cadence®

HiFi 5 Neural Network Library

Programmer's Guide - API

For HiFi DSPs

Version: 2.3
Last Updated: September 2023

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

Figures

Tables

Abbreviations

CNN	Convolutional Neural Networks
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
TFLM	TensorFlow Lite for Micro-controllers
VFPU	Vector Floating Point Unit
LSH	Locality Sensitive Hashing
RNN	Recurrent Neural Network
SVDF	Singular Value Decomposition Filters

Document Change History

Version	Changes
1.0	Initial version
1.1	Added quantized 8 bit variants for depthwise convolution, fully connected and softmax
1.2	Added quantized 8 bit kernels for SVDF support and for standard convolution, average pooling, and quantization.
1.3	Added quantized 8 bit kernels (asymmetric int8) for pointwise convolution, max pooling, elementwise addtion and multiplication.
1.4	Added description of 8 bit kernels (asymmetric int8) for elementwise compare operations, elementwise subtraction, maximum, minimum, tanh, prelu and hardswish.
1.5	Added description of logical operation kernels (Boolean 1-byte) for elementwise logical_and, logical_or, logical_not operations. Added per channel quantized sym8s kernel, asym8s input, asym8s output – dilated 2D convolution (without stride support). Added desciption of L2 Normalization, reduce mean, reduce max kernels (asymmetric int8). Added description of boradcast variants of elementwise maximum, minumum kernels.
1.6	Updated Tensorflow Lite For Microcontrollers (TFLM) operator support table with newly supported operators. Added a separate table for TFLM operators which are optimized without any NNLib kernels. Standard 2D convolution with Dilation description is updated to support stride. Added description for matXvec batch kernels with accumulation. Added descriptions for 16-bit input/output sigmoid and tanh. Added description for following quantized int8 kernel: leaky_relu. Elementwise Quantize kernels are renamed to Elementwise Requantize, two new variants are also added. Added description for Elementwise Dequantize kernels (quantized int8 to float32). Added descriptions for following float32 kernels: add, abs, sin, cos, log, sqrt, rsqrt, square, fill, ceil, round, neg. Added description for 8-bit broadcast kernel. Added descriptions for following memory operation kernels: memset (float32), memmove.

		L2 Normalization kernel description moved to "Normalization Kernels" section from "Basic Operations and Miscellaneous Kernels".
		Added description for following 8-bit reorg kernels: depth_to_space, space_to_depth, pad, batch_to_space, space_to_batch.
		Added sample testbench descriptions for reorg sample testbench. Updated matXvec, conv, activation, basic testbench descriptions.
1.7		Updated the API for dilated conv2d std getsize function.
		Created separate performance document, and removed the performance data from this document.
1.8		Updated Tensorflow Lite For Microcontrollers (TFLM) operator support table with newly supported operators and precisions.
		Single Rounding support for Tensorflow Lite Micro operators' quantized datatype kernels.
		Improved optimization for TensorFlow Lite Micro variants of Sigmoid and Tanh kernels.
		Matrix X Vector Multiplication and Fully Connected kernels added with asym8sxasym8s_asym8s datatype support.
		Matrix Multiplication kernels added with per_chan_sym8sxsym16s_sym16s and asym8sxasym8s_asym8s datatype support.
		Added Transpose Convolution and Strided Slice kernels for Int16 datatype.
		Added Int16 support for following kernels : Standard 2D Convolution, Pointwise 2D Convolution, Leaky ReLU, Pad.
		Added Quantize single precision float to quantized Int8 and Requantize quantized Int8 to quantized Int8 kernels.
		Added following quantized datatype elementwise kernels with 4D broadcasting: Add (Int8 and Int16), Sub (Int8 and Int16), Mul (Int8), Squared Diff (Int8).
		Updated matXvec, conv, activation, basic and reorg testbench descriptions.
1.9	■	Added matXvec, fully connected, conv2d_depth for sym8sxsym16s_sym16s
	■	Added elm_requantize_asym16s_asym16s, strided_slice_int8
		Updated Tensorflow Lite For Microcontrollers (TFLM) operator support table with newly supported operators and precisions.
2.0	■	Added get_softmax_scratch_size helper API in softmax section. Reviewed and corrected some minor errors/typos.
	■	Updated the TFLM operator support table. Also sorted the table alphabetically.
	■	Added matmul 8x16_16, 16x16_16 and f32xf32_f32 variants.
2.1	■	Added sigmoid and tanh kernels for sym16sxsym16s precisions.
	■	Added matmul kernel for sym8sxsym16s_sym16s precision.
	■	Added elm_mul_broadcast_4D_sym16sxsym16s_sym16s, elm_dequantize_asym16s_f32, elm_quantize_f32_asym16s, elm_sub_broadcast_4D_f32xf32_f32 kernels.

	■ Added transpose_8_8, pad_32_32, strided_slice_int32 kernels. ■ Added dilated_conv2d_depthwise kernel for f32 and sym8sxasym8s precisions. Also added dilated_conv2d_depthwise_getsize helper API for this kernel. Also added transpose_conv_f32 kernel. ■ Added LSTM helper API kernels elm_add_16x16_16, elm_mul_sym16sxsym16s_asym8s, lstm_cell_state_update_16. ■ Updated Error codes for GRU API. Added pytorch equations for GRU.
2.2	■ Added kernel transpose convoluion kernel for sym8sxasym8s_asym8s precision. ■ Added kernel squared difference broadcast 4D for precision sym16sxsym16s_sym16s. ■ Added kernels reduce mean 4D & reduce max 4D for asym16s precision. ■ Added memmove kernel for int16 precision.
2.3	■ Added kernels fully_connected, conv2d_std, conv2d_depthwise, matmul and conv2d_pointwise for half precision floating point (f16) precision ■ Added kernel fully_connected for asym4sxasym8s_asym8s precision, and conv2d_std for sym4sxasym8s_asym8s precision. ■ Added kernel conv2d_group (for group convolution) for sym8sxasym8s_asym8s precision

1. Introduction to the HiFi 5 NN Library

The HiFi 5 Neural Network (NN) Library is a HiFi-optimized implementation of various NN layers and low-level NN kernels. The library is designed with speech and audio neural network domain focus. The low-level NN kernels are HiFi-optimized building blocks for NN layer implementation with a generic and simple interface. The NN layers are built using low-level kernels and accept input in the form of 'shapes'[1] (up to four dimensions) and produce the output, also in the form of shapes. The layers use the weights or coefficients and biases stored 'externally'[2] for their operation. The shape of the input, output, weights, and biases are as per the layer's design.

This guide refers to the HiFi 5 NN Library as HiFi NN Library, NN layers simply as layers and low-level NN kernels as low-level kernels. The current version of the library implements GRU, LSTM (forward path), and CNN layers. It also implements matrix vector multiply, activation, pooling, convolution, fully connected, basic operation, normalization, and reorg functions as low-level kernels.

Note	This version of the library supports HiFi 5 DSPs with the NN Extension enabled. The SP-VFPU (Single Precision Vector Floating Point Unit) is optional. The library can be compiled for HiFi 5 DSPs with or without the SP-VFPU enabled.
Note	This version of the HiFi5 NN Library is tested with the xt-clang/xt-clang++ compilers using Xtensa Software Tools from RI-2022.9 release.

1.1   Organization of the HiFi 5 NN Library Package

The HiFi NN Library package includes the HiFi NN library containing all layers and low-level kernels implementations, and a set of sample test applications (for layers and low-level kernels).

The HiFi NN library implements a set of NN layers. The application can instantiate these layers and connect inputs and outputs across the layers to form a Neural Network system.

The HiFi NN library also provides a set of low-level NN kernels. The application can use these kernels to implement or optimize performance of other NN layers.

The sample test applications implement a file-based application to test an instance of a layer or low-level NN kernels for the given specification using pre-generated input, weight or coefficients and bias shapes stored in files in raw binary format.

---

[1] Refer to Section 2.1 Shape

[2] Refer to Section 2.2.3 Weights and Biases Memory

1.1.1 Document Overview

This document covers all the information required to integrate the HiFi NN Library into a Neural Network system. All the layers implement "HiFi NN layer APIs", which is generic and explained in Section 2. The low-level NN kernels are explained in Section 3. Section 4 describes the APIs for each layer. Section 5 provides details about available sample testbenches. Section 6 lists the references.

1.2   HiFi 5 NN Library Specification

The current version of the HiFi NN Library provides the following HiFi-optimized low-level kernels and layer implementations.

1.2.1 Low Level Kernels

- Matrix-vector multiplication kernels
- Convolution kernels
- Activation kernels
- Pooling kernels
- Basic operations kernels
- Fully connected kernel
- Normalization kernel
- Reorg kernels

These kernels support fixed point 8 bit, 16 bit, and single precision floating point data types for weights or coefficients, biases, input, and output. For more information, see Section 3.

Additionally, 8-bit and 16-bit quantized datatypes as defined in TensorFlow (TF), TensorFlow Lite for Microcontrollers (TFLM) are also supported for select kernels [3]. These datatypes use 8-bit/16-bit quantized values (asym8u – asymmetric 8-bit unsigned, asym8s – asymmetric 8-bit signed, sym8s – symmetric 8-bit signed) for weights or coefficients, input, and output. Biases are 32-bit quantized values.

8-bit quantized types are either unsigned (0, 255) or signed (-128, 127) 8-bit integer with 3 additional parameters.

Three numbers are associated with a quantized 8-bit value that can be used to convert the 8-bit integer to the real value and vice versa. These numbers are:

- Shift: an integer value indicating the amount of shift. If the value is positive, it is left shift and if negative, it is right shift
- Multiplier: a 32 bit (Q31) fixed point value greater than zero.
- Zero point: a 32 bit integer, in range [0, 255] for unsigned type, in range [-128, 127] for signed type.

The formula is:

$$real\_value = (quantized\_value - zero\_point) * 2^{shift} * multiplier$$

The 'sym8s' type is symmetrical around 0, this means that quantized values are between -127 to 127 and zero point is 0, so all the calculation required due to zero point is avoided.

To match the asym8u/asym8s/sym8s APIs with TensorFlow, we define zero point as zero_bias in the NN library APIs. The zero_bias is an integer value having range asym8u - [0, 255], asym8s – [-128, 127] (or asym8u - [-255, 0], asym8s – [-127, 128] in case of the reverse operation depending on the corresponding TensorFflow kernel).

In addition to the quantized 8-bit datatypes, a similar 16-bit quantized datatype (asym16s) is used for few kernels. The zero_bias for asym16s datatype is an integer value having range – [-32768, 32767].

1.2.2 Layers

- GRU layer (8x16, 16x16 precision)
- LSTM (forward path) layer (8x16, 16x16 precision)
- CNN layer (8x8, 8x16, 16x16, and float32xfloat32 precision)

Note   MxN precision above denotes (weights or coefficients) x (input, output, bias) precision. For more information, see Section 4.

1.2.3 Support for TensorFlow Lite Micro Operators

The HiFi 5 NN Library low-level kernels can be used to implement the following operators of TensorFlow Lite Micro. HiFi5 NN Library supports both rounding modes available in TensorFlow Lite Micro for applicable operators[5.1] :

No.	Operator	Float32 Datatype Support	Uint8 (asymmetric quantized uint8) Datatype Support	Int8 (quantized int8) Datatype Support	Int16 (quantized int16) Datatype Support	Boolean (1 Byte) Datatype Support
1	ABS	Yes				
2	ADD	Yes		Yes	Yes	
3	AVERAGE_POOL_2D	Yes	Yes	Yes		
4	BATCH_TO_SPACE_ND			Yes		
5	CEIL	Yes				
6	CIRCULAR_BUFFER			Yes		
7	CONV_2D	Yes	Yes	Yes	Yes	
8	COS	Yes				
9	DEPTH_TO_SPACE			Yes		

No.	Operator	Float32 Datatype Support	Uint8 (asymmetric quantized uint8) Datatype Support	Int8 (quantized int8) Datatype Support	Int16 (quantized int16) Datatype Support	Boolean (1 Byte) Datatype Support
10	DEPTHWISE_CONV_2D	Yes	Yes	Yes		
11	DEQUANTIZE			Yes[3]	Yes	
12	EQUAL			Yes		
13	FILL	Yes				
14	FLOOR	Yes				
15	FULLY_CONNECTED	Yes	Yes	Yes	Yes	
16	GREATER			Yes		
17	GREATEREQUAL			Yes		
18	HARDSWISH			Yes		
19	L2 NORM	Yes		Yes		
20	LEAKY_RELU			Yes	Yes	
21	LESS			Yes		
22	LESSEQUAL			Yes		
23	LOG	Yes				
24	LOGICALAND					Yes
25	LOGICALNOT					Yes
26	LOGICALOR					Yes
27	LOGISTIC	Yes		Yes	Yes	
28	MAX_POOL_2D	Yes		Yes		
29	MAXIMUM			Yes		
30	MEAN			Yes		
31	MINIMUM			Yes		
32	MUL	Yes		Yes	Yes	
33	NEG	Yes				
34	NOTEQUAL			Yes		
35	PAD	Yes		Yes	Yes	
36	PADV2			Yes	Yes	
37	PRELU			Yes		
38	QUANTIZE[4]			Yes	Yes	
39	REDUCEMAX			Yes	Yes	
40	RELU	Yes	Yes	Yes		
41	RELU6	Yes	Yes	Yes		
42	ROUND	Yes				

[3] For TFLM DEQUANTIZE operator output is always single precision float whereas multiple input data types are supported. HiFi 5 NN Library has kernel for quantized Int8 and quantized Int16 input data type.

[4] QUANTIZE operator has different input and output quantized data types, HiFi5 NN Library has kernels for Int16 to Int8, Int8 to Int32, Int16 to Int32, int8 to int8, Float32 to Int8, Float32 to Int16 and Int16 to Int16.

No.	Operator	Float32 Datatype Support	Uint8 (asymmetric quantized uint8) Datatype Support	Int8 (quantized int8) Datatype Support	Int16 (quantized int16) Datatype Support	Boolean (1 Byte) Datatype Support
43	RSQRT	Yes				
44	SIN	Yes				
45	SOFTMAX		Yes	Yes		
46	SPACE_TO_BATCH_ND			Yes		
47	SQRT	Yes				
48	SQUARE	Yes				
49	SQUARED_DIFF			Yes	Yes	
50	STRIDED_SLICE	Yes		Yes	Yes	
51	SUB	Yes		Yes	Yes	
52	SVDF			Yes		
53	TANH	Yes		Yes	Yes	
54	TRANSPOSE			Yes		
55	TRANSPOSE_CONV	Yes		Yes	Yes[5]	
56	UnidirectionSequenceLSTM			Yes		

The following TFLM operators get optimized out of box on HiFi 5 and do not require any HiFi 5 NNLib kernels:

No.	Operator	Float32 Datatype Support	Uint8 (asymmetric quantized uint8) Datatype Support	Int8 (quantized int8) Datatype Support	Int32	Int64	Boolean (1 Byte) Datatype Support
1	PACK	Yes	Yes	Yes	Yes	Yes	
2	EXPAND_DIMS	Yes		Yes			
3	RESHAPE[6]						
4	ELU			Yes			
5	SQUEEZE[6]						

[5] Two variants are available – sym8s kernel with sym16s input, float 32 bit kernel with float 32 bit output.

[6] For RESHAPE and SQUEEZE datatype is not specified in TensorFlow Lite Micro.

2. Generic HiFi NN Layer API

Note	This section explains an API standard that is evolving. The APIs may undergo some changes in future versions.

This section describes the API that is common to all the HiFi NN layers. The API facilitates any layer instance that works in the overall method shown in Figure 2-1.
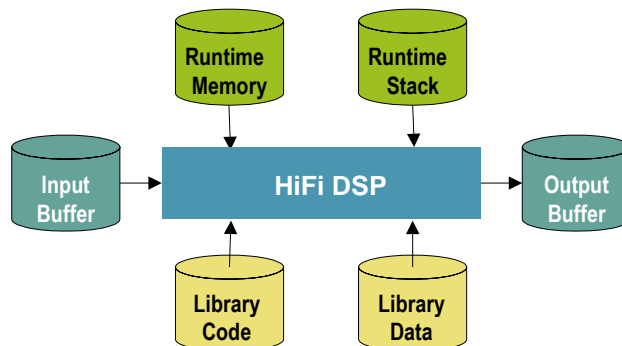
Figure 2-1 HiFi NN Layer Interfaces

All the buffers, input, output, weights, and biases are described as shapes. Section 2.1 explains the shape structure.

Section 2.2 discusses all the types of runtime memory required by the layer instances. There is no state information held in static memory, therefore a single thread can perform time division processing of multiple layer instances. Additionally, multiple threads can perform concurrent layer instance processing.

The output from one instance can be fed as input to the next instance if the precision and the dimension matches.

The data types, structures, and error codes explained in this section are declared/defined in `xa_nnlib_standard.h`. By default, the API header file of each layer includes this header file. The application need not include this file.

2.1 Shape

The shapes are used to describe any buffer used in the NN library. The structure xa_nnlib_shape_t is defined in `xa_nnlib_standard.h`. The shape can be vector, matrix, or cube.

- Vector is a one-dimensional shape specified by length.

- Matrix is a two-dimensional shape specified by rows, columns, and row_offset. This assumes that the elements in a row are stored at consecutive addresses in memory.

- Cube is a three-dimensional shape specified by height, width, depth, height_offset, width_offset, and depth offset. Cube supports the following shape types:

  - SHAPE_CUBE_DWH_T

    This assumes that elements are stored in depth, width, and height order; that is, elements with the same height and width indices are stored consecutively.

  - SHAPE_CUBE_WHD_T

    This assumes that elements are stored in width, height, and depth order; that is, elements with the same height and depth are stored consecutively.

The Figure 2-2 shows the dimension variables of matrix and cube shapes.
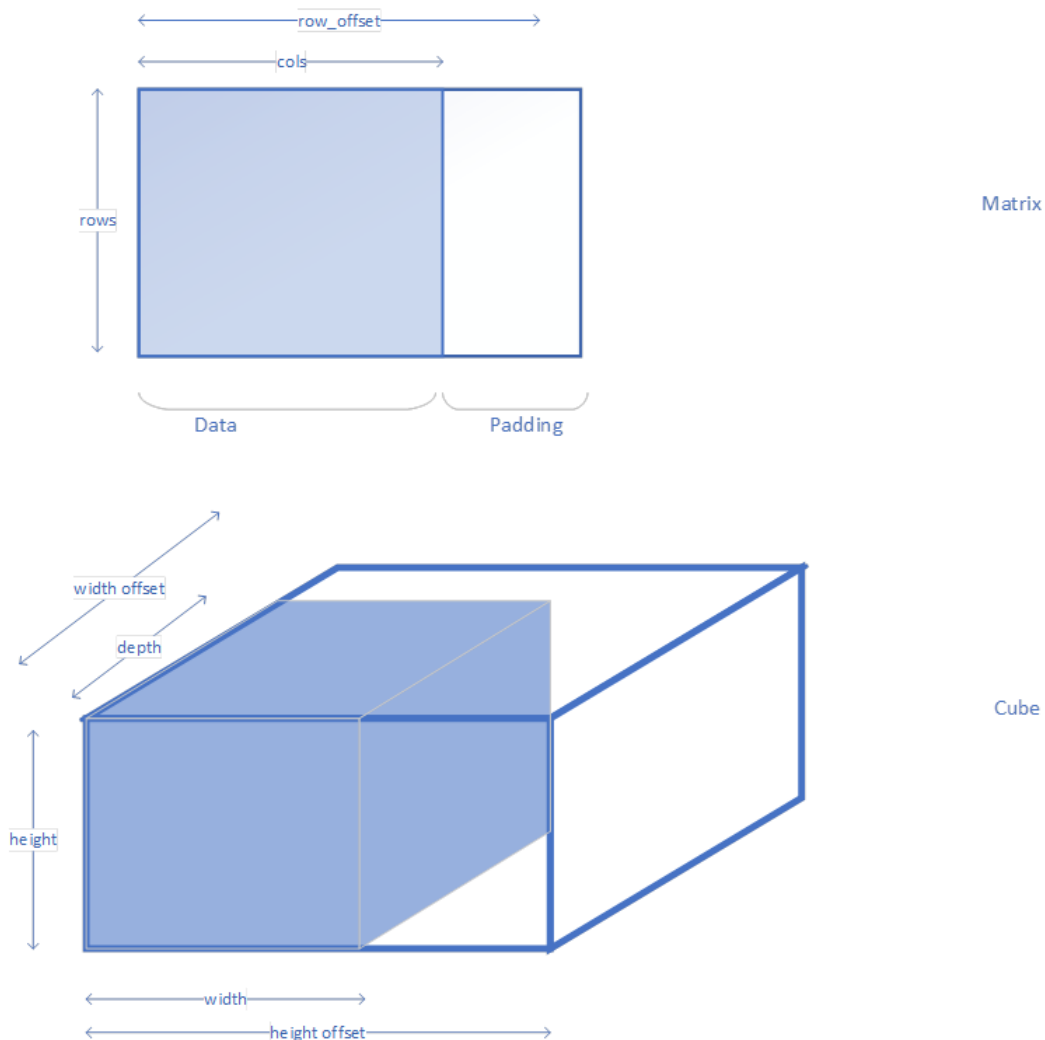
Figure 2-2 Matrix and Cube (SHAPE_CUBE_DWH_T) Shape Representation

2.2   Memory Management

The HiFi NN layer API supports a flexible memory scheme and a simple interface that eases the integration into the final application. The API allows the layers to request the required memory for their operations during runtime.

The runtime memory requirement consists primarily of the scratch and persistent memory. The components also require an input buffer and output buffer for the passing of data into and out of the layer.

2.2.1 API Handle / Persistent Memory

The layer API stores persistent state information in a structure that is referenced via an opaque handle. The handle is passed by the application for each API call. This object contains all state and history information that is maintained from one-layer frame invocation to the next within the same thread or instance. The layers expect that the contents of the persistent memory be unchanged by the system apart from the layer itself for the complete lifetime of the layer.

2.2.2 Scratch Memory

The Scratch Memory is the temporary buffer used by the layer during a single frame processing call. The contents of this memory region must not be changed if the actual layer execution process is active; that is, if the thread running the layer is inside any API call. This region can be used freely by the system between successive calls to the layer.

2.2.3 Weights and Biases Memory

The weights or coefficients and biases must be managed by the application, and memory must not be requested by the API. If the design requires DMA access from or to the internal memory for better performance, a ping-pong or circular buffer is allocated as part of the scratch into which the weights, biases, input, and output are copied using DMA. If required, these memories can also be persistent.

2.2.4 Input Buffer

The Input buffer is the buffer from which the layer reads the input. This buffer must be made available for the layer before its execution call. The input buffer must have an associated shape information to describe the input data format. The input buffer pointer can be changed by the application between calls to the layer, but shape information cannot be changed. This allows the layer to read directly from the output of another layer.

2.2.5 Output Buffer

The Output buffer is the buffer to which the layer writes the output. This buffer must be made available for the layer before its execution call. The output buffer must have an associated shape information to which

the layer can describe the output data format. The output buffer pointer can be changed by the application between calls to the layer. This allows the layer to write directly to the input of another layer.

2.3   Generic API Errors

The Layer API functions return an error code of type `Int32,` which is of type `signed int`. The format of the error codes is defined in the following table.

31	30 − 27	26–12	11 − 7	6 - 0
Fatal	Class	Reserved	Component	Sub code

The errors that can be returned from the API are subdivided into those that are fatal, which require resetting the layer; and those that are nonfatal and are provided for information to the application.

The class of an error can be API, Config, or Execution. The API category errors are concerned with the incorrect use of the API. The Config errors are produced when the layer parameters are incorrect or outside the supported usage. The Execution errors are returned after a call to the main process and indicate situations that have arisen due to the input data.

2.3.1 Common API Errors

The following errors are fatal and must not be encountered during normal application operation. They signal that a serious error has occurred in the application that is calling the layer.

- XA_NNLIB_FATAL_MEM_ALLOC

  At least one of the pointers passed into the API function is NULL.

- XA_NNLIB_FATAL_MEM_ALIGN

  At least one of the pointers passed into the API function is not properly aligned.

- XA_NNLIB_FATAL_INVALID_SHAPE

  At least one of the shapes passed to the API function is invalid.

2.4   C Language API

An overview of the NN layer flow is shown in Figure 2-3. The NN layer API consists of query, initialization, and execution functions.
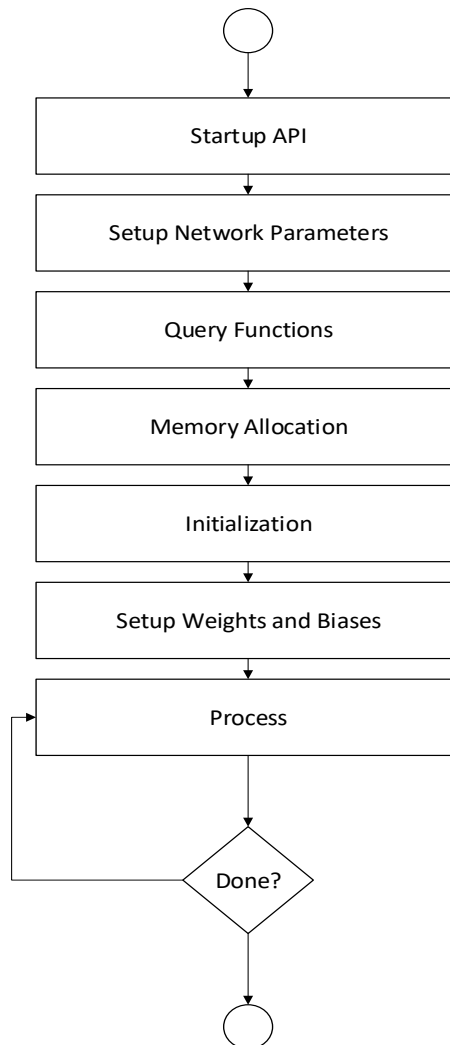
Figure 2-3 NN Layer Flow Overview

2.4.1 Startup Functions

The API startup functions shown in Table 2-1 get the various identification strings from the component library. They are for information only and their usage is optional. These functions do not take any input arguments and return `const char *`.

Table 2-1 Library Identification Functions

Function	Description
`xa_nnlib_get_lib_name_string`	Get the name of the library.
`xa_nnlib_get_lib_version_string`	Get the version of the library.
`xa_nnlib_get_lib_api_version_string`	Get the version of the API.

Example

```
const char *name = xa_nnlib_get_lib_name_string();
const char *ver = xa_nnlib_get_lib_version_string();
const char *aver = xa_nnlib_get_lib_api_version_string();
```

Errors

■ None

2.4.2 Query Functions

The query functions are used in the startup and the memory allocation stages to obtain information about the memory requirements of the library.

The following is the naming convention for query functions:

```
xa_nnlib_<layer>_get_{persistent | scratch}_<placement>
```

Where:

<layer> indicates the module name (such as gru).

<placement> specifies fast or slow.

2.4.3 Initialization Functions

The initialization functions are used to reset the layer to its initial state. Because the layers are fully reentrant, the application can initialize the layer multiple times.

The following is the naming convention for initialization functions:

```
xa_nnlib_<layer>_init
```

2.4.4 Execution Functions

The execution functions are used to generate the output shape by processing one input shape.

The following is the naming convention for execution functions:

```
xa_nnlib_<layer>_process
```

3. HiFi 5 NN Library – Low-Level Kernels

This section explains the low-level kernels provided in the NN library. All the low-level kernels have a generic and simple interface.

The NN library is a single archive containing all low-level kernels and layers implementations. The following sections explain each low-level kernel in detail.

3.1 Matrix X Vector Multiplication Kernels

3.1.1 Matrix X Vector Kernels

Description

The Matrix X Vector kernels perform the dual matXvec operation with bias addition; that is, `z = mat1*vec1 + mat2*vec2 + bias`. The column dimension of `mat1` must match the row dimension of `vec1` and similarly for `mat2`, `vec2`. Bias and resulting output vector `z` have as many rows as `mat1` and `mat2`.

The `bias_shift` and `acc_shift` arguments are provided in the kernel API to adjust Q format of bias and output, respectively. Both `bias_shift` and `acc_shift` can be either positive or negative, where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as matXvec multiplication – accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note   The `acc_shift` and `bias_shift` arguments are not relevant in case of floating-point kernels and quantized 8-bit kernels.

The `row_stride1` and `row_stride2` arguments are provided in kernel API for row offsets of `mat1` and `mat2`, respectively.

Note   The input matrices are expected to be appropriately padded in case of `row_stride > cols`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The arguments, `mat1_zero_bias, mat2_zero_bias, vec1_zero_bias, vec2_zero_bias,` are provided to convert the quantized 8-bit inputs into their real values and perform matXvec operation. The `out_zero_bias, out_multiplier,` and `out_shift` values are used to quantize real values of output back to 8-bit.

The function variants are available as `xa_nn_matXvec_[p]x[q]_[r]`, where:

- ▪ `[p]`: Matrix precision in bits
- ▪ `[q]`: Vector precision in bits
- ▪ `[r]`: Output precision in bits

Precision

The following fourteen variants are available:

Type	Description
`16x16_16`	16-bit matrix inputs, 16-bit vector inputs, 16-bit output
`16x16_32`	16-bit matrix inputs, 16-bit vector inputs, 32-bit output
`16x16_64`	16-bit matrix inputs, 16-bit vector inputs, 64-bit output
`8x16_16`	8-bit matrix inputs, 16-bit vector inputs, 16-bit output
`8x16_32`	8-bit matrix inputs, 16-bit vector inputs, 32-bit output
`8x16_64`	8-bit matrix inputs, 16-bit vector inputs, 64-bit output
`8x8_8`	8-bit matrix inputs, 8-bit vector inputs, 8-bit output
`8x8_16`	8-bit matrix inputs, 8-bit vector inputs, 16-bit output
`8x8_32`	8-bit matrix inputs, 8-bit vector inputs, 32-bit output
`f32xf32_f32`	float32 matrix inputs, float32 vector inputs, float32 output
`asym8uxasym8u_asym8u`	asym8u matrix inputs, asym8u vector inputs, asym8u output
`sym8sxasym8s_asym8s`	sym8s matrix inputs, asym8s vector inputs, asym8s output
`asym8sxasym8s_asym8s`	asym8s matrix inputs, asym8s vector inputs, asym8s output
`sym8sxsym16s_sym16s`	`sym8s matrix inputs, sym16s vector inputs, sym16s output`

Algorithm

$$z_n = 2^{acc\text{-}shift} \left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot vec1_m + \sum_{m=0}^{cols2-1} mat2_{n,m} \cdot vec2_m + 2^{bias\text{-}shift} bias_n \right)$$

For floating-point and quantized 8-bit routines, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

Prototype

```
WORD32 xa_nn_matXvec_16x16_16
(WORD16 * p_out,         WORD16 * p_mat1,        WORD16 * p_mat2,
 WORD16 * p_vec1,        WORD16 * p_vec2,        WORD16 * p_bias,
 WORD32 rows,            WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,     WORD32 row_stride2,
 WORD32 acc_shift,       WORD32 bias_shift);
WORD32 xa_nn_matXvec_16x16_32
(WORD32 * p_out,         WORD16 * p_mat1,        WORD16 * p_mat2,
 WORD16 * p_vec1,        WORD16 * p_vec2,        WORD16 * p_bias,
 WORD32 rows,            WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,     WORD32 row_stride2,
 WORD32 acc_shift,       WORD32 bias_shift);
WORD32 xa_nn_matXvec_16x16_64
(WORD64 * p_out,         WORD16 * p_mat1,        WORD16 * p_mat2,
 WORD16 * p_vec1,        WORD16 * p_vec2,        WORD16 * p_bias,
 WORD32 rows,            WORD32 cols1,           WORD32 cols2,
```

```
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x16_16
(WORD16 * p_out,          WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,         WORD16 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x16_32
(WORD32 * p_out,          WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,         WORD16 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x16_64
(WORD64 * p_out,          WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,         WORD16 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x8_8
(WORD8 * p_out,           WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD8 * p_vec1,          WORD8 * p_vec2,          WORD8 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x8_16
(WORD16 * p_out,          WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD8 * p_vec1,          WORD8 * p_vec2,          WORD8 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_8x8_32
(WORD32 * p_out,          WORD8 * p_mat1,          WORD8 * p_mat2,
 WORD8 * p_vec1,          WORD8 * p_vec2,          WORD8 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,
 WORD32 acc_shift,        WORD32 bias_shift);
WORD32 xa_nn_matXvec_f32xf32_f32
(FLOAT32 * p_out,         const FLOAT32 * p_mat1, const FLOAT32 * p_mat2,
 const FLOAT32 * p_vec1, const FLOAT32 * p_vec2, const FLOAT32 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2);
WORD32 xa_nn_matXvec_asym8uxasym8u_asym8u
(UWORD8 * p_out,          const UWORD8 * p_mat1,  const UWORD8 * p_mat2,
 const UWORD8 * p_vec1,  const UWORD8 * p_vec2,  const WORD32 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,      WORD32 mat1_zero_bias,
 WORD32 mat2_zero_bias,  WORD32 vec1_zero_bias,  WORD32 vec2_zero_bias,
 WORD32 out_multiplier,  WORD32 out_shift,        WORD32 out_zero_bias);
WORD32 xa_nn_matXvec_sym8sxasym8s_asym8s
(WORD8 * p_out,           const WORD8 * p_mat1,   const WORD8 * p_mat2,
 const WORD8 * p_vec1,   const WORD8 * p_vec2,   const WORD32 * p_bias,
 WORD32 rows,             WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,      WORD32 vec1_zero_bias,
 WORD32 vec2_zero_bias,  WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias);
WORD32 xa_nn_matXvec_asym8sxasym8s_asym8s
(WORD8 * p_out,           const WORD8 * p_mat1,   const WORD8 * p_mat2,
 const WORD8 * p_vec1,   const WORD8 * p_vec2,   const WORD32 * p_bias,
```

```
WORD32 rows,            WORD32 cols1,            WORD32 cols2,
WORD32 row_stride1,     WORD32 row_stride2,      WORD32 mat1_zero_bias,
WORD32 vec1_zero_bias,  WORD32 vec2_zero_bias,   WORD32 out_multiplier,
WORD32 out_shift,       WORD32 out_zero_bias);
WORD32 xa_nn_matXvec_sym8sxsym16s_sym16s
(WORD16 * p_out,         const WORD8 * p_mat1,    const WORD8 * p_mat2,
 const WORD16 * p_vec1,  const WORD16 * p_vec2,   const WORD64 * p_bias,
 WORD32 rows,            WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,     WORD32 row_stride2,      WORD32 out_multiplier,
 WORD32 out_shift);
```

Arguments

Type	Name	Size	Description
Input			
`WORD16 *,` `WORD8 *,` `const` `FLOAT32 *` `const` `UWORD8 *,` `const` `WORD8 *`	`p_mat1`	`rows*cols1`	Input matrix 1, fixed, floating point, asym8u or sym8s
`WORD16 *,` `WORD8 *,` `const` `FLOAT32 *` `const` `UWORD8 *,` `const` `WORD8 *`	`p_mat2`	`rows*cols2`	Input matrix 2, fixed, floating point, asym8u or sym8s
`WORD16 *,` `WORD8 *,` `const` `FLOAT32 *` `const` `UWORD8 *,` `const` `WORD8 *`	`p_vec1`	`cols1*1`	Input vector 1, fixed, floating point, asym8u, sym16s or asym8s
`WORD16 *,` `WORD8 *,` `const` `FLOAT32 *` `const` `UWORD8 *,` `const` `WORD8 *`	`p_vec2`	`cols2*1`	Input vector 2, fixed, floating point, asym8u, sym16s or asym8s
`WORD16 *,` `WORD8 *,` `const` `WORD32 *,` `const` `FLOAT32 *,` `const` `WORD64 *`	`p_bias`	`rows*1`	Bias vector, fixed or floating point
`WORD32`	`rows`		Number of rows in matrix 1, 2 and bias
`WORD32`	`cols1`		Number of columns in matrix 1 and rows in vector 1
`WORD32`	`cols2`		Number of columns in matrix 2 and rows in vector 2
`WORD32`	`row_stride1`		Row offset of matrix 1
`WORD32`	`row_stride2`		Row offset of matrix 2

Type	Name	Size	Description
WORD32	acc_shift		Shift applied to accumulator
WORD32	bias_shift		Shift applied to bias
WORD32	mat1_zero_bias		Zero offset of matrix 1
WORD32	mat2_zero_bias		Zero offset of matrix 2
WORD32	vec1_zero_bias		Zero offset of vector 1
WORD32	vec2_zero_bias		Zero offset of vector 2
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32	out_zero_bias		Zero offset of output
Output			
WORD8 *, UWORD8 *, WORD16 *, WORD32 *, WORD64 *, FLOAT32 *	p_out	rows*1	Output, fixed, floating point, asym8u, sym16s or asym8s.

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
row_stride1, row_stride2, cols1, cols2	row_stride1 >= cols1 row_stride2 >= cols2
p_mat1, p_mat2, p_vec1, p_vec2, p_bias, p_out	Aligned on \<size of one element\> boundary Should not overlap
p_mat1, p_vec1, p_out	Cannot be NULL
p_bias	Cannot be NULL (except for asym8uxasym8u and sym8sxasym8s precisions)
acc_shift, bias_shift, out_shift	{-31, ..., 31}
mat1_zero_bias, mat2_zero_bias, vec1_zero_bias, vec2_zero_bias	{-255……, 0} for asym8u, {-127……, 128} for asym8s
out_multiplier	Greater than 0
out_zero_bias	{0,........,255} if out type is asym8u, {-128….,127} if out type is asym8s

3.1.2 Fused (Activation) Matrix X Vector Kernels

Description

The Fused (Activation) Matrix X Vector kernels perform the fused dual matXvec operation with an activation function, that is, `z = activation (mat1*vec1 + mat2*vec2 + bias)`. The column dimension of `mat1` must match the row dimension of `vec1` and similarly for `mat2`, `vec2`. Bias and resulting output vector `z` have as many rows as `mat1` and `mat2`.

Intermediate output of `(mat1*vec1 + mat2*vec2 + bias)` is stored in temporary memory provided by the `p_scratch` argument to kernel API. Activation function is applied on this intermediate output to get final output.

Note   For fixed point kernels, the activation function always takes input in Q6.25 format.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and intermediate output respectively. Both `bias_shift` and `acc_shift` can be either positive or negative, where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as matXvec multiplication – accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the intermediate output in Q6.25 format.

Note   The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels.

The `row_stride1` and `row_stride2` arguments are provided in kernel API for row offsets of `mat1` and `mat2` respectively.

