struct _vx_tensor_view_t∗ vx_tensor_view

The multi dimensional view data structure.
Used to split tensors into several views. Or concatenate several view into one tensor.

See Also

vxCreateTensorFromView

Definition at line 96 of file vx_khr_cnn.h.

2.2.3 Enumeration Type Documentation

enum vx_context_attribute_e

A list of context attributes.

Enumerator

VX_CONTEXT_MAX_TENSOR_DIMENSIONS tensor Data max num of dimensions supported by HW.

Definition at line 71 of file vx_khr_cnn.h.

enum vx_tensor_attribute_e

tensor Data attributes.

Enumerator

VX_TENSOR_NUM_OF_DIMS Number of dimensions.

VX_TENSOR_DIMS Dimension sizes.

VX_TENSOR_DATA_TYPE tensor Data element data type. `vx_type_e`

VX_TENSOR_FIXED_POINT_POS fixed point position when the input element type is int16.

Definition at line 55 of file vx_khr_cnn.h.

2.2.4 Function Documentation

vx_status vxCopyTensorPatch (vx_tensor *tensor,* vx_tensor_view *view,* vx_tensor_addressing *user_addr,* void ∗ *user_ptr,* vx_enum *usage,* vx_enum *user_mem_type*)

Allows the application to copy a view patch from/into an tensor object .

Parameters

`in`	*tensor*	The reference to the tensor object that is the source or the destination of the copy.
`in`	*view*	Optional parameter of type `vx_tensor_view`. The coordinates of the view patch. The patch must be within the bounds of the tensor. (start[index],end[index]) gives the coordinates of the view element out of the patch. Must be 0 <= start < end <= number of elements in the tensor dimension. see `vxCreateTensorView`. If NULL is given instead of the object. Then the function behaves as if view was the size of the full tensor.
`in`	*user_addr*	The address of a structure describing the layout of the user memory location pointed by user_ptr. In the structure, dim[index], stride[index] fields must be provided, other fields are ignored by the function. The layout of the user memory must follow a row major order. see `vxCreateTensorAddressing`
`in`	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the tensor object if the copy was requested in write mode. The accessible memory must be large enough to contain the specified patch with the specified layout: accessible memory in bytes >= (end[last_dimension] - start[last_dimension]) ∗ stride[last_dimension]. see `vxCreateTensorAddressing` and `vxCreateTensorView`.

in	*usage*	This declares the effect of the copy with regard to the tensor object using the `vx_accessor_e` enumeration. Only VX_READ_ONLY and VX_WRITE_ONLY are supported: • VX_READ_ONLY means that data is copied from the tensor object into the application memory • VX_WRITE_ONLY means that data is copied into the tensor object from the application memory
in	*user_mem_type*	A `vx_memory_type_e` enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_OPTIMIZED_AWAY	This is a reference to a virtual tensor that cannot be accessed by the application.
VX_ERROR_INVALID_REFERENCE	The tensor reference is not actually an tensor reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

vx_object_array vxCreateImageObjectArrayFromTensor (vx_tensor *tensor,* vx_rectangle_t *rect,* vx_uint32 *array_size,* vx_uint32 *stride,* vx_df_image *image_format*)

Creates an array of images into the multi-dimension data, this can be a adjacent 2D images or not depending on the stride value. The stride value is representing bytes in the third dimension. The OpenVX image object that points to a three dimension data and access it as an array of images. This has to be portion of the third lowest dimension, and the stride correspond to that third dimension. The returned Object array is an array of images. Where the image data is pointing to a specific memory in the input tensor.

Parameters

in	*tensor*	The tensor data from which to extract the images. Has to be a 3d tensor.
in	*rect*	Image coordinates within tensor data.
in	*array_size*	Number of images to extract.
in	*stride*	Delta between two images in the array.
in	*image_format*	The requested image format. Should match the tensor data's data type.

Returns

An array of images pointing to the tensor data's data.

vx_tensor vxCreateTensor (vx_context *context,* vx_uint32 *num_of_dims,* vx_uint32 $*$ *sizes,* vx_enum *data_format,* vx_uint8 *fixed_point_pos*)

Creates an opaque reference to a tensor data buffer.

Not guaranteed to exist until the `vx_graph` containing it has been verified.

