The OpenVX™ Neural Network Extension

The Khronos® OpenVX Working Group, Editors: Radhakrishna Giduthuri, Tomer Schwartz, Mostafa Hagog

Version 1.3, Thu, 08 Aug 2019 20:28:09 +0000

Table of 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:

- Radhakrishna Giduthuri - Intel
- Frank Brill - Cadence Design Systems
- Thierry Lepley - Cadence Design Systems
- Kari Pulli - Intel
- Mostafa Hagog - Intel
- Tomer Schwartz - Intel
- Victor Eruhimov - Itseez3D
- Chuck Pilkington - Synopsis
- Jesse Villarreal - Texas Instruments
- 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/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 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.

OpenVX objects (tensors, scalars, enums) for weights, biases and other static parameters of CNN layers must have actual data loaded into them before `vxVerifyGraph()` is called, therefore implementation may cache them prior to execution or use them for other optimizations. Optionally, implementation may explicitly define support to change weights after `vxVerifyGraph()` was called or between `vxProcessGraph()` calls. For convenience we tag [static] the parameters that must have actual data loaded into them before `vxVerifyGraph()`.

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.local_response_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

- vx_nn_convolution_params_t
- vx_nn_deconvolution_params_t
- vx_nn_roi_pool_params_t

Macros

- VX_LIBRARY_KHR_NN_EXTENSION

Enumerations

- vx_kernel_nn_ext_e
- vx_nn_activation_function_e
- vx_nn_enum_e
- vx_nn_norm_type_e
- vx_nn_pooling_type_e
- vx_nn_rounding_type_e
- vx_nn_type_e

Functions

- vxActivationLayer
- vxConvolutionLayer
- vxDeconvolutionLayer
- vxFullyConnectedLayer
- vxLocalResponseNormalizationLayer
- vxPoolingLayer
- vxROIPoolingLayer
- vxSoftmaxLayer

2.1.1. Data Structures

vx_nn_convolution_params_t

Input parameters for a convolution operation.

```
typedef struct _vx_nn_convolution_params_t {
    vx_size    padding_x;
    vx_size    padding_y;
    vx_enum    overflow_policy;
    vx_enum    rounding_policy;
    vx_enum    down_scale_size_rounding;
    vx_size    dilation_x;
    vx_size    dilation_y;
} vx_nn_convolution_params_t;
```

Data Field	Definition
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.
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.
down_scale_size_rounding	Rounding method for calculating output dimensions. See vx_nn_rounding_type_e.
overflow_policy	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
padding_x	Number of elements added at each side in the x dimension of the input.
padding_y	Number of elements added at each side in the y dimension of the input.
rounding_policy	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.

vx_nn_deconvolution_params_t

Input parameters for a deconvolution operation.

```
typedef struct _vx_nn_deconvolution_params_t {
    vx_size    padding_x;
    vx_size    padding_y;
    vx_enum    overflow_policy;
    vx_enum    rounding_policy;
    vx_size    a_x;
    vx_size    a_y;
} vx_nn_deconvolution_params_t;
```

Data Field	Definition
a_x	user-specified quantity used to distinguish between the $upscale_x$ different possible output sizes.
a_y	user-specified quantity used to distinguish between the $upscale_y$ different possible output sizes.
overflow_policy	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
padding_x	Number of elements subtracted at each side in the x dimension of the output.
padding_y	Number of elements subtracted at each side in the y dimension of the output.
rounding_policy	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.

vx_nn_roi_pool_params_t

Input parameters for ROI pooling operation.

```
typedef struct _vx_nn_roi_pool_params_t {
    vx_enum    pool_type;
} vx_nn_roi_pool_params_t;
```

Data Field	Definition
pool_type	Of type vx_nn_pooling_type_e. Only VX_NN_POOLING_MAX pooling is supported.