Note   The input matrices are expected to be appropriately padded in case of `row_stride > cols`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

Function variants available are `xa_nn_matXvec_[p]x[q]_[r]_<activation>`, where:

- `[p]`: Matrix precision in bits
- `[q]`: Vector precision in bits
- `[r]`: Output precision in bits
- `<activation>`: activation tag 'tanh' or 'sigmoid'

Precision

The following eight variants are available:

Type	Description
`16x16_16_tanh`	16-bit matrix inputs, 16-bit vector inputs, 16-bit output with tanh activation function
`16x16_16_sigmoid`	16-bit matrix inputs, 16-bit vector inputs, 16-bit output with sigmoid activation function
`8x16_16_tanh`	8-bit matrix inputs, 16-bit vector inputs, 16-bit output with tanh activation function
`8x16_16_sigmoid`	8-bit matrix inputs, 16-bit vector inputs, 16-bit output with sigmoid activation function
`8x8_8_tanh`	8-bit matrix inputs, 8-bit vector inputs, 8-bit output with tanh activation
`8x8_8_sigmoid`	8-bit matrix inputs, 8-bit vector inputs, 8-bit output with sigmoid activation

f32xf32_f32_tanh	float32 matrix inputs, float32 vector inputs, float32 output with tanh activation
f32xf32_f32_sigmoid	float32 matrix inputs, float32 vector inputs, float32 output with sigmoid activation

Algorithm

$$z_n = activation\left(2^{acc\text{-}shift}\left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot vec1_m + \sum_{m=0}^{cols2-1} mat2_{n,m} \cdot vec2_m + 2^{bias\text{-}shift} bias_n \right) \right), \qquad n = 0, \dots, \overline{rows - 1}$$

In case of floating point routine, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

activation is `tanh` or `sigmoid`

Prototype

```
WORD32 xa_nn_matXvec_16x16_16_tanh
(WORD16 * p_out,          WORD16 * p_mat1,        WORD16 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,        VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 bias_precision,  VOID * p_scratch);
WORD32 xa_nn_matXvec_16x16_16_sigmoid
(WORD16 * p_out,          WORD16 * p_mat1,        WORD16 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,        VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 bias_precision,  VOID * p_scratch);
WORD32 xa_nn_matXvec_8x16_16_tanh
(WORD16 * p_out,          WORD8 * p_mat1,         WORD8 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,        VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 bias_precision,  VOID * p_scratch);
WORD32 xa_nn_matXvec_8x16_16_sigmoid
(WORD16 * p_out,          WORD8 * p_mat1,         WORD8 * p_mat2,
 WORD16 * p_vec1,         WORD16 * p_vec2,        VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 bias_precision,  VOID * p_scratch);
WORD32 xa_nn_matXvec_8x8_8_tanh
(WORD8 * p_out,           WORD8 * p_mat1,         WORD8 * p_mat2,
 WORD8 * p_vec1,          WORD8 * p_vec2,         VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 bias_precision,  VOID * p_scratch);
WORD32 xa_nn_matXvec_8x8_8_sigmoid
(WORD8 * p_out,           WORD8 * p_mat1,         WORD8 * p_mat2,
 WORD8 * p_vec1,          WORD8 * p_vec2,         VOID * p_bias,
 WORD32 rows,             WORD32 cols1,           WORD32 cols2,
 WORD32 row_stride1,      WORD32 row_stride2,     WORD32 acc_shift,
```

```
 WORD32 bias_shift,        WORD32 bias_precision,   VOID * p_scratch);
WORD32 xa_nn_matXvec_f32xf32_f32_tanh
(FLOAT32 * p_out,          FLOAT32 * p_mat1,        FLOAT32 * p_mat2,
 FLOAT32 * p_vec1,         FLOAT32 * p_vec2,        FLOAT32 * p_bias,
 WORD32 rows,              WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,       WORD32 row_stride2       FLOAT32 * p_scratch);
WORD32 xa_nn_matXvec_f32xf32_f32_sigmoid
(FLOAT32 * p_out,          FLOAT32 * p_mat1,        FLOAT32 * p_mat2,
 FLOAT32 * p_vec1,         FLOAT32 * p_vec2,        FLOAT32 * p_bias,
 WORD32 rows,              WORD32 cols1,            WORD32 cols2,
 WORD32 row_stride1,       WORD32 row_stride2       FLOAT32 * p_scratch);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_mat1`	`rows*cols1`	Input matrix 1, fixed or floating point

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_mat2`	`rows*cols2`	Input matrix 2, fixed or floating point

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_vec1`	`cols1*1`	Input vector 1, fixed or floating point

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_vec2`	`cols2*1`	Input vector 2, fixed or floating point

`VOID *,` `FLOAT32 *`	`p_bias`	`rows*1`	Bias vector, fixed or floating point
`WORD32`	`rows`		Number of rows in matrix 1,2, bias and output
`WORD32`	`cols1`		Number of columns in matrix 1 and rows in vector 1
`WORD32`	`cols2`		Number of columns in matrix 2 and rows in vector 2
`WORD32`	`row_stride1`		Row offset of matrix 1
`WORD32`	`row_stride2`		Row offset of matrix 2
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`bias_precision`		Precision of bias in bytes
Output			

`WORD8 *,`
`WORD16 *,`
`FLOAT32 *`
	`p_out`	`rows*1`	Output, fixed (Q7, Q15) or floating point

Temporary			
`VOID *,` `FLOAT32 *`	`p_scratch`	`rows*4`	Scratch (temporary) memory pointer

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`row_stride1, row_stride2,` `cols1, cols2`	Multiples of 4 (2 in case of floating point)
`p_mat1, p_mat2, p_vec1,` `p_vec2, p_bias, p_out`	Aligned on 16-byte boundary Should not overlap
`p_mat1, p_vec1, p_bias,` `p_out`	Cannot be NULL
`p_scratch`	Cannot be NULL Should not overlap For 16x16 and 8x16 -> Aligned on 4-bytes boundary

Arguments	Restrictions
	For 8x8 and f32xf32 -> Aligned on 16-bytes boundary
acc_shift, bias_shift	{-31, ..., 31}
bias_precision	{-1, 8, 16, 32, 64} (-1 in case of floating point)

3.1.3 Matrix X Vector Batch Kernels

Description

The Matrix X Vector Batch kernels perform the operation of multiplication of a single matrix with a series of vectors along with bias addition; that is, $zi = mat1*vec1i + bias$. These kernels can also be viewed as matrix X matrix-transpose multiplication kernels. The column dimension of `mat1` must match the row dimension of vectors in `vec1`. Bias and resulting output vector sequence `z` have as many numbers of rows as `mat1`. `vec1` is a sequence of `vec_count` number of input vectors and bias is added to each resulting vector after multiplication with `mat1`. Thus, output `z` has dimensions `rows*vec_count`. `vec_count` number of input vectors and output vectors are provided as array of pointers arguments to kernel API.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output respectively. Both `bias_shift` and `acc_shift` can be either positive or negative where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as matXvec multiplication – accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note   The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels.

The `row_stride1` argument is provided in kernel API for row offset of `mat1`.

Note   The input matrix is expected to be appropriately padded in case of `row_stride1 > cols1`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

Function variants available are `xa_nn_matXvec_batch_[p]x[q]_[r]`, where:

- `[p]`: Matrix precision in bits
- `[q]`: Vector precision in bits
- `[r]`: Output precision in bits

Precision

There are five variants available:

Type	Description
`16x16_64`	16-bit matrix inputs, 16-bit vector inputs, 64-bit output vectors
`8x16_64`	8-bit matrix inputs, 16-bit vector inputs, 64-bit output vectors
`8x8_32`	8-bit matrix inputs, 8-bit vector inputs, 32-bit output vectors
`f32xf32_f32`	float32 matrix inputs, float32 vector inputs, float32 output
`asym8uxasym8u_asym8u`	asym8u matrix inputs, asym8u vector inputs, asym8u output vectors

Algorithm

$$z_{n,i} = 2^{acc\text{-}shift}\left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot vec1_{m,i} + 2^{bias\text{-}shift}bias_n\right),$$

$$n = 0, \ldots, \overline{rows - 1} \; ; \quad i = 0, \ldots, \overline{vec\text{-}count - 1}$$

In case of floating point routine, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

Prototype

```
WORD32 xa_nn_matXvec_batch_16x16_64
(WORD64 ** p_out,          WORD16 * p_mat1,         WORD16 ** p_vec1,
 WORD16 * p_bias,          WORD32 rows,             WORD32 cols1,
 WORD32 row_stride1,       WORD32 acc_shift,        WORD32 bias_shift,
 WORD32 vec_count);
WORD32 xa_nn_matXvec_batch_8x16_64
(WORD64 ** p_out,          WORD8 * p_mat1,          WORD16 ** p_vec1,
 WORD16 * p_bias,          WORD32 rows,             WORD32 cols1,
 WORD32 row_stride1,       WORD32 acc_shift,        WORD32 bias_shift,
 WORD32 vec_count);
WORD32 xa_nn_matXvec_batch_8x8_32
(WORD32 ** p_out,          WORD8 * p_mat1,          WORD8 ** p_vec1,
 WORD8 * p_bias,           WORD32 rows,             WORD32 cols1,
 WORD32 row_stride1,       WORD32 acc_shift,        WORD32 bias_shift,
 WORD32 vec_count);
WORD32 xa_nn_matXvec_batch_f32xf32_f32
(FLOAT32 ** p_out,         FLOAT32 * p_mat1,        FLOAT32 ** p_vec1,
 FLOAT32 * p_bias,         WORD32 rows,             WORD32 cols1,
 WORD32 row_stride1,       WORD32 vec_count);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_mat1`	`rows*cols1`	Input matrix, fixed or floating point

Type	Name	Size	Description

`WORD16 **,`
`WORD8 **,`
`FLOAT32 **`
	`p_vec1`	`cols1*vec_count`	Input vector pointers, fixed or floating point

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *`
	`p_bias`	`rows*1`	Bias vector, fixed or floating point

`WORD32`	`rows`		Number of rows in input matrix, bias and output
`WORD32`	`cols1`		Number of columns in input matrix and rows in input vector
`WORD32`	`row_stride1`		Row offset of input matrix
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`vec_count`		Number of input vectors
Output			

`WORD32 **,`
`WORD64 **,`
`FLOAT32 **`
	`p_out`	`rows*vec_count`	Output vector pointers, fixed or floating point

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`row_stride1, cols1`	Multiples of 4 (2 in case of floating point)
`p_mat1, p_vec1, p_bias,` `p_out`	Aligned on 16-byte boundary Should not overlap Cannot be NULL
`acc_shift, bias_shift` `,out_shift`	{-31, ..., 31}
`vec_count, out_multiplier`	Greater than Zero
`out_zero_bias`	{0, ..., 255}
`mat1_zero_bias,` `vec1_zero_bias`	{-255, ..., 0}

3.1.4 Matrix Multiplication Kernels

Description

The Matrix Multiplication kernels perform the operation of multiplication of a matrix `mat1` with another matrix `mat2` along with bias addition; that is, $z = mat1 * mat2 + bias$. The first matrix must be stored in row major order and the second matrix must be stored in column major order. The first matrix is of dimensions `rows x cols`. The second matrix `mat2` is of dimensions `cols x vec_count`. These kernels can also be viewed as a modification of the Matrix X Vector Batch kernels. The column dimension of `mat1` matches the row dimension of `mat2`, that is, the length of each vector in `p_mat2`. Bias and resulting output vector sequence `z` have as many numbers of rows as `mat1`. `mat2` is a sequence of `vec_count` number of input vectors and bias is added to each resulting vector after multiplication with `mat1`. Thus, output `z` has

dimensions `rows * vec_count`. The arguments `vec_offset` and `out_offset` are offsets to the next vector and output addresses. The argument `out_stride` defines the row offset for the output matrix. For standard matrix multiplication, `vec_offset` must be equal to `cols`, `out_offset` equal to 1 and `out_stride` must be equal to `vec_count`, that is, columns of mat2.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output respectively. Both `bias_shift` and `acc_shift` can be either positive or negative where positive value denotes a left shift and negative value denotes a right shift.

The `bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as multiplication – accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note	The `acc_shift` and `bias_shift` arguments are not relevant in case of quantized 8-bit kernels.

The `row_stride` argument indicates the offset to next row of `mat1`.

The `vec_offset` argument refers to the column offset of `mat2`.

Similarly, the `out_offset` and `out_stride` arguments refer to the column offset and row offset of the output matrix `rows * vec_count` respectively.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The arguments, `mat1_zero_bias`, `mat2_zero_bias`, are provided to convert the quantized 8-bit inputs into their real values and perform matXvec batch operation. The `out_zero_bias`, `out_multiplier`, and `out_shift` values are used to quantize real values of output back to quantized 8-bit values.

For the quantized int8 variant, we have per-row quantized input `mat1`.

Function variants available are `xa_nn_matmul_[p]x[q]_[r]`, where:

- `[p]`: Matrix 1 precision in bits
- `[q]`: Matrix 2 precision in bits
- `[r]`: Output precision in bits

Precision

There are nine variants available:

Type	Description
16x16_16	16-bit matrix inputs, 16-bit matrix inputs, 16-bit output matrix
8x16_16	8-bit matrix inputs, 16-bit matrix inputs, 16-bit output matrix
8x8_8	8-bit matrix inputs, 8-bit matrix inputs, 8-bit output matrix
f32xf32_f32	float32 matrix inputs, float32 matrix inputs, float32 output matrix
asym8uxasym8u_asym8u	asym8u matrix inputs, asym8u matrix inputs, asym8u output matrix

`per_chan_sym8sxasym8s_asym8s`	per channel quantized sym8s vector inputs, asym8s vector inputs, asym8s output vectors
`per_chan_sym8sxsym16s_sym16s`	per channel quantized sym8s vector inputs, sym16s vector inputs, sym16s output vectors
`asym8sxasym8s_asym8s`	asym8s matrix inputs, asym8s matrix inputs, asym8s output matrix
`sym8sxsym16s_sym16s`	sym8s matrix inputs, sym16s matrix inputs, sym16s output matrix
`f16xf16_f16`	float16 matrix inputs, float16 matrix inputs, float16 output matrix

Algorithm

$$z_{n,i} = 2^{acc\text{-}shift} \left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot mat2_{m,i} \ + \ 2^{bias\text{-}shift} bias_n \right),$$

$$n = 0, \dots, \overline{rows - 1} \ ; \quad i = 0, \dots, \overline{vec\text{-}count - 1}$$

In case of quantized 8-bit routines, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

Prototype

```
WORD32 xa_nn_matmul_16x16_16
(WORD16 * p_out,        WORD16 * p_mat1,      WORD16 * p_mat2,
 WORD16 * p_bias,       WORD32 rows,          WORD32 cols,
 WORD32 row_stride,     WORD32 acc_shift,     WORD32 bias_shift,
 WORD32 vec_count,      WORD32 vec_offset,    WORD32 out_offset,
 WORD32 out_stride);
WORD32 xa_nn_matmul_8x16_16
(WORD16 * p_out,        WORD8 * p_mat1,       WORD16 * p_mat2,
 WORD16 * p_bias,       WORD32 rows,          WORD32 cols,
 WORD32 row_stride,     WORD32 acc_shift,     WORD32 bias_shift,
 WORD32 vec_count,      WORD32 vec_offset,    WORD32 out_offset,
 WORD32 out_stride);
WORD32 xa_nn_matmul_8x8_8
(WORD8 * p_out,         WORD8 * p_mat1,        WORD8 * p_mat2,
 WORD8 * p_bias,        WORD32 rows,           WORD32 cols,
 WORD32 row_stride,     WORD32 acc_shift,      WORD32 bias_shift,
 WORD32 vec_count,      WORD32 vec_offset,     WORD32 out_offset,
 WORD32 out_stride);
WORD32 xa_nn_matmul_f32xf32_f32
(FLOAT32 * p_out,       FLOAT32 * p_mat1,      FLOAT32 * p_mat2,
 FLOAT32 * p_bias,      WORD32 rows,           WORD32 cols,
 WORD32 row_stride,     WORD32 acc_shift,      WORD32 bias_shift,
 WORD32 vec_count,      WORD32 vec_offset,     WORD32 out_offset,
 WORD32 out_stride);
WORD32 xa_nn_matmul_asym8uxasym8u_asym8u
(UWORD8 * p_out,        const UWORD8 * p_mat1,  const UWORD8 * p_mat2,
 const WORD32 * p_bias,  WORD32 rows,           WORD32 cols,
```

```
 WORD32 row_stride,       WORD32 vec_count,       WORD32 vec_offset,
 WORD32 out_offset,       WORD32 out_stride,      WORD32 mat1_zero_bias,
 WORD32 vec1_zero_bias,   WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias);
WORD32 xa_nn_matmul_per_chan_sym8sxasym8s_asym8s
(WORD8  * p_out,          const WORD8 * p_mat1,   const WORD8 * p_mat2,
 const WORD32 * p_bias,   WORD32 rows,            WORD32 cols,
 WORD32 row_stride,       WORD32 vec_count,       WORD32 vec_offset,
 WORD32 out_offset,       WORD32 out_stride,      WORD32 vec1_zero_bias
 const WORD32 *p_out_multiplier, const WORD32 *p_out_shift,
 WORD32 out_zero_bias);
WORD32 xa_nn_matmul_per_chan_sym8sxsym816s_sym16s
(WORD16  * p_out,         const WORD8 * p_mat1,   const WORD16 * p_mat2,
 const WORD64 * p_bias,   WORD32 rows,            WORD32 cols,
 WORD32 row_stride,       WORD32 vec_count,       WORD32 vec_offset,
 WORD32 out_offset,       WORD32 out_stride,      WORD32 vec1_zero_bias
 const WORD32 *p_out_multiplier, const WORD32 *p_out_shift,
 WORD32 out_zero_bias);
WORD32 xa_nn_matmul_asym8sxasym8s_asym8s
(WORD8  * p_out,          const WORD8 * p_mat1,   const WORD8 * p_mat2,
 const WORD32 * p_bias,   WORD32 rows,            WORD32 cols,
 WORD32 row_stride,       WORD32 vec_count,       WORD32 vec_offset,
 WORD32 out_offset,       WORD32 out_stride,      WORD32 mat1_zero_bias,
 WORD32 vec1_zero_bias    WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias);
WORD32 xa_nn_matmul_sym8sxsym16s_sym16s
(WORD16 * p_out,          const WORD8 * p_mat1,   const WORD16 * p_vec1,
 const WORD64 * p_bias,   WORD32 rows,            WORD32 cols1,
 WORD32 row_stride1,      WORD32 vec_count,       WORD32 vec_offset,
 WORD32 out_offset,       WORD32 out_stride,      WORD32 vec1_zero_bias,
 WORD32 out_multiplier,   WORD32 out_shift,       WORD32 out_zero_bias);
WORD32 xa_nn_matmul_f16xf16_f16
 (WORD16 * p_out,         const WORD16 * p_mat1,  const WORD16 * p_vec1,
  const WORD16 * p_bias,  WORD32 rows,            WORD32 cols1,
  WORD32 row_stride1,     WORD32 vec_count,       WORD32 vec_offset,
  WORD32 out_offset,      WORD32 out_stride);
```

Arguments

Type	Name	Size	Description
Input			
WORD8 *, WORD16 *, UWORD8 *, const WORD8 *, FLOAT32 *	p_mat1	rows*cols	Input matrix 1, f32, f16, fixed point, asym8u or sym8s
WORD8 *, WORD16 *, UWORD8 *, const WORD8 *, FLOAT32 *	p_mat2	cols * vec_count	Input matrix 2, f32, f16, fixed , asym8u or sym8s
WORD8 *, WORD16 *,	p_bias	rows*1	Bias vector, fixed point, f32, f16

Type	Name	Size	Description
`const WORD32 *, FLOAT32 *`			
`WORD32`	`rows`		Number of rows in input matrix, bias and output
`WORD32`	`cols`		Number of columns in input matrix and rows in input vector
`WORD32`	`row_stride`		Row offset of input matrix
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`vec_count`		Number of vectors (columns) in matrix 2
`WORD32`	`vec_offset`		Offset to the next vector address
`WORD32`	`out_offset`		Offset to the next output address
`WORD32`	`out_stride`		Row offset of output matrix
`WORD32`	`mat1_zero_bias`		Zero offset of matrix 1
`WORD32`	`vec1_zero_bias`		Zero offset of matrix 2
`WORD32`	`out_multiplier`		Multiplier value of output
`WORD32`	`out_shift`		Shift value of output
`WORD32`	`out_zero_bias`		Zero offset of output
Output			
`WORD8 *, WORD16 *, UWORD8 *, FLOAT32 *`	`p_out`	`rows*vec_ count`	Output matrix, fixed-point, f32, f16 or asym8u

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_mat1, p_mat2, p_out`	Aligned on (size of one element)-byte boundary Cannot be NULL Should not overlap
`p_bias`	Aligned on (size of one element)-byte boundary
`acc_shift, bias_shift, out_shift`	{-31, ..., 31}
`vec_count`	Greater than 0
`vec_offset, out_offset, out_stride`	Should not be 0
`mat1_zero_bias,`	{-255, ..., 0} for asym8u, {-127......, 128} for asym8s,
`vec1_zero_bias`	{-255......, 0} for asym8u, {-127......, 128} for asym8s, 0 for sym16s
`out_multiplier`	Greater than 0
`p_out_multiplier,`	Aligned on (size of one element)-byte boundary

`p_out_shift`	Cannot be NULL (range of values are specified for out_multiplier and out_shift)
`out_zero_bias`	{0,........,255} if out type is asym8u, {-128….,127} if out type is asym8s, 0 for sym16s

3.1.5 Matrix X Vector Kernels with Output Stride

Description

The Matrix X Vector kernels with output stride perform a single matXvec operation with bias addition; that is, `z = mat1*vec1 + bias`. The column dimension of `mat1` must match the row dimension of `vec1`. Bias and resulting output vector `z` have as many rows as `mat1`.

`row_stride1` is provided in kernel API for row offsets of `mat1`.

Note   The input matrices are expected to be appropriately padded in case of `row_stride > cols`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The argument `out_stride` is helpful in storing the output at a given offset.

The argument `vec1_zero_bias` is provided to convert the quantized 8-bit inputs into their real values and perform matXvec operation. The `out_multiplier` and `out_shift` values are used to convert real values of output to 16-bit.

Function variants available are `xa_nn_matXvec_[p]x[q]_[r]`, where:

- `[p]`: Matrix precision in bits
- `[q]`: Vector precision in bits
- `[r]`: Output precision in bits

Precision

There is one variant available:

Type	Description
`sym8sxasym8s_16`	sym8s matrix inputs, asym8s vector inputs, asym8s output

Algorithm

$$z_n = \left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot vec1_m \ + \ bias_n \right)$$

Prototype

```
WORD32 xa_nn_matXvec_out_stride_sym8sxasym8s_16
(WORD16 * p_out,          const WORD8  * p_mat1,  const WORD8  * p_vec1,
 const WORD32 * p_bias,  WORD32 rows,             WORD32 cols1,
 WORD32 row_stride1,     WORD32 out_stride,       WORD32 vec1_zero_bias,
 WORD32 out_multiplier,  WORD32 out_shift);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_mat1	rows*cols1	Input matrix, sym8s
const WORD8 *	p_vec1	cols1*1	Input vector, asym8s
const WORD32 *	p_bias	rows*1	Bias vector
WORD32	rows		Number of rows in matrix and number of elements in bias
WORD32	cols1		Number of columns in matrix and elements in vector
WORD32	row_stride1		Row offset of matrix
WORD32	out_stride		Row offset of output
WORD32	vec1_zero_bias		Zero offset of vector
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
Output			
WORD16 *	p_out	rows*1	Output, 16-bit

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
row_stride1, cols1	row_stride1 >= cols1
p_mat1, p_vec1, p_bias, p_out	Aligned on <size of one element> boundary Should not overlap
p_mat1, p_vec1, p_out	Cannot be NULL
out_shift	{-31, ..., 31}
vec1_zero_bias	{-127……, 128} for asym8s
out_multiplier	Greater than 0

3.1.6 Matrix X Vector Batch Kernels with Accumulation

The Matrix X Vector Batch kernels with accumulation perform the operation of multiplication of a single matrix with a series of vectors along with bias addition; that is, $zi = zi + mat1*vec1i + bias$. These kernels can also be viewed as matrix X matrix-transpose multiplication kernels. The column dimension of

`mat1` must match the row dimension of vectors in `vec1`. Bias and resulting output vector sequence `z` have as many numbers of rows as `mat1`. `vec1` is a sequence of `vec_count` number of input vectors and bias is added to each resulting vector after multiplication with `mat1`. Thus, output `z` has dimensions `rows*vec_count`. `vec_count` number of input and output vectors are provided as pointers to the start of first vector, subsequent vectors are supposed to be stored contiguously in memory. The result of matrix X vector batch operation is accumulated to the values present at the output.

The `row_stride1` argument is provided in kernel API for row offset of `mat1`.

Note   The input matrix is expected to be appropriately padded in case of `row_stride1 > cols1`.

The `out_zero_bias`, `out_multiplier`, and `out_shift` values are used to quantize the output to 16-bits.

Function variants available are `xa_nn_matXvec_acc_batch_[p]x[q]_[r]`, where:

- `[p]`: Matrix precision in bits
- `[q]`: Vector precision in bits
- `[r]`: Output precision in bits

Precision

There is one variant available:

Type	Description
`sym8sx8_asym16s`	sym8s matrix inputs, 8-bit vector inputs, asym16s output vectors

Algorithm

$$z_{n,i} = z_{n,i} + \left(\sum_{m=0}^{cols1-1} mat1_{n,m} \cdot vec1_{m,i} + bias_n \right),$$

$$n = 0, \dots, \overline{rows - 1} \ ; \quad i = 0, \dots, \overline{vec\text{-}count - 1}$$

Prototype

```
WORD32 xa_nn_matXvec_acc_batch_sym8sx8_asym16s
(WORD16 * p_out,          const WORD8 * p_mat1,     const WORD8 * p_vec1,
 const WORD32 * p_bias, WORD32 rows,               WORD32 cols1,
 WORD32 row_stride1,     WORD32 out_multiplier,    WORD32 out_shift,
 WORD32 out_zero_bias,   WORD32 vec_count);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_mat1	rows*cols1	Input matrix, sym8s
const WORD8 *	p_vec1	cols1*vec_count	Input vectors, 8-bit
const WORD32 *	p_bias	rows*1	Bias vector, 32-bit
WORD32	rows		Number of rows in input matrix, bias and output
WORD32	cols1		Number of columns in input matrix and rows in input vector
WORD32	row_stride1		Row offset of input matrix
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32	out_zero_bias		Zero offset of output
WORD32	vec_count		Number of input vectors
Output			
WORD16	p_out	rows*vec_count	Output vectors, asym16s

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_mat1, p_vec1, p_bias, p_out`	Aligned on <size of one element> boundary
	Cannot be NULL
	Should not overlap
`rows, cols1, vec_count`	Should be greater than 0.
`row_stride1`	Cannot be less than `cols1`
`out_shift`	{-31, ..., 31}
`out_zero_bias`	{-32768, ..., 32767}

3.2   Convolution Kernels

3.2.1 Standard 2D Convolution Kernels

Description

The Standard 2D Convolution kernels perform the 2D convolution operation as `z = inp(*)kernel + bias`. A 3D input cube (`input_height x input_width x input_channels`), is convolved with a 3D kernel cube (`kernel_height x kernel_width x input_channels`) to produce a 2D convolution output plane (`out_height x out_width`). With `out_channels` number of such 3D kernels, output cube (`out_height x out_width x out_channels`) is produced. The bias having dimension (`out_channels`) is added after the convolution (one bias value is added to each output channel) to produce the final output.

Note   The depth or channels dimension (`input_channels`) of input and kernel must be identical for 2D convolution.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output, respectively. Both `bias_shift` and `acc_shift` can be either positive or negative where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as convolution - accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note   The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels and quantized datatype kernels.

The `x_stride` and `y_stride` arguments in kernel API define the step size of the kernel when traversing the input in width and height dimensions respectively.

The `x_padding` argument defines padding to the left of the input in the width dimension and the `y_padding` argument defines padding to the top of the input in the height dimension.

The right padding is calculated based on `out_width` as `right_paddding = kernel_width + (out_width – 1) * x_stride – (x_padding + input_width)`.

The bottom padding is calculated based on `out_height` as `bottom_paddding = kernel_height + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The kernel is expected to be padded in the depth or channels dimension if the number of `input_channels` is not a multiple of 4 in case of fixed point variants other than the 8x8, asym8uxasym8u, per_chan_sym8sxasym8s and per_chan_sym8sxsym16s_sym16s variant, and 2 in case of floating point variant. No padding is needed for 8x8 and quantized 8-bit variants.

These kernels require temporary buffer for convolution computation. This temporary buffer is provided by `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_conv2d_std_getsize()` helper API for kernels except for sym4sxasym8s variant. For the sym4sxasym8s variant, xa_nn_conv2d_std_getsize_sym4s() helper API must be used to query size of temporary buffer.

The arguments `input_zero_bias`, `kernel_zero_bias, out_zero_bias, out_multiplier` and `out_shift` are provided to handle scaling and quantization in the quantized Standard 2D Convolution variants.

These kernels expect input and kernel cubes in SHAPE_CUBE_DWH_T shape type and can produce output cube in either SHAPE_CUBE_DWH_T or SHAPE_CUBE_WHD_T shape type. The `out_data_format` argument to kernel API controls the output cube shape type.

Function variants available are `xa_nn_conv2d_std_[p]x[q]`, where:

- `[p]`: Kernel precision in bits
- `[q]`: Input precision in bits

Precision

There are seven variants available.

Type	Description
`16x16`	16-bit kernel, 16-bit input, 16-bit output
`8x16`	8-bit kernel, 16-bit input, 16-bit output
`8x8`	8-bit kernel, 8-bit input, 8-bit output
`f32`	float32 kernel, float32 input, float32 output
`asym8uxasym8u`	asym8u kernel, asym8u input, asym8u output
`per_chan_sym8sxasym8s`	per channel quantized sym8s kernel, asym8s input, asym8s output
`per_chan_sym8sxsym16s`	per channel quantized sym8s kernel, sym16s input, sym16s output
`f16`	float16 kernel, float16 input, float16 output
`per_chan_sym4sxasym8s`	per channel quantized sym4s kernel, asym8s input, asym8s output

Algorithm

$$z_{h,w,d} = 2^{acc\text{-}shift} \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} \sum_{k=0}^{I_C-1} in_{pad_{(h*y\text{-}stride+i),(w*x\text{-}stride+j),k}} \cdot ker_{pad_{d,i,j,k}} \right.$$

$$\left. + 2^{bias\text{-}shift} b_d \right)$$

$$h = 0, \ldots, \overline{out\text{-}height - 1}, w = 0, \ldots, \overline{out\text{-}width - 1},$$
$$d = 0, \ldots, \overline{out\text{-}channels - 1}$$

In case of floating-point kernels and quantized 8-bit kernels, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

in_{pad}, ker_{pad} denote the padded `p_inp` and padded `p_ker` shapes, respectively.

K_H, K_W, I_C denote kernel_height, kernel_width, and input_channels, respectively.

b denotes the `bias` shape.

Prototype

```
WORD32 xa_nn_conv2d_std_getsize
(WORD32 input_height,     WORD32 input_channels,WORD32 kernel_height,
 WORD32 kernel_width,     WORD32 y_stride,      WORD32 y_padding,
 WORD32 out_height,       WORD32 out_channels,  WORD32 input_precision);
WORD32 xa_nn_conv2d_std_getsize_sym4s
(WORD32 input_height,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,     WORD32 y_stride,        WORD32 y_padding,
 WORD32 out_height,       WORD32 out_channels,    WORD32 input_precision);


WORD32 xa_nn_conv2d_std_16x16
(WORD16 * p_out,          WORD16 * p_inp,       WORD16 * p_ker,
 WORD16 * p_bias,         WORD32 input_height,  WORD32 input_width,
 WORD32 input_channels,   WORD32 kernel_height, WORD32 kernel_width ,
 WORD32 out_channels,     WORD32 x_stride,      WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,     WORD32 out_height,
 WORD32 out_width,        WORD32 bias_shift,    WORD32 acc_shift,
 WORD32 out_data_format,  VOID   * p_scratch);
WORD32 xa_nn_conv2d_std_8x16
(WORD16 * p_out,          WORD16  * p_inp,      WORD8 * p_ker,
 WORD16 * p_bias,         WORD32 input_height,  WORD32 input_width,
 WORD32 input_channels,   WORD32 kernel_height, WORD32 kernel_width,
 WORD32 out_channels,     WORD32 x_stride,      WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,     WORD32 out_height,
 WORD32 out_width,        WORD32 bias_shift,    WORD32 acc_shift,
 WORD32 out_data_format,  VOID   * p_scratch);
WORD32 xa_nn_conv2d_std_8x8
(WORD8  * p_out,          WORD8 * p_inp,        WORD8  * p_ker,
 WORD8  * p_bias,         WORD32 input_height,  WORD32 input_width,
 WORD32 input_channels,   WORD32 kernel_height, WORD32 kernel_width,
 WORD32 out_channels,     WORD32 x_stride,      WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,     WORD32 out_height,
 WORD32 out_width,        WORD32 bias_shift,    WORD32 acc_shift,
 WORD32 out_data_format,  VOID   * p_scratch);
```

```
WORD32 xa_nn_conv2d_std_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp, const FLOAT32 * p_ker,
 const FLOAT32 * p_bias,   WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,    WORD32 kernel_height,  WORD32 kernel_width,
 WORD32 out_channels,      WORD32 x_stride,       WORD32 y_stride,
 WORD32 x_padding,         WORD32 y_padding,      WORD32 out_height,
 WORD32 out_width,         WORD32 out_data_format,VOID  * p_scratch);
WORD32 xa_nn_conv2d_std_asym8uxasym8u
(UWORD8 * p_out,           const UWORD8 * p_inp,  const UWORD8 * p_ker,
 const WORD32 * p_bias,    WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,    WORD32 kernel_height,  WORD32 kernel_width,
 WORD32 out_channels,      WORD32 x_stride,       WORD32 y_stride,
 WORD32 x_padding,         WORD32 y_padding,      WORD32 out_height,
 WORD32 out_width,         WORD32 input_zero_bias,WORD32 kernel_zero_bias,
 WORD32 out_multiplier,    WORD32 out_shift,      WORD32 out_zero_bias,
 WORD32 out_data_format,   VOID  * p_scratch);
WORD32 xa_nn_conv2d_std_per_chan_sym8sxasym8s
(WORD8  * p_out,           const WORD8 * p_inp,   const WORD8 * p_ker,
 const WORD32 * p_bias,    WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,    WORD32 kernel_height,  WORD32 kernel_width,
 WORD32 out_channels,      WORD32 x_stride,       WORD32 y_stride,
 WORD32 x_padding,         WORD32 y_padding,      WORD32 out_height,
 WORD32 out_width,         WORD32 input_zero_bias,WORD32 * p_out_multiplier,
 WORD32 * p_out_shift,     WORD32 out_zero_bias,  WORD32 out_data_format,
 VOID  * p_scratch);
WORD32 xa_nn_conv2d_std_per_chan_sym8sxsym16s
(WORD16* p_out,            const WORD16* p_inp,    const WORD8* p_kernel,
 const WORD64* p_bias,     WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,    WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 out_channels,      WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,         WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,         WORD32 input_zero_bias, WORD32 * p_out_multiplier,
 WORD32 * p_out_shift,     WORD32 out_zero_bias,   WORD32 out_data_format,
 VOID  * p_scratch);
WORD32 xa_nn_conv2d_std_f16
 (WORD16* p_out,           const WORD16* p_inp,    const WORD16* p_kernel,
  const WORD16* p_bias,  WORD32 input_height,    WORD32 input_width,
  WORD32 input_channels, WORD32 kernel_height,   WORD32 kernel_width,
  WORD32 out_channels,   WORD32 x_stride,        WORD32 y_stride,
  WORD32 x_padding,      WORD32 y_padding,       WORD32 out_height,
  WORD32 out_width,      WORD32 out_data_format, VOID *p_scratch);
WORD32 xa_nn_conv2d_std_per_chan_sym4sxasym8s
 (WORD8* p_out,           const WORD8* p_inp,     const WORD8* p_kernel,
 const WORD32* p_bias,   WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels, WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 out_channels,   WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,      WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,      WORD32 input_zero_bias, WORD32 * p_out_multiplier,
 WORD32 * p_out_shift,  WORD32 out_zero_bias,   WORD32 out_data_format,
 VOID *p_scratch);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`const`
`FLOAT32 *,`
	`p_inp`	`input_height*`
		`input_width*`
		`input_channels`
			Input cube, fixed, floating point, asym8u, asym8s or sym16s, in SHAPE_CUBE_DWH_T

Type	Name	Size	Description
`const UWORD8 *,` `const WORD8 *,` `const WORD16 *`			
`WORD16 *,` `WORD8 *,` `const FLOAT32 *,` `const UWORD8 *` `const WORD8 *`	`p_ker`	`out_channels*` `(kernel_height` `*` `kernel width*` `input_channels` `)`	Kernel cube, fixed, floating point, sym4s, asym8u or sym8s, in SHAPE_CUBE_DWH_T
`WORD16 *,` `WORD8 *,` `FLOAT32 *,` `const WORD32 *,` `const WORD64 *`	`p_bias`	`out_channels`	Bias vector, fixed or floating point
`WORD32`	`input_height`		Input height
`WORD32`	`input_width`		Input width
`WORD32`	`input_channels`		Number of input channels
`WORD32`	`kernel_height`		Kernel height
`WORD32`	`kernel_width`		Kernel width
`WORD32`	`out_channels`		Number of output channels
`WORD32`	`x_stride`		Horizontal stride over input
`WORD32`	`y_stride`		Vertical stride over input
`WORD32`	`x_padding`		Left padding width on input
`WORD32`	`y_padding`		Top padding height on input
`WORD32`	`out_height`		Output height
`WORD32`	`out_width`		Output width
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`input_zero_bias`		Zero offset of input
`WORD32`	`kernel_zero_bia s`		Zero offset of kernel
`WORD32`	`out_multiplier`		Multiplier value of output
`WORD32`	`out_shift`		Shift value of output
`const WORD32 *`	`p_out_multiplie r`		Vector having multiplier values of output for per channel quantization
`const WORD32 *`	`p_out_shift`		Vector having shift values of output for per channel quantization
`WORD32`	`out_zero_bias`		Zero offset of output
`WORD32`	`out_data_format`		Output data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T

Type	Name	Size	Description
`VOID *`	`p_scratch`	`xa_nn_conv2d_std_getsize()`	Scratch memory pointer
Output			

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
`UWORD8 *`
	`p_out`	`(out_height*`
		`out_width)*`
		`out_channels`
			Output cube, fixed, floating point, asym8u, asym8s or sym16s, as per the `out_data_format` argument.

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_out, p_inp, p_ker, p_bias,` `p_scratch`	Cannot be NULL
	Should not overlap
	Aligned on 16-byte boundary except for quantized 8-bit kernels where only p_scratch is required to be 16-byte aligned.and other (size of one element) byte aligned
	For p_scratch - memory size >= size returned by xa_nn_conv2d_std_getsize()
`p_out_multiplier,` `p_out_shift`	Cannot be NULL
	Should not overlap
	Aligned on 4-byte boundary
`input_height, input_width,` `input_channels`	Greater than or equal to 1
`kernel_height`	{1, 2, …, input_height}
`kernel_width`	{1, 2, …, input_width}
`out_channels`	Greater than or equal to 1
`x_stride`	{1, 2, …, kernel_width}
`y_stride`	Greater than or equal to 1
`x_padding, y_padding`	Greater than or equal to 0
`out_height, out_width`	Greater than or equal to 1
`acc_shift, bias_shift,` `out_shift`	{-31 …. 31} for fixed point and quantized datatype APIs
`input_zero_bias`	{-255,….., 0} for asym8u input, {-127….., 128} for asym8s input, 0 for sym16s input
`kernel_zero_bias`	{-255……, 0} for asym8u kernel
`out_zero_bias`	{0,........,255} for asym8u output, {-128….., 127} for asym8s output, 0 for sym16s output
`out_multiplier`	Greater than 0
`out_data_format`	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T
`kernel_height*kernel_width*` `input_channels`	This multiplication must be even for sym4s data-type. No restriction for other data-types

3.2.2 Standard 2D Convolution Kernels with Dilation

Description

The Standard 2D Convolution kernels with dilation perform the dilated 2D convolution operation as `z = inp(*)kernel + bias`. A 3D input cube (`input_height x input_width x input_channels`), is convolved with a 3D dilated kernel cube to produce a 2D convolution output plane (`out_height x out_width`). With `out_channels` number of such 3D kernels, output cube (`out_height x out_width x out_channels`) is produced. Before convolution, the 3D kernel cube (`kernel_height x kernel_width x input_channels`) is dilated by skipping `dilation_height-1` elements in height dimension and `dilation_width-1` elements in width dimension with, `dilation_height>=1` and/or `dilation_width>=1`. Post dilation, the kernel cube is of size `kernel_height_dilation = kernel_height + (kernel_height-1)*( dilation_height-1)` in height dimension and `kernel_width_dilation = kernel_width + (kernel_width-1)*( dilation_width-1)` in width dimension. The bias having dimension (`out_channels`) is added after the convolution (one bias value is added to each output channel) to produce the final output.

Note	The depth or channels dimension (`input_channels`) of input and kernel must be identical for 2D convolution.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output, respectively. Both `bias_shift` and `acc_shift` can be either positive or negative where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as convolution - accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

The `x_stride` and `y_stride` arguments in kernel API define the step size of the kernel when traversing the input in width and height dimensions respectively.

The `x_padding` argument defines padding to the left of the input in the width dimension and the `y_padding` argument defines padding to the top of the input in the height dimension.

The right padding is calculated based on `out_width` as `right_paddding = kernel_width_dilation + (out_width – 1) * x_stride – (x_padding + input_width)`.

The bottom padding is calculated based on `out_height` as `bottom_paddding = kernel_height_dilation + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

These kernels require temporary buffer for convolution computation. This temporary buffer is provided by `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_dilated_conv2d_std_getsize()` helper API.

These kernels expect input and kernel cubes in SHAPE_CUBE_DWH_T shape type and can produce output cube in either SHAPE_CUBE_DWH_T or SHAPE_CUBE_WHD_T shape type. The `out_data_format` argument to kernel API controls the output cube shape type.

Precision

Type	Description
`per_chan_sym8sxasym8s`	per channel quantized sym8s kernel, asym8s input, asym8s output

Algorithm

$$z_{h,w,d}$$
$$= 2^{acc\text{-}shift} \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} \sum_{k=0}^{I_C-1} in_{pad_{(h*y\text{-}stride+i*dilation\text{-}height),(w*x\text{-}stride\,+j*dilation\text{-}width),k}}} \right.$$
$$\left. \cdot\ ker_{d,i,j,k} + 2^{bias\text{-}shift}\, b_d \right)$$
$$h = 0, \ldots, \overline{out\text{-}height - 1}, w = 0, \ldots, \overline{out\text{-}width - 1},$$
$$d = 0, \ldots, \overline{out\text{-}channels - 1}$$

in_{pad}, ker denote the padded `p_inp` and kernel `p_ker` shapes, respectively.

K_H, K_W, I_C denote kernel_height, kernel_width, and input_channels, respectively.

b denotes the `bias` shape.

Prototype

```
WORD32 xa_nn_dilated_conv2d_std_getsize
(WORD32 input_height,      WORD32 input_channels,    WORD32 kernel_height,
 WORD32 kernel_width,      WORD32 y_stride,          WORD32 y_padding,
 WORD32 out_height,        WORDD32 out_channels,     WORD32 input_precision,
 WORD32 dilation_height);


WORD32 xa_nn_dilated_conv2d_std_per_chan_sym8sxasym8s
(WORD8  * p_out,           const WORD8 * p_inp,      const WORD8 * p_ker,
 const WORD32 * p_bias,    WORD32 input_height,      WORD32 input_width,
 WORD32 input_channels,    WORD32 kernel_height,     WORD32 kernel_width,
 WORD32 out_channels,      WORD32 x_stride,          WORD32 y_stride,
 WORD32 x_padding,         WORD32 y_padding,         WORD32 out_height,
 WORD32 out_width,         WORD32 input_zero_bias,   WORD32 * p_out_multiplier,
 WORD32 * p_out_shift,     WORD32 out_zero_bias,     WORD32 out_data_format,
 VOID   * p_scratch,       WORD32 dilation_height,   WORD32 dilation_width);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`const`
`FLOAT32 *,`
`const`
`UWORD8 *,`
`const`
`WORD8 *`
	`p_inp`	`input_height*`
		`input width*`
		`input_channels`
			Input cube, fixed, floating point, asym8u or asym8s, in SHAPE_CUBE_DWH_T

`WORD16 *,`
`WORD8 *,`
`const`
`FLOAT32 *,`
`const`
`UWORD8 *`
`const`
`WORD8 *`
	`p_ker`	`out_channels*`
		`(kernel_height`
		`*`
		`kernel width*`
		`input_channels`
		`)`
			Kernel cube, fixed, floating point, asym8u or sym8s, in SHAPE_CUBE_DWH_T

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
`const`
`WORD32 *`
	`p_bias`	`out_channels`	Bias vector, fixed or floating point

`WORD32`	`input_height`		Input height
`WORD32`	`input_width`		Input width
`WORD32`	`input_channels`		Number of input channels
`WORD32`	`kernel_height`		Kernel height
`WORD32`	`kernel_width`		Kernel width
`WORD32`	`out_channels`		Number of output channels
`WORD32`	`x_stride`		Horizontal stride over input
`WORD32`	`y_stride`		Vertical stride over input
`WORD32`	`x_padding`		Left padding width on input
`WORD32`	`y_padding`		Top padding height on input
`WORD32`	`out_height`		Output height
`WORD32`	`out_width`		Output width
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`input_zero_bias`		Zero offset of input
`WORD32`	`kernel_zero_bias`		Zero offset of kernel
`WORD32`	`out_multiplier`		Multiplier value of output
`WORD32`	`out_shift`		Shift value of output
`WORD32`	`out_zero_bias`		Zero offset of output
`WORD32`	`out_data_format`		Output data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T

`VOID *`	`p_scratch`	`xa_nn_dilated_`
		`conv2d_std_get`
		`size()`
			Scratch memory pointer

Type	Name	Size	Description
WORD32	dilation_height		Kernel height dilation factor
WORD32	dilation_width		Kernel width dilation factor
Output			
WORD16 *, WORD8 *, FLOAT32 *, UWORD8 *	p_out	(out_height* out_width)* out_channels	Output cube, fixed, floating point, asym8u or asym8s, as per the out_data_format argument.