Parameters

in	*context*	The reference to the implementation context.
in	*num_of_dims*	The number of dimensions.

`in`	*sizes*	Dimensions sizes in elements.
`in`	*data_format*	The `vx_type_t` that represents the data type of the tensor data elements.
`in`	*fixed_point_pos*	Specifies the fixed point position when the input element type is int16, if 0 calculations are performed in integer math

Returns

A tensor data reference or zero when an error is encountered.

vx_tensor_addressing vxCreateTensorAddressing (vx_context *context,* vx_uint32 ∗ *addressing_array_dimension,* vx_uint32 ∗ *addressing_array_stride,* vx_uint8 *numViewDimensions*)

Create an opaque reference to a tensor addressing object.

Not guaranteed to exist until the `vx_graph` containing it has been verified.

Parameters

`in`	*context*	The reference to the implementation context.
`in`	*addressing_-array_dimension*	a vx_uint32 array of sLength of patch in all dimensions in elements.
`in`	*addressing_-array_stride*	a vx_uint32 arrayStride in all dimensions in bytes.
`in`	*numView-Dimensions*	number of dimensions of view_array_start and view_array_end.

Returns

A tensor data view reference or zero when an error is encountered.

vx_tensor vxCreateTensorFromView (vx_tensor *tensor,* vx_tensor_view *view*)

Creates a tensor data from another tensor data given a view. This second reference refers to the data in the original tensor data. Updates to this tensor data updates the parent tensor data. The view must be defined within the dimensions of the parent tensor data.

Parameters

`in`	*tensor*	The reference to the parent tensor data.
`in`	*view*	The region of interest of a tensor view. Must contain points within the parent tensor data dimensions. `vx_tensor_view`

Returns

The reference to the sub-tensor or zero if the view is invalid.

vx_tensor_view vxCreateTensorView (vx_context *context,* vx_uint32 ∗ *view_array_start,* vx_uint32 ∗ *view_array_end,* vx_uint8 *numViewDimensions*)

Create an opaque reference to a tensor view object.

Not guaranteed to exist until the `vx_graph` containing it has been verified.

Parameters

`in`	*context*	The reference to the implementation context.
`in`	*view_array_start*	a vx_uint32 array of start values of the view.
`in`	*view_array_end*	a vx_uint32 array of end values of the view.

in	*numView-Dimensions*	number of dimensions of view_array_start and view_array_end.

Returns

A tensor data view reference or zero when an error is encountered.

vx_tensor vxCreateVirtualTensor (vx_graph *graph,* vx_uint32 *num_of_dims,* vx_uint32 ∗ *sizes,* vx_enum *data_format,* vx_uint8 *fixed_point_pos*)

Creates an opaque reference to a tensor data buffer with no direct user access. This function allows setting the tensor data dimensions or data format.

Virtual data objects allow users to connect various nodes within a graph via data references without access to that data, but they also permit the implementation to take maximum advantage of possible optimizations. Use this API to create a data reference to link two or more nodes together when the intermediate data are not required to be accessed by outside entities. This API in particular allows the user to define the tensor data format of the data without requiring the exact dimensions. Virtual objects are scoped within the graph they are declared a part of, and can't be shared outside of this scope.

Parameters

in	*graph*	The reference to the parent graph.
in	*num_of_dims*	The number of dimensions.
in	*sizes*	Dimensions sizes in elements.
in	*data_format*	The `vx_type_t` that represents the data type of the tensor data elements.
in	*fixed_point_pos*	Specifies the fixed point position when the input element type is int16, if 0 calculations are performed in integer math

Returns

A tensor data reference or zero when an error is encountered.

Note

Passing this reference to `vxCopyTensorPatch` will return an error.

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

Retrieves various attributes of a tensor data.

Parameters

in	*tensor*	The reference to the tensor data to query.
in	*attribute*	The attribute to query. Use a `vx_tensor_attribute_e`.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If data is not a `vx_tensor`.

VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

vx_status vxReleaseTensor (vx_tensor ∗ *tensor*)

Releases a reference to a tensor data object. The object may not be garbage collected until its total reference count is zero.

Parameters

`in`	*tensor*	The pointer to the tensor data to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_SUCCESS	Success
∗	An error occurred. See `vx_status_e`.

vx_status vxReleaseTensorAddressing (vx_tensor_addressing ∗ *tensor_addr*)

Releases a reference to a tensor data addressing object. The object may not be garbage collected until its total reference count is zero.