2.1.2. Macros

VX_LIBRARY_KHR_NN_EXTENSION

The Neural Network Extension Library Set.

```
#define VX_LIBRARY_KHR_NN_EXTENSION       (0x1)
```

2.1.3. Enumerations

vx_kernel_nn_ext_e

The list of Neural Network Extension Kernels.

```
enum vx_kernel_nn_ext_e {
    VX_KERNEL_CONVOLUTION_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x0,
    VX_KERNEL_FULLY_CONNECTED_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x1,
    VX_KERNEL_POOLING_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x2,
    VX_KERNEL_SOFTMAX_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x3,
    VX_KERNEL_NORMALIZATION_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x4,
    VX_KERNEL_ACTIVATION_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x5,
    VX_KERNEL_ROI_POOLING_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x6,
    VX_KERNEL_DECONVOLUTION_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_NN_EXTENSION) + 0x7,
    VX_KERNEL_LOCAL_RESPONSE_NORMALIZATION_LAYER = VX_KERNEL_BASE(VX_ID_KHRONOS,
VX_LIBRARY_KHR_NN_EXTENSION) + 0x8,
};
```

Enumerator

- VX_KERNEL_CONVOLUTION_LAYER - The Neural Network Extension convolution Kernel.

  See also: Extension: Deep Convolutional Networks API

- VX_KERNEL_FULLY_CONNECTED_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_LOCAL_RESPONSE_NORMALIZATION_LAYER - The Neural Network Extension Local Response 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_ROI_POOLING_LAYER - The Neural Network POI 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

vx_nn_activation_function_e

The Neural Network activation functions list.

```
enum vx_nn_activation_function_e {
    VX_NN_ACTIVATION_LOGISTIC = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x0,
    VX_NN_ACTIVATION_HYPERBOLIC_TAN = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) +
0x1,
    VX_NN_ACTIVATION_RELU = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x2,
    VX_NN_ACTIVATION_BRELU = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x3,
    VX_NN_ACTIVATION_SOFTRELU = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x4,
    VX_NN_ACTIVATION_ABS = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x5,
    VX_NN_ACTIVATION_SQUARE = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x6,
    VX_NN_ACTIVATION_SQRT = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x7,
    VX_NN_ACTIVATION_LINEAR = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE) + 0x8,
};
```

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	$f(x) = \lvert x \rvert$		
square	$f(x) = x^2$		
square root	$f(x) = \sqrt{x}$		
linear	$f(x) = a x + b$	a, b	VX_FLOAT32

Enumerator

- VX_NN_ACTIVATION_LOGISTIC
- VX_NN_ACTIVATION_HYPERBOLIC_TAN
- VX_NN_ACTIVATION_RELU
- VX_NN_ACTIVATION_BRELU
- VX_NN_ACTIVATION_SOFTRELU
- VX_NN_ACTIVATION_ABS
- VX_NN_ACTIVATION_SQUARE
- VX_NN_ACTIVATION_SQRT
- VX_NN_ACTIVATION_LINEAR

vx_nn_enum_e

NN extension type enums.

```
enum vx_nn_enum_e {
    VX_ENUM_NN_ROUNDING_TYPE = 0x1A,
    VX_ENUM_NN_POOLING_TYPE = 0x1B,
    VX_ENUM_NN_NORMALIZATION_TYPE = 0x1C,
    VX_ENUM_NN_ACTIVATION_FUNCTION_TYPE = 0x1D,
};
```

Enumerator

- 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

The Neural Network normalization type list.

```
enum vx_nn_norm_type_e {
    VX_NN_NORMALIZATION_SAME_MAP = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_NORMALIZATION_TYPE) + 0x0,
    VX_NN_NORMALIZATION_ACROSS_MAPS = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_NORMALIZATION_TYPE) + 0x1,
};
```

Enumerator

- VX_NN_NORMALIZATION_SAME_MAP - normalization is done on same IFM
- VX_NN_NORMALIZATION_ACROSS_MAPS - Normalization is done across different IFMs.

vx_nn_pooling_type_e

The Neural Network pooling type list.

```
enum vx_nn_pooling_type_e {
    VX_NN_POOLING_MAX = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_POOLING_TYPE) + 0x0,
    VX_NN_POOLING_AVG = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_POOLING_TYPE) + 0x1,
};
```

kind of pooling done in pooling function

Enumerator

- VX_NN_POOLING_MAX - max pooling
- VX_NN_POOLING_AVG - average pooling

vx_nn_rounding_type_e

down scale rounding.

```
enum vx_nn_rounding_type_e {
    VX_NN_DS_SIZE_ROUNDING_FLOOR = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ROUNDING_TYPE) + 0x0,
    VX_NN_DS_SIZE_ROUNDING_CEILING = VX_ENUM_BASE(VX_ID_KHRONOS, VX_ENUM_NN_ROUNDING_TYPE) + 0x1,
};
```