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_out, p_inp, p_ker, p_bias, p_scratch	Cannot be NULL
	Should not overlap
	Aligned on 16-byte boundary except for quantized 8-bit kernels where only p_scratch is required to be 16-byte aligned.
	For p_scratch - memory size >= size returned by xa_nn_conv2d_std_getsize()
input_height, input_width, input_channels	Greater than or equal to 1
kernel_height	{1, 2, …, input_height}
kernel_width	{1, 2, …, input_width}
out_channels	Greater than or equal to 1
x_stride	Greater than or equal to 1
y_stride	Greater than or equal to 1
x_padding, y_padding	Greater than or equal to 0
dilation_height, dilation_width	Greater than or equal to 1
out_height, out_width	Greater than or equal to 1
acc_shift, bias_shift, out_shift	{-31 …. 31} for fixed point and quantized 8-bit APIs
input_zero_bias	{-255,….., 0} for asym8u input, {-127….., 128} for asym8s input
kernel_zero_bias	{-255……, 0} for asym8u kernel
out_zero_bias	{0,........,255} for asym8u output, {-128….., 127} for asym8s output
out_multiplier	Greater than 0
out_data_format	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T

3.2.3 Standard 1D Convolution Kernels

Description

The Standard 1D Convolution kernels perform the 1D convolution operation as `z = inp(*)kernel + bias`. A 3D input cube (`input_height x input_width x input_channels`) is convolved with a 3D kernel cube (`kernel_height x input_width x input_channels`) to produce a 1D convolution output vector (`out_height`). With `out_channels` number of such 3D kernels, output matrix (`out_height x out_channels`) is produced. The bias having dimension (`out_channels`) is added after the convolution (one bias value is added to each output column) to produce the final output.

Note	The depth or channels dimension (`input_channels`) of input and kernel must be identical, and width dimension (`input_width`) of input and kernel also must be identical for 1D convolution.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output, respectively. Both `bias_shift` and `acc_shift` can be either positive or negative, where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as convolution - accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note	The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels.

The `y_stride` argument to kernel API defines the step size of the kernel when traversing the input in height dimension.

The `y_padding` argument defines padding to the top of the input in the height dimension.

The bottom padding is calculated based on `out_height` as `bottom_paddding = kernel_height + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The kernel is expected to be padded if the product `input_channels*input_width` is not a multiple of 4 in case of fixed point variants, and 2 in case of floating point variant.

These kernels require temporary buffer for convolution computation. This temporary buffer is provided by `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_conv1d_std_getsize()` helper API.

These kernels expect input and kernel cubes in SHAPE_CUBE_DWH_T shape type and can produce output matrix with either (`out_height x out_channels`) or (`out_channels x out_height`) dimensions. The `out_data_format` argument to kernel API controls the output matrix height and width order.

Function variants available are `xa_nn_conv1d_std_[p]`, where:

- ■ `[p]`: precision in bits

Precision

There are five variants available:

Type	Description
`16x16`	16-bit kernel, 16-bit input, 16-bit output
`8x16`	8-bit kernel, 16-bit input, 16-bit output
`8x8`	8-bit kernel, 8-bit input, 8-bit output
`f32`	float32 kernel, float32 input, float32 output
`asym8uxasym8u_asym8u`	asym8u matrix inputs, asym8u vector inputs, asym8u output vectors

Algorithm

$$z_{h,d} = 2^{acc\text{-}shift} \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{I_W-1} \sum_{k=0}^{I_C-1} in_{pad\,(h*y\text{-}stride+i),j,k} \cdot ker_{pad\,d,i,j,k} \right.$$

$$\left. + 2^{bias\text{-}shift} b_d \right)$$

$$h = 0, \dots, \overline{out\text{-}height - 1}, d = 0, \dots, \overline{out\text{-}channels - 1}$$

In case of floating point kernel, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

n_{pad}, ker_{pad} denote the padded `p_inp` and padded `p_ker` shapes, respectively.

K_H, I_W, I_C denote kernel_height, input_width, and input_channels, respectively.

b denotes the `bias` shape.

Prototype

```
WORD32 xa_nn_conv1d_std_getsize
(WORD32 kernel_height,  WORD32 input_width,   WORD32 input_channels,
 WORD32 input_precision);

WORD32 xa_nn_conv1d_std_16x16
(WORD16 * p_out,         WORD16 * p_inp,        WORD16 * p_ker,
 WORD16 * p_bias,        WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,  WORD32 kernel_height,  WORD32 out_channels,
 WORD32 y_stride,        WORD32 y_padding,      WORD32 out_height,
 WORD32 bias_shift,      WORD32 acc_shift,      WORD32 out_data_format,
 VOID   * p_scratch);
WORD32 xa_nn_conv1d_std_8x16
(WORD16 * p_out,         WORD16 * p_inp,        WORD8  * p_ker,
 WORD16 * p_bias,        WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,  WORD32 kernel_height,  WORD32 out_channels,
 WORD32 y_stride,        WORD32 y_padding,      WORD32 out_height,
 WORD32 bias_shift,      WORD32 acc_shift,      WORD32 out_data_format,
 VOID   * p_scratch);
```

```
WORD32 xa_nn_conv1d_std_8x8
(WORD8  * p_out,         WORD8  * p_inp,        WORD8  * p_ker,
 WORD8  * p_bias,        WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,  WORD32 kernel_height,  WORD32 out_channels,
 WORD32 y_stride,        WORD32 y_padding,      WORD32 out_height,
 WORD32 bias_shift,      WORD32 acc_shift,      WORD32 out_data_format,
 VOID   * p_scratch);
WORD32 xa_nn_conv1d_std_f32
(FLOAT32 * p_out,         FLOAT32 * p_inp,        FLOAT32 * p_ker,
 FLOAT32 * p_bias,        WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 kernel_height,   WORD32 out_channels,
 WORD32 y_stride,         WORD32 y_padding,       WORD32 out_height,
 WORD32 out_data_format,  VOID   * p_scratch);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
	`p_inp`	`input_height*`
		`input width*`
		`input_channels`
			Input cube, fixed or floating point, in SHAPE_CUBE_DWH_T

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
	`p_ker`	`out_channels*`
		`(kernel_height*`
		`input width*`
		`input_channels)`
			Kernel cube, fixed or floating point, in SHAPE_CUBE_DWH_T

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
	`p_bias`	`out_channels`	Bias vector, fixed or floating point

`WORD32`	`input_height`		Input height
`WORD32`	`input_width`		Input width
`WORD32`	`input_channels`		Number of input channels
`WORD32`	`kernel_height`		Kernel height
`WORD32`	`out_channels`		Number of output channels
`WORD32`	`y_stride`		Vertical stride over input
`WORD32`	`y_padding`		Top padding height on input
`WORD32`	`out_height`		Output height
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`out_data_format`		Output matrix order 0: `out_height x out_channels` 1: `out_channels x out_height`
`VOID *`	`p_scratch`	`xa_nn_conv1d_std_getsize()`	Scratch memory pointer
Output			

`WORD16 *,`
`WORD8 *,`
`FLOAT32 *,`
	`p_out`	`out_height*`
		`out_channels`
			Output matrix, fixed or floating point, as per the `out_data_format` argument.

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_out, p_inp, p_ker, p_bias, p_scratch`	Cannot be NULL
	Should not overlap
	Aligned on 16-byte boundary
	For `p_scratch` – memory size >= size returned by `xa_nn_conv1d_std_getsize()`
`input_height, input_width, input_channels`	Greater than or equal to 1
`kernel_height`	{1, 2, ..., `input_height`}
`out_channels`	Greater than or equal to 1
`y_stride`	{1, 2, ..., `kernel_height`}
`y_padding`	Greater than or equal to 0
`out_height`	Greater than or equal to 1
`acc_shift, bias_shift`	{-31 .... 31} for fixed point APIs
`out_data_format`	Can be 0: `out_height x out_channels` or 1: `out_channels x out_height`

3.2.4 Depthwise Separable 2D Convolution Kernels

Depthwise Separable 2D Convolution is computed in two steps using following two low-level kernels:

- First step: xa_nn_conv2d_depthwise_xx() low-level kernel

  These kernels convolve each input 2D plane (`input_height x input_width`) from input cube (`input_height x input_width x input_channels`) with channels_multiplier number of 2D kernels (`kernel_height x kernel_width`) to produce channels_multiplier number of 2D output planes (`out_height x out_width`). Thus, with kernel cube of dimension (`kernel_height x kernel_width x (channels_multiplier * input_channels)`), output cube of dimension (`out_height x out_width x (channels_multiplier * input_channels)`) is produced. Bias is added to the convolution output. There is one bias value for each output 2D plane; that is, bias is a vector of dimension (`channels_multiplier * input_channels`).

- Second step: xa_nn_conv2d_pointwise_xx() low-level kernel

  These kernels take output cube (`out_height x out_width x (channels_multiplier * input_channels)`) of first step as input and perform pointwise multiplication with kernel vector (`channels_multiplier * input_channels`) in depth dimension to produce output 2D plane (`out_height x out_width`). Thus, with `out_channels` kernel vectors, output cube of

dimension (`out_height x out_width x out_channels`) is produced. Bias is added to the pointwise multiplication output. There is one bias value for each output 2D plane; that is, bias is a vector of dimension `out_channels`.

Following are the descriptions for these two low-level kernels.

Depthwise 2D Convolution Kernel

Description

These kernels perform the 2D depthwise convolution operation as `z = inp (*) kernel + bias`. These kernels convolve each input 2D plane (`input_height  x  input_width`) from input cube (`input_height x input_width x input_channels`) with channels_multiplier number of 2D kernels (`kernel_height  x  kernel_width`) to produce channels_multiplier number of 2D output planes (out_height x out_width). Thus, with kernel cube of dimension (`kernel_height x kernel_width x (channels_multiplier  *  input_channels)`), output cube of dimension (`out_height  x out_width x (channels_multiplier  *  input_channels)`) is produced. Bias is added to the convolution output. There is one bias value for each output 2D plane; that is, bias is a vector of dimension (`channels_multiplier * input_channels`).

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output respectively. Both `bias_shift` and `acc_shift` can be either positive or negative where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as convolution - accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note	The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels and quantized 8-bit kernels.

The `x_stride` and `y_stride` arguments in kernel API define the step size of the kernel when traversing the input in width and height dimensions, respectively.

The `x_padding` argument defines padding to the left of the input in the width dimension, and `y_padding` argument defines padding to the top of the input in the height dimension.

The right padding is calculated based on `out_width` as `right_paddding  =  kernel_width + (out_width – 1) * x_stride – (x_padding + input_width)`.

The bottom padding is calculated based on `out_height` as `bottom_paddding = kernel_height + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

These kernels require a temporary buffer for convolution computation. This temporary buffer is provided by the `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_conv2d_depthwise_getsize()` helper API.

The arguments `input_zero_bias`, `kernel_zero_bias` are provided to convert the quantized 8-bit inputs into their real values and perform Depthwise 2D Convolution operation. The `out_zero_bias`, `out_multiplier`, and `out_shift` values are used to quantize real values of output back to 8-bit.

The depthwise kernels expect input cube in SHAPE_CUBE_DWH_T and SHAPE_CUBE_WHD_T shape type and produce output cube in SHAPE_CUBE_DWH_T shape type respectively. The `inp_data_format` argument to the kernel API can be 0 or 1 to indicate input cube shape respectively.

The `out_data_format` argument to the kernel API must be 0 for all the kernels to indicate output cube shape.

Function variants available are `xa_nn_conv2d_depthwise_[p]`, where:

- `[p]`: precision in bits

Precision

There are seven variants available:

Type	Description
`16x16`	16-bit kernel, 16-bit input, 16-bit output
`8x16`	8-bit kernel, 16-bit input, 16-bit output
`8x8`	8-bit kernel, 8-bit input, 8-bit output
`f32`	float32 kernel, float32 input, float32 output
`asym8uxasym8u`	asym8u kernel, asym8u input, asym8u output
`per_chan_sym8sxasym8s`	per channel quantized sym8s kernel, asym8s input, asym8s output
`per_chan_sym8sxsym16s`	per channel quantized sym8s kernel, sym16s input, sym16s output
`f16`	float16 kernel, float16 input, float16 output

Algorithm

$$z_{h,w,d*C_M+m} = 2^{acc\text{-}shift}\left(\sum_{i=0}^{K_H-1}\sum_{j=0}^{K_W-1} in_{pad_{(h*y\text{-}stride+i),(w*x\text{-}stride+j),d}}\right.$$

$$\left. \cdot\, ker_{pad_{i,j,(d*C_M+m)}} + 2^{bias\text{-}shift} b_{0,0,d*C_M+m}\right)$$

$h = 0, \dots, \overline{out\text{-}height\ -\ 1}, w = 0, \dots, \overline{out\text{-}width\ -\ 1}$,
$d = 0, \dots, \overline{input\text{-}channels\ -\ 1}$,
$m = 0, \dots, \overline{channels\text{-}multiplier\ -\ 1}$

In case of floating-point kernel and quantized 8-bit kernels, `acc_shift=0` and `bias_shift=0`.

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

in_{pad}, ker_{pad} denote the padded `p_inp` and padded `p_ker` shapes, respectively.

K_H, K_W, C_M denote `kernel_height`, `kernel_width`, and `channels_multiplier`, respectively.

b denotes the `bias shape.`

Prototype

```
WORD32 xa_nn_conv2d_depthwise_getsize
(WORD32 input_height,       WORD32 input_width      WORD32 input_channels,
 WORD32 kernel_height,      WORD32 kernel_width,    WORD32 channels_multiplier,
 WORD32 x_stride,           WORD32 y_stride,        WORD32 x_padding,
 WORD32 y_padding,          WORD32 output_height,   WORD32 output_width,
 WORD32 circ_buf_precision, WORD32 inp_data_format);

WORD32 xa_nn_conv2d_depthwise_16x16
(WORD16 * p_out,            WORD16 *  p_ker,        WORD16 *   p_inp,
 WORD16 * p_bias,           WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,          WORD32 acc_shift,       WORD32 bias_shift,
 WORD32 inp_data_format,    WORD32 out_data_format, VOID   * p_scratch);
WORD32 xa_nn_conv2d_depthwise_8x16
(WORD16 * p_out,            WORD8 * p_ker,          WORD16 * p_inp,
 WORD16 * p_bias,           WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,          WORD32 acc_shift,       WORD32 bias_shift,
 WORD32 inp_data_format,    WORD32 out_data_format, VOID   * p_scratch);
WORD32 xa_nn_conv2d_depthwise_8x8
(WORD8  * p_out,            WORD8  * p_ker,         WORD8  * p_inp,
 WORD8  * p_bias,           WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,          WORD32 acc_shift,       WORD32 bias_shift,
 WORD32 inp_data_format,    WORD32 out_data_format, VOID   * p_scratch);
WORD32 xa_nn_conv2d_depthwise_f32
(FLOAT32 * p_out,           const FLOAT32 * p_ker,  const FLOAT32 * p_inp,
 const FLOAT32 * p_bias,    WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,   WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,          WORD32 inp_data_format, WORD32 out_data_format,
 VOID   * p_scratch);
WORD32 xa_nn_conv2d_depthwise_asym8uxasym8u
(pUWORD8 p_out,             const UWORD8 * p_kernel, const UWORD8 * p_inp,
 const WORD32 * p_bias,     WORD32 input_height,     WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,    WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,          WORD32 input_zero_bias,  WORD32 kernel_zero_bias,
 WORD32 out_multiplier,     WORD32 out_shift,        WORD32 out_zero_bias,
 WORD32 inp_data_format,    WORD32 out_data_format,  pVOID p_scratch);
WORD32 xa_nn_conv2d_depthwise_per_chan_sym8sxasym8s
(pWORD8 p_out,              const WORD8 * p_kernel,  const WORD8 * p_inp,
 const WORD32 * p_bias,     WORD32 input_height,     WORD32 input_width,
 WORD32 input_channels,     WORD32 kernel_height,    WORD32 kernel_width,
 WORD32 channels_multiplier,WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,          WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,          WORD32 input_zero_bias,  const WORD32 * p_out_multiplier,
```

```
 const WORD32 * p_out_shift,WORD32 out_zero_bias,    WORD32 inp_data_format,
 WORD32 out_data_format,    pVOID p_scratch);
WORD32 xa_nn_conv2d_depthwise_per_chan_sym8sxsym16s
 (pWORD16  p_out,                const WORD8 * p_kernel,    const WORD16 * p_inp,
  const WORD64 * p_bias,         WORD32  input_height,      WORD32  input_width,
  WORD32  input_channels,        WORD32  kernel_height,     WORD32  kernel_width,
  WORD32  channels_multiplier,   WORD32  x_stride,          WORD32  y_stride,
  WORD32  x_padding,             WORD32  y_padding,         WORD32  out_height,
  WORD32  out_width,             WORD32  input_zero_bias,   const WORD32 *p_out_multiplier,
  const WORD32 *p_out_shift,     WORD32  out_zero_bias,     WORD32  inp_data_format,
  WORD32  out_data_format,       pVOID p_scratch);
WORD32 xa_nn_conv2d_depthwise_f16
 (WORD16* p_out,                 const WORD16* p_kernel,    const WORD16* p_inp,
  const WORD16* p_bias,          WORD32  input_height,      WORD32  input_width,
  WORD32  input_channels,        WORD32  kernel_height,     WORD32  kernel_width,
  WORD32  channels_multiplier,   WORD32  x_stride,          WORD32  y_stride,
  WORD32  x_padding,             WORD32  y_padding,         WORD32  out_height,
  WORD32  out_width,             WORD32  inp_data_format,   WORD32  out_data_format,
  pVOID p_scratch);
```

Arguments

Type	Name	Size	Description
Input			
WORD16 *, WORD8 *, FLOAT32 *, const UWORD8 *, const WORD8 *	p_ker	kernel_height* kernel width* input_channels* channels_multiplier	Kernel cube, fixed, floating point, asym8u or sym8s, in SHAPE_CUBE_DWH or SHAPE_CUBE_WHD_T
WORD16 *, WORD8 *, FLOAT32 *, const UWORD8 *, const WORD8 *	p_inp	input_height* input width* input_channels	Input cube, fixed, floating point, asym8u or asym8s in SHAPE_CUBE_DWH or SHAPE_CUBE_WHD_T
WORD16 *, WORD8 *, FLOAT32 *, const WORD32 * const WORD64 *	p_bias	input_channels*channels_multiplier	Bias vector, fixed or floating point
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Number of input channels
WORD32	kernel_height		Kernel height
WORD32	kernel_width		Kernel width
WORD32	channels_multiplier		Multiplier value for each input channel
WORD32	x_stride		Horizontal stride over input

Type	Name	Size	Description
WORD32	y_stride		Vertical stride over input
WORD32	x_padding		Left padding width on input
WORD32	y_padding		Right padding height on input
WORD32	out_height		Output height
WORD32	out_width		Output width
WORD32	acc_shift		Shift applied to accumulator
WORD32	bias_shift		Shift applied to bias
WORD32	input_zero_bias		Zero offset of input
WORD32	kernel_zero_bias		Zero offset of kernel
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32 *	p_out_multiplier	input_channels*channels_multiplier	Array of multiplier values of output
WORD32 *	p_out_shift	input_channels*channels_multiplier	Array of shift values of output
WORD32	out_zero_bias		Zero offset of output
WORD32	inp_data_format		Input and Kernel data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
WORD32	out_data_format		Output data format 0:SHAPE_CUBE_DWH_T
VOID *	p_scratch	xa_nn_conv2d_depthwise_getsize()	Scratch memory pointer
Output			
WORD16 *, WORD8 *, UWORD8 * FLOAT32 *,	p_out	out_height* out width* input_channels* channels_multiplier	Output cube, fixed, floating point, asym8u or asym8s, in SHAPE_CUBE_DWH_T

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_out, p_ker, p_inp, p_bias,`	Cannot be NULL
	Should not overlap
	Aligned on <size of one element> boundary
`p_scratch`	Cannot be NULL
	Should not overlap with other buffers
	Aligned on 16-byte boundary
	For `p_scratch` – memory size >= size returned by `xa_nn_conv2d_depthwise_getsize( )`
`p_out_multiplier`	Cannot be NULL
	Should not overlap
	Aligned on 4-byte boundry
`p_out_shift`	Cannot be NULL
	Should not overlap
	Aligned on 4-byte boundry
	Each 32-bit value must be in range [-31 ... 31]
`input_height, input_width, input_channels`	Greater than or equal to 1
`kernel_height`	{1,2, ..., `input_height`}
`kernel_width`	{1,2, ..., `input_width`}
`channels_multiplier`	Greater than or equal to 1
`x_stride`	{1,2, ..., `kernel_width`}
`y_stride`	{1,2, ..., `kernel_height`}
`x_padding, y_padding`	Greater than or equal to 0
`out_height, out_width`	Greater than or equal to 1
`acc_shift,bias_shift, out_shift`	{-31 .... 31} for fixed point and quantized 8-bit APIs
`input_zero_bias`	{-255,....., 0} for asym8u input, {-127....., 128} for asym8s input Must be 0 for sym16s input
`kernel_zero_bias`	{-255......, 0} for asym8u kernel
`out_zero_bias`	{0,........,255} for asym8u output, {-128....., 127} for asym8s output Must be 0 for sym16s output
`out_multiplier`	Greater than 0
`inp_data_format`	can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T
`out_data_format`	must be 0: SHAPE_CUBE_DWH_T

Pointwise 2D Convolution Kernel

Description

These kernels perform pointwise multiplication of input cube (`input_height x input_width x input_channels`) with kernel vector (`input_channels`) in depth dimension to produce output 2D plane (`input_height x input_width`). Thus, with `out_channels` kernel vectors, output cube of dimension (`input_height x input_width x out_channels`) is produced. Bias is added to the pointwise

multiplication output. There is one bias value for each output 2D plane; that is, bias is a vector of dimension `out_channels`.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output respectively. Both `bias_shift` and `acc_shift` can be either positive or negative, where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as convolution - accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note	The `acc_shift` and `bias_shift` are not relevant in case of floating point kernels and quantized 8-bit kernels.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

These kernels expect input cube in SHAPE_CUBE_DWH_T shape type, kernel as matrix, bias as vector and produce output cube in SHAPE_CUBE_WHD_T or SHAPE_CUBE_DWH_T (only for 8x8, asym8uxasym8u and per_chan_sym8sxasym8s kernels) shape type. The `out_data_format` argument to kernel API must be always 1 except for 8x8 and quantized 8-bit kernels for which it can be 0 or 1 indicating SHAPE_CUBE_DWH_T and SHAPE_CUBE_WHD_T respectively.

Function variants available are `xa_nn_conv2d_pointwise_[p]`, where:

- `[p]`: precision in bits

Precision

There are seven variants available:

Type	Description
`16x16`	16-bit kernel, 16-bit input, 16-bit output
`8x16`	8-bit kernel, 16-bit input, 16-bit output
`8x8`	8-bit kernel, 8-bit input, 8-bit output
`f32`	float32 kernel, float32 input, float32 output
`asym8uxasym8u`	asym8u kernel, asym8u input, asym8u output
`per_chan_sym8sxasym8s`	per channel quantized sym8s kernel, asym8s input, asym8s output
`per_chan_sym8sxsym16s`	per channel quantized sym8s kernel, sym16s input, sym16s output
`f16`	float16 kernel, float16 input, float16 output

Algorithm

$$z_{h,w,d} = 2^{acc\text{-}shift} \left(\sum_{k=0}^{I_C-1} in_{h,w,k} \cdot ker_{d,0,0,k} + 2^{bias\text{-}shift} b_{0,0,d} \right)$$

$h = 0,..\overline{input\text{-}height-1}, w = 0,..\overline{input\text{-}width-1},$
$d = 0,..\overline{out_{channels}-1}$

In case of floating-point kernel and quantized 8-bit kernels, `acc_shift=0` and `bias_shift=0`. Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

in, ker denote the `p_inp`, and `p_ker` shapes respectively.

I_C denotes `input_channels`

b denotes the `bias` shape

Prototype

```
WORD32 xa_nn_conv2d_pointwise_16x16
(WORD16 * p_out,          WORD16 * p_ker,        WORD16 * _inp,
 WORD16 * p_bias,         WORD32  input_height,  WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,   WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_8x16
(WORD16 * p_out,          WORD8  * p_ker,        WORD16 * p_inp,
 WORD16 * p_bias,         WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,   WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_8x8
(WORD8  * p_out,          WORD8  * p_ker,        WORD8  * p_inp,
 WORD8  * p_bias,         WORD32 input_height,   WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,   WORD32 acc_shift,
 WORD32 bias_shift,       WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_f32
(FLOAT32 * p_out,         const FLOAT32 * p_ker,  const FLOAT32 *  p_inp,
 const FLOAT32 *  p_bias,       WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,
 WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_asym8uxasym8u
(UWORD8 * p_out,          const UWORD8 * p_ker,   const UWORD8 * p_inp,
 WORD32 * p_bias,         WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,    WORD32 input_zero_bias,
 WORD32 kernel_zero_bias, WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias,    WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_asym8uxasym8u
(UWORD8 * p_out,          const UWORD8 * p_ker,   const UWORD8 * p_inp,
 const WORD32 * p_bias,   WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,    WORD32 input_zero_bias,
 WORD32 kernel_zero_bias, WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias,    WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_per_chan_sym8sxasym8s
(WORD8  * p_out,          const WORD8 * p_ker,    const WORD8 * p_inp,
 const WORD32 * p_bias,   WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,    WORD32 input_zero_bias,
 WORD32 * p_out_multiplier,WORD32 * p_out_shift,  WORD32 out_zero_bias,
 WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_per_chan_sym8sxsym16s
(WORD16  * p_out,         const WORD8 * p_ker,    const WORD8 * p_inp,
 const WORD64 * p_bias,   WORD32 input_height,    WORD32 input_width,
 WORD32 input_channels,   WORD32 out_channels,    WORD32 input_zero_bias,
 WORD32 * p_out_multiplier,WORD32 * p_out_shift,  WORD32 out_zero_bias,
 WORD32 out_data_format);
WORD32 xa_nn_conv2d_pointwise_f16
 (WORD16* p_out,          WORD16* p_kernel,       WORD16* p_inp,
  WORD16* p_bias,         WORD32  input_height,  WORD32  input_width,
```

```
WORD32  input_channels,  WORD32  out_channels,  WORD32  out_data_format);
```

Arguments

Type	Name	Size	Description
Input			
WORD16 *, WORD8 *, const FLOAT32 *, const UWORD8 *, const WORD8 *	p_ker	out_channels * input_channels	Kernel matrix, fixed, floating point, asym8u or asym8s, (out_channels x input_channels)
WORD16 *, WORD8 *, const FLOAT32 *, const UWORD8 *, const WORD8 *, const WORD16 *	p_inp	input_height* input width* input_channels	Input cube, fixed or floating point, asym8u or sym8s, in SHAPE_CUBE_DWH_T
WORD16 *, WORD8 *, FLOAT32 *, const WORD32 *, const WORD64 *	p_bias	out_channels	Bias vector, fixed or floating point
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Number of input channels
WORD32	out_channels		Number of output channels
WORD32	acc_shift		Shift applied to accumulator
WORD32	bias_shift		Shift applied to bias
WORD32	input_zero_bias		Zero offset of input
WORD32	kernel_zero_bias		Zero offset of kernel
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32	out_zero_bias		Zero offset of output
WORD32	out_data_format		Output data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
Output			
WORD16 *, WORD8 *, FLOAT32 *, UWORD8 *	p_out	(out_height* out_width)* out_channels	Output cube, fixed, floating point, asym8u or asym8s, as per the out_data_format argument.

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_out, p_ker, p_inp, p_bias`	Cannot be NULL
	Should not overlap
	Aligned on 16-byte boundary except for 8x8 and quantized 8-bit/16-bit kernels
`input_height, input_width`	Greater than or equal to 1
`input_channels`	Greater than or equal to 4, multiple of 4 except for 8x8 and and quantized datatype kernels
`out_channels`	Greater than or equal to 1
`acc_shift, bias_shift, out_shift`	{-31 …. 31} for fixed point and quantized datatype APIs
`input_zero_bias`	{-255,….., 0} for asym8u input, {-127….., 128} for asym8s input, 0 for sym16s input
`kernel_zero_bias`	{-255……, 0} for asym8u kernel
`out_zero_bias`	{0,........,255} for asym8u output, {-128….., 127} for asym8s output, 0 for sym16s output
`out_multiplier`	Greater than 0
`out_data_format`	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T for 8x8 and quantized 8-bit kernels. Must be 1 for other kernels.

3.2.5 Depthwise Separable 2D Convolution Kernels with Dilation

Description

These kernels perform the dilated 2D depthwise convolution operation as `z = inp (*) kernel + bias`. These kernels convolve each input 2D plane (`input_height x input_width`) from input cube (`input_height x input_width x input_channels`) with channels_multiplier number of 2D dilated kernels (`dilated_kernel_height x dilated_kernel_width`) to produce channels_multiplier number of 2D output planes (out_height x out_width). Thus, with kernel cube of dimension (`dilated_kernel_height x dilated_kernel_width x (channels_multiplier * input_channels)`), output cube of dimension (`out_height x out_width x (channels_multiplier * input_channels)`) is produced. Bias is added to the convolution output. There is one bias value for each output 2D plane; that is, bias is a vector of dimension (`channels_multiplier * input_channels`).

Kernel is dilated by inserting (dilation_height – 1) zeros between consecutive height elements and (dilation_width -1) zeros between consecutive width elements. Post dilation, the kernel cube is of size dilated_kernel_height = kernel_height + (kernel_height-1)*(dilation_height-1) in height dimension and dilated_kernel_width = kernel_width + (kernel_width-1)*(dilation_width-1) in width dimension.

The `x_stride` and `y_stride` arguments in kernel API define the step size of the kernel when traversing the input in width and height dimensions, respectively.

The `x_padding` argument defines padding to the left of the input in the width dimension, and `y_padding` argument defines padding to the top of the input in the height dimension.

The right padding is calculated based on `out_width` as `right_padding = dilated_kernel_width + (out_width – 1) * x_stride – (x_padding + input_width)`.

The bottom padding is calculated based on `out_height` as `bottom_padding = dilated_kernel_height + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

These kernels require a temporary buffer for convolution computation. This temporary buffer is provided by the `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_dilated_conv2d_depthwise_getsize()` helper API.

The argument `input_zero_bias` is provided to convert the asym8s inputs into their real values and perform Dilated Depthwise 2D Convolution operation. The `out_zero_bias`, `p_out_multiplier`, and `p_out_shift` arguments are used to quantize real values of output back to asym8s.

The depthwise kernels expect input cube in SHAPE_CUBE_DWH_T and SHAPE_CUBE_WHD_T shape type and produce output cube in SHAPE_CUBE_DWH_T shape type respectively. The `inp_data_format` argument to the kernel API can be 0 or 1 to indicate input cube shape, respectively.

The `out_data_format` argument to the kernel API must be 0 for all the kernels to indicate output cube shape.

Precision

The following two variants are available:

Type	Description
per_chan_sym8sxasym8s_asym8s	per channel quantized sym8s kernel, asym8s input, asym8s output
f32xf32_f32	Float 32-bit kernel, Float 32-bit input, Float 32-bit output

Algorithm

$$z_{h,w,d*C_M+m}$$
$$= \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} in_{pad_{(h*y_{stride}+i*dilation\_height),(w*x\text{-}stride +j*dilation\_width),d}} \right.$$
$$\left. \cdot ker_{pad_{i,j,(d*C_M+m)}} + b_{0,0,d*C_M+m} \right)$$

$h = 0, ..., \overline{out\text{-}height - 1}, w = 0, ..., \overline{out\text{-}width - 1},$
$d = 0, ..., \overline{input\text{-}channels - 1},$
$m = 0, ..., \overline{channels\text{-}multiplier - 1}$

in_{pad}, ker_{pad} denote the padded `p_inp` and padded `p_ker` shapes, respectively.

K_H, K_W, C_M denote `kernel_height`, `kernel_width`, and `channels_multiplier`, respectively.

b denotes the `bias shape`.

Prototype

```
WORD32 xa_nn_dilated_conv2d_depthwise_getsize
(WORD32 input_height,        WORD32 input_width,        WORD32 input_channels,
 WORD32 kernel_height,       WORD32 kernel_width,       WORD32 channels_multiplier,
 WORD32 dilation_height,     WORD32 dilation_width,     WORD32 x_stride,
 WORD32 y_stride,            WORD32 x_padding,          WORD32 y_padding,
 WORD32 output_height        WORD32 output_width        WORD32 circ_buf_precision
 WORD32 inp_data_format);


WORD32 xa_nn_dilated_conv2d_depthwise_f32
(FLOAT32* p_out,             const FLOAT32* p_kernel,        const FLOAT32* p_inp,
 const FLOAT32* p_bias,      WORD32  input_height,           WORD32  input_width,
 WORD32  input_channels,     WORD32  kernel_height,          WORD32  kernel_width,
 WORD32  channels_multiplier,WORD32  dilation_height,        WORD32  dilation_width,
 WORD32  x_stride,           WORD32  y_stride,               WORD32  x_padding,
 WORD32  y_padding,          WORD32  out_height,             WORD32  out_width,
 WORD32  inp_data_format,    WORD32  out_data_format,        pVOID p_scratch);


WORD32 xa_nn_dilated_conv2d_depthwise_per_chan_sym8sxasym8s
(pWORD8* p_out,              const WORD8* p_kernel,          const WORD8 * p_inp,
 const WORD32 *p_bias,       WORD32  input_height,           WORD32  input_width,
 WORD32  input_channels,     WORD32  kernel_height,          WORD32  kernel_width,
 WORD32  channels_multiplier,WORD32  dilation_height,        WORD32  dilation_width,
 WORD32  x_stride ,          WORD32  y_stride,               WORD32  x_padding,
 WORD32  y_padding,          WORD32  out_height,             WORD32  out_width,
 WORD32  input_zero_bias,    const WORD32 *p_out_multiplier, const WORD32 *p_out_shift,
 WORD32  out_zero_bias,      WORD32  inp_data_format,        WORD32  out_data_format,
 pVOID p_scratch);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const FLOAT32 *,	p_ker	kernel_height* kernel width* input_channels* channels_multiplier	Kernel matrix, sym8s or floating point in SHAPE_CUBE_DWH or SHAPE_CUBE_WHD_T
const WORD8 *, const FLOAT32 *,	p_inp	input_height* input width* input_channels	Input cube, asym8s or floating point, in SHAPE_CUBE_DWH or SHAPE_CUBE_WHD_T
const WORD32 *, const FLOAT32 *,	p_bias	input_channels* channels_multiplier	Bias vector, fixed or floating point
VOID *	p_scratch	xa_nn_dilated_conv2d _depthwise getsize()	Scratch memory pointer
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Number of input channels
WORD32	kernel_height		Kernel height
WORD32	kernel_width		Kernel width
WORD32	channels_multipl ier		Multiplier value for each input channel
WORD32	dilation_height		Kernel height dilation factor

WORD32	dilation_width		Kernel width dilation factor
WORD32	x_stride		Horizontal stride over input
WORD32	y_stride		Vertical stride over input
WORD32	x_padding		Left padding width on input
WORD32	y_padding		Top padding height on input
WORD32	out_height		Output height
WORD32	out_width		Output width
WORD32	input_zero_bias		Input offset
WORD32	output_zero_bias		Output offset
WORD32 *	p_out_multiplier		Vector having multiplier values of ouput for per channel quantization
WORD32 *	p_out_shift		Vector having shift values of output for per channel quantization
WORD32	inp_data_format		input data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
WORD32	out_data_format		Output data format 0:SHAPE_CUBE_DWH_T
Output			
WORD8 *, FLOAT32 *	p_out	(out_height* out_width)* input_channels* channels_multiplier	Output cube, floating point or asym8s, in SHAPE_CUBE_DWH_T

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_out, p_ker, p_inp	Cannot be NULL
	Must not overlap
p_bias, p_out_multiplier, p_out_shift	Cannot be NULL, Aligned on 4 byte boundary p_out_shift[i] {-31, …, 31}
p_scratch	Cannot be NULL, aligned on 16 byte boundary.
input_height, input_width, kernel_height, kernel_width, channel_multiplier,	Greater than 0
input_channels	Greater than 0
dilation_height, dilation_width,	Greater than 0
y_stride,x_stirde	Greater than 0
x_padding, y_padding	Greater than or equal to 0
out_height, out_width	Greater than 0
input_zero_bias	{-127, …, 128} for sym8sxasym8s variant
output_zero_bias	{-128, …, 127} for sym8sxasym8s variant
input_data_format	can be 0 or 1

output_data_format	Should be 0

3.2.6 Transpose Convolution

Description

The Transpose Convolution kernels perform a reverse convolution operation only in the sense that the transpose convolution output has the same spatial dimension as that of input in standard convolution. A transpose convolution layer is generally used for upsampling, that is, to generate an output which has more samples than the input.