Parameters

`in`	*tensor_addr*	The pointer to the tensor data addressing to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_SUCCESS	Success
∗	An error occurred. See `vx_status_e`.

vx_status vxReleaseTensorView (vx_tensor_view ∗ *tensor_view*)

Releases a reference to a tensor data view object. The object may not be garbage collected until its total reference count is zero.

Parameters

`in`	*tensor_view*	The pointer to the tensor data view to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

vx_node vxTensorAddNode (vx_graph *graph,* vx_tensor *in1,* vx_tensor *in2,* vx_enum *policy,* vx_tensor *out*)

[Graph] Performs arithmetic addition on element values in the input tensor data's.
Parameters

`in`	*graph*	The handle to the graph.
`in`	*in1*	input tensor data,.
`in`	*in2*	input tensor data, inputs must be of equal in dimensions. else, If in one of the vx_mddata dimension is 1. That dimension is considered as a const on all the dimension terms. And will perform as if the values are duplicated on all terms in that dimensions. After the expansion. The dimensions are equal.
`in`	*policy*	A vx_convert_policy_e enumeration.
`out`	*out*	The output tensor data with the same dimensions as the input tensor data's.

Returns

`vx_node`.

Return values

0	Node could not be created.
*	Node handle.

vx_node vxTensorMultiplyNode (vx_graph *graph,* vx_tensor *in1,* vx_tensor *in2,* vx_scalar *scale,* vx_enum *overflow_policy,* vx_enum *rounding_policy,* vx_tensor *out*)

[Graph] Performs element wise multiplications on element values in the input tensor data's with a scale.
Parameters

`in`	*graph*	The handle to the graph.
`in`	*in1*	input tensor data.
`in`	*in2*	input tensor data, inputs must be of equal in dimensions. else, If in one of the vx_mddata dimension is 1. That dimension is considered as a const on all the dimension terms. And will perform as if the values are duplicated on all terms in that dimensions. After the expansion. The dimensions are equal.
`in`	*scale*	The scale value.
`in`	*overflow_policy*	A `vx_convert_policy_e` enumeration.
`in`	*rounding_policy*	A `vx_round_policy_e` enumeration.
`out`	*out*	The output tensor data with the same dimensions as the input tensor data's.

Returns

`vx_node`.

Return values

0	Node could not be created.
*	Node handle.

vx_node vxTensorSubtractNode (vx_graph *graph,* vx_tensor *in1,* vx_tensor *in2,* vx_enum *policy,* vx_tensor *out*)

[Graph] Performs arithmetic subtraction on element values in the input tensor data's.

Parameters

`in`	*graph*	The handle to the graph.
`in`	*in1*	input tensor data.
`in`	*in2*	input tensor data, inputs must be of equal in dimensions. else, If in one of the vx_mddata dimension is 1. That dimension is considered as a const on all the dimension terms. And will perform as if the values are duplicated on all terms in that dimensions. After the expansion. The dimensions are equal.
`in`	*policy*	A vx_convert_policy_e enumeration.
`out`	*out*	The output tensor data with the same dimensions as the input tensor data's.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

vx_node vxTensorTableLookupNode (vx_graph *graph,* vx_tensor *in1,* vx_lut *lut,* vx_tensor *out*)

[Graph] Performs LUT on element values in the input tensor data's.
Parameters

`in`	*graph*	The handle to the graph.
`in`	*in1*	input tensor data.
`in`	*lut*	of type `vx_lut`
`out`	*out*	The output tensor data with the same dimensions as the input tensor data's.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

vx_node vxTensorTransposeNode (vx_graph *graph,* vx_tensor *in,* vx_tensor *out,* vx_uint32 *dim1,* vx_uint32 *dim2*)

[Graph] Performs transpose on the input tensor. The node transpose the tensor according to a specified 2 indexes in the tensor (0-based indexing)
Parameters

`in`	*graph*	The handle to the graph.
`in`	*in*	input tensor data,
`out`	*out*	output tensor data,
`in`	*dim1*	that is transposed with dim 2.
`in`	*dim2*	that is transposed with dim 1.

Returns

`vx_node`.

Return values

0	Node could not be created.
∗	Node handle.

Index