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

vx_nn_type_e

The type enumeration lists all NN extension types.

```
enum vx_nn_type_e {
    VX_TYPE_NN_CONVOLUTION_PARAMS = 0x025,
    VX_TYPE_NN_DECONVOLUTION_PARAMS = 0x026,
    VX_TYPE_NN_ROI_POOL_PARAMS = 0x027,
};
```

Enumerator

- VX_TYPE_NN_CONVOLUTION_PARAMS - A vx_nn_convolution_params_t.
- VX_TYPE_NN_DECONVOLUTION_PARAMS - A vx_nn_deconvolution_params_t.
- VX_TYPE_NN_ROI_POOL_PARAMS - A vx_nn_roi_pool_params_t.

2.1.4. Functions

vxActivationLayer

[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 vxActivationLayer(
    vx_graph                                graph,
    vx_tensor                               inputs,
    vx_enum                                 function,
    vx_float32                              a,
    vx_float32                              b,
    vx_tensor                               outputs);
```

Parameters

- [in] *graph* - The handle to the graph.
- [in] *inputs* - 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] *function* - [static] 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] *a* - [static] Function parameters a. must be positive.
- [in] *b* - [static] Function parameters b. must be positive.
- [out] *outputs* - The output tensor data. Output will have the same number of dimensions as input.

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

vxConvolutionLayer

[Graph] Creates a Convolutional Network Convolution Layer Node.

```
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);
```

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:

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

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 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. In case of ambiguity in the inverse calculation of the skip. The minimum solution is chosen. Skip must be a positive non zero integer. rounding is done according to `vx_nn_rounding_type_e`. Notice that this node creation function has more parameters than the corresponding kernel. Numbering of kernel parameters (required if you create this node using the generic interface) is explicitly specified here.

Parameters

- `[in]` *graph* - The handle to the graph.
- `[in]` *inputs* - 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"`. (Kernel parameter #0)

- `[in]` *weights* - [static] Weights are 4d tensor with dimensions [kernel_x, kernel_y, #IFM, #OFM]. see `vxCreateTensor` and `vxCreateVirtualTensor`

  Weights data type must match the data type of the inputs. (Kernel parameter #1)

- `[in]` *biases* - [static] 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. (Kernel parameter #2)
- `[in]` *convolution_params* - [static] Pointer to parameters of type `vx_nn_convolution_params_t`. (Kernel parameter #3)
- `[in]` *size_of_convolution_params* - [static] Size in bytes of convolution_params. Note that this parameter is not counted as one of the kernel parameters.
- `[out]` *outputs* - 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. (Kernel parameter #4)

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

vxGetStatus.

vxDeconvolutionLayer

[Graph] Creates a Convolutional Network Deconvolution 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);
```

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:

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

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. 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:

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

and

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

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. a_x and a_y must be positive and smaller then $upscale_x$ and $upscale_y$ respectively. Since the padding parameter is on the output, the effective input padding is:

$$padding_{input_x} = kernel_x - padding_x - 1$$

$$padding_{input_y} = kernel_y - padding_y - 1$$

Therfore the following constraints apply: $kernel_x \geq padding_x - 1$ and $kernel_y \geq padding_y - 1$. Rounding is done according to vx_nn_rounding_type_e. Notice that this node creation function has more parameters than the corresponding kernel. Numbering of kernel parameters (required if you create this node using the generic interface) is explicitly specified here.

Parameters

- [in] *graph* - The handle to the graph.
- [in] *inputs* - 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 vxCreateTensor and vxCreateVirtualTensor. Implementations must support input tensor data types indicated by the extension strings "KHR_NN_8" or "KHR_NN_8 KHR_NN_16". (Kernel parameter #0)
- [in] *weights* - [static] The 4d weights with dimensions [width, height, #IFM, #OFM]. See vxCreateTensor and vxCreateVirtualTensor. (Kernel parameter #1)
- [in] *biases* - [static] Optional, ignored if NULL. The biases have one dimension [#OFM]. Implementations must support input tensor data type same as the inputs. (Kernel parameter #2)
- [in] *deconvolution_params* - [static] Pointer to parameters of type vx_nn_deconvolution_params_t (Kernel parameter #3)
- [in] *size_of_deconv_params* - [static] Size in bytes of deconvolution_params. Note that this parameter is not counted as one of the kernel parameters.
- [out] *outputs* - The output tensor. The output has the same number of dimensions as the input. (Kernel parameter #4)