As illustrated below, the input is multiplied with every value in the kernel and accumulated at appropriate indices in the output.
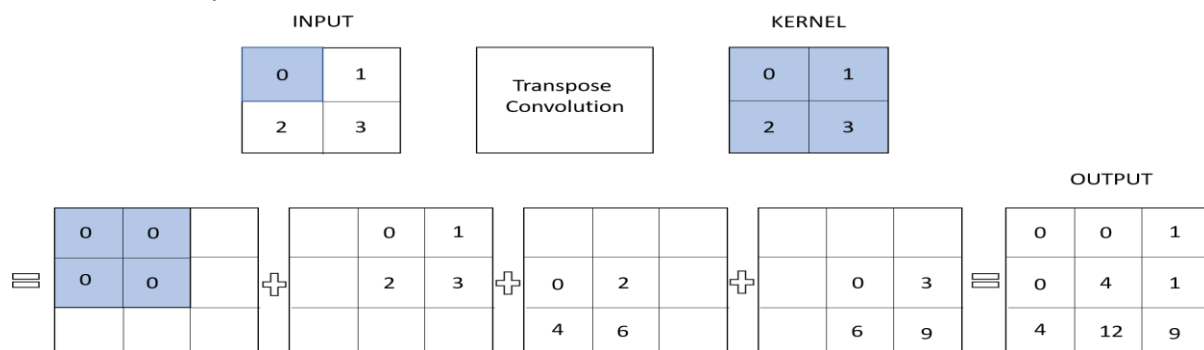
Figure 3-1 Example of Transpose Convolution (with padding 0 and stride 1)

These kernels require temporary buffer for convolution computation. This temporary buffer is provided by scratch_buffer argument of kernel API. The size of temporary buffer must be queried using xa_nn_transpose_conv_getsize() helper API.

The stride_width and stride_height arguments in kernel API define the step size to store intermediate multiplications in the width and height dimensions of the output respectively.

The pad_width and pad_height arguments define padding at the transpose convolution output, that is, original input to standard convolution.

Function variants available are xa_nn_transpose_conv_[p], where:

- [p]: precision

Precision

There are two variants available.

Type	Description
sym8sxsym16s	sym8s kernel, sym16s input, sym16s output
sym8sxasym8s	sym8s kernel, asym8s input, asym8s output
f32	f32 kernel, f32 input, f32 output

Algorithm

$$for\ iny = 0, \dots, \overline{input\_height - 1}$$
$$for\ inx = 0, \dots, \overline{input\_width - 1}$$
$$for\ inz = 0, \dots, \overline{input\_depth - 1}$$
$$for\ ky = 0, \dots, \overline{filter\_height - 1}$$
$$for\ kx = 0, \dots, \overline{filter\_width - 1}$$
$$for\ outz = 0, \dots, \overline{output\_depth - 1}$$

$$if\ (outx \in [0, out\_width - 1]\ \&\&\ outy \in [0, out\_height - 1]$$
$$Z_{outy,outx,outz} += \left(input_{iny,inx,inz} \cdot kernel_{outz,ky,kx,inz}\right)$$

Where,

$$outx = (inx * stride\_width) - pad\_width + kx$$

$$outy = (iny * stride\_height) - pad\_height + ky$$

Prototype

```
WORD32 xa_nn_transpose_conv_getsize
(WORD32 input_height,       WORD32 input_width,        WORD32 input_channels,
 WORD32 kernel_height,      WORD32 kernel_width,       WORD32 x_stride,
 WORD32 y_stride,           WORD32 output_height,      WORD32 output_width,
 WORD32 output_channels,    WORD32 kernel_precision,   WORD32 output_precision);


WORD32 xa_nn_transpose_conv_sym8sxsym16s
(WORD16 * output_data,      const WORD16 * input_data,const WORD8* filter_data,
 const WORD64 * bias_data,  WORD32 stride_width,       WORD32 stride_height,
 WORD32 pad_width,          WORD32 pad_height,         WORD32 input_depth,
 WORD32 output_depth,       WORD32 input_height,       WORD32 input_width,
 WORD32 filter_height,      WORD32 filter_width,       WORD32 output_height,
 WORD32 output_width,       WORD32 num_elements,       WORD32 * output_shift,
 WORD32 * output_multiplier,WORD64 * scratch_buffer);
int xa_nn_transpose_conv_sym8sxasym8s
(WORD8* output_data,        const WORD8* input_data, const WORD8* filter_data,
 const WORD32* bias_data,   int stride_width,          int stride_height,
 int pad_width,             int pad_height,            int input_depth,
 int output_depth,          int input_height,          int input_width,
 int filter_height,         int filter_width,          int output_height,
 int output_width,          int num_elements,          int input_offset,
 int output_offset,         int *output_shift,         int *output_multiplier,
 int32_t* scratch_buffer);
WORd32 xa_nn_transpose_conv_f32
(FLOAT32* output_data,      const FLOAT32* input_data,const FLOAT32* filter_data,
 const FLOAT32* bias_data,  int stride_width,          int stride_height,
 int pad_width,             int pad_height,            int input_depth,
 int output_depth,          int input_height,          int input_width,
 int filter_height,         int filter_width,          int output_height,
 int output_width,          int num_elements,          FLOAT32* scratch_buffer)
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const WORD16 * , const FLOAT32 *	input_data	input_height* input width* input_depth	Input cube, asym8s, sym16s or f32 SHAPE_CUBE_DWH_T

Type	Name	Size	Description
`const WORD8 * , const FLOAT32 *`	`filter_data`	`out_depth* (kernel_height * kernel width* input_depth)`	Kernel cube, f32 or fixed sym8s in SHAPE_CUBE_DWH_T
`const WORD64 *, const FLOAT32 *`	`bias_data`	`out_channels`	Bias vector, fixed point
`WORD32`	`input_offset`		Zero offset of input
`WORD32`	`output_offset`		Zero offset of output
`WORD32`	`input_height`		Input height
`WORD32`	`input_width`		Input width
`WORD32`	`input_depth`		Number of input channels
`WORD32`	`filter_height`		Kernel height
`WORD32`	`filter_width`		Kernel width
`WORD32`	`output_depth`		Number of output channels
`WORD32`	`pad_width`		Left padding width on output
`WORD32`	`pad_height`		Top padding height on output
`WORD32`	`stride_width`		Horizontal stride over output
`WORD32`	`stride_height`		Vertical stride over output
`WORD32`	`out_height`		Output height
`WORD32`	`out_width`		Output width
`WORD32 *`	`output_multiplier`		Multiplier value of output
`WORD32 *`	`output_shift`		Shift value of output
`WORD32`	`num_elements`	`(out_height* out_width)* output_depth`	Number of output points
`WORD64 * FLOAT32 *`	`scratch_buffer`	`xa_nn_transpose_conv_getsize()`	Scratch memory pointer
Output			
`WORD8 * WORD16 * FLOAT32 *`	`output_data`	`(out_height* out_width)* output_depth`	Output cube, asym8s, sym16s or f32.

Returns

- ◼ 0: no error
- ◼ -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`input_data, output_data`	Cannot be NULL Aligned on <size of one element> boundary. Should not overlap
`filter_data`	Cannot be NULL Aligned on <size of one element> boundary.
`scratch_buffer`	Cannot be NULL

Arguments	Restrictions
	Aligned on 16-byte boundary
`bias_data`	Cannot be NULL Aligned on 8-byte boundary for sym8sxsym16s and 4-byte for f32 and sym8sxasym8s.
`input_offset`	{-127, ..., 128}
`output_offset`	{-128, ..., 127}
`input_height,input_width, input_depth,filter_height, filter_width,output_depth, stride_height,stride_width, output_height,output_width, num_elements`	Greater than zero
`pad_height, pad_width`	Greater than or equal to zero

3.2.7 Group Convolution Kernel

Description

The Group Convolution kernels perform the 2D convolution operation as `z = inp(*)kernel + bias`. A 3D input cube (`input_height x input_width x input_channels`), is convolved with a 3D kernel cube (`kernel_height x kernel_width x kernel_channels`) to produce a 2D convolution output plane (`out_height  x  out_width`). With `out_channels` number of such 3D kernels, output cube (`out_height x out_width x out_channels`) is produced. The bias having the same dimensions as that of the output is added after the convolution to produce the final output.

Note   The depth or channels dimension (`input_channels`) must be multiple of kernel depth or channels(`kernel_channels`) for group convolution.

Note   The depth or channels dimension (`out_channels`) must be multiple of (`input_channels /kernel_channels`) for group convolution.

The `x_stride` and `y_stride` arguments in kernel API define the step size of the kernel when traversing the input in width and height dimensions respectively.

The `x_padding` argument defines padding to the left of the input in the width dimension and the `y_padding` argument defines padding to the top of the input in the height dimension.

The right padding is calculated based on `out_width` as `right_padding  =  kernel_width + (out_width – 1) * x_stride – (x_padding + input_width)`.

The bottom padding is calculated based on `out_height` as `bottom_padding = kernel_height + (out_height – 1) * y_stride – (y_padding + input_height)`.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

These kernels require temporary buffer for convolution computation. This temporary buffer is provided by `p_scratch` argument of kernel API. The size of temporary buffer must be queried using

`xa_nn_conv2d_std_getsize()` helper API, which is the same API used in standard Convolution. For Group convolution, the kernel_channels is passed as second argument.

The argument `input_zero_bias` is provided to convert the asym8s inputs into their real values and perform Group Convolution operation. The `out_zero_bias`, `p_out_multiplier`, and `p_out_shift` values are used to quantize real values of output back to asym8s.

These kernels expect input, kernel, and bias cubes in SHAPE‗CUBE‗DWH‗T shape type and can produce output cube in either SHAPE‗CUBE‗DWH‗T or SHAPE‗CUBE‗WHD‗T shape type. The `out_data_format` argument to kernel API controls the output cube shape type.

The function variants are available as `xa_nn_conv2d_group_[p]`, where:

- [p]: precision in bits

Precision

The following seven variants are available:

Type	Description
sym8sxasym8s	per channel quantized sym8s kernel, asym8s input, asym8s output

Algorithm

$$z_{h,w,d} = 2^{acc\text{-}shift} \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} \sum_{k=0}^{Kc-1} in_{pad_{(h*y\text{-}stride+i),(w*x\text{-}stride+j),(g*Kc+k)}} \cdot ker_{pad_{d,i,j,k}} \right.$$

$$\left. + 2^{bias\text{-}shift} b_{h,w,d} \right)$$

$h = 0, \ldots, \overline{out\text{-}height - 1}, w = 0, \ldots, \overline{out\text{-}width - 1},$
$d = 0, \ldots, \overline{out_{channels} - 1}$
$g = floor\left(\dfrac{d}{G}\right), where\ G = \dfrac{Ic}{Kc}$

in_{pad}, ker_{pad} denote the padded `p_inp` and padded `p_ker` shapes, respectively.

K_H, K_W, Kc, Ic denote kernel‗height, kernel‗width, kernel‗channels and input channels, respectively.

b denotes the `bias` shape.

Prototype

```
WORD32 xa_nn_conv2d_std_getsize
(WORD32 input_height,      WORD32 kernel_channels,WORD32 kernel_height,
 WORD32 kernel_width,      WORD32 y_stride,       WORD32 y_padding,
 WORD32 out_height,        WORD32 out_channels,   WORD32 input_precision);
```

```
WORD32 xa_nn_conv2d_group_sym8sxasym8s
(WORD8* __restrict__ p_out,                          const WORD8* __restrict__ p_inp,
 const WORD8* __restrict__ p_kernel,                 const WORD32* __restrict__ p_bias,
 WORD32 input_height,        WORD32 input_width,      WORD32 input_channels,
 WORD32 kernel_height,       WORD32 kernel_width,     WORD32 kernel_channels,
 WORD32 out_channels,        WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,           WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,           WORD32 input_zero_bias,  WORD32 * p_out_multiplier,
 WORD32 * p_out_shift,       WORD32 out_zero_bias,    WORD32 out_data_format,
 VOID *p_scratch);
```

Arguments

Type	Name	Size	Description
Input			
WORD8 *	p_inp	input_height* input width* input_channels	Input cube, fixed, floating point, asym8u or asym8s, in SHAPE_CUBE_DWH_T
WORD8 *	p_ker	out_channels* (kernel_height * kernel width* kernel_channels)	Kernel cube, fixed, floating point, asym8u or sym8s in SHAPE_CUBE_DWH_T
WORD32 *	p_bias	out_channels	Bias vector, fixed or floating point
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Number of input channels
WORD32	kernel_height		Kernel height
WORD32	kernel_width		Kernel width
WORD32	kernel_channels		Kernel channels
WORD32	out_channels		Number of output channels
WORD32	x_stride		Horizontal stride over input
WORD32	y_stride		Vertical stride over input
WORD32	x_padding		Left padding width on input
WORD32	y_padding		Top padding height on input
WORD32	out_height		Output height
WORD32	out_width		Output width
WORD32	input_zero_bias		Zero offset of input
const WORD32 *	p_out_multiplier		Vector having multiplier values of ouput for per channel quantization
const WORD32 *	p_out_shift		Vector having shift values of output for per channel quantization
WORD32	out_zero_bias		Zero offset of output
WORD32	out_data_format		Output data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
VOID *	p_scratch	xa_nn_conv2d_std_getsize()	Scratch memory pointer
Output			
WORD8 *	p_out	(out_height* out_width)* out_channels	Output cube, fixed, floating point, asym8u or asym8s as per the out_data_format argument.

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_ker, p_scratch`	Cannot be NULL
	Must not overlap
	Aligned on 8-byte boundary (p_bias needs to be only 4-byte aligned for asym8 variant)
	For `p_scratch` – memory size >= size returned by `xa_nn_conv2d_std_getsize()`
`p_out, p_inp, p_bias`	Cannot be NULL
	Must not overlap
	Aligned on (size of one element)-byte boundary
`input_height, input_width`	Greater than or equal to 1
`p_out_multiplier, p_out_shift`	Cannot be NULL, must not overlap, aligned to 4-byte boundary
`kernel_height`	{1, 2, ..., `input_height`}
`kernel_width`	{1, 2, ..., `input_width`}
`Kernel_channels`	Greater than or equal to 1
`input_channels`	Greater than or equal to 1
	input channels must be multiple of kernel_channels
`out_channels`	Greater than or equal to 1
	out_channels must be multiple of (input_channels/kernel_channels) i.e groups.
`x_stride`	Greater than or equal to 1
`y_stride`	Greater than or equal to 1
`x_padding, y_padding`	Greater than or equal to 0
`out_height, out_width`	Greater than or equal to 1
`input_zero_bias`	{-127, ..., 128} for sym8sxasym8s variant
`out_zero_bias`	{-128 ..., 127} for sym8sxasym8s variant
`out_data_format`	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T

3.3   Activation Kernels

3.3.1 Sigmoid

Description

The Sigmoid kernels perform the sigmoid operation on input vector `x` and give output vector as `y = sigmoid(x)`. Both the input and output vectors have size `vec_length`.

The 32-bit input fixed-point kernels accept 32-bit input in Q6.25 format and give output in Q16.15 (32-bit), Q15 (16-bit), or Q7 (8-bit) format. The 16-bit input/output fixed-point kernel accepts the input in Q3.12 and give output in Q15 (16-bit) format.

For the sym16s, asym8u and asym8s kernels both the input and output are of sym16s, asym8u and asym8s datatype respectively.

The 16-bit fixed point variant and the quantized 8-bit variants of sigmoid are based on TensorFlow implementations. 16-bit fixed point and quantized signed 8-bit variants support improved optimization (but 1-bit difference with TensorFlow implementation) for HiFi5 cores which have activation tie instructions [5.1].

Similarly, the sigmoid sym16s kernel supports improved optimization (but 1-bit difference with respect to Tensorflow implementation) for HiFi5 cores with activation tie instructions when the actual inp values (dequantized) are in the range -8 to 8. [5.1].

Note	The `input_range_radius` argument for quantized 8-bit variants is derived from other input parameters in TensorFlow. The kernel does not perform dependency check on the `input_range_radius` and the user will have to ensure that correct value is passed.

Function variants available are `xa_nn_vec_sigmoid_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are eight variants available.

Type	Description
`32_32`	32-bit input, 32-bit output
`32_16`	32-bit input, 16-bit output
`32_8`	32-bit input, 8-bit output
`16_16`	16-bit input, 16-bit output
`f32_f32`	float32 input, float32 output
`asym8u_asym8u`	asym8u input, asym8u output
`asym8s_asym8s`	asym8s input, asym8s output
`sym16s_sym16s`	sym16s input, sym16s output

Algorithm

$$y_n = \frac{1}{1 + \exp(-x_n)} \ , \qquad n = 0, \ldots, \overline{vec\text{-}length - 1}$$

Prototype

```
WORD32 xa_nn_vec_sigmoid_32_32
(WORD32 * p_out,          const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_32_16
(WORD16 * p_out,          const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_32_8
(WORD8 * p_out,           const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_f32_f32
(FLOAT32 * p_out,         const FLOAT32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_asym8u_asym8u
(UWORD8 * p_out,          const UWORD8 * p_vec,    WORD32 zero_point,
 WORD32 input_range_radius, WORD32 input_multiplier, WORD32 input_left_shift,
 WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_asym8s_asym8s
(WORD8 * p_out,           const WORD8 * p_vec,     WORD32 zero_point,
 WORD32 input_range_radius, WORD32 input_multiplier, WORD32 input_left_shift,
 WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_16_16
(WORD16 * p_out,          const WORD16 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_sigmoid_sym16s_sym16s
(WORD16 *p_out,           const WORD16 *p_vec,     WORD32 input_multiplier,
 WORD32 input_left_shift,  WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *, const WORD16 *, const UWORD8 *, const FLOAT32 *, const WORD8 *	p_vec	vec_length	Input vector, Q6.25, Q3.12, floating point, asym8u, asym8s or sym16s
WORD32	zero_point		bias value
WORD32	input_range_radius		Range radius: For asym8u output = ((x_i - zero$_$point) < radius)? sigmoid() : 255 output = ((x_i - zero$_$point) > (-radius))? sigmoid() : 0 For asym8s output = ((x_i - zero$_$point) < radius)? sigmoid() : 127 output = ((x_i - zero$_$point) > (-radius))? sigmoid() : -128
WORD32	input_multiplier		Multiplier value of input
WORD32	input_left_shift		Left Shift value of input

WORD32	`vec_length`		Length of input vector
Output			
WORD32 *, WORD16 *, WORD8 *, UWORD8 *, FLOAT32 *	`p_out`	`vec_length`	Output vector, fixed (Q16.15, Q15, Q7), floating point, asym8u, asym8s or sym16s

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out`	Should not overlap
	Cannot be NULL
`zero_point`	[0, 255] for asym8u [-128, 127] for asym8s
`input_range_radius`	Greater than or equal to 0
`input_left_shift`	[-31, 31] for asym8u and asym8s Should be greater than or equal to 0 for sym16s kernel
`input_multiplier`	Should not be less than 0.
`vec_length`	Greater than 0

3.3.2 Tanh

Description

The Tanh kernels perform the hyperbolic tangent operation on input vector x and give output vector as y = `tanh(x)`. Both the input and output vectors have size `vec_length`.

The 32-bit input fixed-point kernels accept 32-bit input in Q6.25 format and give output in Q16.15 (32-bit), Q15 (16-bit), or Q7 (8-bit) format. The 16-bit fixed-point kernel has input argument `integer_bits` to specify the number of integer bits in input so input Q format is Q(`integer_bits`).(15 - `integer_bits`), output is given in Q15 (16-bit) format.

For the sym16s and asym8s kernels both the input and output are of sym16s and asym8s datatype respectively.

The 16-bit fixed point variant and the quantized 8-bit variants of tanh are based on Tensorflow implementations. 16-bit fixed point and quantized signed 8-bit variants support improved optimization (but 1-bit difference with TensorFlow implementation) for HiFi5 cores which have activation tie instructions [5.1].

Similarly, the tanh sym16s kernel supports improved optimization (but 1-bit difference with respect to Tensorflow implementation) for HiFi5 cores with activation tie instructions when the actual inp values (dequantized) are in the range -8 to 8. [5.1].

Note	The `input_range_radius` argument for quantized 8-bit variant is derived from other input parameters in TensorFlow. The kernel does not perform dependency check on the `input_range_radius` and the user will have to ensure that correct value is passed.

Function variants available are `xa_nn_vec_tanh_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are seven variants available:

Type	Description
`32_32`	32-bit input, 32-bit output
`32_16`	32-bit input, 16-bit output
`32_8`	32-bit input, 8-bit output
`16_16`	16-bit input, 16-bit output
`f32_f32`	float32 input, float32 output
`asym8s_asym8s`	asym8s input, asym8s output
`sym16s_sym16s`	sym16s input, sym16s output

Algorithm

$$y_n = \tanh(x_n), \qquad n = 0, \ldots, \overline{vec\text{-}length - 1}$$

Prototype

```
WORD32 xa_nn_vec_tanh_32_32
(WORD32 * p_out,          const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_tanh_32_16
(WORD16 * p_out,          const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_tanh_32_8
(WORD8 * p_out,           const WORD32 * p_vec,    WORD32 vec_length);
WORD32 xa_nn_vec_tanh_f32_f32
(FLOAT32 * p_out,         const FLOAT32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_tanh_asym8s_asym8s
(WORD8 * p_out,           const WORD8 * p_vec,     WORD32 zero_point,
 WORD32 input_range_radius, WORD32 input_multiplier, WORD32 input_left_shift,
 WORD32 vec_length);
WORD32 xa_nn_vec_tanh_16_16
(WORD16 * p_out,          const WORD16 *p_vec,     WORD32 integer_bits,
 WORD32 vec_length);
WORD32 xa_nn_vec_tanh_sym16s_sym16s
(WORD16 *p_out,           const WORD16 *p_vec,     WORD32 input_multiplier,
 WORD32 input_left_shift,  WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			

Type	Name	Size	Description
`const WORD32 *, const WORD16 *, const FLOAT32 *, const WORD8 *`	`p_vec`	`vec_length`	Input vector, Q6.25, Q(integer_bits).(15-integer_bits), floating point, asym8s or sym16s
`WORD32`	`zero_point`		Bias value
`WORD32`	`input_range_radius`		Range radius: output = ((x_i - zero_point) < radius)? tanh() : 127 output = ((x_i - zero_point) > (-radius))? tanh() : -128
`WORD32`	`input_multiplier`		Multiplier value of input
`WORD32`	`input_left_shift`		Left shift value of input
`WORD32`	`vec_length`		Length of input vector
`WORD32`	`integer_bits`		Number of integer bits in the 16-bit input
Output			
`WORD32 *, WORD16 *, WORD8 *, FLOAT32 *`	`p_out`	`vec_length`	Output vector, fixed (Q16.15, Q15, Q7), floating point, asym8s or sym16s

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out`	Should not overlap
	Cannot be NULL
`zero_point`	[-128, 127]
`input_range_radius`	Greater than or equal to 0
`input_multiplier`	Should not be less than 0
`input_left_shift`	[-31,31] for asym8s kernel Should be greater than or equal to 0 for sym16s kernel
`vec_length`	Greater than 0
`integer_bits`	[0, 6]

3.3.3 Rectifier Linear Unit (ReLU)

Description

The Rectifier Linear Unit (ReLU) kernels compute the rectifier linear unit function of input vector `x` and give output vector as `y = relu(x)`. Both the input and output vectors have size `vec_length`.

The fixed-point routines accept 32-bit input in Q6.25 format and gives 32-bit output in Q16.15 format.

The `threshold` argument to `relu` kernel API allows to set upper threshold for proper compression of output signal and is expected in Q16.15 format. In `relu1` and `relu6` kernels, the thresholds are set to 1 and 6, respectively.

For the asym8u and asym8s kernels, the quantized input is requantized and applied the standard ReLU function to give the output. The `threshold` argument is not applicable for quantized ReLU kernels.

The standard ReLU kernels `relu_std` can be used when the `threshold` is not required.

Function variants available are `xa_nn_vec_relu_[p]_[q]`, `xa_nn_vec_relu1_[p]_[q]`, and `xa_nn_vec_relu6_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are six variants available:

Type	Description
`32_32`	32-bit input, 32-bit output
`f32_f32`	float32 input, float32 output
`16_16`	16-bit input, 16-bit output
`8_8`	8-bit input, 8-bit output
`asym8u_asym8u`	asym8u input, asym8u output
`asym8s_asym8s`	asym8s input, asym8s output

Algorithm

$$y_n = \max(0, \min(x_n, K)), \qquad n = 0, \ldots, \overline{vec\text{-}length - 1}$$

K represents `threshold`

Prototype

```
WORD32 xa_nn_vec_relu_32_32
(WORD32 * p_out,      const WORD32 * p_vec,    WORD32 threshold,
 WORD32 vec_length);
WORD32 xa_nn_vec_relu_f32_f32
(FLOAT32 * p_out,     const FLOAT32 * p_vec,  FLOAT32 threshold,
 WORD32 vec_length);
```

```
WORD32 xa_nn_vec_relu_16_16
(WORD16 * p_out,      const WORD16 * p_vec,   WORD16 threshold,
 WORD32 vec_length);
WORD32 xa_nn_vec_relu_8_8
(WORD8 * p_out,       const WORD8  * p_vec,   WORD8 threshold,
 WORD32 vec_length);
WORD32 xa_nn_vec_relu_asym8u_asym8u
(UWORD8 * p_out,        const UWORD8 * p_vec,WORD32 inp_zero_bias,
 WORD32 out_multiplier, WORD32 out_shift,    WORD32 out_zero_bias,
 WORD32 quantized_activation_min, WORD32 quantized_activation_max,
 WORD32 vec_length);
WORD32 xa_nn_vec_relu_asym8s_asym8s
(WORD8 * p_out,         const WORD8 * p_vec, WORD32 inp_zero_bias,
 WORD32 out_multiplier, WORD32 out_shift,    WORD32 out_zero_bias,
 WORD32 quantized_activation_min, WORD32 quantized_activation_max,
 WORD32 vec_length);
WORD32 xa_nn_vec_relu1_32_32
(WORD32 * p_out,     const WORD32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_relu1_f32_f32
(FLOAT32 * p_out,    const FLOAT32 * p_vec,  WORD32 vec_length);
WORD32 xa_nn_vec_relu6_32_32
(WORD32 * p_out,     const WORD32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_relu6_f32_f32
(FLOAT32 * p_out,    const FLOAT32 * p_vec,  WORD32 vec_length);
WORD32 xa_nn_vec_relu_std_32_32
(WORD32 * p_out,     const WORD32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_relu_std_f32_f32
(FLOAT32 * p_out,    const FLOAT32 * p_vec,  WORD32 vec_length);
WORD32 xa_nn_vec_relu_std_16_16
(WORD16 * p_out,     const WORD16 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_relu_std_8_8
(WORD8 * p_out,      const WORD8  * p_vec,   WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *, const FLOAT32 *, const WORD16 *, const WORD8 *, const UWORD8 *	p_vec	vec_length	Input vector, fixed-point, floating point, asym8u or asym8s
WORD32	inp_zero_bias		Zero bias value for input vector
WORD32	out_multipler		Fixed-point multiplier value for output
WORD32	out_shift		Shift value for output
WORD32	vec_length		length of input vector
WORD32	out_zero_bias		Zero bias value for output vector
WORD32	quantized_activation_min		Lower threshold value, quantized.
WORD32, FLOAT32	quantized_activation_max		Upper threshold value, quantized
WORD32 FLOAT32	threshold		threshold, fixed or floating point

Type	Name	Size	Description
`WORD16` `WORD8`			
Output			

`WORD32 *,`
`FLOAT32 *,`
`WORD16 *,`
`WORD8 *,`
`UWORD8 *`
	`p_out`	`vec_length`	Output vector, fixed-point, floating point, asym8u or asym8s

Returns

- ◼ 0: no error

- ◼ -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out`	Should not overlap
	Cannot be NULL
`inp_zero_bias,` `out_zero_bias`	{0,........,255} for asym8u, {-128….., 127} for asym8s input
`out_multiplier`	Should not be less than 0.
`out_shift`	{-31, …, 31}
`quantized_activation_min` `quantized_activation_max`	{0,........,255} for asym8u output, {-128….., 127} for asym8s output
	quantized_activation_min < quantized_activation_max

3.3.4 Softmax

Description

The Softmax kernels compute the softmax (normalized exponential function) of input vector `x` and give output vector as `y = softmax(x)`. Both the input and output vectors have size `vec_length`.

The fixed-point kernels accept 32-bit input in Q6.25 format and give 32-bit output in Q16.15 format.

For the asym8u kernels, both the input and output are of the same precision and for asym8s kernels, the input is asym8s and the output precision can be asym8s or 16-bit.

These kernels require temporary buffer for softmax computation. This temporary buffer is provided by p_scratch argument of kernel API. The size of temporary buffer must be queried using get_softmax_scratch_size() helper API.

Function variants available are `xa_nn_vec_softmax_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are five variants available:

Type	Description
`32_32`	32-bit input, 32-bit output
`f32_f32`	float32 input, float32 output
`asym8u_asym8u`	asym8u input, asym8u output
`asym8s_asym8s`	asym8s input, asym8s output
`asym8s_16`	asym8s input, 16-bit output

Algorithm

$$y_n = \frac{\exp(x_n)}{\sum_k \exp(x_k)}, \qquad n = 0, \ldots, \overline{vec_{length} - 1}\ , \qquad k = 0, \ldots, \overline{vec\text{-}length - 1}$$

Prototype

```
WORD32 xa_nn_vec_softmax_32_32
(WORD32 * p_out,      const WORD32 * p_vec,   WORD32 vec_length);
WORD32 xa_nn_vec_softmax_f32_f32
(FLOAT32 * p_out,     const FLOAT32 * p_vec,  WORD32 vec_length);
WORD32 xa_nn_vec_softmax_asym8u_asym8u
(UWORD8 * p_out,      const UWORD8 * p_vec,   WORD32 diffmin,
 WORD32 input_left_shift,     WORD32 input_multiplier,
 WORD32  vec_length,          pVOID  p_scratch);
WORD32 xa_nn_vec_softmax_asym8s_asym8s
(WORD8 * p_out,       const WORD8 * p_vec,    WORD32 diffmin,
 WORD32 input_left_shift,     WORD32 input_multiplier,
```

```
 WORD32  vec_length,             pVOID  p_scratch);
WORD32 xa_nn_vec_softmax_asym8s_16
(WORD16 * p_out,      const WORD8 * p_vec,    WORD32 diffmin,
 WORD32 input_left_shift,     WORD32 input_multiplier,
 WORD32  vec_length,             pVOID  p_scratch);
int get_softmax_scratch_size
(int inp_precision,           int out_precision,      int length);
```

Arguments

Type	Name	Size	Description
Input			

`WORD32 *,`
`FLOAT32 *,`
`const`
`UWORD8 *,`
`const`
`WORD8 *`
	`p_vec`	`vec_length`	Input vector, Q6.25, floating point, asym8u or asym8s

`WORD32`	`diffmin`		Diffmin value: output = ((x_i – max) > diffmin) ? softmax() : 0
`WORD32`	`input_` `left_shift`		left shift value of input
`WORD32`	`input_` `multiplier`		multiplier value of input
`WORD32`	`vec_length`		Length of input vector
Output			

`WORD32 *,`
`FLOAT32 *,`
`UWORD8 *,`
`WORD8 *,`
`WORD16 *`
	`p_out`	`vec_length`	Output vector, Q16.15, floating point, asym8u, asym8s or 16-bit.

Temporary			
`VOID *,`	`p_scratch`		Scratch (temporary) memory pointer

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out`	Should not overlap
	Cannot be NULL
`input_left_shift`	[-31, 31]
`input_multiplier`	Should not be less than 0.
`vec_length`	Greater than 0

3.3.5 Activation Min Max

Description

The Activation Min Max kernels compute the activation minimum and maximum value of input vector `x` and give output vector as `y` = `activation_min_max(x)`. Both the input and output vectors have size `num_elm`.

For activation min max kernels, the input precision and the output precision are the same.

The `activation_min` and `activation_max` arguments to the kernel API allow to set the threshold for proper compression of the output. The kernel is a generic implementation of the ReLU function.

Function variant available is `xa_nn_vec_activation_min_max_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are four variants available:

Type	Description
`f32_f32`	float32 input, float32 output
`asym8uxasym8u`	asym8u input, asym8u output
`16_16`	16-bit input, 16-bit output
`8_8`	8-bit input, 8-bit output

Algorithm

$$y_n = \max(activation\text{-}min, \min(x_n, activation\text{-}max)), \qquad n = 0, \ldots, \overline{vec\text{-}length - 1}$$

activation-min represents lower threshold.

activation-max represents upper threshold.

Prototype

```
WORD32 xa_nn_vec_activation_min_max_f32_f32
(FLOAT32 * p_out,        const FLOAT32 * p_vec, FLOAT32 activation_min,
 FLOAT32 activation_max,WORD32  vec_length);
WORD32 xa_nn_vec_activation_min_max_asym8u_asym8u
(UWORD8 * p_out,        const UWORD8 * p_vec,  int activation_min,
 int activation_max,    WORD32 vec_length);
WORD32 xa_nn_vec_activation_min_max_16_16
(WORD16 * p_out,        const WORD16 * p_vec,  int activation_min,
 int activation_max,    WORD32 vec_length);
WORD32 xa_nn_vec_activation_min_max_8_8
(WORD8 * p_out,         const WORD8 * p_vec,   int activation_min,
 int activation_max,    WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
`const UWORD8 *,` `const FLOAT32 *,` `const WORD16 *,` `const WORD8 *`	`p_vec`	`vec_length`	Input vector, floating-point,asym8u or fixed point.
`WORD32`	`vec_length`		Length of input vector

Type	Name	Size	Description
WORD32, FLOAT32	activation_min		Lower threshold value, floating-point, asym8u or fixed point.
WORD32, FLOAT32	activation_max		Upper threshold value, floating-point, asym8u or fixed point
Output			
UWORD8 *, FLOAT32 *, WORD16 *, WORD8 *	p_out	vec_length	Output vector, floating-point, asym8u or fixed point

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_vec, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL

3.3.6 Hard Swish

Description

The Hard Swish kernels compute the hard-swish function of input vector `x` and give output vector as `y = hard_swish(x)`. Both the input and output vectors have size `vec_length`.

The hard-swish activation function is a type of activation function based on swish but replaces the computationally expensive sigmoid function by ReLU6.

Function variants available are `xa_nn_vec_hard_swish_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There is one variant available:

Type	Description
asym8s_asym8s	asym8s input, asym8s output

Algorithm

$$y_n = x_n * [\text{ReLU6}(x_n + 3)/6], \qquad n = 0, \ldots, \overline{vec\text{-}length - 1}$$

Prototype

```
WORD32 xa_nn_vec_hard_swish_asym8s_asym8s
(WORD8 * p_out,            const WORD8 * p_vec,    WORD32 inp_zero_bias,
 WORD16 reluish_multiplier,WORD32 reluish_shift,   WORD16 out_multiplier,
 WORD32 out_shift,         WORD32 out_zero_bias,   WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_vec	vec_length	Input vector, asym8s
WORD32	inp_zero_bias		Zero bias value for input vector
WORD16	reluish_multipler		Fixed-point multiplier value for reluish scale
WORD32	reluish_shift		Shift value for reluish scale
WORD16	out_multipler		Fixed-point multiplier value for output
WORD32	out_shift		Shift value for output
WORD32	out_zero_bias		Zero bias value for output vector
WORD32	vec_length		length of input vector
Output			
WORD8 *	p_out	vec_length	Output vector, asym8s

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_vec, p_out	Cannot be NULL Should not overlap (the two pointers could be same, inplace operation is possible)
inp_zero_bias, out_zero_bias	{-128….., 127} for asym8s datatype
out_multiplier, reluish_multiplier	Should not be less than 0
out_shift,reluish_shift	{-31, ..., 31}

3.3.7 Parametric ReLU (PReLU)

Description

The Parametric ReLU kernels compute the Parametric ReLU function of input vector x and give output vector as $y = \text{prelu}(x)$. Both the input and output vectors have size vec_length.

The PReLU activation function acts like a standard ReLU function for input values greater than or equal to 0. For input values less than 0, a learnable negative slope parameter alpha(a) is multiplied with input to get the output. This slope value for all the input elements is determined based on the alpha input vector.

Function variants available are `xa_nn_vec_prelu_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There is one variant available:

Type	Description
`asym8s_asym8s`	asym8s input, asym8s output

Algorithm

$$y_n = x_n, \quad when\ x_n \geq 0 \qquad n = 0, ..., \overline{vec\text{-}length - 1}$$
$$y_n = ax_n, \quad when\ x_n < 0$$

where a is the learnable negative slope parameter: alpha.

Prototype

```
WORD32 xa_nn_vec_prelu_asym8s_asym8s
(WORD8 * p_out,        const WORD8 * p_vec,    const WORD8 * p_vec_alpha,
 WORD32 inp_zero_bias, WORD32 alpha_zero_bias, WORD32 alpha_multiplier,
 WORD32 alpha_shift,   WORD32 out_multiplier,  WORD32 out_shift,
 WORD32 out_zero_bias, WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_vec	vec_length	Input vector, asym8s
const WORD8 *	p_vec_alpha	vec_length	alpha input vector, asym8s
WORD32	inp_zero_bias		Zero bias value for input vector
WORD32	alpha_zero_bias		Zero bias value for alpha input vector
WORD16	alpha_multipler		Fixed-point multiplier value for alpha input.
WORD32	alpha_shift		Shift value for alpha input.
WORD16	out_multipler		Fixed-point multiplier value for output
WORD32	out_shift		Shift value for output
WORD32	out_zero_bias		Zero bias value for output vector
WORD32	vec_length		length of input vector

Output			
`WORD8 *`	`p_out`	`vec_length`	Output vector, asym8s

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out, p_vec_alpha`	Cannot be NULL
	Should not overlap (the two pointers could be same, inplace operation is possible)
`inp_zero_bias,` `alpha_zero_bias`	{-127....., 128} for asym8s datatype
`out_zero_bias`	{-128....., 127} for asym8s datatype
`out_multiplier,` `alpha_multiplier`	Should not be less than 0
`out_shift,alpha_shift`	{-31, ..., 31}

3.3.8 Leaky ReLU

Description

The Leaky ReLU kernels compute the Leaky ReLU function of input vector `x` and give output vector as `y = leaky_relu(x)`. Both the input and output vectors have size `vec_length`.

The Leaky ReLU activation function acts like a standard ReLU function for input values greater than or equal to 0. For input values less than 0, a negative slope parameter alpha(a) is multiplied with input to get the output. The slope value is constant for all the input elements.

Function variants available are `xa_nn_vec_leaky_relu_[p]_[q]`, where:

- `[p]`: Input precision in bits
- `[q]`: Output precision in bits

Precision

There are two variants available:

Type	Description
`asym8s_asym8s`	asym8s input, asym8s output
`asym16s_asym16s`	asym16s input, asym16s output

Algorithm

$$y_n = x_n, \qquad when \; x_n \geq 0 \qquad n = 0, ..., \overline{vec\text{-}length - 1}$$
$$y_n = ax_n, \qquad when \; x_n < 0$$

where a is the negative slope parameter: alpha.

Prototype

```
WORD32 xa_nn_vec_leaky_relu_asym8s_asym8s
(WORD8 * p_out,          const WORD8 * p_vec, WORD32 inp_zero_bias,
 WORD32 alpha_multiplier,WORD32 alpha_shift,  WORD32 out_multiplier,
 WORD32 out_shift,       WORD32 out_zero_bias,WORD32 vec_length);
WORD32 xa_nn_vec_leaky_relu_asym16s_asym16s
(WORD16 * p_out,          const WORD16 * p_vec,WORD32 inp_zero_bias,
 WORD32 alpha_multiplier,WORD32 alpha_shift,  WORD32 out_multiplier,
 WORD32 out_shift,       WORD32 out_zero_bias,WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const WORD16 *	p_vec	vec_length	Input vector, asym8s or asym16s
WORD32	inp_zero_bias		Zero bias value for input vector
WORD16	alpha_multipler		Fixed-point multiplier value for alpha input.
WORD32	alpha_shift		Shift value for alpha input.
WORD16	out_multipler		Fixed-point multiplier value for output
WORD32	out_shift		Shift value for output
WORD32	out_zero_bias		Zero bias value for output vector
WORD32	vec_length		length of input vector
Output			
WORD8 *, WORD16 *	p_out	vec_length	Output vector, asym8s or asym16s

Returns

- ■ 0: no error
- ■ -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_vec, p_out`	Cannot be NULL
	Aligned on size of one element byte boundary
	Should not overlap (the two pointers could be same, inplace operation is possible)
`inp_zero_bias`	{-128....., 127} for asym8s datatype {-32768, ... 32767} for asym16s datatype
`out_zero_bias`	{-128....., 127} for asym8s datatype {-32768, ... 32767} for asym16s datatype
`out_multiplier, alpha_multiplier`	Should not be less than 0
`out_shift,alpha_shift`	{-31, ..., 31}

3.4 Pooling Kernels

3.4.1 Average Pool Kernels

Description

The Average Pool kernels compute 2D average pool on a set of input planes (matrices) `x` and give a set of planes `y` as output.

The pooling region is defined by `kernel_height` and `kernel_width`. It is shifted over the input plane in steps of `x_stride` horizontally and in steps of `y_stride` vertically to generate the specified output plane size. The input is extended by zero padding as specified by the padding region. The padding is determined by the parameters `x_padding`, `y_padding` for left and top side padding respectively, and `out_width`, `out_height` for right and bottom padding respectively. Around the edges of input planes, if only a part of pooling region is covering input plane then only the average of those elements is calculated, and the denominator is the number of elements from input in current pooling region.

The average pool kernels accept input as 8-bit, 16-bit integer, asym8u or single precision floating point format and give output in the same precision as input.

These kernels require temporary buffer for average pool computation. This temporary buffer is provided by the `p_scratch` argument of kernel API. The size of the temporary buffer must be queried using `xa_nn_avgpool_getsize()` helper API.

These kernels expect input cube in SHAPE_CUBE_WHD_T and SHAPE_CUBE_DWH_T shape type and produce output cube in SHAPE_CUBE_WHD_T and SHAPE_CUBE_DWH_T shape type respectively. The `inp_data_format` and `out_data_format` arguments to the kernel API can be 0 or 1 to indicate input and output cube shapes respectively.

The value of `inp_data_format` and `out_data_format` must be equal.

Note	The fixed-point 8-bit average pool kernel, `xa_nn_avgpool_8` can be used for the quantized int8 datatype.

Function variants available are `xa_nn_avgpool_[p]`, where:

- `[p]`: Input and Output precision in bits

Precision

There are four variants available:

Type	Description
`8`	8-bit input, 8-bit output
`16`	16-bit input, 16-bit output
`f32`	float32 input, float32 output
`asym8u`	asym8u input, asym8u output

Algorithm

$$z_{h,w,d} = \frac{1}{K_H K_W} \left(\sum_{i=0}^{K_H-1} \sum_{j=0}^{K_W-1} in_{(h*y\text{-}stride+i),(w*x\text{-}stride+j),d)} \right)$$

$$h = 0, \ldots, \overline{out\text{-}height - 1}, \quad w = 0, \ldots, \overline{out\text{-}width - 1},$$
$$d = 0, \ldots, \overline{out\text{-}channels - 1}$$

in denotes padded input cube, z denotes output

K_H, K_W denote `kernel_height`, `kernel_width` respectively.

Prototype

```
WORD32 xa_nn_avgpool_getsize
(WORD32 input_channels,  WORD32 inp_precision,   WORD32 out_precision,
 WORD32 input_height,    WORD32 input_width,     WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 inp_data_format, WORD32 out_data_format);

WORD32 xa_nn_avgpool_8
(WORD8 * p_out,          const WORD8 * p_inp,    WORD32 input_height,
 WORD32 input_width,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 inp_data_format, WORD32 out_data_format,
 VOID * p_scratch);
```

```
WORD32 xa_nn_avgpool_16
(WORD16 * p_out,          const WORD16 * p_inp,    WORD32 input_height,
 WORD32 input_width,      WORD32 input_channels,   WORD32 kernel_height,
 WORD32 kernel_width,     WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,        WORD32 inp_data_format,  WORD32 out_data_format,
 VOID * p_scratch);
WORD32 xa_nn_avgpool_f32
(FLOAT32 * p_out,         const FLOAT32 * p_inp,   WORD32 input_height,
 WORD32 input_width,      WORD32 input_channels,   WORD32 kernel_height,
 WORD32 kernel_width,     WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,        WORD32 inp_data_format,  WORD32 out_data_format,
 VOID * p_scratch);
WORD32 xa_nn_avgpool_asym8u
(UWORD8* p_out,           const UWORD8* p_inp,     WORD32 input_height,
 WORD32 input_width,      WORD32 input_channels,   WORD32 kernel_height,
 WORD32 kernel_width,     WORD32 x_stride,         WORD32 y_stride,
 WORD32 x_padding,        WORD32 y_padding,        WORD32 out_height,
 WORD32 out_width,        WORD32 inp_data_format,  WORD32 out_data_format,
 VOID *p_scratch);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const WORD16 *, const UWORD8 *, const FLOAT32 *	p_inp	input_height * input_width * input_channels	Input cube
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Input number of channels
WORD32	kernel_height		Pooling window height
WORD32	kernel_width		Pooling window width
WORD32	x_stride		Horizontal stride over input
WORD32	y_stride		Vertical stride over input
WORD32	x_padding		Left padding width on input
WORD32	y_padding		Top padding height on input
WORD32	out_height		Output height
WORD32	out_width		Output width
WORD32	inp_data_format		Input data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
WORD32	out_data_format		Output data format: 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
Output			
WORD8 *, WORD16 *, UWORD8 *, FLOAT32 *	p_out	out_height * out_width * input_channels	Output cube
Temporary			
VOID *	p_scratch	xa_nn_avgpool_ getsize()	Temporary / scratch memory

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_inp, p_out`	Cannot be NULL
	Should not overlap
`p_scratch`	Cannot be NULL
	Should not overlap
	Memory size ≥ size returned by `xa_nn_avgpool_getsize()`
`input_height, input_width`	Greater than or equal to 1
`input_channels`	Greater than or equal to 1
`kernel_height`	{1, 2, ..., min(`input_height`, 256)} (for 8-bit and 16-bit) {1, 2, ..., `input_height`} (for float32)
`kernel_width`	{1, 2, ..., min(`input_width`, 256)} (for 8-bit and 16-bit) {1, 2, ..., `input_width`} (for float32)
`x_stride, y_stride`	Greater than or equal to 1
`x_padding, y_padding`	Greater than or equal to 0
`out_height, out_width`	greater than or equal to 1
`inp_data_format`	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T
`out_data_format`	Must be equal to `inp_data_format`

3.4.2 Max Pool Kernels

Description

The Max Pool kernels perform 2D max pooling operation over a set of input planes `x` and give as output, a set of planes `y`.

The pooling region is defined by `kernel_height` and `kernel_width`. It is shifted over the input plane horizontally in steps of `x_stride` and vertically in steps of `y_stride` to generate the specified output plane size.

The input plane, padded with the maximum negative values is considered while performing the max pooling operation. The padding region is determined by the parameters `x_padding`, `y_padding` for left and top side padding respectively, and `out_width`, `out_height` for right and bottom padding respectively.

The max pool kernels accept input as 8-bit, 16-bit integer, asym8u or single precision floating point format and give output in the same precision as input.

These kernels require temporary buffer for max pool computation. This temporary buffer is provided by the `p_scratch` argument of kernel API. The size of the temporary buffer must be queried using the `xa_nn_maxpool_getsize()` helper API.

These kernels expect input cube in SHAPE_CUBE_WHD_T and SHAPE_CUBE_DWH_T shape type and produce output cube in SHAPE_CUBE_WHD_T and SHAPE_CUBE_DWH_T shape type respectively. The

`inp_data_format` and `out_data_format` arguments to the kernel API can be 0 or 1 to indicate input and output cube shapes respectively.

The value of `inp_data_format` and `out_data_format` must be equal.

Note   The fixed-point 8-bit max pool kernel, `xa_nn_maxpool_8` can be used for the quantized int8 datatype.

Function variants available are `xa_nn_maxpool_[p]`, where:

- `[p]`: Input and Output precision in bits

Precision

There are four variants available:

Type	Description
8	8-bit input, 8-bit output
16	16-bit input, 16-bit output
f32	float32 input, float32 output
asym8u	asym8u input, asym8u output

Algorithm

$$z_{h,w,d} = \max\left(in_{(h*y\text{-}stride+i),(w*x\text{-}stride+j),d)}\right)$$
$$h = 0, \ldots, \overline{out\text{-}height - 1}, \quad w = 0, \ldots, \overline{out\text{-}width - 1},$$
$$d = 0, \ldots, \overline{out\text{-}channels - 1}$$
$$i = 0, \ldots, K_H - 1, \quad j = 0, \ldots, K_W - 1$$

in denotes padded input cube, z denotes output.

K_H, K_W denote `kernel_height`, `kernel_width` respectively.

Prototype

```
WORD32 xa_nn_maxpool_getsize
(WORD32 input_channels,  WORD32 inp_precision,   WORD32 out_precision,
 WORD32 input_height,    WORD32 input_width,     WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 inp_data_format, WORD32 out_data_format);

WORD32 xa_nn_maxpool_8
(WORD8 * p_out,          const WORD8 * p_inp,    WORD32 input_height,
 WORD32 input_width,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 out_data_format,
 VOID * p_scratch);
WORD32 xa_nn_maxpool_16
(WORD16 * p_out,         const WORD16 * p_inp,   WORD32 input_height,
 WORD32 input_width,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 out_data_format,
 VOID * p_scratch);
WORD32 xa_nn_maxpool_f32
(FLOAT32 * p_out,        const FLOAT32 * p_inp,  WORD32 input_height,
 WORD32 input_width,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 out_data_format,
 VOID * p_scratch);
WORD32 xa_nn_maxpool_asym8u
(UWORD8* p_out,          const UWORD8* p_inp,    WORD32  input_height,
 WORD32 input_width,     WORD32 input_channels,  WORD32 kernel_height,
 WORD32 kernel_width,    WORD32 x_stride,        WORD32 y_stride,
 WORD32 x_padding,       WORD32 y_padding,       WORD32 out_height,
 WORD32 out_width,       WORD32 inp_data_format, WORD32 out_data_format,
 VOID  *p_scratch);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const WORD16 *, const UWORD8 *, const FLOAT32 *	p_inp	input_height * input_width * input_channels	Input cube
WORD32	input_height		Input height
WORD32	input_width		Input width
WORD32	input_channels		Input number of channels
WORD32	kernel_height		Pooling window height
WORD32	kernel_width		Pooling window width
WORD32	x_stride		Horizontal stride over input
WORD32	y_stride		Vertical stride over input
WORD32	x_padding		Left padding width on input

Type	Name	Size	Description
`WORD32`	`y_padding`		Top padding height on input
`WORD32`	`out_height`		Output height
`WORD32`	`out_width`		Output width
`WORD32`	`inp_data_format`		Input data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
`WORD32`	`out_data_format`		Input data format 0:SHAPE_CUBE_DWH_T 1:SHAPE_CUBE_WHD_T
Output			

`WORD8 *,`
`WORD16 *,`
`UWORD8 *,`
`FLOAT32 *`
	`p_out`	`out_height *`
		`out_width *`
		`input_channels`
			Output cube

Temporary			
`VOID *`	`p_scratch`	`xa_nn_maxpool_` `getsize()`	Temporary / scratch memory

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_inp, p_out`	Cannot be NULL
	Should not overlap
	Should not overlap
	Memory size ≥ size returned by `xa_nn_maxpool_getsize()`
`input_height, input_width`	Greater than or equal to 1
`input_channels`	Greater than or equal to 1
`kernel_height`	{1, 2, ..., `input_height`}
`kernel_width`	{1, 2, ..., `input_width`}
`x_stride, y_stride`	Greater than or equal to 1
`x_padding, y_padding`	Greater than or equal to 0
`out_height, out_width`	Greater than or equal to 1
`inp_data_format`	Can be 0: SHAPE_CUBE_DWH_T or 1: SHAPE_CUBE_WHD_T
`out_data_format`	Must be equal to `inp_data_format`

3.5 Fully Connected Layer

3.5.1 Fully Connected Kernels

Description

The Fully Connected kernels perform the operation of multiplication of weight matrix with input vectors in a fully connected neural network layer, that is, `z = weight*input + bias`. The column dimension of `weight` must match the row dimension of `input`. Bias and resulting output vector `z` have as many numbers of rows as `weight` matrix.

The `bias_shift` and `acc_shift` arguments are provided in kernel API to adjust Q format of bias and output, respectively. Both `bias_shift` and `acc_shift` can be either positive or negative, where positive value denotes a left shift and negative value denotes a right shift.

`bias_shift` is the shift in number of bits applied to the bias to make it in the same Q format as weight X input multiplication – accumulation result. `acc_shift` is the shift in number of bits applied to the accumulator to obtain the output in the required Q format.

Note	`acc_shift` and `bias_shift` are not relevant in the case of floating point and quantized 8-bit kernels.

For conversion from higher precision accumulator to lower precision output, symmetric rounding is used.

The precision of output is the same as precision of input vector.

The arguments `input_zero_bias`, `weight_zero_bias` are provided to convert the quantized 8-bit inputs into their real values and perform Fully Connected kernel operation. The `out_zero_bias`, `out_multiplier`, and `out_shift` values are used to quantize real values of output back to 8-bit.

Function variants available (for fixed point) are `xa_nn_fully_connected_[p]x[q]_[r]`, where:

- ■ `[p]`: Weight matrix precision in bits
- ■ `[q]`: Input vector precision in bits
- ■ `[r]`: Output vector precision in bits

Precision

There are eight variants available:

Type	Description
`16x16_16`	16-bit weight matrix, 16-bit input vector, 16-bit output
`8x16_16`	8-bit weight matrix, 16-bit input vector, 16-bit output
`8x8_8`	8-bit weight matrix, 8-bit input vector, 8-bit output
`f32`	float32 weight matrix, float32 input vector, float32 output
`asym8uxasym8u_asym8u`	asym8u weight matrix, asym8u input vector, asym8u output
`sym8sxasym8s_asym8s`	sym8s weight matrix, asym8s input vector, asym8s output
`asym8sxasym8s_asym8s`	asym8s weight matrix, asym8s input vector, asym8s output
`sym8sxsym16s_sym16s`	sym8s weight matrix, sym16s input vector, sym16s output
`f16`	float16 weight matrix, float16 input vector, float16 output
`asym4sxasym8s_asym8s`	asym4s weight matrix, asym8s input vector, asym8s output

Algorithm

$$z_n = 2^{acc\text{-}shift} \left(\sum_{m=0}^{W_D-1} weight_{n,m} \cdot input_m \; + \; 2^{bias\text{-}shift} bias_n \right),$$

$$n = 0, \dots, \overline{out\text{-}depth - 1}$$

where W_D represents `weight_depth`

For floating point and quantized 8-bit routines, `acc_shift=0` and `bias_shift=0`

Thus, $2^{acc\text{-}shift} = 2^{bias\text{-}shift} = 1$

Prototype

```
WORD32 xa_nn_fully_connected_16x16_16
(WORD16 * p_out,        WORD16 * p_weight,      WORD16 * p_inp,
 WORD16 * p_bias,       WORD32 weight_depth,    WORD32 out_depth,
 WORD32 acc_shift,      WORD32 bias_shift);
WORD32 xa_nn_fully_connected_8x16_16
(WORD16 * p_out,        WORD8 * p_weight,       WORD16 * p_inp,
 WORD16 * p_bias,       WORD32 weight_depth,    WORD32 out_depth,
 WORD32 acc_shift,      WORD32 bias_shift);
WORD32 xa_nn_fully_connected_8x8_8
(WORD8 * p_out,         WORD8 * p_weight,       WORD8 * p_inp,
 WORD8 * p_bias,        WORD32 weight_depth,    WORD32 out_depth,
 WORD32 acc_shift,      WORD32 bias_shift);
WORD32 xa_nn_fully_connected_f32
(FLOAT32 * p_out,       FLOAT32 * p_weight,     FLOAT32 * p_inp,
 FLOAT32 * p_bias,      WORD32 weight_depth,    WORD32 out_depth);
WORD32 xa_nn_fully_connected_asym8uxasym8u_asym8u
(UWORD8 * p_out,        const UWORD8 * p_weight, const UWORD8 * p_inp,
 const WORD32 * p_bias, WORD32 weight_depth,    WORD32 out_depth,
 WORD32 input_zero_bias,WORD32 weight_zero_bias, WORD32 out_multiplier,
 WORD32 out_shift,      WORD32 out_zero_bias);
WORD32 xa_nn_fully_connected_sym8sxasym8s_asym8s
(WORD8 * p_out,         const WORD8 * p_weight, const WORD8 * p_inp,
 const WORD32 * p_bias, WORD32 weight_depth,    WORD32 out_depth,
```

```
 WORD32 input_zero_bias,WORD32 out_multiplier,    WORD32 out_shift,
 WORD32 out_zero_bias);


WORD32 xa_nn_fully_connected_asym8sxasym8s_asym8s
(WORD8 * p_out,          const WORD8 * p_weight,   const WORD8 * p_inp,
 const WORD32 * p_bias, WORD32 weight_depth,      WORD32 out_depth,
 WORD32 input_zero_bias,WORD32 weight_zero_bias,  WORD32 out_multiplier,
 WORD32 out_shift,        WORD32 out_zero_bias);
WORD32 xa_nn_fully_connected_sym8sxsym16s_sym16s
(pWORD16  p_out,          const WORD8 * p_weight,      const WORD16 * p_inp,
 const WORD64 * p_bias,    WORD32  weight_depth,       WORD32  out_depth,
 WORD32  out_multiplier,   WORD32  out_shift);
WORD32 xa_nn_fully_connected_f16
 (WORD16 * p_out ,        const WORD16 * p_weight, const WORD16 * p_inp ,
  const WORD16 * p_bias, WORD32  weight_depth,    WORD32  out_depth );
WORD32 xa_nn_fully_connected_asym4sxasym8s_asym8s
 (WORD8 * p_out,           const WORD8 * p_weight,    const WORD8 * p_inp,
  const WORD32 * p_bias,    WORD32  weight_depth,      WORD32  out_depth,
  WORD32  input_zero_bias,  WORD32  weight_zero_bias,  WORD32  out_multiplier,
  WORD32  out_shift,        WORD32  out_zero_bias,     VOID *p_scratch);
```

Arguments

Type	Name	Size	Description
Input			

`WORD16 *,`
`WORD8 *,`
`pFLOAT32,`
`const`
`UWORD8 *,`
`const`
`WORD8 *`
	`p_weight`	`out_depth*`
		`weight_depth`
			Weight matrix, fixed, floating point, asym4s, asym8u or sym8s

`WORD16 *,`
`WORD8 *,`
`pFLOAT32,`
`const`
`UWORD8 *,`
`const`
`WORD8 *`
	`p_inp`	`weight_depth*`
		`1`
			Input vector, fixed, floating point, asym8u or asym8s

`WORD16 *,`
`WORD8 *,`
`pFLOAT32,`
`WORD32 *,`
`const`
`WORD64 *`
	`p_bias`	`out_depth*1`	Bias vector, fixed or floating point, 32-bit for quantized kernels

`WORD32`	`out_depth`		Number of rows in weight matrix, bias and output vector
`WORD32`	`weight_depth`		Number of columns in weight matrix and rows in input vector
`WORD32`	`acc_shift`		Shift applied to accumulator
`WORD32`	`bias_shift`		Shift applied to bias
`WORD32`	`input_zero_bias`		Zero offset of input
`WORD32`	`weight_zero_bias`		Zero offset of weights
`WORD32`	`out_multiplier`		Multiplier value of output
`WORD32`	`out_shift`		Shift value of output
`WORD32`	`out_zero_bias`		Zero offset of output
Output			

| WORD8 *,
WORD16 *,
pFLOAT32,
WORD8 *,
UWORD8* | p_out | out_depth*1 | Output vector, fixed, floating point, asym8u or asym8s |

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
weight_depth	Multiple of 4 for fixed point and floating point kernels. No restriction for quantized 8-bit kernels.
p_weight, p_inp, p_bias, p_out	Aligned on 16-byte boundary, must not overlap.. (size of one element)-byte boundary in case of floating point and quantized 8-bit kernels).
p_weight, p_inp, p_out	Cannot be NULL
p_bias	Cannot be NULL (except for sym8sxasym8s precision)
out_depth	Greater than or equal to 1
out_multiplier	Greater than 0
acc_shift, bias_shift, out_shift	{-31, ...,31}
input_zero_bias	{-255, ...,0} for asym8u, {-127, ...,128} for asym8s
weight_zero_bias	{-255, ...,0} for asym8u, {-127, ...,128} for asym8s and asym4s
out_zero_bias	{0, ...,255} for asym8u, {-128, ...,127} for asym8s

3.6 Basic Operations and Miscellaneous Kernels

3.6.1 Interpolation Kernel

Description

The Interpolation kernel performs interpolation between two input vectors h and y using interpolation factor from vector x to get output vector z.

The interpolation kernel accepts 16-bit inputs and 16-bit interpolation factor in Q15 format and produces 16-bit output in Q15 format.

Precision

Type	Description
16-bit	16-bit input, 16-bit interpolation factor, 16-bit output

Algorithm

$$z_n = x_n * y_n + (1 - x_n) * h_n \; , \qquad n = 0 \, ..., \overline{num\text{-}elements - 1}$$

x_n represents interpolation factor.

y_n represents first input, h_n represents second input.

z_n represents output.

Prototype

```
WORD32 xa_nn_vec_interpolation_q15
(WORD16 * p_out,      WORD16 * p_ifact,    WORD16 * p_inp1,WORD16 * p_inp2,       WORD32
 num_elements);
```

Arguments

Type	Name	Size	Description
Input			
WORD16 *	p_ifact	num_elements	Interpolation factor vector
WORD16 *	p_inp1	num_elements	First input vector
WORD16 *	p_inp2	num_elements	Second input vector
WORD32	num_elements		Number of elements
Output			
WORD16 *	p_out	num_elements	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_ifact, p_inp1, p_inp2, p_out	Aligned on 8-byte boundary
	Should not overlap
	Cannot be NULL
num_elements	Multiple of 4

3.6.2 Dot Product Kernels

Description

The Dot Product kernels perform the dot product operations between two sets of input vectors `p_inp1` and `p_inp2` to get output vector `p_out`. The supported precisions are: f32xf32_f32 and 16x16_asym8s.

Function variants available are `xa_nn_elm_dot_prod_[p]x[q]_[r]`, where:

- `[p],[q]`: Input precision
- `[r]`: Output precision

Precision

There are two variants available:

Type	Description
f32xf32_f32	float32 input, float32 output
16x16_asym8s	16-bit input, asym8s output

Algorithm

$$z_n = \left(\sum_{m=0}^{vec\text{-}length-1} inp1_m \cdot inp2_m \; + \; bias_n \right)$$

$$n = 0, \ldots, \overline{vec\text{-}count \; - \; 1}$$

Prototype

```
WORD32 xa_nn_dot_prod_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp1,      const FLOAT32 * p_inp2,
 WORD32 vec_length,        WORD32 num_vecs);
WORD32 xa_nn_dot_prod_16x16_asym8s
(WORD8 * p_out,            const WORD16 * p_inp1,       const WORD16 * p_inp2,
 const WORD32 * bias_ptr,  WORD32 vec_length,           WORD32 out_multiplier,
 WORD32 out_shift,         WORD32 out_zero_bias,        WORD32 vec_count);
```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 * const WORD16 *	p_inp1	vec_length	First input vector
const FLOAT32 * const WORD16 *	p_inp2	vec_length	Second input vector
const WORD32 *	bias_ptr	vec_count	Bias vector
WORD32	vec_length		Length of each vector
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32	out_zero_bias		Zero offset of output
WORD32	num_vecs, vec_count		Number of input vectors
Output			
FLOAT32 * WORD8 *	p_out	num_vecs	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp1,p_inp2, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
vec_length, num_vecs	Greater than 0
out_shift	{-31, ..., 31}
out_multiplier	Greater than 0
out_zero_bias	{-128….,127} for out type asym8s

3.6.3 Elementwise Quantize Kernels

Description

The Elementwise Quantize kernels perform the quantization operation of the `p_inp1` input vector elements to get the output vector `p_out`. The kernels are developed in reference to the Quantize operator implementation in TensorFlow Lite Micro.

Function variants available are `xa_nn_elm_quantize_[p]_[q]`, where:

- `[p]`: Input precision
- `[p]`: Output precision

Algorithm

for itr = 0:(num_elm-1)

$$p\text{-}out[itr] = (p\text{-}inp[itr] \, / \, out\_scale) \; + \; out\text{-}zero\text{-}bias$$

Precision

Type	Description
f32_asym8s	single precision float input, asym8s output
f32_asym16s	single precision float input, asym16s output

Prototype

```
WORD32 xa_nn_elm_quantize_f32_asym8s
(WORD8 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,   FLOAT32  out_scale,
 WORD32  out_zero_bias,    WORD32 num_elm);
WORD32 xa_nn_elm_quantize_f32_asym16s
(WORD16 * __restrict_p_out, const FLOAT32 * __restrict__p_inp,   FLOAT32  out_scale,
 WORD32  out_zero_bias,    WORD32 num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp	num_elm	Input vector

FLOAT32	out_scale		Scale of output
WORD32	out_zero_bias		Zero offset of output
WORD32	num_elm		Number of input elements
Output			
WORD8 * WORD16 *	p_out	num_elm	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
num_elm	Greater than 0
out_scale	Not equal to zero and finite single precision float value
out_zero_bias	{-128....,127} for out type asym8s {-32768....,32767} for out type asym16s

3.6.4 Elementwise Requantize Kernels

Description

The Elementwise Requantize kernels perform the requantization operation of the `p_inp1` input vector elements to get the output vector `p_out`. The kernels are developed in reference to the Quantize operator implementation in TensorFlow Lite Micro.

Function variants available are `xa_nn_elm_requantize_[p]_[q]`, where:

- `[p]`: Input precision
- `[p]`: Output precision

Algorithm

for itr = 0:(num_elm-1)

$$p\text{-}out[itr] = ((2 \char94 out\text{-}shift) * (out\text{-}multiplier) * (p\text{-}inp[itr] - inp\text{-}zero\text{-}bias)) + out\text{-}zero\text{-}bias$$

Precision

Type	Description
asym16s_asym8s	asym16s input, asym8s output
asym8s_asym32s	asym8s input, asym32s output
asym16s_asym32s	asym16s input, asym32s output
asym8s_asym8s	asym8s input, asym8s output
asym16s_asym16s	asym16s input, asym16s output

Prototype

```
WORD32 xa_nn_elm_requantize_asym16s_asym8s
(WORD8 *__restrict__ p_out, const WORD16 *__restrict__ p_inp, WORD32  inp_zero_bias,
 WORD32  out_zero_bias,    WORD32  out_shift,          WORD32  out_multiplier,
 WORD32  num_elm);
WORD32 xa_nn_elm_requantize_asym8s_asym32s
(WORD32 * __restrict__ p_out, const WORD8 * __restrict__ p_inp,  WORD32  inp_zero_bias,
 WORD32  out_zero_bias,      WORD32  out_shift,          WORD32  out_multiplier,
 WORD32  num_elm);
WORD32 xa_nn_elm_requantize_asym16s_asym32s
(WORD32 * __restrict__ p_out, const WORD16 * __restrict__ p_inp, WORD32  inp_zero_bias,
 WORD32  out_zero_bias,      WORD32  out_shift,          WORD32  out_multiplier,
 WORD32  num_elm);
WORD32 xa_nn_elm_requantize_asym8s_asym8s
(WORD8 * __restrict__ p_out,  const WORD8 * __restrict__ p_inp,  WORD32  inp_zero_bias,
 WORD32  out_zero_bias,      WORD32  out_shift,          WORD32  out_multiplier,
 WORD32  num_elm);
WORD32 xa_nn_elm_requantize_asym16s_asym16s
(WORD16 * __restrict__ p_out, const WORD16 * __restrict__ p_inp,  WORD32  inp_zero_bias,
 WORD32  out_zero_bias,      WORD32 out_shift,           WORD32  out_multiplier,
 WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const WORD16 *, const WORD8 *	p_inp	num_elm	Input vector
WORD32	inp_zero_bias		Zero offset of input
WORD32	out_zero_bias		Zero offset of output
WORD32	out_shift		Shift value of output
WORD32	out_multiplier		Multiplier value of output
WORD32	num_elm		Number of input elements
Output			
WORD8 *, WORD16 *, WORD32 *	p_out	num_elm	Output vector

Returns

- ■ 0: no error
- ■ -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap

num_elm	Greater than 0
out_shift	{-31, …, 31}
out_multiplier	Greater than 0
inp_zero_bias	{-32768….,32767} for inp type asym16s {-128,….,127} for inp type asym8s
out_zero_bias	{-32768….,32767} for inp type asym16s {-128….,127} for out type asym8s Signed 32-bit integer value for out type asym32s

3.6.5 Elementwise Dequantize Kernels

Description

The Elementwise Dequantize kernels perform the dequantization operation of the `p_inp1` input vector elements to get the output vector `p_out`. The kernels are developed in reference to the Dequantize operator implementation in Tensorflow Lite Micro.

Function variants available are `xa_nn_elm_dequantize_[p]_[q]`, where:

- `[p]`: Input precision
- `[p]`: Output precision

Precision

Type	Description
asym8s_f32	asym8s input, float output
asym16s_f32	asym16s input, float output

Algorithm

for itr = 0:(num_elm-1)

$$p\text{-}out[itr] = (p\text{-}inp[itr] - inp\text{-}zero\text{-}bias) * inp\text{-}scale$$

Prototype

```
WORD32 xa_nn_elm_dequantize_asym8s_f32
(FLOAT32 * __restrict__ p_out, const WORD8 * __restrict__ p_inp, WORD32  inp_zero_bias,
 FLOAT32  inp_scale,          WORD32  num_elm);
WORD32 xa_nn_elm_dequantize_asym16s_f32
(FLOAT32 * __restrict__ p_out, const WORD16 *__restrict__ p_inp, WORD32  inp_zero_bias,
 FLOAT32  inp_scale,          WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 * const WORD16 *	p_inp	num_elm	Input vector

WORD32	inp_zero_bias		Zero offset of input
FLOAT32	inp_scale		Input scale
WORD32	num_elm		Number of input elements
Output			
FLOAT32 *	p_out	num_elm	Output vector

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
num_elm	Greater than 0
inp_zero_bias	{-128….,127} for inp type asym8s
	{-32768….,32767} for inp type asym16s

3.6.6 Basic Kernels

Description

The Basic kernels perform basic elementwise operations on one or two input vectors `x` and `y` to get output vector `z`. The supported operations are: add, subtract, multiply, floor, minimum, maximum, sine, cosine, log (natural), absolute, ceil, round (banker's), negative, square, square-root and inverse square-root. The supported precisions are: 8-bit, float32 and asym8s.

The 8-bit elementwise minimum and maximum kernels can be also used for asym8s datatype.

Function variants available are `xa_nn_[o]_[p]_[q]`, where:

- `[o]`: Operations: `elm_add`, `elm_sub`, `elm_mul`, `elm_floor`, `elm_min`, `elm_max`, `elm_sine`, `elm_cosine`, `elm_logn`, `elm_abs`, `elm_ceil`, `elm_round`, `elm_neg`, `elm_square`, `elm_sqrt`, `elm_rsqrt`

- `[p]`: Input Precision in bits- input1xinput2 or input1

- `[q]`: Output Precision in bits

Precision

Type	Description
`f32xf32_f32`	2 float32 inputs, float32 output
`f32_f32`	float32 input, float32 output
`8x8_8`	2 8-bit input, 8-bit output
`16x16_16`	2 16-bit input, 16-bit output
`asym8sxasym8s_asym8s`	2 asym8s inputs, asym8s output
`sym16sxsym16s_asym8s`	2 sym16s inputs, asym8s output

Algorithm

elm_add:	$z_n = x_n + y_n$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_sub:	$z_n = x_n - y_n$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_mul:	$z_n = x_n * y_n$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_floor:	$z_n = \lfloor x_n \rfloor$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_min:	$z_n = \min(x_n, y_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_max:	$z_n = \max(x_n, y_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_sine:	$z_n = \sin(x_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_cosine:	$z_n = \cos(x_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_logn:	$z_n = log_e(x_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_abs:	$z_n = \text{abs}(x_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_ceil:	$z_n = \lceil x_n \rceil$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_round[7]:	$z_n = \text{round}(x_n)$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_neg:	$z_n = -x_n$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$
elm_square:	$z_n = x_n * x_n$,	$n = 0 \dots, \overline{num\text{-}elm - 1}$

---

[7] The round variant is banker's rounding. It is also called "Round half to even". In this rounding method, if fractional part of input is 0.5, then output is the even integer nearest to input. Thus, for example, +23.5 becomes 24, as does 24.5; while -23.5 becomes -24, as does -24.5.

elm_sqrt: $z_n = \sqrt{x_n},$ $n = 0 \ldots, \overline{num\text{-}elm - 1}$
elm_rsqrt: $z_n = 1 \div \sqrt{x_n},$ $n = 0 \ldots, \overline{num\text{-}elm - 1}$

x_n represents first input, y_n represents second input.

z_n represents output.

Prototype

```
WORD32 xa_nn_elm_floor_f32_f32
(FLOAT32 * p_out,           const FLOAT32 * p_inp,       WORD32  num_elm);
WORD32 xa_nn_elm_add_asym8sxasym8s_asym8s
(WORD8 * p_out,             WORD32 out_zero_bias,       WORD32 out_shift,
 WORD32 out_multiplier,     WORD32 out_activation_min,  WORD32 out_activation_max,
 const  WORD8 * p_inp1,     WORD32 inp1_zero_bias,      WORD32 inp1_shift,
 WORD32 inp1_multiplier,    const  WORD8 * p_inp2,      WORD32 inp2_zero_bias,
 WORD32 inp2_shift,         WORD32 inp2_multiplier,     WORD32 left_shift,
 WORD32 num_elm);
WORD32 xa_nn_elm_sub_asym8sxasym8s_asym8s
(WORD8 * p_out,             WORD32  out_zero_bias,      WORD32  out_left_shift,
 WORD32  out_multiplier,    WORD32  out_activation_min, WORD32  out_activation_max,
 const WORD8 * p_inp1,      WORD32  inp1_zero_bias,     WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,   const WORD8 * p_inp2,       WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,   WORD32  inp2_multiplier,    WORD32  left_shift,
 WORD32  num_elm);
WORD32 xa_nn_elm_mul_asym8sxasym8s_asym8s
(WORD8 * p_out,             WORD32 out_zero_bias,       WORD32 out_shift,
 WORD32 out_multiplier,     WORD32 out_activation_min,  WORD32 out_activation_max,
 const  WORD8 * p_inp1,     WORD32 inp1_zero_bias,      const  WORD8 * p_inp2,
 WORD32 inp2_zero_bias,     WORD32 num_elm);
WORD32 xa_nn_elm_mul_sym16sxsym16s_asym8s
(WORD8 * p_out,             WORD32  out_zero_bias,      WORD32  out_shift,
 WORD32  out_multiplier,    WORD32  out_activation_min, WORD32  out_activation_max,
 const  WORD16 * p_inp1,    const   WORD16 * p_inp2,    WORD32  num_elm);
WORD32 xa_nn_elm_min_8x8_8
(WORD8* p_out,              const WORD8* p_in1,         const WORD8* p_in2,
 WORD32 num_element);
WORD32 xa_nn_elm_max_8x8_8
(WORD8* p_out,              const WORD8* p_in1,         const WORD8* p_in2,
 WORD32 num_element);
WORD32 xa_nn_elm_sine_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_cosine_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_logn_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_abs_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_ceil_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_round_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_neg_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_square_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,   WORD32  num_elm);
WORD32 xa_nn_elm_sqrt_f32_f32
```

```
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,  WORD32  num_elm);
WORD32 xa_nn_elm_add_f32xf32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp1,
const FLOAT32 * __restrict__ p_inp2, WORD32  num_elm);
WORD32 xa_nn_elm_add_16x16_16
(WORD16 * __restrict__ p_out,  const WORD16 * __restrict__ p_inp1,
 const WORD16 * __restrict__ p_inp2, WORD32 num_elm);
WORD32 xa_nn_elm_rsqrt_f32_f32
(FLOAT32 * __restrict__ p_out,  const FLOAT32 * __restrict__ p_inp,  WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 * WORD16 * FLOAT32 *	p_inp1, p_inp, p_in1	num_elm	First input vector
const WORD8 * WORD16 * FLOAT32 *	p_inp2, P_in2	num_elm	Second input vector
WORD32	num_elm/num_element		Number of elements
WORD32	out_zero_bias		Zero bias of output
WORD32	out_shift		Shift value of output
WORD32	out_multiplier		Multiplier value of output
WORD32	out_activation_min		Activation min of output
WORD32	out_activation_max		Activation max of output
WORD32	inp1_zero_bias		Zero bias of input 1
WORD32	inp1_shift		Shift value of input 1
WORD32	inp1_multiplier		Multiplier value of input 1
WORD32	inp2_zero_bias		Zero bias of input 2
WORD32	inp2_shift		Shift value of input 2
WORD32	inp2_multiplier		Multiplier value of input 2
WORD32	left_shift		Global left shift value for inputs.
Output			
WORD8 * WORD16 * FLOAT32 *	p_out	num_elm	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp1,p_inp2, p_inp,p_in1,p_in2 p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL

Arguments	Restrictions
p_out	Should not overlap with the input pointers (could be same as one of the input pointers, inplace operation is possible)
num_elm, num_element	Greater than 0
inp1_zero_bias, inp2_zero_bias	{-127....., 128} for asym8s input
inp1_shift, inp2_shift, out_shift	{-31 .... 0} for add,sub quantized datatype kernels, {-31 .... 31} for other fixed point and quantized 8-bit APIs
left_shift	{0 .... 31}
inp1_multiplier, inp2_multiplier out_multiplier	Should not be less than 0.
out_zero_bias	{-128....., 127} for asym8s output
out_activation_min, out_activation_max	{-128....., 127} for asym8s output out_activation_min < out_activation_max

3.6.7 Basic Kernels with Broadcasting

Description

The Basic kernels with broadcasting perform a broadcast operation and apply an arithmetic operator. The supported operators are: elementwise minimum and maximum.

Details of the broadcast operation can be found at [Tensorflow Broadcasting semantics](#) [4].

There are two variants of these kernels, one for 4-dimensional and another for 8-dimensional input/output tensors. Input tensors smaller than these dimensions must be have their shapes extended[4.1] to match either of these two.

Tensors must also be broadcast compatible (as these kernels do not perform any runtime checks and depend on the TensorFlow infrastructure)

The input to these kernels are the IO pointers to tensors stored in row-major format, the shape of the resulting broadcasted output and the input 'strides' [5].

Function variants available are `xa_nn_[op]_[d]_Bcast_[p]`, where:

- [op]: Operation: `elm_min`, `elm_max`
- [d]: Number of IO dimensions: `4D`, `8D`
- [p]: Input/Output precision in bits as `[in1_precision]`x`[in2_precision]_[out_precision]`

Precision

Type	Description
8x8_8	Signed 8-bit inputs, signed 8-bit output

Algorithm

$$p\text{-}out[i_0][i_1]\ldots[i_N] =$$
$$[op](\, p\text{-}in1(\,[i_0\ i_1\ \ldots\ i_N]\cdot[s1_0\ s1_1\ \ldots\ s1_N])\,,\ p\text{-}in2(\,[i_0\ i_1\ \ldots\ i_N]\cdot[s2_0\ s2_1\ \ldots\ s2_N])\,))$$

Where,

- $i_n \in (0\ \text{out-extents}[n]]$, and, $n \in (0\ 4]$ for 4D tensors, or,
  $(0\ 8]$ for 8D Tensors
- $s1_n = \text{in1-strides}[n]$, with n defined the same as above
- $s2_n = \text{in2-strides}[n]$, with n defined the same as above

Prototypes

```
WORD32 xa_nn_elm_min_4D_Bcast_8x8_8(
        WORD8* __restrict__ p_out,  const int* const out_extents,
    const WORD8* __restrict__ p_in1,  const int* const in1_strides,
    const WORD8* __restrict__ p_in2,  const int* const in2_strides )
WORD32 xa_nn_elm_max_4D_Bcast_8x8_8(
        WORD8* __restrict__ p_out,  const int* const out_extents,
    const WORD8* __restrict__ p_in1,  const int* const in1_strides,
    const WORD8* __restrict__ p_in2,  const int* const in2_strides )
WORD32 xa_nn_elm_min_8D_Bcast_8x8_8(
        WORD8* __restrict__ p_out,  const int* const out_extents,
    const WORD8* __restrict__ p_in1,  const int* const in1_strides,
    const WORD8* __restrict__ p_in2,  const int* const in2_strides )
WORD32 xa_nn_elm_max_8D_Bcast_8x8_8(
        WORD8* __restrict__ p_out,  const int* const out_extents,
    const WORD8* __restrict__ p_in1,  const int* const in1_strides,
    const WORD8* __restrict__ p_in2,  const int* const in2_strides )
```

Arguments

Type	Name	Size	Description
Input			
const WORD8*	p_in1	–	First input tensor in row-major
const int* const	in1_strides	4 or 8	Strides for first input tensor
const WORD8*	p_in2	–	Second input tensor in row-major
const int* const	in2_strides	4 or 8	Strides for second input tensor
const int* const	out_extents	4 or 8	Broadcasted output shape
Output			
WORD8*	p_out	prod(out_extents)	Output tensor in row-major

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_in1,p_in2` `p_out`	Aligned on byte boundary Cannot be NULL
`out_extents, in1_strides,` `in2_strides`	Positive integers

3.6.8 Basic Kernels with 4D Broadcasting

Description

The Basic kernels with 4D broadcasting perform a broadcast operation and apply an arithmetic operator. The supported operators are: elementwise add, sub, mul and squared_diff.