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

vxFullyConnectedLayer

[Graph] 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);
```

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:

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

Where j is a index on the input feature and i is a index on the output.

There two possible input tensor 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".

Parameters

- [in] *graph* - The handle to the graph.
- [in] *inputs* - The input tensor data.
- [in] *weights* - [static] 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* - [static] Optional, ignored if NULL. The biases have one dimension [#OFM]. Implementations must support input tensor data type same as the inputs.
- [in] *overflow_policy* - [static] A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
- [in] *rounding_policy* - [static] 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: A node reference vx_node. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vxLocalResponseNormalizationLayer

[Graph] Creates a Convolutional Network Local Response Normalization Layer Node. This function is optional for 8-bit extension with the extension string "KHR_NN_8".

```
vx_node vxLocalResponseNormalizationLayer(
    vx_graph                                graph,
    vx_tensor                               inputs,
    vx_enum                                 type,
    vx_size                                 normalization_size,
    vx_float32                              alpha,
    vx_float32                              beta,
    vx_float32                              bias,
    vx_tensor                               outputs);
```

Local Response Normalizing over local input regions. Each input value is divided by

$(bias + \frac{\alpha}{n}\sum_i x_i^2)^\beta$

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] *graph* - The handle to the graph.
- [in] *inputs* - 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 vxCreateTensor and vxCreateVirtualTensor. 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 VX_TYPE_INT16 with fixed_point_position 8.
- [in] *type* - [static] Either same map or across maps (see vx_nn_norm_type_e).
- [in] *normalization_size* - [static] Number of elements to normalize across. Must be a positive odd number with maximum size of 7 and minimum of 3.
- [in] *alpha* - [static] Alpha parameter in the local response normalization equation. must be positive.
- [in] *beta* - [static] Beta parameter in the local response normalization equation. must be positive.
- [in] *bias* - [static] Bias parameter in the local response normalization equation. must be positive.
- [out] *outputs* - The output tensor data. Output will have the same number of dimensions as input.

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

vxPoolingLayer

[Graph] 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);
```

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] *graph* - The handle to the graph.
- [in] *inputs* - 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 vxCreateTensor and vxCreateVirtualTensor Implementations must support input tensor data types indicated by the extension strings

"KHR_NN_8" or "KHR_NN_8 KHR_NN_16".

- [in] *pooling_type* - [static] Either max pooling or average pooling (see vx_nn_pooling_type_e).
- [in] *pooling_size_x* - [static] Size of the pooling region in the x dimension
- [in] *pooling_size_y* - [static] Size of the pooling region in the y dimension.
- [in] *pooling_padding_x* - [static] Padding size in the x dimension.
- [in] *pooling_padding_y* - [static] Padding size in the y dimension.
- [in] *rounding* - [static] Rounding method for calculating output dimensions. See vx_nn_rounding_type_e
- [out] *outputs* - 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: A node reference vx_node. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vxROIPoolingLayer

[Graph] Creates a Convolutional Network ROI pooling 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);
```

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. Notice that this node creation function has more parameters than the corresponding kernel. Numbering of kernel parameters (required if you create this node using the generic interface) is explicitly specified here.

Parameters

- [in] *graph* - The handle to the graph.
- [in] *inputs* - 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 vxCreateTensor and vxCreateVirtualTensor. Implementations must support input tensor data types indicated by the extension strings "KHR_NN_8" or "KHR_NN_8 KHR_NN_16". (Kernel parameter #0)
- [in] *inputs_rois* - 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. (Kernel parameter #1)
- [in] *pool_type* - [static] Of type vx_nn_pooling_type_e. Only VX_NN_POOLING_MAX pooling is supported. (Kernel parameter #2)
- [in] *size_of_roi_params* - [static] Size in bytes of roi_pool_params. Note that this parameter is not counted as one of the kernel parameters.

- [out] *output_arr* - The output tensor. Output will have [output_width, output_height, #IFM, #batches] dimensions. #batches is optional and must be the same as in inputs. (Kernel parameter #3)

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

vxSoftmaxLayer

[Graph] Creates a Convolutional Network Softmax Layer Node.

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

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 σ(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] *graph* - The handle to the graph.
- [in] *inputs* - 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 vxCreateTensor and vxCreateVirtualTensor. 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] *outputs* - The output tensor. Output will have the same number of dimensions as input. Output tensor data type must be same as the inputs.

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