Details of the broadcast operation can be found at [Tensorflow Broadcasting semantics](#) [4].

These kernels support 4-dimensional input/output tensors. Input/output tensors having less than than 4 dimensions must have their shapes extended[4.1] to have 4 dimensions.

Tensors must also be broadcast compatible (that is, either their dimensions must match or be equal to 1) otherwise kernels return error.

Function variants available are `xa_nn_[op]_broadcast_4D_[p]`, where:

- `[op]`: Operation: `elm_add, elm_sub, elm_mul, elm_squared_diff`

- `[p]`: Input/Output precision in bits as `[in1_precision]x[in2_precision]_[out_precision]`

Precision

Type	Description
`asym8sxasym8s_asym8s`	asym8s inputs, asym8s output
`asym16sxasym16s_asym16s`	asym16s inputs, asym16s output
`sym16sxsym16s_sym16s`	sym16s inputs, sym16s output
`f32xf32_f32`	f32 inputs,_f32 output

Algorithm

$$p\_out[i_0][i_1] \ldots [i_3] = [op](p\_inp1[i1_0][i1_1] \ldots [i1_3], \ p\_inp2[i2_0][i2_1] \ldots [i2_3])$$

Where,

- $i_n = [0, p\_out\_shape[n] - 1]; \ n = [0, 3]$

- $i1_n = i_n \ if \ p\_out\_shape[n] = p\_inp1\_shape[n] \ else \ 0; \ n = [0, 3]$

- $i2_n = i_n \ if \ p\_out\_shape[n] = p\_inp2\_shape[n] \ else \ 0; \ n = [0, 3]$

Ops are:

elm_add: $z_n = x_n + y_n$
elm_sub: $z_n = x_n - y_n$
elm_mul: $z_n = x_n * y_n$
elm_squared_diff: $z_n = (x_n - y_n)^2$

Prototypes

```
WORD32 xa_nn_elm_add_broadcast_4D_asym8sxasym8s_asym8s
(WORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD8 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD8 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);


WORD32 xa_nn_elm_sub_broadcast_4D_asym8sxasym8s_asym8s
(WORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD8 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD8 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);


WORD32 xa_nn_elm_mul_broadcast_4D_asym8sxasym8s_asym8s
(WORD8 * __restrict__ p_out,
```

```
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const    WORD8 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 const    WORD8 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias);


WORD32 xa_nn_elm_squared_diff_broadcast_4D_asym8sxasym8s_asym8s
(WORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD8 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD8 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);


WORD32 xa_nn_elm_add_broadcast_4D_asym16sxasym16s_asym16s
(WORD16 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD16 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD16 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);

WORD32 xa_nn_elm_sub_broadcast_4D_asym16sxasym16s_asym16s
(WORD16 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
```

```
 WORD32  out_zero_bias,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD16 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_zero_bias,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD16 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_zero_bias,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);


WORD32 xa_nn_elm_mul_broadcast_4D_sym16sxsym16s_sym16s
(WORD16 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_zero_bias,
 WORD32  out_shift,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD16 * p_inp1,
 const WORD32 *const p_inp1_shape,
 const WORD16 * p_inp2,
 const WORD32 *const p_inp2_shape);


WORD32 xa_nn_elm_sub_broadcast_4D_f32xf32_f32
(FLOAT32 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 const FLOAT32 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 const FLOAT32 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape);


WORD32 xa_nn_elm_squared_diff_broadcast_4D_sym16sxsym16s_sym16s
(WORD16 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 WORD32  out_left_shift,
 WORD32  out_multiplier,
 WORD32  out_activation_min,
 WORD32  out_activation_max,
 const WORD16 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 WORD32  inp1_left_shift,
 WORD32  inp1_multiplier,
 const WORD16 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32  inp2_left_shift,
 WORD32  inp2_multiplier,
 WORD32  left_shift);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const WORD16 * const FLOAT32 *	p_inp1	$\prod_{i=0}^{i=3} p\text{-}inp1\text{-}shape[i]$	First input tensor
const WORD8 *, const WORD16 * const FLOAT32 *	p_inp2	$\prod_{i=0}^{i=3} p\text{-}inp2\text{-}shape[i]$	Second input tensor
const WORD32 *const	p_out_shape	4	Shape of output (array of size 4) (first dimension is outer most)
const WORD32 *const	p_inp1_shape	4	Shape of first input (array of size 4) (first dimension is outer most)
const WORD32 *const	p_inp2_shape	4	Shape of second input (array of size 4) (first dimension is outer most)
WORD32	out_zero_bias		Zero bias of output
WORD32	out_shift		Shift value of output
WORD32	out_multiplier		Multiplier value of output
WORD32	out_activation_min		Activation min of output
WORD32	out_activation_max		Activation max of output
WORD32	inp1_zero_bias		Zero bias of input 1
WORD32	inp1_shift		Shift value of input 1
WORD32	inp1_multiplier		Multiplier value of input 1
WORD32	inp2_zero_bias		Zero bias of input 2
WORD32	inp2_shift		Shift value of input 2
WORD32	inp2_multiplier		Multiplier value of input 2
WORD32	left_shift		Global left shift value for inputs.
Output			
WORD8 * FLOAT32 * WORD16 *	p_out	$\prod_{i=0}^{i=3} p\text{-}out\text{-}shape[i]$	Output tensor

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_inp1, p_inp2, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL

Arguments	Restrictions
`p_out`	Should not overlap with the input pointers (could be same as one of the input pointers, inplace operation is possible)

`p_out_shape,`
`p_inp1_shape,`
`p_inp2_shape`
	Cannot be NULL

	Aligned on 4-byte boundary
	Shapes must be broadcast compatible, that is, p_out_shape[i] must be max(p_inp1_shape[i], p_inp2_shape[i]) p_inp1_shape[i] must be either equal to p_inp2_shape[i] or 1 p_inp2_shape[i] must be either equal to p_inp1_shape[i] or 1
`inp1_zero_bias,` `inp2_zero_bias`	{-127….., 128} for asym8s input {-32767 ….. 32768} for asym16s input
`inp1_shift, inp2_shift,` `out_shift`	{-31 …. 0} for add,sub quantized datatype kernels, {-31 …. 31} for other fixed point and quantized datatype kernels
`left_shift`	{0 …. 31}

`inp1_multiplier,`
`inp2_multiplier`
`out_multiplier`
	Should not be less than 0.

`out_zero_bias`	{-128….., 127} for asym8s output {-32768 ….. 32767} for asym16s output
`out_activation_min,` `out_activation_max`	{-128….., 127} for asym8s output {-32768 ….. 32767} for asym16s output out_activation_min < out_activation_max

3.6.9 Elementwise Comparison Kernels

Description

The Elementwise Comparison kernels perform elementwise comparison operations on two input vectors x and y to get the output vector z. The supported operations are: equal, not equal, greater, greater equal, less, less equal. The output for all the comparison kernels is a Boolean value that requires 1-byte space.The supported precisions are: asym8s.

Function variants available are `xa_nn_[o]_[p]`, where:

- `[o]`: Operations: `elm_equal`, `elm_notequal`, `elm_greater`, `elm_greaterequal`, `elm_less`, `elm_lessequal`
- `[p]`: Input Precision in bits- input1xinput2

Precision

Type	Description

`asym8sxasym8s`	asym8s inputs, Boolean(1-byte) output

Algorithm

elm_equal: $z_n = (x_n == y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_notequal: $z_n = (x_n\ ! = y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_greater: $z_n = (x_n > y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_greaterequal: $z_n = (x_n \geq y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_less: $z_n = (x_n < y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_lessequal: $z_n = (x_n \leq y_n)$, $n = 0 \ldots, \overline{num\text{-}elm - 1}$

x_n represents first input, y_n represents second input.

z_n represents output.

Prototype

```
WORD32 xa_nn_elm_equal_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
WORD32 xa_nn_elm_notequal_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
WORD32 xa_nn_elm_greater_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
WORD32 xa_nn_elm_greaterequal_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
WORD32 xa_nn_elm_less_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
WORD32 xa_nn_elm_lessequal_asym8sxasym8s
(WORD8 * p_out,          const WORD8 * p_inp1,    WORD32  inp1_zero_bias,
 WORD32  inp1_shift,     WORD32  inp1_multiplier, const WORD8 * p_inp2,
 WORD32  inp2_zero_bias, WORD32  inp2_shift,      WORD32  inp2_multiplier,
 WORD32  left_shift,     WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
`const WORD8 *`	`p_inp1`	`num_elm`	First input vector

const WORD8 *	p_inp2	num_elm	Second input vector
WORD32	num_elm		Number of elements
WORD32	inp1_zero_bias		Zero bias of input 1
WORD32	inp1_shift		Shift value of input 1
WORD32	inp1_multiplier		Multiplier value of input 1
WORD32	inp2_zero_bias		Zero bias of input 2
WORD32	inp2_shift		Shift value of input 2
WORD32	inp2_multiplier		Multiplier value of input 2
WORD32	left_shift		Global left shift value for inputs.
Output			
WORD8 *	p_out	num_elm	Output vector

Returns

- 0: no error

- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
`p_inp1,p_inp2,p_out,`	Aligned on (size of one element)-byte boundary Cannot be NULL
`num_elm`	Greater than 0
`inp1_zero_bias,` `inp2_zero_bias`	{-127….., 128} for asym8s input
`inp1_shift, inp2_shift`	{-31 …. 31} for fixed point and quantized 8-bit APIs
`inp1_multiplier,` `inp2_multiplier`	Should not be less than 0.
`left_shift`	{0 …. 31}

3.6.10 Elementwise Logical Kernels

Description

The Elementwise Logical kernels perform elementwise logical operations on two Boolean input vectors `x` and `y` to get the Boolean output vector `z`. The supported operations are: logical_and, logical_or, logical_not. The inputs and output for all the logical kernels are Boolean values that requires 1-byte space each. The supported precisions are: bool.

Function variants available are `xa_nn_[o]_[p]`, where:

- `[o]`: Operations: `elm_logicaland`, `elm_logicalor`, `elm_logicalnot`
- `[p]`: Input Precision in bits- input1xinput2

Precision

Type	Description
`boolxbool`	Boolean(1-byte) inputs, Boolean(1-byte) output

Algorithm

elm_logicaland: $\quad z_n = (x_n \text{ \&\& } y_n),\quad n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_logicalor: $\quad z_n = (x_n \,||\, y_n),\quad n = 0 \ldots, \overline{num\text{-}elm - 1}$

elm_logicalnot: $\quad z_n = (!\, x_n),\quad n = 0 \ldots, \overline{num\text{-}elm - 1}$

x_n represents first input, y_n represents second input.

z_n represents output.

Prototype

```
WORD32 xa_nn_elm_logicaland_boolxbool_bool
(WORD8 * __restrict__ p_out, const   WORD8 * __restrict__ p_inp1,
 const   WORD8 * __restrict__ p_inp2, WORD32  num_elm);


WORD32 xa_nn_elm_logicalor_boolxbool_bool
(WORD8 * __restrict__ p_out, const   WORD8 * __restrict__ p_inp1,
 const   WORD8 * __restrict__ p_inp2, WORD32  num_elm);


WORD32 xa_nn_elm_logicalnot_bool_bool
(WORD8 * __restrict__ p_out, const   WORD8 * __restrict__ p_inp,
 WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_inp1 / p_inp	num_elm	First input vector
const WORD8 *	p_inp2	num_elm	Second input vector
WORD32	num_elm		Number of elements
Output			
WORD8 *	p_out	num_elm	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_inp1/p_inp,p_inp2,p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
num_elm	Greater than 0

3.6.11 Reduce Kernels

Description

The Reduce kernels perform reduction operations on an input vector x based on the dimensions given in axis vector and get the output vector z. The supported operations are: reduce_max and reduce_mean. The supported precisions are: asym8s & asym16s. The kernels presently support up to 4 dimensions and the input data is assumed to be in "NHWC" or "DWHN" data format (Depth or channels dimension is written first).

Note	The axis vector must have non-duplicate values to avoid larger execution time and poor performance.

For the reduce_max kernel, the input and output quantization are expected to be same. Thus, the API does not include quantization specific multiplier, shift and zero bias arguments. For the dimensions mentioned in the axis vector, max operation is carried out thereby reducing the dimension size to 1.

For the reduce_mean kernel, the input and output quantization can be different. The arguments `inp_zero_bias`, `out_zero_bias`, `out_multiplier`, and `out_shift` are provided for the Mean operation and requantization into asym8s output. For the dimensions mentioned in the axis vector, mean operation is carried out thereby reducing the dimension size to 1.

The reduce_mean kernel expects the multiplication factor 1/(Number of elements in axis) to be adjusted in `out_multiplier` and `out_shift` parameters.

These kernels require temporary buffer for reduce operation. This temporary buffer is provided by `p_scratch` argument of kernel API. The size of temporary buffer must be queried using `xa_nn_reduce_getsize_nhwc()` helper API. The `reduce_ops` argument accepts an enumerator that states the reduce operation type. It can take the following values: REDUCE_MAX and REDUCE_MEAN.

Function variants available are `xa_nn_reduce_[o]_[n]_[p]`, where:

- `[o]`: Operations: `reduce_max, reduce_mean`
- `[n]`: Number of dimentions: `4D`
- `[p]`: Input Precision in bits- `input_output`

Precision

Type	Description
`asym8s_asym8s`	asym8s input, asym8s output
`asym16s_asym16s`	asym16s input, asym16s output

Algorithm

Reduce Max:

- For every dimension r in axis:

$$Z_{N,H,W,C} = \max\left(in_{n,h,w,c}[\boldsymbol{r}_i], \ in_{n,h,w,c}[\boldsymbol{r}_j] \right)$$

Where,

- The values of output dimensions (N, H, W, C) if reduced will be equal to 1
- $\boldsymbol{r} \in$ dimensions along which reduce max is to be performed .
- $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ are the elements in the input shape along the $\boldsymbol{r}$ dimension.

Reduce Mean:

- For every dimension r in axis:

$$S_{N,H,W,C} = sum(\ in_{n,h,w,c}[r_i],\ \ in_{n,h,w,c}[r_j]\ )$$

- Then, we compute the mean

$$Z_{N,H,W,C} = \frac{1}{\Pi\ nElem_r} S_{N,H,W,C}$$

Where,

- The values of output dimensions(N, H, W, C) if reduced will be equal to 1

- $r \in$ dimensions along which reduce mean is to be performed .

- r_i and r_j are the elements in the input shape along the r dimension.

- $\Pi\ nElem_r$ is the product of number of elements in every r dimension.

    o   Also refered to as 'Number of elements in axis'.

$S_{N,H,W,C}$ represents the intermediate reduce sum output required for reduce mean.

$Z_{N,H,W,C}$ represents the reduce operation output and $\ in_{n,h,w,c}$ represents the input vector.

Prototype

```
WORD32 xa_nn_reduce_getsize_nhwc
(WORD32 inp_precision,  const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,
 const WORD32 *p_axis,  WORD32 num_axis_dims,               WORD32 reduce_ops);

WORD32 xa_nn_reduce_max_4D_asym8s_asym8s
(WORD8 * p_out,          const WORD32 *const p_out_shape, const WORD8 * p_inp,
 const WORD32 *const p_inp_shape,     const WORD32 * p_axis,
 WORD32 num_out_dims,   WORD32 num_inp_dims,         WORD32 num_axis_dims,
 pVOID p_scratch_in);

WORD32 xa_nn_reduce_max_4D_asym16s_asym16s
(WORD16 * p_out,         const WORD32 *const p_out_shape,const WORD16 * p_inp,
 const WORD32 *const p_inp_shape,     const WORD32 * p_axis,
 WORD32 num_out_dims,   WORD32 num_inp_dims,          WORD32 num_axis_dims,
 pVOID p_scratch_in);

WORD32 xa_nn_reduce_mean_4D_asym8s_asym8s
(WORD8 * p_out,          const WORD32 *const p_out_shape, const WORD8 * p_inp,
 const WORD32 *const p_inp_shape,     const WORD32 * p_axis,
 WORD32 num_out_dims,   WORD32 num_inp_dims,         WORD32 num_axis_dims,
 WORD32 inp_zero_bias,  WORD32 out_multiplier,       WORD32 out_shift,
 WORD32 out_zero_bias,  pVOID p_scratch_in);

WORD32 xa_nn_reduce_mean_4D_asym16s_asym16s
(WORD16 * _p_out,        const WORD32 *const p_out_shape, const WORD16 * p_inp,
 const WORD32 *const p_inp_shape,     const WORD32 * p_axis,
 WORD32 num_out_dims,  WORD32 num_inp_dims,          WORD32 num_axis_dims,
 WORD32 inp_zero_bias, WORD32 out_multiplier,         WORD32 out_shift
 WORD32 out_zero_bias, pVOID p_scratch_in);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *const	p_out_shape	num_out_dims	Output shape vector containing size in each output dimension.
const WORD8 *, const WORD16 *	p_inp	Product of all dims in p_inp_shape	Input vector, asym8s/asym16s
const WORD32 *const	p_inp_shape	num_inp_dims	Input shape vector containing size in each input dimension.
const WORD32 *	p_axis	num_axis_dims	Axis vector, contains dimensions for reduce operation
WORD32	num_out_dims		Number of output dimension
WORD32	num_inp_dims		Number of input dimension
WORD32	num_axis_dims		Number of axis dimension
WORD32	inp_zero_bias		Zero offset of input
WORD32	out_multiplier		Multiplier value of output
WORD32	out_shift		Shift value of output
WORD32	out_zero_bias		Zero offset of output
pVOID	p_scratch	xa_nn_reduce_getsize_nhwc()	Scratch memory pointer
Output			
WORD8 * WORD16 *	p_out	Product of all dims in p_out_shape	Output vector, asym8s/asym16s

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
reduce_ops	Should be REDUCE_MAX or REDUCE_MEAN.
p_inp,p_axis,p_out,p_inp_shape,p_out_shape	Aligned on (size of one element)-byte boundary
	Cannot be NULL and cannot overlap
num_inp_dims, num_out_dims,	Should be more than 0 and less than equal to 4.
num_axis_dims	Should not be less than 0 and more than 4.
p_axis	The axis values must be between 0 and ($num\_inp\_dims - 1$).
p_inp_shape,p_out_shape	The shape values must be greater than 0.
p_inp_shape	Input shape values which are axis p_inp_shape[p_axis[0:num_axis_dims]] must be less than or equal to 1024.
inp_zero_bias out_zero_bias	{-128….,127} for asym8s, {-32768….,32767} for asym16s
out_multiplier	Greater than 0
out_shift	{-31, …, 31}

3.6.12 Broadcast Kernel

Description

The Broadcast kernel broadcasts an input shape into the specified output shape. The input and output shapes must be compatible for the broadcast operation to succeed.

Details of the broadcast operation can be found at [Tensorflow Broadcasting semantics](#) [4].

The dimensions of input and output tensors are passed as `in_shape` and `out_shape` and the number of dimensions specified by `numDims` must be the same for both. In case, the number of input and output dimensions are unequal, the empty leading dimensions of the smaller shape must be filled with ones to equalize them. For example, if the input dimension is 2x1x3 and the output dimension is 4x2x5x3, then `in_shape` must be passed as 1x2x1x3.

Figure 3-2 shows a simple illustration for broadcasting a 1x4x1 tensor into 1x4x3 and 2x4x3.
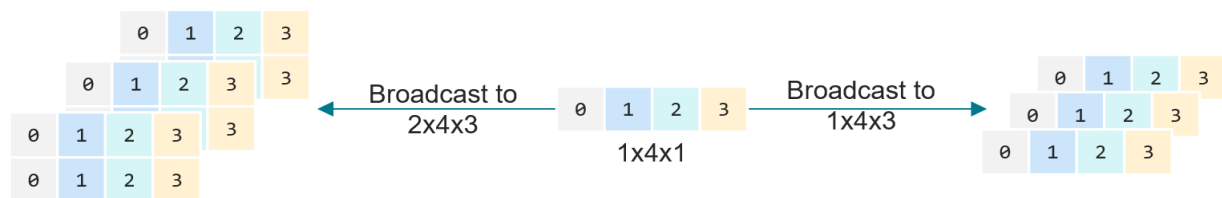
Figure 3-2 Broadcasting a 1x4x1 Tensor to 1x4x3 and 2x4x3

Precision

Type	Description
8_8	8-bit input, 8-bit output

Prototype

```
WORD32 xa_nn_broadcast_8_8
(WORD8* __restrict__ p_out,  const int* const out_shape,
 const WORD8* __restrict_p_in,  const int* const  in_shape,
 int numDims);
```

Arguments

Type	Name	Size	Description
Input			
`const WORD8 *`	`p_in`	$\prod_{i=0}^{i=num\text{-}dims-1} in\text{-}shape[i]$	Input tensor
`const int * const`	`in_shape` `out_shape`	`num_dims`	Input/output shapes
`int`	`num_dims`	–	Number of dimensions
Output			
`WORD8 *`	`p_out`	$\prod_{i=0}^{i=num\text{-}dims-1} out\text{-}shape[i]$	Output tensor

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
`p_in, p_out`	Aligned on (size of one element)-byte boundary
	Cannot be NULL
`inp_shape, out_shape`	Aligned on 4-byte boundary
	Cannot be NULL
	All elements must be greater than zero
	inp_shape[i] must be either equal to out_shape[i] or 1 for i = [0, numDims-1]
`num_dims`	In the range [1, 8]

3.6.13 Memory Operation Kernels

Description

The Memory Operation kernels perform basic memory related operations. The supported operations are: memmove and memset. The supported precisions are: 8-bit, float32.

Memmove kernel does byte-level transfer and takes generic pointers, `num_elm` must be set to number of 1-byte elements or simply number of bytes to be transferred for data types with sizes bigger than 1-byte.

Function variants available are `xa_nn_[o]_[p]_[q]`, where:

- `[o]`: Operations: `memmove`, `memset`
- `[p]`: Input Precision in bits
- `[q]`: Output Precision in bits

HiFi 5 Neural Network Library Programmer's Guide - API

Precision

Type	Description
`f32_f32`	float32 input, float32 output
`16`	16-bit input, 16-bit output
`8_8`	8-bit input, 8-bit output

Algorithm

memmove: $\quad z_n = x_n$, $\qquad\qquad n = 0\,....,\overline{num\text{-}elm - 1}$

memset: $\quad\;\; z_n = x_0$, $\qquad\qquad n = 0\,....,\overline{num\text{-}elm - 1};\; x_0 < scalar >$

x_n represents input

z_n represents output.

Prototype

```
WORD32 xa_nn_memset_f32_f32
(FLOAT32 * __restrict__ p_out,  FLOAT32 val,      WORD32  num_elm);
WORD32 xa_nn_memmove_16
(void * pdst,          const void *psrc,        WORD32 n);
WORD32 xa_nn_memmove_8_8
(void * p_out,         const void * p_inp,      WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
`const FLOAT32 * void *`	`p_inp`	num_elm	First input vector
`FLOAT32`	`val`		Memset value
`WORD32`	`num_elm`		Number of 1-byte elements or Number of bytes
Output			
`FLOAT32 * void *`	`p_out`	num_elm	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
`p_inp, p_out`	Aligned on (size of one element)-byte boundary
	Cannot be NULL

© CADENCE DESIGN SYSTEMS, INC. 125

num_elm	Greater than 0

3.6.14 LSTM Cell State Update

Description

This is a helper function for LSTM operator in TFLM. It updates the LSTM cell state based on the values of gate vectors : input_gate, forget_gate, cell_gate.

Function variants available are xa_nn_lstm_cell_update_[p], where:

[p]: Input and Output precision

Precision

Type	Description
16	16 bit cell state, forget gate, cell gate & input_gate

Algorithm

$$c_t = f_t . c_{t-1} + i_t . cg_t$$

where:

f_t : forget gate vector at time t

i_t : input gate vector at time t

c_t : cell state vector at time t

c_{t-1} : cell state vector at time t-1(Previous cell state)

cg_t : cell gate vector at time t

Prototype

```
WORD32 xa_nn_lstm_cell_state_update_16
(WORD16* p_cell_state,       const WORD16* p_forget_gate,  const WORD16* p_cell_gate,
 const WORD16* p_input_gate, WORD32 cell_to_forget_shift,  WORD32 cell_to_input_shift,
 WORD32 quantized_cell_clip, WORD32 num_elms);
```

Arguments

Type	Name	Size	Description
Input			

Type	Name	Size	Description
const WORD16 *	p_forget_gate	num_elms	Forget gate vector
const WORD16 *	p_cell_state	num_elms	Cell state vector. This argument is both an input and an output
const WORD16 *	p_cell_gate	num_elms	Cell gate vector
const WORD16 *	p_input_gate	num_elms	Input gate vector
WORD32	cell_to_forget_shift		Shift required for cell_state * forget_gate
WORD32	cell_to_input_shift		Shift required for input_gate * cell_gate
WORD32	quantized_cell_clip		Value to clamping the output
WORD32	num_elms	num_elms	Vector length
Output			
WORD16 *	p_cell_state	num_elms	Cell state vector. This argument is both an input and an output

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_forget_gate, p_cell_state, p_cell_gate, p_input_gate	Aligned on (size of one element)-byte boundary
	Cannot be NULL
num_elms	Greater than 0
cell_to_forget_shift	{-31, ..., -1}
cell_to_input_shift	{-31, ..., -1}

3.7   Normalization Kernels

3.7.1 L2 Normalization Kernel

Description

The L2 Normalization kernel performs L2 normalization of an input vector x to get output vector z, which means every element of input vector x is divided by L2 norm of x, this gives an output vector z whose L2 norm is 1.

The L2 Normalization kernel accepts asym8s input vector and produces asym8s output vector.

Precision

Type	Description
`asym8s`	asym8s input, asym8s output

Algorithm

$$z_n = \frac{x_n}{\sqrt{\sum_{n=1}^{N}|x_n|^2}}, \qquad n = 1\ldots,\overline{num\text{-}elements}$$

x_n represents input vector.

z_n represents output vector.

Prototype

```
WORD32 xa_nn_l2_norm_asym8s_asym8s
(WORD8 * p_out,      const WORD8 * p_inp,    WORD32 zero_point,    WORD32 num_elm);
```

Arguments

Type	Name	Size	Description
Input			
`const WORD8 *`	`p_inp`	num_elm	Input vector
`WORD32`	`zero_point`		Input zero bias
`WORD32`	`num_elm`		Number of elements
Output			
`WORD8 *`	`p_out`	num_elm	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_inp, p_out`	Aligned on (size of one element)-byte boundary
	Should not overlap
	Cannot be NULL
`zero_point`	{-128….., 127} for asym8s input
`num_elm`	Greater than 0

3.8   Reorg Kernels

3.8.1 Depth to Space Kernels

Description

The Depth to Space kernels convert the depth dimension of an input cube into the spatial dimensions of an output cube controlled by a block size parameter.

These kernels are based on DEPTH_TO_SPACE operator in TFLM[3], which collects all elements from the input depth dimension and spreads it across the output spatial dimension using a `block_size` factor. The operation is illustrated in Figure 3-3.
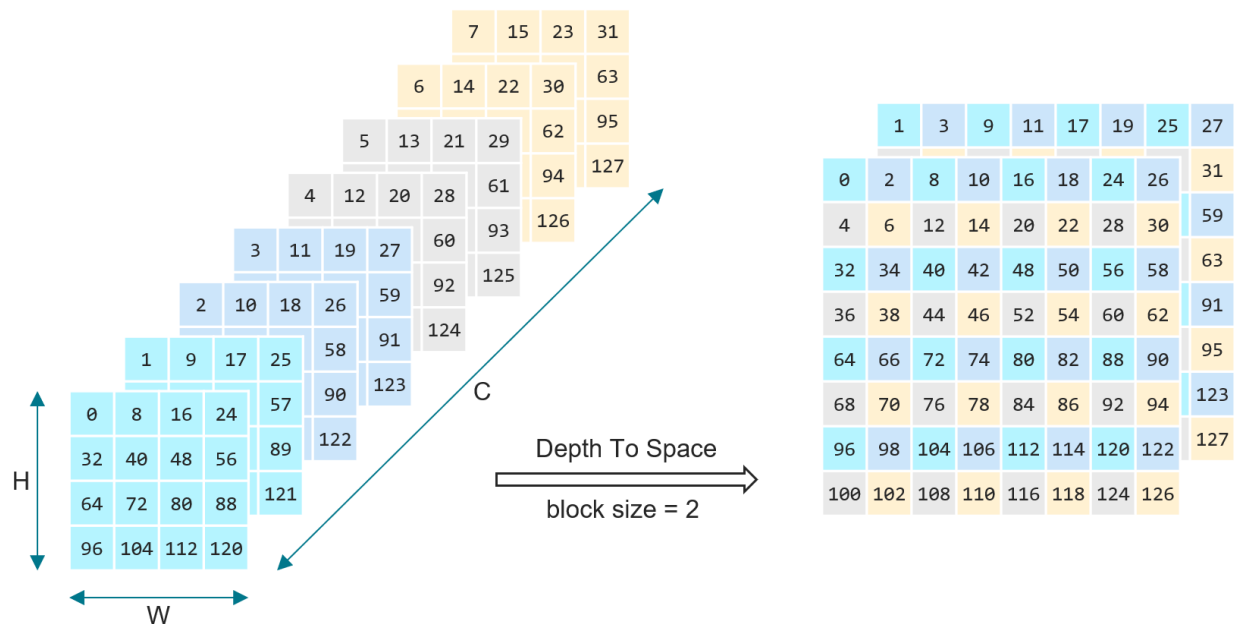

Figure 3-3 Depth to Space Conversion for 4x4x8 Input with Block Size of 2

Given an input cube of shape HxWxC and a `block_size` of K, this kernel gives an output cube of dimensions HKxWKxC/K². The specified output shape i.e `out_height/width/channels` must therefore equal HK, WK and C/K² respectively.

Since the elements collected from one dimension must be spread across two, the input depth dimension C (that is, `input_channels`) must be divisible by K² (that is, `block_size²`).

Precision

Type	Description
`8_8`	8-bit input, 8-bit output

Prototype

```
WORD32 xa_nn_depth_to_space_8_8
(pWORD8 __restrict__ p_out,  const WORD8 *__restrict__ p_inp,
 WORD32 input_height, WORD32 input_width, WORD32 input_channels,
 WORD32 block_size,
 WORD32 out_height,   WORD32 out_width,   WORD32 out_channels,
 WORD32 inp_data_format, WORD32 out_data_format);
```

Arguments

Type	Name	Size	Description
Input			
`const WORD8 *`	`p_inp`	`input_height* input_width* input_channels`	Input cube data
`WORD32`	`input_height`		Input cube height
`WORD32`	`input_width`		Input cube width
`WORD32`	`input_channels`		Input cube channels
`WORD32`	`block_size`		Spatial dimension block size
`WORD32`	`out_height`		Output cube height
`WORD32`	`out_width`		Output cube width
`WORD32`	`out_channels`		Output cube channels
`WORD32`	`inp_data_format`		Input data format
`WORD32`	`out_data_format`		Output data format
Output			
`WORD8 *`	`p_out`	`output_height* output_width* output_channels`	Output cube data

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_inp, p_out`	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
`input_height`	Must be greater than 0
`input_width`	Must be greater than 0
`input_channels`	Must be greater than 0 and divisible by `block_size`^{2}
`block_size`	Must be greater than 0
`out_height`	Must be `input_height*block_size`
`out_width`	Must be `input_width*block_size`
`out_channels`	Must be `input_channels/(block_size`^{2}`)`
`inp_data_format`	Must be 0 (NHWC)
`out_data_format`	Must be 0 (NHWC)

3.8.2 Space to Depth Kernels

Description

The Space to Depth kernels convert the spatial dimension of an input cube into the depth dimensions of an output cube controlled by a block size parameter.

These kernels perform the opposite operation of depth_to_space kernels which is illustrated in Figure 3-4.

Figure 3-4 Space to Depth Conversion for a 8x8x2 Input with a Block Size of 2
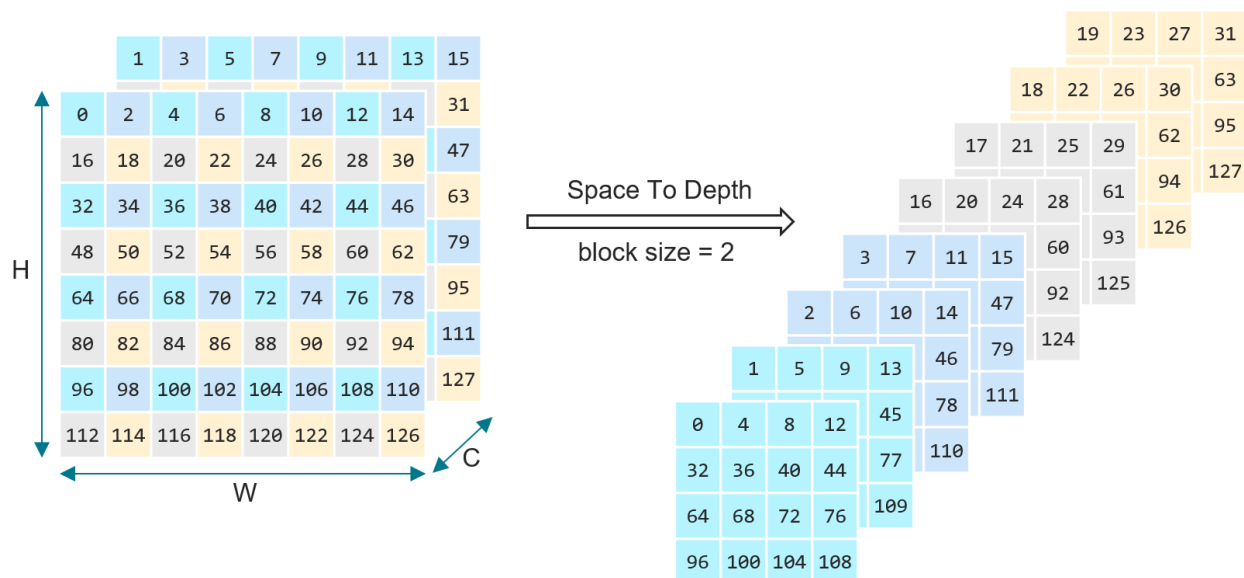
Given an input of shape HxWxC with a `block_size` of K, this kernel collects KxKxC elements from the input cube and serialize it into CK^2 elements across the depth dimension of the output resulting in an output of shape (H/K)x(W/K)x(CK^2).

The output shape specified i.e `out_height/width/channels` must equal H/K, W/K, and CK^2 respectively.

Because the elements collected from input 2D spatial dimension must be serialized into one output depth dimension, `output_channels` specified must equal `input_channels*block_size`^{2}.

Precision

Type	Description
8_8	8-bit input, 8-bit output

Prototype

```
WORD32 xa_nn_space_to_depth_8_8
(pWORD8 __restrict__ p_out,  const WORD8 *__restrict__ p_inp,
 WORD32 input_height, WORD32 input_width, WORD32 input_channels,
 WORD32 block_size,
 WORD32 out_height,   WORD32 out_width,   WORD32 out_channels,
 WORD32 inp_data_format, WORD32 out_data_format);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_inp	input_height* input_width* input_channels	Input cube data
WORD32	input_height		Input cube height
WORD32	input_width		Input cube width
WORD32	input_channels		Input cube channels
WORD32	block_size		Spatial dimension block size
WORD32	out_height		Output cube height
WORD32	out_width		Output cube width
WORD32	out_channels		Output cube channels
WORD32	inp_data_format		Input data format
WORD32	out_data_format		Output data format
Output			
WORD8 *	p_out	output_height* output_width* output_channels	Output cube data

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions

Arguments	Restrictions
`p_inp, p_out`	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
`input_height`	Must be greater than 0 and divisible by `block_size`
`input_width`	Must be greater than 0 and divisible by `block_size`
`input_channels`	Must be greater than 0
`block_size`	Must be greater than 0
`out_height`	Must be `input_height/block_size`
`out_width`	Must be `input_width/block_size`
`out_channels`	Must be `input_channels*(block_size`^{2}`)`
`inp_data_format`	Must be 0 (NHWC)
`out_data_format`	Must be 0 (NHWC)

3.8.3 Pad Kernels

Description

The Pad kernels pad an input with the given `pad_value` according to the values specified in `p_pad_values`. `p_pad_values` is an integer array with size (2 * input_dimensions), giving a pair of values for each input dimension. For each dimension of input, `p_pad_values` contains a pair of values which indicates how many values to add before the contents of input in that dimension and how many values to add after the contents of input in that dimension. This kernel is based on Pad and PadV2 operators in TFLM.

Input dimensions must be less than or equal to 4. 1/2/3-dimensional input can be scaled up to 4D. Output dimension must be equal to input dimension. Size of `p_pad_values` must be exactly (2 * input_dimensions). The value to be padded can be given through `pad_value`.

Naming convention used for pad kernel is:

`xa_nn_pad_[p]`

Where [p] = `[input_precision]_[out_precision]`

Precision

Type	Description
`8_8`	Signed 8-bit input, signed 8-bit output
`16_16`	Signed 16-bit input, signed 16-bit output
`32_32`	Signed 32-bit input, signed 32-bit output

Algorithm

If

ob = ib + p_pad_values[0] ; ib = [0, p_inp_shape[0]-1]

oh = ih + p_pad_values[2]; ih = [0, p_inp_shape[1]-1]

ow = iw + p_pad_values[4]; iw = [0, p_inp_shape[2]-1]

od = id + p_pad_values[6]; id = [0, p_inp_shape[3]-1]

$$Output_{ob,oh,ow,od} = Input_{ib,ih,iw,id}$$

else

$$Output_{ob,oh,ow,od} = pad\text{-}value$$

The shape of output after padding is as follows:

for D=0:(`num_inp_dims`-1)

$$p\text{-}out\text{-}shape[D] = p\text{-}pad\text{-}values[2*D] + p\text{-}inp\text{-}shape[D] + p\text{-}pad\text{-}values[2*D+1]$$

Prototype

```
WORD32 xa_nn_pad_8_8
(WORD8 *__restrict__ p_out, const WORD32 *const p_out_shape,
 const WORD8 *__restrict__ p_inp, const WORD32 *const p_inp_shape,
 const WORD32 *__restrict__ p_pad_values, const WORD32 *const p_pad_shape,
 WORD32 num_out_dims, WORD32 num_inp_dims, WORD32 num_pad_dims,
 WORD32 pad_value);
WORD32 xa_nn_pad_16_16
(WORD16 *__restrict__ p_out, const WORD32 *const p_out_shape,
 const WORD16 *__restrict__ p_inp, const WORD32 *const p_inp_shape,
 const WORD32 *__restrict__ p_pad_values, const WORD32 *const p_pad_shape,
 WORD32 num_out_dims, WORD32 num_inp_dims, WORD32 num_pad_dims,
 WORD32 pad_value);
WORD32 xa_nn_pad_32_32
(WORD32 *__restrict__ p_out,  const WORD32 *const p_out_shape,
 const WORD32 *__restrict__ p_inp, const WORD32 *const p_inp_shape,
 const WORD32 * _restrict__ p_pad_values, const WORD32 *const p_pad_shape,
 WORD32 num_out_dims, WORD32 num_inp_dims, WORD32 num_pad_dims,
 WORD32 pad_value);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *const	p_out_shape	num_out_dims	Shape of output
const WORD8 *, const WORD16 * const WORD32 *	p_inp	$\prod_{i=0}^{i=num\text{-}inp\text{-}dims-1} p\text{-}inp\text{-}shape[i]$	Input (set of cubes)
const WORD32 *const	p_inp_shape	num_inp_dims	Shape of input
const WORD32 *	p_pad_values	$\prod_{i=0}^{i=num\text{-}pad\text{-}dims-1} p\text{-}pad\text{-}shape[i]$	Pair of values (corresponds to before pad value and after pad value) for each input dimension
const WORD32 *const	p_pad_shape	num_pad_dims	Shape of pad_values

WORD32	num_out_dims		Number of output dimensions
WORD32	num_inp_dims		Number of input dimensions
WORD32	num_pad_dims		Number of pad dimensions
WORD32	pad_value		Value for padding
Output			
WORD8 *, WORD16 *, WORD32 *	p_out	$\prod_{i=0}^{i=num\text{-}out\text{-}dims-1} p\text{-}out\text{-}shape[i]$	Output (set of cubes)

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_out, p_inp	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
p_out_shape, p_inp_shape, p_pad_shape	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap
	All elements must be greater than zero
p_pad_values	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap with other buffers
	All elements must be greater than or equal to zero
	Pair of values for each input dimension
num_out_dims	Must be in range [1, 4]
num_inp_dims	Must be in range [1, 4]
num_pad_dims	Must be in range [1, 4]
pad_value	Must be in range [-128, 127]

3.8.4 Batch to Space Kernels

Description

The Batch to Space kernels perform batch to space conversion on a set of input cube `in` (`input_batch x input_height x input_width x input_depth`) and outputs a set of output cubes `out` of dimension (`out_batch x out_height x out_width x out_depth`). These kernels are based on BATCH_TO_SPACE_ND operator in TFLM[3].

Input can be 4 dimensional (dimensions are in order – batch, height, width and depth) or 3 dimensional (for 3 dimensional input width is assumed to be 1), output is always 4 dimensional. The conversion is determined by parameters `block_sizes` (`num_inp_dims - 2`) which determine conversion of a set of vectors in input (`input_batch x input_depth`) to a set of cubes (`out_batch x block_size_height x block_size_width x out_depth`) (`out_depth` must be equal to

input_depth), this conversion is repeated over all (input_height x input_width) sets of vectors in input. Additionally, some parts of output in height and width dimensions can be cropped by using crop_sizes.

For 4 dimensional input, number of block_sizes are 2 (in order - block_size_height, block_size_width), for 3 dimensional input only block_size_height is used and block_size_width is ignored.

For 4 dimensional input, number of crop_sizes are 4 (in order – crop_top, crop_bottom, crop_left, crop_right), crop_top and crop_left are used for 4 dimensional input, and only crop_top is used for 3 dimensional input.

Naming convention used for batch_to_space_nd kernels is:

xa_nn_batch_to_space_nd_[p]

Where [p] = [input_precision]_[out_precision]

Precision

Type	Description
8_8	Signed 8-bit input, signed 8-bit output

Algorithm

$$out_{ob,oh,ow,d} = in_{ib,ih,iw,d}$$

$$ob = ib \% out\text{-}batch$$

$$oh = ih * block\text{-}size\text{-}height - \left(\frac{ib}{out\text{-}batch}\right)/block\text{-}size\text{-}width - crop\text{-}left$$

$$ow = iw * block\text{-}size\text{-}width - \left(\frac{ib}{out\text{-}batch}\right)\% block\text{-}size\text{-}width - crop\text{-}top$$

% represents mod operator in C.

/ represents integer division in C.

For visualization of batch to space conversion, see Figure 3-5.

Prototype

```
WORD32 xa_nn_batch_to_space_nd_8_8
(WORD8 *__restrict__ p_out, const WORD32 *const p_out_shape,
 const WORD8 *__restrict__ p_inp, const WORD32 *const p_inp_shape,
 const WORD32 *const p_block_sizes, const WORD32 *const p_crop_sizes,
 WORD32  num_out_dims, WORD32  num_inp_dims);
```

Arguments

Type	Name	Size	Description
Input			

Type	Name	Size	Description
const WORD32 *const	p_out_shape	num_out_dims	Shape of output
const WORD8 *	p_inp	$\displaystyle\prod_{i=0}^{i=num\text{-}inp\text{-}dims-1} p\text{-}inp\text{-}shape[i]$	Input (set of cubes)
const WORD32 *const	p_inp_shape	num_inp_dims	Shape of input
const WORD32 *const	p_block_sizes	num_inp_dims - 2	Block sizes for spatial dimension.
const WORD32 *const	p_crop_sizes	2*(num_inp_dims - 2)	Crop sizes for cropping output
WORD32	num_out_dims		Number of output dimensions
WORD32	num_inp_dims		Number of input dimensions
Output			
WORD8 *	p_out	$\displaystyle\prod_{i=0}^{i=num\text{-}out\text{-}dims-1} p\text{-}out\text{-}shape[i]$	Output (set of cubes)

Returns

■ 0: no error

■ -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_out, p_inp	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
p_out_shape, p_inp_shape	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap
	All elements must be greater than zero
	p_out_shape[num_out_dims – 1] == p_inp_shape[num_inp_dims – 1] (depth for input and output must be equal.
p_block_sizes	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap with other buffers
	All elements must be greater than zero
	p_inp_shape[0] == p_out_shape[0]*p_block_sizes[0]*p_block_sizes[1][8]
p_crop_sizes	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap with other buffers
	All elements must be greater than or equal to zero
num_out_dims	Must be equal to 4
num_inp_dims	Must be in range {3, 4}

---

[8] This restriction is for num_inp_dims 4, if num_inp_dims is 3, it becomes p_inp_shape[0] == p_out_shape[0]*p_block_size[0]

batch to space, with block_sizes = {2, 3}

space to batch, with block_sizes = {2, 3}

ib -> input_batch = 6;   block_size_height = 2
ih -> input_height = 3;   block_size_width = 3
iw -> input_width = 3
id -> input_depth
output_batch = 1 (6/(2*3))
oh -> output_height = 6 (3*2)
ow -> output_width = 9 (3*3)
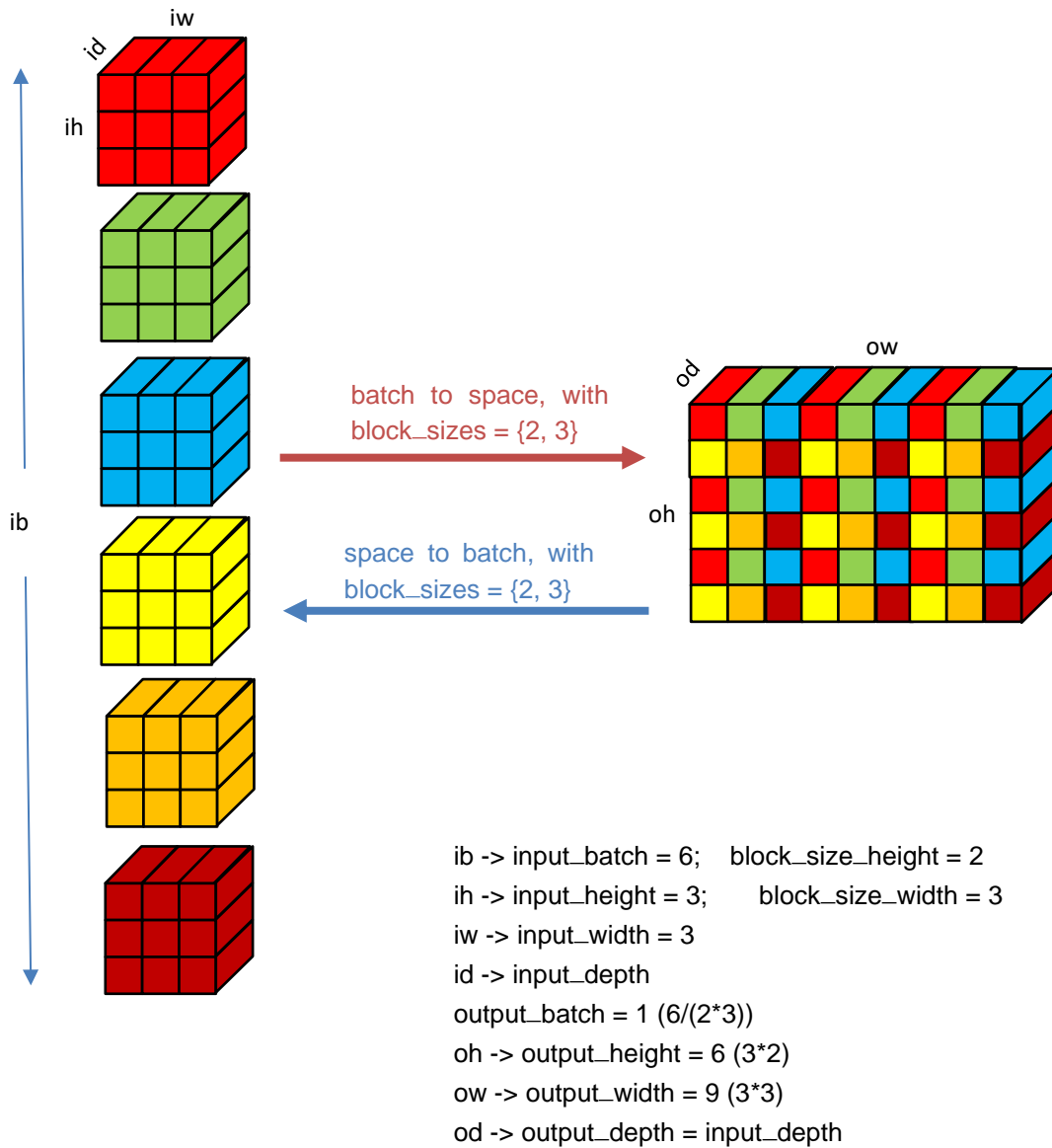od -> output_depth = input_depth

Figure 3-5 batch_to_space and space to batch Conversion

For simplicity, crop_sizes and pad_sizes are assumed to be 0.

3.8.5 Space to Batch Kernels

Description

The Space to Batch kernels perform space to batch conversion on a set of input cube `in` (`input_batch x input_height x input_width x input_depth`) and outputs a set of output cubes `out` of dimension (`out_batch x out_height x out_width x out_depth`). These kernels are based on SPACE_TO_BATCH_ND operator in TensorFlow Lite Micro[3].

Input can be 4 dimensional (dimensions are in order – batch, height, width and depth) or 3 dimensional (for 3 dimensional input width is assumed to be 1), output must have same number of dimensions as input. The conversion is determined by parameters `block_sizes` (`num_inp_dims - 2`) which determine conversion of a set of cubes in input (`input_batch x block_size_height x block_size_width x input_depth`) to a set of vectors (`out_batch x out_depth`) (`out_depth` must be equal to `input_depth`), this conversion is repeated over all of input. Additionally, output can be padded in height and width dimensions according to `pad_sizes`.

For 4 dimensional input, number of `block_sizes` are 2 (in_order - `block_size_height`, `block_size_width`), for 3 dimensional input only `block_size_height` is used and `block_size_width` is ignored.

For 4 dimensional input, number of `pad_sizes` are 4 (in order – `pad_top, pad_bottom, pad_left, pad_right`), `pad_top` and `pad_left` are used for 4 dimensional input, and only `pad_top` is used for 3 dimensional input.

The value to be filled in padding regions can be specified by `pad_value`.

Naming convention used for space_to_batch_nd kernels is:

`xa_nn_batch_to_space_nd_[p]`

Where [p] = `[input_precision]_[out_precision]`

Precision

Type	Description
8_8	Signed 8-bit input, signed 8-bit output

Algorithm

$$out_{ob,oh,ow,d} = in_{ib,ih,iw,d}$$

$$ib = ob \% out\text{-}batch$$

$$ih = oh * block\text{-}size\text{-}height - \left(\frac{ob}{input\text{-}batch}\right)/block\text{-}size\text{-}width - crop\text{-}left$$

$$iw = ow * block\text{-}size\text{-}width - \left(\frac{ob}{input\text{-}batch}\right) \% block\text{-}size\text{-}width - crop\text{-}top$$

% represents mod operator in C.

/ represents integer division in C.

For visualization of space to batch conversion, see Figure 3-5.

Prototype

```
WORD32 xa_nn_space_to_batch_nd_8_8
(WORD8 *__restrict__ p_out, const WORD32 *const p_out_shape,
 const WORD8 *__restrict__ p_inp, const WORD32 *const p_inp_shape,
 const WORD32 *const p_block_sizes, const WORD32 *const p_pad_sizes,
 WORD32 num_out_dims, WORD32 num_inp_dims
 WORD32 pad_value);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *const	p_out_shape	num_out_dims	Shape of output
const WORD8 *	p_inp	$\prod_{i=0}^{i=num\_inp\_dims-1} p\text{-}inp\text{-}shape[i]$	Input (set of cubes)
const WORD32 *const	p_inp_shape	num_inp_dims	Shape of input
const WORD32 *const	p_block_sizes	num_inp_dims – 2	Block sizes for spatial dimension.
const WORD32 *const	p_pad_sizes	2*(num_inp_dims – 2)	Crop sizes for cropping output
WORD32	num_out_dims		Number of output dimensions
WORD32	num_inp_dims		Number of input dimensions
WORD32	pad_value		Value for padding
Output			
WORD8 *	p_out	$\prod_{i=0}^{i=num\_out\_dims-1} p\text{-}out\text{-}shape[i]$	Output (set of cubes)

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
p_out, p_inp	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Should not overlap
p_out_shape, p_inp_shape	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap
	All elements must be greater than zero

Arguments	Restrictions
	p_out_shape[num_out_dims – 1] == p_inp_shape[num_inp_dims – 1] (depth for input and output must be equal.
p_block_sizes	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap with other buffers
	All elements must be greater than zero
	p_out_shape[0] == p_inp_shape[0]*p_block_sizes[0]*p_block_sizes[1][9]
p_pad_sizes	Aligned on 4-byte boundary
	Cannot be NULL
	Should not overlap with other buffers
	All elements must be greater than or equal to zero
num_out_dims	Must be in range {3, 4}
num_inp_dims	Must be in range {3, 4}
pad_value	Must be in range [-128, 127]

3.8.6 Strided Slice

Description

The Strided Slice kernels slice the given input based on the start ,stop, and stride parameters. It begins at the location specified by the start parameter and picks elements according to stride value untill it reaches stop point in that dimention. Input dimensions must be less than or equal to 4. 1/2/3/4 -dimensional input can be scaled up to 5D. The stride value can be negative, which represents the slice in backward direction. This kernel is based on Strided Slice operator in TFLM.

Precision

Type	Description
8_8	Signed 8-bit input, signed 8-bit output
16_16	Signed 16-bit input, signed 16-bit output
32_32	Signed 32-bit input, signed 32-bit output

Algorithm

for I = start_0 * input_dim_1 : strides_0 * input_dim_1 : ((stop_0 * input_dim_1)-offset_0)
 for J = (I + start_1) * input_dim_2 : strides_1 * input_dim_2 : (((I + stop_1) * input_dim_2)-offset_1)
  for K = (J + start_2) * input_dim_3 : strides_2 * input_dim_3 : (((J + stop_2) * input_dim_3)-offset_2)
   for L = (K + start_3) * input_dim_4 : strides_3 * input_dim_4 : (((K + stop_3) * input_dim_4-offset_3)
    for M = L + start_4 : strides_4 : ((L + stop_4)-offset_4)
     p_out++ = p_inp[M+1];
    end
   end

---

[9] This restriction is for num_inp_dims 4, if num_inp_dims is 3, it becomes p_out_shape[0] == p_inp_shape[0]*p_block_size[0]

```
    end
   end
 end
```

where, offset_x = ((stride_x)<0) ? -1 : 1;   x = {0,1,2,3,4}

Prototype

```
WORD32 xa_nn_strided_slice_int16(WORD16 * __restrict__ p_out, const   WORD16 * __restrict__
p_inp,
WORD32 start_0, WORD32 stop_0, WORD32 start_1, WORD32 stop_1,
WORD32 start_2, WORD32 stop_2, WORD32 start_3, WORD32 stop_3,
WORD32 start_4, WORD32 stop_4, WORD32 stride_0, WORD32 stride_1,
WORD32 stride_2, WORD32 stride_3, WORD32 stride_4,
WORD32 dims_1, WORD32 dims_2,  WORD32 dims_3, WORD32 dims_4);
WORD32 xa_nn_strided_slice_int8
(WORD8 * __restrict__ p_out, const   WORD8 * __restrict__ p_inp,
WORD32 start_0, WORD32 stop_0, WORD32 start_1, WORD32 stop_1,
WORD32 start_2, WORD32 stop_2, WORD32 start_3, WORD32 stop_3,
WORD32 start_4, WORD32 stop_4, WORD32 stride_0, WORD32 stride_1,
WORD32 stride_2, WORD32 stride_3, WORD32 stride_4,
WORD32 dims_1, WORD32 dims_2,  WORD32 dims_3, WORD32 dims_4);
WORD32 xa_nn_strided_slice_int32
(WORD32 * __restrict__ p_out, const   WORD32 *_restrict__ p_inp,
WORD32 start_0, WORD32 stop_0, WORD32 start_1, WORD32 stop_1,
WORD32 start_2, WORD32 stop_2, WORD32 start_3, WORD32 stop_3,
WORD32 start_4, WORD32 stop_4, WORD32 stride_0, WORD32 stride_1,
WORD32 stride_2, WORD32 stride_3, WORD32 stride_4, WORD32 dims_1,
WORD32 dims_2, WORD32 dims_3, WORD32 dims_4);
```

Arguments

Type	Name	Size	Description
Input			
const WORD16 *, const WORD8 *, const WORD32 *	p_inp		Input vector
WORD32	start_0		begin point for dimention 0
WORD32	start_1		begin point for dimention 1
WORD32	start_2		begin point for dimention 2
WORD32	start_3		begin point for dimention 3
WORD32	start_4		begin point for dimention 4
WORD32	stop_0		end point for dimention 0;
WORD32	stop_1		end point for dimention 1
WORD32	stop_2		end point for dimention 2
WORD32	stop_3		end point for dimention 3

Type	Name	Size	Description
WORD32	stop_4		end point for dimention 4
WORD32	stride_0		stride for dimention 0
WORD32	stride_1		stride for dimention 1
WORD32	stride_2		stride for dimention 2
WORD32	stride_3		stride for dimention 3
WORD32	stride_4		stride for dimention 4
WORD32	dims_1		dimention 1
WORD32	dims_2		dimention 2
WORD32	dims_3		dimention 3
WORD32	dims_4		dimention 4
Output			
WORD16 *, WORD8 *, WORD32 *	p_out	ceil(((stop_0 – start_0)/stride_0))) * ceil(((stop_1 – start_1)/stride_1))) * ceil(((stop_2 – start_2)/stride_2))) * ceil(((stop_3 – start_3)/stride_3))) * ceil(((stop_4 – start_4)/stride_4)))	Output vector

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions
`p_inp, p_out`	Should not overlap Cannot be NULL Aligned on size of element boundary
`dims_1, dims_2, dims_3, dims_4`	Greater than Zero
`stride_0,`	Equal to one (As we are only supporting 4D input)
`stride_1,stride_2, stride_3, stride_4`	Not Equal to Zero
`start_0`	Equal to Zero (As we are only supporting 4D input)
`stop_0`	Equal to One (As we are only supporting 4D input)
`start_1, stop_1`	if stride_1 > 0 then {0 … dims_1} else {-1 … dims_1 - 1}
`start_2, stop_2`	if stride_2 > 0 then {0 … dims_2} else {-1 … dims_2 - 1}
`start_3, stop_3`	if stride_3 > 0 then {0 … dims_3} else {-1 … dims_3 - 1}
`start_4, stop_4`	if stride_4 > 0 then {0 … dims_4} else {-1 … dims_4 - 1}

3.8.7 Transpose

Description

This kernel performs transpose operation on a N-dimentional input tensor(upto 5D) as per the combination of dimensions specified in the permute vector. The output tensor's dimension `i` will correspond to the input dimension `permute_vec[i]`. For a 2D tensor, this operation performs a regular matrix transpose.

Number of input dimensions must be less than or equal to 5. 1/2/3/4-dimensional input is scaled up to 5D. The output shape should be conformant with respect to the values in permute vector.

The naming convention used for the transpose kernel is as follows:

```
xa_nn_transpose_[p]
```

Where [p] = `[input_precision]_[out_precision]`

Precision

Type	Description
`8_8`	Signed 8-bit input, signed 8-bit output

Algorithm

For input P and output Q,
size(Q) = [dim3,dim2,dim4,dim0,dim1] for size(P) = [dim0,dim1,dim2,dim3,dim4] if permute_vec = [3,2,4,0,1]
For point *p* in P and point *q* in Q,
$q(y,x,z,v,w) = p(v,w,x,y,z)$
where,

$v = 0....dim0 - 1$
$w = 0....dim1 - 1$
$x = 0....dim2 - 1$
$y = 0....dim3 - 1$
$z = 0....dim4 - 1$

Prototype

```
WORD32 xa_nn_transpose_8_8
(WORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 const WORD8 * __restrict__ p_inp,
 const WORD32 *const p_inp_shape,
 const WORD32 * __restrict__ p_permute_vec,
 WORD32 num_out_dims,
 WORD32 num_inp_dims);
```

Arguments

Type	Name	Size	Description
Input			
const WORD32 *	p_out_shape	num_out_dims	Shape of output
const WORD8 *	p_inp	$\prod_{i=0}^{i=num\text{-}inp\text{-}dims-1} p\text{-}inp\text{-}shape[i]$	Input (set of cubes)
const WORD32 *	p_inp_shape	num_inp_dims	Shape of input
const WORD32 *	p_permute_vec	num_inp_dims	Permute Vector
WORD32	num_out_dims		Number of output dimensions
WORD32	num_inp_dims		Number of input dimensions
Output			
WORD8 *	p_out	$\prod_{i=0}^{i=num\text{-}out\text{-}dims-1} p\text{-}out\text{-}shape[i]$	Output (set of cubes)

Returns

- 0: no error
- -1: error, invalid parameters

Restrictions:

Arguments	Restrictions

`p_out, p_inp`	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Must not overlap
`p_out_shape, p_inp_shape`	Aligned on 4-byte boundary
	Cannot be NULL
	Must not overlap
	All elements must be greater than zero
`p_out_shape`	p_out_shape[i] = p_inp_shape[p_permute_vec[i]]
`p_permute_vec`	Cannot be NULL
`num_out_dims`	Must be in range [1, 5], should be equal to num_inp_dims.
`num_inp_dims`	Must be in range [1, 5], should be equal to num_out_dims.

4. HiFi 5 NN Library – Layers

This section explains the APIs of each layer implementation in the NN library. All the layers conform to the "generic NN Layer API" and flow explained in Section 2.

The NN library is a single archive containing all layers and low-level kernels implementations. Each layer has its own header file that defines the APIs specific to the layer. The following sections explain each layer in detail.

Note   This version of the library supports GRU, LSTM, and CNN layers.

4.1   GRU Layer

The GRU APIs are defined in xa_nnlib_gru_api.h. Refer to the overall signal flow diagram of GRU in [1].

4.1.1 GRU Layer Specification

GRU layer implements the following input-output equations when split_bias parameter is set as 0.

$$z_t = sigmoid(W_z * x_t + U_z * prev\text{-}h + b_z)$$
$$r_t = sigmoid(W_r * x_t + U_r * prev\text{-}h + b_r)$$
$$g = \tanh(W_h * x_t + U_h * (r_t \cdot prev\text{-}h) + b_h)$$
$$y_t = h_t = z_t \cdot g + (1 - z_t) \cdot prev\text{-}h$$
$$prev\text{-}h = h_t$$

GRU layer implements the following input-output equations when split_bias parameter is set as 1.

$$z_t = sigmoid(W_z * x_t + b_{sz} + U_z * prev\text{-}h + b_z)$$
$$r_t = sigmoid(W_r * x_t + b_{sr} + U_r * prev\text{-}h + b_r)$$
$$g = \tanh(W_h * x_t + b_{sh} + r_t * (U_h \cdot prev\text{-}h + b_h))$$
$$y_t = h_t = z_t \cdot prev\text{-}h + (1 - z_t) \cdot g$$
$$prev\text{-}h = h_t$$

x_t : input vector	z_t : update gate vector
$y_t,\ h_t$: output vector	r_t : reset gate vector
W, U : weight matrices	b : bias vectors
$prev\text{-}h$: previous output vector	

The biases b_{sr}, b_{sz}, b_{sh} are not used when split_bias = 0.

4.1.2 Error Codes Specific to GRU

Other than common error codes explained in Section 2.3, the GRU layer may also report the following error codes, which may be generated during the initialization stage.

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IN_FEATS[10]

  Number of input features is not supported

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_OUT_FEATS

  Number of output features is not supported

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PRECISION

  I/O precision is not supported

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_COEFF_QFORMAT

  Number of fractional bits for coefficients is not supported.

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IO_QFORMAT

  Number of fractional bits for input-output is not supported.

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_MEMBANK_PADDING

  Membank padding must be 0 or 1.

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PARAM_ID

  Parameter identifier (param_id) is not valid

- ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_SPLIT_BIAS

  Parameter split bias must be 0 or 1.

The following error codes may be generated during the execution stage.

- ■ XA_NNLIB_GRU_EXECUTE_FATAL_INSUFFICIENT_DATA

  Input data passed in is insufficient

- ■ XA_NNLIB_GRU_EXECUTE_FATAL_INSUFFICIENT_OUTPUT_BUFFER_SPACE

  Output Buffer Size is not sufficient

---

[10] FEATS := features

4.1.3 API Functions Specific to GRU

Query Functions

Table 4-1 GRU Get Persistent Size Function

Function	xa_nnlib_gru_get_persistent_fast
Syntax	Int32 xa_nnlib_gru_get_persistent_fast(　　　　　xa_nnlib_gru_init_config_t *config)
Description	Returns persistent memory size in bytes required by GRU layer.
Parameters	Input: config Initial configuration parameters (see Table 4-7).
Errors	If return value is less than 0, then it is an error. Following are the possible error codes: ■　XA_NNLIB_FATAL_MEM_ALLOC ■　XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IN_FEATS 　　Number of input features is not supported ■　XA_NNLIB_GRU_CONFIG_FATAL_INVALID_OUT_FEATS 　　Number of output features is not supported ■　XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PRECISION 　　I/O precision is not supported ■　XA_NNLIB_GRU_CONFIG_FATAL_INVALID_COEFF_QFORMAT 　　Number of fractional bits for coefficients is not supported. ■　XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IO_QFORMAT 　　Number of fractional bits for input-output is not supported.

Table 4-2 GRU Get Scratch Size Function

Function	`xa_nnlib_gru_get_scratch_fast`
Syntax	`Int32 xa_nnlib_gru_get_scratch_fast(` `           xa_nnlib_gru_init_config_t *config)`
Description	Returns scratch memory size in bytes required by GRU layer.
Parameters	Input: `config` Initial configuration parameters (see Table 4-7).

Errors	If return value is less than 0, then it is an error. Following are the possible error codes:
	
	■ XA_NNLIB_FATAL_MEM_ALLOC
	
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IN_FEATS
	
	  Number of input features is not supported
	
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_OUT_FEATS
	
	  Number of output features is not supported
	
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PRECISION
	
	  I/O precision is not supported
	
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_COEFF_QFORMAT
	
	  Number of fractional bits for coefficients is not supported
	
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IO_QFORMAT
	
	  Number of fractional bits for input-output is not supported

Initialization Stage

Table 4-3 GRU Init Function

Function	`xa_nnlib_gru_init`

Syntax	`Int32`
	
	`xa_nnlib_gru_init (`
	
	`    xa_nnlib_handle_t handle,`
	
	`    xa_nnlib_gru_init_config_t *config)`

Description	Reset the GRU Layer API handle into its initial state. Set up the GRU Layer to the specified initial configuration parameters. This function sets prev_h vector to 0; the user can enter the required values in prev_h by using set config XA_NNLIB_GRU_RESTORE_CONTEXT (for more information, seeTable 4-11).
Parameters	Input: `handle` Pointer to the component persistent memory. This is the opaque handle. Required size: see `xa_nnlib_gru_get_persistent_fast`. Required alignment: 8 bytes. Input: `config` Initial configuration parameters (see Table 4-7). **Note:** The initial configuration parameters must be identical to those passed to query functions.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers is invalid. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers is not properly aligned. ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IN_FEATS Number of input features is not supported ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_OUT_FEATS Number of output features is not supported ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PRECISION I/O precision is not supported. ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_COEFF_QFORMAT Number of fractional bits for coefficients is not supported.

■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_IO_QFORMAT

Number of fractional bits for input-output is not supported.

■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_MEMBANK_PADDING

Membank padding must be 0 or 1.

Execution Stage

Table 4-4 GRU Execution Function

Function	xa_nnlib_gru_process

Syntax	`Int32 xa_nnlib_gru_process(`
	`                xa_nnlib_handle_t handle,`
	`                void *scratch,`
	`                void *input,`
	`                void *output,`
	`                xa_nnlib_shape_t *p_in_shape,`
	`                xa_nnlib_shape_t`
	`                *p_out_shape)`

Description	Processes one input shape to generate one output shape.
Parameters	Input: `handle` The opaque component handle. Required alignment: 8 bytes. Input: `scratch` A pointer to the scratch buffer. Required alignment: 8 bytes. Input: `input` A pointer to the input buffer. Input buffer contains input data. Required alignment: 8 bytes. Output: `output` A pointer to the output buffer. Output is written to output buffer. Required alignment: 8 bytes. Input/Output: `p_in_shape` Pointer to the shape containing input buffer dimensions. Contains the length of input data passed to GRU layer. Required alignment: 4 bytes. Input/Output: `p_out_shape` Pointer to the shape for output buffer dimensions. On return, `*p_out_shape` is filled with the length of output generated by HiFi GRU Layer.

	Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers is NULL. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers is not properly aligned. ■ XA_NNLIB_FATAL_INVALID_SHAPE Either input or output shape is invalid. ■ XA_NNLIB_GRU_EXECUTE_FATAL_INSUFFICIENT_DATA Input data passed in insufficient. ■ XA_NNLIB_GRU_EXECUTE_FATAL_INSUFFICIENT_OUTPUT_BUFFER_SPACE Output buffer size is not sufficient.

Table 4-5 GRU Set Parameter Function Details

Function	xa_nnlib_gru_set_config

Syntax	`Int32`
	`xa_nnlib_gru_set_config (`
	`    xa_nnlib_handle_t handle,`
	`    xa_nnlib_gru_param_id_t param_id,`
	`    void *params)`

Description	Sets the parameter specified by `param_id` to the value passed in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be written. For more information on the list of supported parameters, see Table 4-11. **Input:** `params` A pointer to a buffer that contains the parameter value. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers (`handle` or `params`) is `NULL`.

	■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers (`handle` or `params`) is not aligned correctly.
	■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PARAM_ID Parameter identifier (`param_id`) is not valid.

Table 4-6 GRU Get Parameter Function Details

Function	`xa_nnlib_gru_get_config`

Syntax	`Int32 xa_nnlib_gru_get_config (`
	`    xa_nnlib_handle_t handle,`
	`    xa_nnlib_gru_param_id_t param_id,`
	`    void *params)`

Description	Gets the value of the parameter specified by `param_id` in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be read. For more information on the list of supported parameters, see Table 4-11. **Output:** `params` A pointer to a buffer that is filled with the parameter value when the function returns. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers (`handle` or `params`) is `NULL`. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers (`handle` or `params`) is not aligned correctly. ■ XA_NNLIB_GRU_CONFIG_FATAL_INVALID_PARAM_ID Parameter identifier (`param_id`) is not valid.

4.1.4 Structures Specific to GRU

Table 4-7 GRU Config Structure xa‿nnlib‿gru‿init‿config‿t

Element Type	Element Name	Range	Default	Description
Int32	in_feats	4-2048	256	Number of input features (must be multiple of 4)
Int32	out_feats	4-2048	256	Number of output features (must be multiple of 4)
Int32	pad	0, 1	1	Padding 16 bytes for HiFi 5
Int32	mat_prec	8, 16	16	Matrix input precision
Int32	vec_prec	16	16	Vector input precision
xa_nnlib_gru_precision_t	precision	XA_NNLIB_GRU_16bx16b, XA_NNLIB_GRU_8bx16b, XA_NNLIB_GRU_flt32xflt32	XA_NNLIB_GRU_16bx16b	Coef and I/O precision. **Note:** The current library supports only 16bx16b, 8bx16b and float32xfloat32 precision for GRU
Int16	coeff_Qformat	0-15	15	Number of fractional bits for weights and biases
Int16	io_Qformat	0-15	12	Number of fractional bits for input and output
Int32	split_bias	0,1	0	0 for Tensorflow equations and 1 for PyTorch equations.

Table 4-8 xa‿nnlib‿gru‿weights‿t Parameter Type

Element Type	Element Name	Range	Default	Description
coeff_t* coeff8_t* float*	w_z	NA	NA	Pointer to coefficient matrix w‿z.
xa_nnlib_shape_t	shape_w_z	NA	NA	Shape information about w‿z.
coeff_t* coeff8_t* float*	u_z	NA	NA	Pointer to coefficient matrix u‿z.
xa_nnlib_shape_t	shape_u_z	NA	NA	Shape information about u‿z.
coeff_t* coeff8_t* float*	w_r	NA	NA	Pointer to coefficient matrix w‿r.
xa_nnlib_shape_t	shape_w_r	NA	NA	Shape information about w‿r.

Element Type	Element Name	Range	Default	Description
coeff_t* coeff8_t* float*	u_r	NA	NA	Pointer to coefficient matrix u_r.
xa_nnlib_ shape_t	shape_u_r	NA	NA	Shape information about u_r.
coeff_t* coeff8_t* float*	w_h	NA	NA	Pointer to coefficient matrix w_h.
xa_nnlib_ shape_t	shape_w_h	NA	NA	Shape information about w_h.
coeff_t* coeff8_t* float*	u_h	NA	NA	Pointer to coefficient matrix u_h.
xa_nnlib_ shape_t	shape_u_h	NA	NA	Shape information about u_h.

Table 4-9 xa_nnlib_gru_biases_t Parameter Type

Element Type	Element Name	Range	Default	Description
void *	b_z	NA	NA	Pointer to bias vector b_z.
xa_nnlib_ shape_t	shape_b_z	NA	NA	Shape information about b_z.
void *	b_r	NA	NA	Pointer to bias vector b_r.
xa_nnlib_ shape_t	shape_b_r	NA	NA	Shape information about b_r.
void *	b_h	NA	NA	Pointer to bias vector b_h.
xa_nnlib_ shape_t	shape_b_h	NA	NA	Shape information about b_h.
void *	bs_z	NA	NA	Pointer to bias vector bs_z.
xa_nnlib_ shape_t	shape_bs_z	NA	NA	Shape information about bs_z.
void *	bs_r	NA	NA	Pointer to bias vector bs_r.
xa_nnlib_ shape_t	shape_bs_r	NA	NA	Shape information about bs_r.
void *	bs_h	NA	NA	Pointer to bias vector bs_h.
xa_nnlib_ shape_t	shape_bs_h	NA	NA	Shape information about bs_h.

Note   GRU requires all weight matrices' and bias vectors' pointers to be 8 bytes aligned.

4.1.5 Enums Specific to GRU

Table 4-10 Enum xa_nnlib_gru_precision_t

Element	Description
XA_NNLIB_GRU_16bx16b	Coef: 16 bits, I/O: 16 bits Fixed Point
XA_NNLIB_GRU_8bx16b	Coef: 8 bits, I/O: 16 bits Fixed Point
XA_NNLIB_flt32xflt32	Coef: float32, I/O: float32
XA_NNLIB_GRU_8bx8b	Not supported
XA_NNLIB_flt16xflt16	Not supported

Note Currently, GRU only supports XA_NNLIB_GRU_16bx16b, XA_NNLIB_GRU_8bx16b precision setting.

Table 4-11 describes parameter IDs for parameters supported by GRU. It contains the following columns:

- Parameter ID: Parameter identifier (`param_id`).
- Value type: A pointer (`params`) to a variable of this type is to be passed.
- RW: Indicates whether the parameter can be read (`get`) and/or written (`set`).
- Range: Indicates valid values of the parameter.
- Default: Default value of the parameter
- Description: Brief description of the parameter.

Table 4-11 GRU Specific Parameters

Parameter ID	Value Type	RW	Range	Default	Description
XA_NNLIB_GRU_RESTORE_CONTEXT	vect_t []	*RW*	NA	NA	Set previous output. This can be used to set prev_h to specific context (size must be equal to number of output features). Upon set config, the buffer passed is copied to persistent memory; upon get config, it returns the prev_h state in the given buffer.
XA_NNLIB_GRU_WEIGHT	xa_nnlib_gru_weights_t	*RW*	NA	NA	Weight matrices, pointers to weight matrices along with shape information must be passed via xa_nnlib_gru_weights_t structure for set config. Upon get config, it returns pointers to weight matrices along with their shape information in same structure.
XA_NNLIB_GRU_BIAS	xa_nnlib_gru_	*RW*	NA	NA	Bias vectors, pointers to bias vectors along with shape information must be passed via

Parameter ID	Value Type	RW	Range	Default	Description
	biases_t				xa_nnlib_gru_biases_t structure for set config. Upon get config, it returns pointers to bias vectors along with their shape information in same structure.
XA_NNLIB_GRU_INPUT_SHAPE	xa_nnlib_shape_t	*R*	NA	NA	Input shape information, get information of the input shape expected by the layer.
XA_NNLIB_GRU_OUTPUT_SHAPE	xa_nnlib_shape_t	*R*	NA	NA	Output shape information, get information of the output shape expected by layer.

4.2   LSTM Layer

The LSTM APIs are defined in xa_nnlib_lstm_api.h.

4.2.1 LSTM Layer Specification

The LSTM layer implements the following forward path input-output equations:

$$f_f = sigmoid(w_{xf} * frame_f + prev\text{-}h * w_{hf} + b_f)$$
$$i_f = sigmoid(w_{xi} * frame_f + prev\text{-}h * w_{hi} + b_i)$$
$$c\text{-}hat_f = \tanh(w_{xc} * frame_f + prev\text{-}h * w_{hc} + b_c)$$
$$c_f = f_f . prev\text{-}c + i_f * c\text{-}hat_f$$
$$o_f = sigmoid(w_{xo} * frame_f + prev\text{-}h * w_{ho} + b_o)$$
$$h_f = o_f * \tanh(c_f)$$

i_f : input gate

h_t : output vector

$c\text{-}hat_f$: intermediate cell state vector

f_f : forget gate

$frame_f$: Input vector

w_x : weight matrices of input
     connections

$prev\text{-}h$: previous output vector

$prev\text{-}c$: previous cell output

b : bias vectors

o_f : output gate

c_f : cell state vector

w_h : weight matrices of recurrent
     connections

4.2.2 Error Codes Specific to LSTM

Other than common error codes explained in Section 2.3, the LSTM layer may also report the following error codes, which may be generated during the initialization stage:

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IN_FEATS[11]

  Number of input features is not supported

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_OUT_FEATS

  Number of output features is not supported

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PRECISION

  I/O precision is not supported

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_COEFF_QFORMAT

  Number of fractional bits for coefficients is not supported.

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_CELL_QFORMAT

  Number of fractional bits for cells is not supported

---

[11] FEATS: = features

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IO_QFORMAT

  Number of fractional bits for input-output is not supported.

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_MEMBANK_PADDING

  Membank padding must be 0 or 1.

- XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PARAM_ID

  Parameter identifier (param_id) is not valid

The following error codes may be generated during the execution stage.

- XA_NNLIB_LSTM_EXECUTE_FATAL_INSUFFICIENT_DATA

  Input data passed in insufficient

- XA_NNLIB_LSTM_EXECUTE_FATAL_INSUFFICIENT_OUTPUT_BUFFER_
  SPACE

  Output Buffer Size is not sufficient

4.2.3 API Functions Specific to LSTM

Query Functions

Table 4-12 LSTM Get Persistent Size Function

Function	`xa_nnlib_lstm_get_persistent_fast`
Syntax	`Int32 xa_nnlib_lstm_get_persistent_fast (` `            xa_nnlib_lstm_init_config_t *config)`
Description	Returns persistent memory size in bytes required by LSTM layer.
Parameters	Input: `config` Initial configuration parameters (see Table 4-18).

Errors	If return value is less than 0 then it is an error. Following are the possible error codes:
	
	■ XA_NNLIB_FATAL_MEM_ALLOC
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IN_FEATS
	  Number of input features is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_OUT_FEATS
	  Number of output features is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PRECISION
	  I/O precision is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_COEFF_QFORMAT
	  Number of fractional bits for coefficients is not supported.
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_CELL_QFORMAT
	  Number of fractional bits for cells is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IO_QFORMAT
	  Number of fractional bits for input-output is not supported.
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_MEMBANK_PADDING
	  Membank padding must be 0 or 1.

Table 4-13 LSTM Get Scratch Size Function

Function	`xa_nnlib_lstm_get_scratch_fast`
Syntax	`Int32 xa_nnlib_lstm_get_scratch_fast (` `            xa_nnlib_lstm_init_config_t *config)`
Description	Returns scratch memory size in bytes required by LSTM layer.
Parameters	Input: `config` Initial configuration parameters (see Table 4-18).
Errors	If return value is less than 0 then it is an error, the possible error codes are: ■ XA_NNLIB_FATAL_MEM_ALLOC ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IN_FEATS Number of input features is not supported ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_OUT_FEATS Number of output features is not supported ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PRECISION I/O precision is not supported ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_COEFF_QFORMAT Number of fractional bits for coefficients is not supported. ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_CELL_QFORMAT Number of fractional bits for cells is not supported ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IO_QFORMAT Number of fractional bits for input-output is not supported. ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_MEMBANK_PADDING Membank padding must be 0 or 1.

Initialization Stage

Table 4-14 LSTM Init Function

Function	`xa_nnlib_lstm_init`

Syntax	`Int32`
	`xa_nnlib_lstm_init (`
	      `xa_nnlib_handle_t handle,`
	      `xa_nnlib_lstm_init_config_t *config)`

Description	Reset the LSTM layer API handle into its initial state. Set up the LSTM layer to the specified initial configuration parameters. This function sets prev_h vector and prev_c vector to 0; the user can enter the required values in prev_h and prev_c by using set config XA_NNLIB_LSTM_RESTORE_CONTEXT_OUTPUT and XA_NNLIB_LSTM_RESTORE_CONTEXT_CELL respectively (for more information, see Table 4-22).
Parameters	Input: `handle` Pointer to the component persistent memory. This is the opaque handle. Required size: see `xa_nnlib_lstm_get_persistent_fast`. Required alignment: 8 bytes. Input: `config` Initial configuration parameters (see Table 4-18). **Note:** The initial configuration parameters must be identical to those passed to query functions.

Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors:
	
	■ XA_NNLIB_FATAL_MEM_ALLOC
	   One of the pointers is invalid.
	
	■ XA_NNLIB_FATAL_MEM_ALIGN
	   One of the pointers is not properly aligned.
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IN_FEATS
	   Number of input features is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_OUT_FEATS
	   Number of output features is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PRECISION
	   I/O precision is not supported
	
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_COEFF_QFORMAT
	   Number of fractional bits for coefficients is not supported.

	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_CELL_QFORMAT
	Number of fractional bits for cells is not supported
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_IO_QFORMAT
	Number of fractional bits for input-output is not supported
	■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_MEMBANK_PADDIN G
	Membank padding must be 0 or 1.

Execution Stage

Table 4-15 LSTM Execution Function

Function	`xa_nnlib_lstm_process`
Syntax	```Int32 xa_nnlib_lstm_process (xa_nnlib_handle_t handle, void *scratch, void *input, void *output, xa_nnlib_shape_t *p_in_shape, xa_nnlib_shape_t *p_out_shape)```
Description	Processes one input shape to generate one output shape.
Parameters	Input: `handle` The opaque component handle. Required alignment: 8 bytes. Input: `scratch` A pointer to the scratch buffer. Required alignment: 8 bytes. Input: `input` A pointer to the input buffer. Input buffer contains input data. Required alignment: 8 bytes. Output: `output` A pointer to the output buffer. Output is written to the output buffer. Required alignment: 8 bytes. Input/Output: `p_in_shape` Pointer to the shape containing input buffer dimensions. Contains the length of input data passed to LSTM layer. Required alignment: 4 bytes. Input/Output: `p_out_shape`

	Pointer to the shape for output buffer dimensions. On return, *p_out_shape is filled with the length of output generated by HiFi LSTM layer. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers is NULL. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers is not having proper alignment. ■ XA_NNLIB_FATAL_INVALID_SHAPE Either input or output shape is invalid. ■ XA_NNLIB_LSTM_EXECUTE_FATAL_INSUFFICIENT_DATA Input data passed in insufficient ■ XA_NNLIB_LSTM_EXECUTE_FATAL_INSUFFICIENT_OUTPUT_ BUFFER_SPACE Output Buffer Size is not sufficient

Table 4-16 LSTM Set Parameter Function Details

Function	`xa_nnlib_lstm_set_config`

Syntax	`Int32`
	`xa_nnlib_lstm_set_config (`
	      `xa_nnlib_handle_t handle,`
	      `xa_nnlib_lstm_param_id_t param_id,`
	      `void *params)`

Description	Sets the parameter specified by `param_id` to the value passed in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be written. For more information on the list of supported parameters, see Table 4-11. **Input:** `params` A pointer to a buffer that contains the parameter value. Required alignment: 4 bytes.

Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors:
	
	  ■ XA_NNLIB_FATAL_MEM_ALLOC
	      One of the pointers (`handle` or `params`) is `NULL`.
	
	  ■ XA_NNLIB_FATAL_MEM_ALIGN
	      One of the pointers (`handle` or `params`) is not aligned correctly.
	
	  ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PARAM_ID
	      Parameter identifier (`param_id`) is not valid.

Table 4-17 LSTM Get Parameter Function Details

Function	`xa_nnlib_lstm_get_config`

Syntax	`Int32 xa_nnlib_lstm_get_config (`
	`        xa_nnlib_handle_t handle,`
	`        xa_nnlib_lstm_param_id_t param_id,`
	`        void *params)`

Description	Gets the value of the parameter specified by `param_id` in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be read. For more information on the list of supported parameters, see Table 4-11. **Output:** `params` A pointer to a buffer that is filled with the parameter value when the function returns. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers (`handle` or `params`) is `NULL`. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers (`handle` or `params`) is not aligned correctly. ■ XA_NNLIB_LSTM_CONFIG_FATAL_INVALID_PARAM_ID Parameter identifier (`param_id`) is not valid.

4.2.4 Structures Specific to LSTM

Table 4-18 LSTM Config Structure xa‗nnlib‗lstm‗init‗config‗t

Element Type	Element Name	Range	Default	Description
Int32	in_feats	4-2048	256	Number of input features (must be multiple of 4)
Int32	out_feats	4-2048	256	Number of output features (must be multiple of 4)
Int32	pad	0, 1	1	Padding 16 bytes for HiFi 5 DSP
Int32	mat_prec	8, 16	16	Matrix input precision
Int32	vec_prec	16	16	Vector input precision
xa_nnlib_lstm_precision_t	precision	XA‗NNLIB‗LSTM‗16bx16b, XA‗NNLIB‗LSTM‗8bx16b	XA‗NNLIB‗LSTM‗16bx16b	Coef and I/O precision. **Note:** The current library supports only 16bx16b and 8bx16b precision for LSTM.
Int16	coeff_Qformat	0-15	15	Number of fractional bits for weights and biases
Int16	cell_Qformat	0-26		Number of fractional bits for cells.
Int16	io_Qformat	0-15	12	Number of fractional bits for input and output

Table 4-19 xa‗nnlib‗lstm‗weights‗t Parameter Type

Element Type	Element Name	Range	Default	Description
coeff_t *	w_xf	NA	NA	Pointer to coefficient matrix w‗xf.
xa_nnlib_shape_t	shape_w_xf	NA	NA	Shape information about w‗xf.
coeff_t *	w_xi	NA	NA	Pointer to coefficient matrix w‗xi.
xa_nnlib_shape_t	shape_w_xi	NA	NA	Shape information about w‗xi.
coeff_t *	w_xc	NA	NA	Pointer to coefficient matrix w‗xc.
xa_nnlib_shape_t	shape_w_xc	NA	NA	Shape information about w‗xc.
coeff_t *	w_xo	NA	NA	Pointer to coefficient matrix w‗xo.
xa_nnlib_shape_t	shape_w_xo	NA	NA	Shape information about w‗xo.
coeff_t *	w_hf	NA	NA	Pointer to coefficient matrix w‗hf.
xa_nnlib_shape_t	shape_w_hf	NA	NA	Shape information about w‗hf.
coeff_t *	w_hi	NA	NA	Pointer to coefficient matrix w‗hi.

Element Type	Element Name	Range	Default	Description
`xa_nnlib_shape_t`	`shape_w_hi`	NA	NA	Shape information about w_hi.
`coeff_t *`	`w_hc`	NA	NA	Pointer to coefficient matrix w_hc.
`xa_nnlib_shape_t`	`shape_w_hc`	NA	NA	Shape information about w_hc.
`coeff_t *`	`w_ho`	NA	NA	Pointer to coefficient matrix w_ho.
`xa_nnlib_shape_t`	`shape_w_ho`	NA	NA	Shape information about w_ho.

Table 4-20 xa_nnlib_lstm_biases_t Parameter Type

Element Type	Element Name	Range	Default	Description
`coeff_t *`	`b_f`	NA	NA	Pointer to coefficient matrix b_f.
`xa_nnlib_shape_t`	`shape_b_f`	NA	NA	Shape information about b_f.
`coeff_t *`	`b_i`	NA	NA	Pointer to coefficient matrix b_i.
`xa_nnlib_shape_t`	`shape_b_i`	NA	NA	Shape information about b_i.
`coeff_t *`	`b_c`	NA	NA	Pointer to coefficient matrix b_c.
`xa_nnlib_shape_t`	`shape_b_c`	NA	NA	Shape information about b_c.
`coeff_t *`	`b_o`	NA	NA	Pointer to coefficient matrix b_o.
`xa_nnlib_shape_t`	`shape_b_o`	NA	NA	Shape information about b_o.

Note   LSTM requires all weight matrices' and bias vectors' pointers to be 8 bytes aligned.

4.2.5 Enums Specific to LSTM

Table 4-21 Enum xa_nnlib_lstm_precision_t

Element	Description
`XA_NNLIB_LSTM_16bx16b`	Coef: 16 bits, I/O: 16 bits Fixed Point
`XA_NNLIB_LSTM_8bx16b`	Coef: 8 bits, I/O: 16 bits Fixed Point
`XA_NNLIB_LSTM_8bx8b`	Not supported
`XA_NNLIB_flt16xflt16`	Not supported

Note   Currently, LSTM only supports the XA_NNLIB_LSTM_16bx16b, XA_NNLIB_LSTM_8bx16b precision setting.

Table 4-22 describes parameter IDs for parameters supported by LSTM. It contains the following columns:

- Parameter ID: Parameter identifier (`param_id`).

- Value type: A pointer (`params`) to a variable of this type is to be passed.

- RW: Indicates whether the parameter can be read (`get`) and/or written (`set`).

- Range: Indicates valid values of the parameter.

- Default: Default value of the parameter.

- Description: Brief description of the parameter.

Table 4-22 LSTM Specific Parameters

Parameter ID	Value Type	RW	Range	Default	Description
XA_NNLIB_LSTM_RESTORE_CONTEXT_OUTPUT	vect_t []	*RW*	NA	NA	Set previous output. This can be used to set prev‗h to specific context (size must be equal to number of output features). Upon set config, the buffer passed is copied to persistent memory; upon get config, it returns the prev‗h state in the given buffer.
XA_NNLIB_LSTM_RESTORE_CONTEXT_CELL	vect_t []	*RW*	NA	NA	Set previous cell state. This can be used to set prev‗c to specific cell context (size must be equal to number of output features). Upon set config, the buffer passed is copied to persistent memory; upon get config, it returns the prev‗c state in the given buffer.
XA_NNLIB_LSTM_WEIGHT	xa_nnlib_ lstm_ weights_t	*RW*	NA	NA	Weight matrices, pointers to weight matrices along with shape information needs to be passed via xa‗nnlib‗lstm‗weights‗t structure for set config. Upon get config, it returns pointers to weight matrices along with their shape information in same structure.
XA_NNLIB_LSTM_BIAS	xa_nnlib_ lstm_ biases_t	*RW*	NA	NA	Bias vectors, pointers to bias vectors along with shape information needs to be passed via xa‗nnlib‗lstm‗biases‗t structure for set config. Upon get config, it returns pointers to bias vectors along with their shape information in same structure.
XA_NNLIB_LSTM_INPUT_SHAPE	xa_nnlib_ shape_t	*R*	NA	NA	Input shape information, get information of the input shape expected by the layer.
A_NNLIB_LSTM_OUTPUT_SHAPE	xa_nnlib_ shape_t	*R*	NA	NA	Output shape information, get information of the output shape expected by layer.

4.3 CNN Layer

The CNN APIs are defined in `xa_nnlib_cnn_api.h`.

4.3.1 CNN Layer Specification

The CNN layer implements Standard 2D Convolution, Standard 1D Convolution, and Depthwise Separable 2D Convolution. For more information on the equations, see Section 3.2.1 for Standard 2D Convolution, Section 3.2.2 for Standard 1D Convolution, and Section 3.2.4 for Depthwise Separable 2D Convolution.

4.3.2 Error Codes Specific to CNN

Other than common error codes explained in Section 2.3, the CNN layer may also report the following error codes, which may be generated during the initialization stage.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ALGO

  Algorithm is not supported

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PRECISION

  I/O precision is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHIFT

  Value of Bias shift is not supported

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ACC_SHIFT

  Value of Accumulator shift is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_STRIDE

  Value of strides is not supported

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PADDING

  Value of padding is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_INPUT_SHAPE

  Input shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_OUTPUT_SHAPE

  Out shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_KERNEL_SHAPE

  Kernel shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHAPE

  Bias shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID

  Parameter identifier (param_id) is not valid

■   XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_COMBINATION

Parameter combination (param_id) is not valid

The following error codes may be generated during the execution stage.

■   XA_NNLIB_CNN_EXECUTE_FATAL_INVALID_INPUT_SHAPE

Input shape passed during execution does not match with the input shape passed during initialization

4.3.3 API Functions Specific to CNN

Query Functions

Table 4-23 CNN Get Persistent Size Function

Function	`xa_nnlib_cnn_get_persistent_fast`
Syntax	`Int32 xa_nnlib_cnn_get_persistent_fast (` `            xa_nnlib_cnn_init_config_t *config)`
Description	Returns persistent memory size in bytes required by CNN layer.
Parameters	Input: `config` Initial configuration parameters (see Table 4-29).
Errors	If return value is less than 0, then it is an error. Following are the possible error codes: ■ XA_NNLIB_FATAL_MEM_ALLOC ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ALGO Algorithm is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PRECISION I/O precision is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHIFT Value of Bias shift is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ACC_SHIFT Value of Accumulator shift is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_STRIDE Value of strides is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PADDING Value of padding is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_INPUT_SHAPE

Input shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_OUTPUT_SHAPE

Out shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_KERNEL_SHAPE

Kernel shape dimension is not supported.

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHAPE

Bias shape dimension is not supported

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID

Parameter identifier (param_id) is not valid

- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_COMBIN ATION

Parameter combination (param_id) is not valid

Table 4-24 CNN Get Scratch Size Function

Function	`xa_nnlib_cnn_get_scratch_fast`
Syntax	`Int32 xa_nnlib_cnn_get_scratch_fast (` `            xa_nnlib_cnn_init_config_t *config)`
Description	Returns scratch memory size in bytes required by CNN layer.
Parameters	Input: `config` Initial configuration parameters (see Table 4-29).
Errors	If return value is less than 0, then it is an error. Following are the possible error codes: ■ XA_NNLIB_FATAL_MEM_ALLOC ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ALGO Algorithm is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PRECISION I/O precision is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHIFT Value of bias shift is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ACC_SHIFT Value of Accumulator shift is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_STRIDE Value of strides is not supported ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PADDING Value of padding is not supported.

	- XA_NNLIB_CNN_CONFIG_FATAL_INVALID_INPUT_SHAPE Input shape dimension is not supported. - XA_NNLIB_CNN_CONFIG_FATAL_INVALID_OUTPUT_SHAPE Out shape dimension is not supported. - XA_NNLIB_CNN_CONFIG_FATAL_INVALID_KERNEL_SHAPE Kernel shape dimension is not supported. - XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHAPE Bias shape dimension is not supported. - XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID Parameter identifier (param_id) is not valid - XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_COMBINATION Parameter combination (param_id) is not valid

Initialization Stage

Table 4-25 CNN Init Function

Function	`xa_nnlib_cnn_init`

Syntax	`int xa_nnlib_cnn_init (`
	`        xa_nnlib_handle_t handle,`
	`        xa_nnlib_cnn_init_config_t *config)`

Description	Reset the CNN layer API handle into its initial state. Set up the CNN layer to the specified initial configuration parameters.
Parameters	Input: `handle` Pointer to the component persistent memory. This is the opaque handle. Required size: see `xa_nnlib_cnn_get_persistent_fast`. Required alignment: 8 bytes. Input: `config` Initial configuration parameters (see Table 4-29). **Note:** The initial configuration parameters must be identical to those passed to query functions.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers is invalid. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers is not properly aligned. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ALGO Algorithm is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PRECISION I/O precision is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHIFT Value of Bias shift is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_ACC_SHIFT Value of Accumulator shift is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_STRIDE Value of strides is not supported. ■ XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PADDING Value of padding is not supported.

	XA_NNLIB_CNN_CONFIG_FATAL_INVALID_INPUT_SHAPE Input shape dimension is not supported. **XA_NNLIB_CNN_CONFIG_FATAL_INVALID_OUTPUT_SHAPE** Out shape dimension is not supported. **XA_NNLIB_CNN_CONFIG_FATAL_INVALID_KERNEL_SHAPE** Kernel shape dimension is not supported. **XA_NNLIB_CNN_CONFIG_FATAL_INVALID_BIAS_SHAPE** Bias shape dimension is not supported. **XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID** Parameter identifier (param_id) is not valid. **XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_COMBINATION** Parameter combination (param_id) is not valid.

Execution Stage

Table 4-26 CNN Execution Function

Function	`xa_nnlib_cnn_process`

Syntax	`int xa_nnlib_cnn_process (`
	`                xa_nnlib_handle_t handle,`
	`                void *scratch,`
	`                void *input,`
	`                void *output,`
	`                xa_nnlib_shape_t *p_in_shape,`
	`                xa_nnlib_shape_t *p_out_shape)`

Description	Processes one input shape to generate one output shape.
Parameters	Input: `handle` The opaque component handle. Required alignment: 8 bytes. Input: `scratch` A pointer to the scratch buffer. Required alignment: 8 bytes. Input: `input` A pointer to the input buffer. Input buffer contains input data. Required alignment: 8 bytes. Output: `output` A pointer to the output buffer. Output is written to the output buffer. Required alignment: 8 bytes. Input/Output: `p_in_shape` Pointer to the shape containing input buffer dimensions. Contains the length of input data passed to the CNN layer. Required alignment: 4 bytes. Output: `p_out_shape` Pointer to the shape for output buffer dimensions. Upon return, `*p_out_shape` is filled with the length of output generated by the CNN layer. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers is NULL

- **XA_NNLIB_FATAL_MEM_ALIGN**

  One of the pointers is not having required alignment

- **XA_NNLIB_FATAL_INVALID_SHAPE**

  Input shape passed during execution does not match with the input shape passed during initialization

Table 4-27 CNN Set Parameter Function Details

Function	`xa_nnlib_cnn_set_config`

Syntax	`int xa_nnlib_cnn_set_config (`
	`    xa_nnlib_handle_t handle,`
	`    xa_nnlib_cnn_param_id_t param_id,`
	`    void *params)`

Description	Sets the parameter specified by `param_id` to the value passed in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be written. For the list of supported parameters, see Table 4-32. **Input:** `params` A pointer to a buffer that contains the parameter value. Required alignment: 4 bytes.

Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors:
	
	- **XA_NNLIB_FATAL_MEM_ALLOC**
	  One of the pointers (`handle` or `params`) is `NULL`.
	- **XA_NNLIB_FATAL_MEM_ALIGN**
	  One of the pointers (`handle` or `params`) is not aligned correctly.
	  XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID
	  Parameter identifier (`param_id`) is not valid.

Table 4-28 CNN Get Parameter Function Details

Function	`xa_nnlib_cnn_get_config`

Syntax	`int xa_nnlib_cnn_get_config(`
	  `xa_nnlib_handle_t handle,`
	  `xa_nnlib_cnn_param_id_t param_id,`
	  `void *params )`

Description	Gets the value of the parameter specified by `param_id` in the buffer pointed to by `params`.
Parameters	**Input:** `handle` The opaque component handle. Required alignment: 8 bytes. **Input:** `param_id` Identifies the parameter to be read. For the list of supported parameters, see Table 4-32. **Output:** `params` A pointer to a buffer that is filled with the parameter value when the function returns. Required alignment: 4 bytes.
Errors	If the return value is not XA_NNLIB_NO_ERROR, it implies that the function has encountered one of the following errors: ■ XA_NNLIB_FATAL_MEM_ALLOC One of the pointers (`handle` or `params`) is `NULL`. ■ XA_NNLIB_FATAL_MEM_ALIGN One of the pointers (`handle` or `params`) is not aligned correctly. XA_NNLIB_CNN_CONFIG_FATAL_INVALID_PARAM_ID Parameter identifier (`param_id`) is not valid.

4.3.4 Structures Specific to CNN

Table 4-29 CNN Config Structure xa_nnlib_cnn_init_config_t

Element Type	Element Name	Range	Default	Description
xa_nnlib_ shape_t	input_ shape	NA	height = 16 width = 16 channels = 4	Input shape dimensions
Int32	output_ height	NA	16	Output height
Int32	output_ width	NA	16	Output width
Int32	output_ channels	NA	4	Output depth or channels
Int32	output_ format	0 or 1	0	Output data format 0: SHAPE_CUBE_DWH_T 1: SHAPE_CUBE_WHD_T
xa_nnlib_ shape_t	kernel_ std_shape	NA	height = 16 width = 16 channels = 4	Standard 1D/2D Convolution Kernel (Filter) shape dimensions output_channels indicate number of kernels
xa_nnlib_ shape_t	kernel_ ds_depth_ shape	NA	NA	Depthwise Separable 2D Convolution - Depthwise Kernel (filter) Dimensions
xa_nnlib_ shape_t	kernel_ds_ point_ shape	NA	NA	Depthwise Separable 2D Convolution - Pointwise Kernel (filter) Dimensions
xa_nnlib_ shape_t	bias_std_ shape	NA	channels = 4	Standard 1D/2D Convolution Bias dimensions
xa_nnlib_ shape_t	bias_ds_ depth_ shape	NA	NA	Depthwise Separable 2D Convolution - Depthwise Bias) Dimensions
xa_nnlib_ shape_t	bias_ds_ point_ shape	NA	NA	Depthwise Separable 2D Convolution – Pointwise Bias Dimensions
xa_nnlib_cnn _precision_t	precision	XA_NNLIB_ CNN_16bx1 6b, XA_NNLIB_ CNN_8bx16 b, XA_NNLIB_ CNN_8bx8b, XA_NNLIB_ CNN_f32xf3 2	XA_NNLIB_CNN_8b x16b	Kernel (filter), input, output precision setting
Int32	bias_ shift	-31 to 31	7	Q-format adjustment for bias before addition into accumulator, +/- value - left/right shift

Element Type	Element Name	Range	Default	Description
`Int32`	`acc_shift`	-31 to 31	-7	Q-format adjustment for accumulator before rounding to result, +/- value - left/right shift
`Int32`	`channels_ multiplier`	NA	NA	Depthwise Separable 2D Convolution - channel multiplier. (`channels_multiplier * input_channels`) must be multiple of 4
`Int32`	`x_padding`	NA	2	Left side padding to be added to input
`Int32`	`y_padding`	NA	2	Top padding to be added to input
`Int32`	`x_stride`	NA	2	Strides over padded input in width dimension
`Int32`	`y_stride`	NA	2	Strides over padded input in height dimension
`xa_nnlib_cnn _algo_t`	`algo`	NA	XA_NNLIB_CNN_ CONV2D_STD	Convolution algorithm

4.3.5 Enums Specific to CNN

Table 4-30 Enum xa_nnlib_cnn_precision_t

Element	Description
`XA_NNLIB_CNN_16bx16b`	Coef: 16 bits, I/O: 16 bits fixed point
`XA_NNLIB_CNN_8bx16b`	Coef: 8 bits, I/O: 16 bits fixed point
`XA_NNLIB_CNN_8bx8b`	Coef: 8 bits, I/O: 8 bits fixed point
`XA_NNLIB_CNN_f32xf32`	Coef: single precision float, I/O: single precision float

Table 4-31 Enum xa_nnlib_cnn_algo_t

Element	Description
`XA_NNLIB_CNN_CONV1D_ST`	Standard 1D Convolution
`XA_NNLIB_CNN_CONV2D_STD`	Standard 2D Convolution
`XA_NNLIB_CNN_CONV2D_DS`	Depthwise Separable 2D Convolution

Table 4-32 describes parameter IDs for parameters supported by CNN. It contains the following columns:

- Parameter ID: Parameter identifier (`param_id`).

- Value type: A pointer (`params`) to a variable of this type is to be passed.

- RW: Indicates whether the parameter can be read (`get`) and/or written (`set`).

- Range: Indicates valid values of the parameter.

- Default: Default value of the parameter

- Description: Brief description of the parameter.

Table 4-32 CNN Specific Parameters

Parameter ID	Value Type	RW	Range	Default	Description
`XA_NNLIB_CNN_KERNEL`	`vect_t []`	*RW*	NA	NA	Kernel shape information, get or set information of the kernel shape expected by the layer
`XA_NNLIB_CNN_BIAS`	`vect_t []`	*RW*	NA	NA	Bias shape information, get or set information of the bias shape expected by the layer
`XA_NNLIB_CNN_INPUT_ SHAPE`	`xa_ nnlib_ shape_ t`	*R*	NA	NA	Input shape information, get information of the input shape expected by the layer.
`XA_NNLIB_CNN_OUTPUT_ SHAPE`	`xa_ nnlib_ shape_ t`	*R*	NA	NA	Output shape information, get information of the output shape produced by layer.

5. Introduction to the Example Testbench

The HiFi 5 NN library is released as .tgz file for linux/makefile based usage and .xws file for Xtensa Xplorer based usage. Both the tgz and xws packages contain various testbenches in addition to the library. These testbenches demonstrate the usage of various APIs, and their performances. The details about building and running the library and testbenches are provided in the following sections.

5.1   Making the Library

If you have source code distribution (that is, .tgz), you must build the NN library before building the testbench. To do so, follow these steps:

1. Go to directory `libxa_nnlib/build`.
2. From the command prompt, enter:
    `xt-make -f makefile clean all install`

The NN library `xa_nnlib.a` is built and copied to the `lib` directory.

To create a debug build, pass DEBUG=1 makefile option in the make command.

The NN Library has TensorFlow Lite Micro double rounding as default option (SINGLE_ROUNDING=0, which is default for TensorFlow Lite Micro as well) and single rounding can be enabled by using makefile option SINGLE_ROUNDING=1.[12]

The NN Library also supports improved optimizations using HiFi5 activation tie instructions for xa_nn_vec_[sigmoid|tanh]_[16|asym8s|sym16s]_[16|asym8s|sym16s] kernels which differs by 1-bit from Tensorflow Lite Micro implementation of corresponding operators, those optimizations are by default enabled for HiFi5 cores which has activation tie instructions, and can be disabled as follows (default is DISABLE_ACT_TIE=0):

` xt-make -f makefile clean all install DISABLE_ACT_TIE=1`

5.1.1 Controlling Library Code Size

The HiFi NN Library code size can be reduced by discarding unused functions at the time of linking.

The library is compiled with the '`-ffunction-sections`' option. With this option, the compiler puts each function in a separate section. This enables the linker to discard unused functions when linking the executable, using the '`-Wl,-gc-sections`' linker option.

Additionally, to remove unused function sections during the library creation, the '`-Wl,-gc-sections`' linker option is enabled while building the testbench. The list of required functions is provided in the linker

---

[12] For XTENSA workspaces, the single-rounding option can be enabled by defining TFLITE_SINGLE_ROUNDING=1 in Build Properties of libxa_nnlib.

script file `build/ldscript_nnlib.txt`. While building the library, the linker discards functions not listed as 'EXTERN' in the linker script file. By appropriately modifying the linker script, the library can be built with only the kernels required for particular application.

5.2 Making the Executable

To build the testbenches, follow these steps:

1. Go to `test/build`.
2. In the command-line prompt, enter:
   `xt-make –f makefile_testbench_sample clean all`

This builds the example testbenches for all the kernels and layers.

The following header files are common and used by all testbenches.

- Testbench header files (`test/include`)

  - `xt_profiler.h`

  - `cmdline_parser.h`

  - `file_io.h`

  - `xt_manage_buffers.h`

To build and execute the example testbenches from xws based release package, check the readme.html files available in the imported example testbench projects.

The following sections describe each low-level kernel and layer testbench.

5.2.1 Controlling Executable Code Size

The code size of the executable binaries can be reduced by discarding unused functions at the time of linking.

The library is compiled with the '-ffunction-sections' option. With this option, the compiler puts each function in a separate section. This enables the linker to discard unused functions when linking the executable, using the '-Wl,-gc-sections' linker option.

5.3 Sample Testbench for Matrix X Vector Multiplication Kernels

The NN library Matrix X Vector Multiplication Kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)
    - `xa_nn_matXvec_testbench.c`

5.3.1 Usage

The NN library Matrix X Vector Multiplication Kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_matXvec_test [options]
```

The following options are available:

Option	Description	Additional Information
-rows	Rows of mat1 (Default=32)	
-cols1	Columns of mat1 and rows of mat2 (Default=32)	Columns of mat1 must be multiple of 4 (except for quantized datatype kernels)
-cols2	Columns of mat2 (Default=32)	Columns of mat2 must be multiple of 4 ((except for quantized datatype kernels)
-row_stride1	Row stride for mat1(Default=32)	
-row_stride2	Row stride for mat2(Default=32)	
-vec_count	Vec count for Time batching (Default=1)	
-acc_shift	Accumulator left shift (Default=0)	
-bias_shift	Bias left shift (Default=0)	
-mat_precision	8, 16, -1(single precision float), -2(half precision float), -3 (asym8u), -13 (asym4s) or -5 (sym8s); (Default=16)	
-inp_precision	8, 16, -1(single precision float), -2(half precision float), -3(asym8u,-8(sym16s) or-4 (asym8s); (Default=16)	
-out_precision	8, 16, 32, 64, -1(single precision float), -2(half precision float), -3(asym8u), -4 (asym8s), -8(sym16s) or -7 (asym16s); (Default=16)	
-bias_precision	8, 16, 64, -1(single precision float), -2(half precision float), 32(asym8); (Default=16)	
-mat1_zero_bias	Matrix1 zero bias for quantized 8-bit, -255 to 0 for asym8u, ignored for sym8s; Default=-128	
-mat2_zero_bias	Matrix2 zero bias for quantized 8-bit, -255 to 0 for asym8u, ignored for sym8s; Default=-128	
-inp1_zero_bias	Input1 zero bias for quantized 8-bit, -255 to 0 for asym8u, -127 to 128 for asym8s, 0 for sym16s, ; Default=-128	
-inp2_zero_bias	Input2 zero bias for quantized 8-bit, -255 to 0 for asym8u, -127 to 128 for asym8s, 0 for sym16s; Default=-128	
-out_multiplier	Output multiplier in Q31 format for quantized 8-bit, 0x0 to 0x7fffffff; Default=0x40000000	

Option	Description	Additional Information
`-out_shift`	Output shift for quantized 8-bit (asym8u and asym8s) 31 to -31; Default=-8	
`-out_zero_bias`	Output zero bias for quantized 8-bit, 0 to 255 for asym8u, -128 to 127 for asym8s, 0 for sym16s; Default=128	
`-out_stride`	Stride for storing the output; Default=1	
`-membank_padding`	0, 1 (Default=1)	
`-frames`	Positive number; (Default=2)	
`-activation`	Sigmoid, tanh (Default= bypass, that is, no activation for output)	
`-write_file`	Set to 1 to write input and output vectors to file; (Default=0)	
`-read_inp_file_name`	Full filename for reading inputs (order - mat1, vec1, mat2, vec2, bias)	
`-read_ref_file_name`	Full filename for reading reference output	
`-write_inp_file_name`	Full filename for writing inputs (order - mat1, vec1, mat2, vec2, bias)	
`-write_out_file_name`	Full filename for writing output	
`-verify`	Verify output against provided reference	0: Disable, 1: Bit exact match (Default=1)
`-batch`	Flag to execute time batching kernels	0: Disable, 1: Enable (Default=0)
`-matmul`	Flag to execute matmul kernels	0: Disable, 1: Enable (Default=0)
`-fc`	Flag to execute fully connected kernels	0: Disable, 1: Enable (Default=0)
`--help, -help, -h`	Prints help	

If no command line arguments are given, the Matrix X Vector Multiplication Kernels sample testbench runs with default values from the paramfile (paramfilesimple_matXvec.txt).

5.4   Sample Testbench for Convolution Kernels

The NN library convolutional kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)
    - `xa_nn_conv_testbench.c`

5.4.1 Usage

The NN Library convolutional kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_conv_test [options]
```

The following options are available:

Option	Description
`-input_height`	Input height (Default=16)
`-input_width`	Input width (Default=16)
`-input_channels`	Input channels (Default=4) Not required for group convolution.
`-kernel_height`	Kernel height (Default=3)
`-kernel_width`	Kernel width (Default=3)
`-kernel_channels`	kernel channels (Default=4)
`-out_channels`	Out channels (Default=4)
`-channels_multiplier`	Channel Multiplier (Default=1)
`-x_stride`	Stride in width dimension (Default=2)
`-y_stride`	Stride in height dimension (Default=2)
`-x_padding`	Left padding in width dimension (Default=2)
`-y_padding`	Top padding in height dimension (Default=2)
`-dilation_height`	Dilation in height dimension (Default=1)
`-dilation_width`	Dilation in width dimension (Default=1)
`-out_height`	Output height (Default=16)
`-out_width`	Output width (Default=16)
`-bias_shift`	Bias left shift (Default=7)
`-groups`	number of groups; Default=1
`-acc_shift`	Accumulator left shift (Default=-7)
`-inp_data_format`	Input data format, 0 (DWH), 1 WHD) Default=1(WHD), ignored for conv2d_std and conv1d_std kernels
`-out_data_format`	Output data format, 0 (DWH), 1 (WHD) Default=0 (DWH)
`-inp_precision`	8, 16, -1(single precision float), -2(half precision float), -3(asymmetric 8-bit unsigned), -8 (sym16s) or -4 (asymmetric 8-bit signed); (Default=16)

Option	Description
`-kernel_precision`	8, 16, -1(single precision float), -2(half precision float), -3(asymmetric 8-bit unsigned) or -5 (symmetric 8-bit signed), -12(Symmetric 4-bit signed); (Default=8)
`-out_precision`	8, 16, -1(single precision float), -2(half precision float), -3(asymmetric 8-bit unsigned), -8 (sym16s) or -4 (asymmetric 8-bit signed); (Default=16)
`-bias_precision`	8, 16, -1(single precision float), -2(half precision float), 32(for quantized 8-bit kernels); (Default=16)
`-input_zero_bias`	Input zero bias for quantized 8-bit, -255 to 0 for asymmetric 8 bit unsigned, -127 to 128 for asymmetric 8 bit signed, ignored for symmetric 16-bit signed; Default=-127
`-kernel_zero_bias`	Kernel zero_bias for quantized 8-bit, -255 to 0 for asymmetric 8 bit unsigned, ignored for symmetric 8 bit signed; Default=-127
`-out_multiplier`	Output multiplier in Q31 format for quantized 8 bit, 0x0 to 0x7fffffff; Default=0x40000000
`-out_shift`	Output shift for quantized 8-bit(asym8u and asym8s), 31 to -31; Default=-8
`-out_zero_bias`	Output zero bias for quantized 8-bit, 0 to 255 for asym8u, -128 to 127 for asym8s, ignored for symmetric 16-bit signed; Default=128
`-frames`	Positive number (Default=2)
`-kernel_name`	conv2d_std, dilated_conv2d_std, conv2d_depth, conv2d_point, conv1d_std, transpose_conv, dilated_conv2d_depth, conv2d_group; Default= : conv2d_std
`-pointwise_profile_only`	Applicable only when kernel_name is conv2d_depth, 0 (print conv2d depthwise and pointwise profile info), 1(print only conv2d pointwise profile info); Default=0
`-write_file`	Set to 1 to write input and output vectors to file; (Default=0)
`-read_inp_file_name`	Full filename for reading inputs (order - input, kernel, bias, (pointwise kernel, pointwise bias for depth separable))
`-read_ref_file_name`	Full filename for reading reference output
`-write_inp_file_name`	Full filename for writing inputs (order - input, kernel, bias, (pointwise kernel, pointwise bias for depth separable))
`-write_out_file_name`	Full filename for writing output
`-verify`	Verify output against provided reference; 0: Disable, 1: Bit exact match (Default=1)
`--help, -help, -h`	Prints help

If no command line arguments are given, the Convolutional Kernels sample testbench runs with default values from the paramfile (paramfilesimple_conv.txt).

5.5   Sample Testbench for Activation Kernels

The NN library activation kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)

    - `xa_nn_activations_testbench.c`

5.5.1 Usage

The NN library activation kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_activation_test [options]
```

The following options are available:

Option	Description
`-num_elements`	Number of elements (Default=32)
`-relu_threshold`	Threshold for relu in Q16.15 (Default= 32768, that is, =1 in Q16.15)
`-inp_precision`	8,16, 32, -1(single precision float), -3(asym8u) -4 (asym8s), -7 (asym16s) or -8 (sym16s); (Default=32)
`-out_precision`	8,16, 32, -1(single precision float), -3(asym8u) or -4 (asym8s), -7 (asym16s) or -8 (sym16s); (Default=32)
`-integer_bits`	Number of integer bits in input for tanh_16_16(0 to 6) (Default = 3)
`-frames`	Positive number (Default=2)
`-activation`	Sigmoid, tanh, relu, relu_std, relu1, relu6, activation_min_max, softmax, hard_swish, prelu or leaky_relu (Default= sigmoid)
`-write_file`	Set to 1 to write input and output vectors to file; (Default=0)
`-read_inp_file_name`	Full filename for reading input
`-read_ref_file_name`	Full filename for reading reference output
`-write_inp_file_name`	Full filename for writing input
`-write_out_file_name`	Full filename for writing output
`-verify`	Verify output against provided reference; 0: Disable, 1: Bit exact match (Default=1)
Quantized 8-bit specific parameters	
`-diffmin`	Diffmin; Default=-15
`-input_left_shift`	Input_left_shift; Default=27
`-input_multiplier`	Input_multiplier; Default=2060158080
`-activation_max`	asym8u/asym8s/16/8 input data activation max; Default=0
`-activation_min`	asym8u/asym8s/16/8 input data activation min; Default=0
`-activation_max_f32`	Float input data activation max (Default=0)

Option	Description
`-activation_min_f32`	Float input data activation min (Default=0)
`-input_range_radius`	sigmoid_asym8u/s input parameter; Default=128
`-zero_point`	sigmoid_asym8u/s input parameter; Default=0
`-input_zero_bias`	Zero bias value for input (Default =0)
`-alpha_zero_bias`	Prelu parameter - Zero bias value for alpha Default=0
`-alpha_multiplier`	Leaky Relu and Prelu parameter - Multiplier value for alpha Default=0x40000000
`-alpha_shift`	Leaky Relu and Prelu parameter - Shift value for alpha Default=0
`-reluish_multiplier`	Hard Swish parameter - Multiplier value for relu scale Default=0x40000000
`-reluish_shift`	Hard Swish parameter - Shift value for relu scale Default=0
`-out_multiplier`	Multiplier value for output Default=0x40000000
`-out_shift`	Shift value for output Default=0
`-out_zero_bias`	Zero bias value for output Default=0
`--help, -help, -h`	Prints help

If no command line arguments are given, the Activation Kernels sample testbench runs with default values from the paramfile (paramfilesimple_activations.txt).

5.6   Sample Testbench for Pooling Kernels

The NN library pooling kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)
  - `xa_nn_pool_testbench.c`

5.6.1 Usage

The NN library pooling kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_pool_test [options]
```

The following options are available:

Option	Description
`-inp_data_format`	Input data format, 0 (SHAPE_CUBE_DWH_T), 1 SHAPE_CUBE_WHD_T); (Default=1 (SHAPE_CUBE_WHD_T))

Option	Description
`-out_data_format`	Output data format, 0 (SHAPE_CUBE_DWH_T), 1 SHAPE_CUBE_WHD_T); (Default=1 (SHAPE_CUBE_WHD_T))
`-input_height`	Input height (Default=16)
`-input_width`	Input width (Default=16)
`-input_channels`	Input channels (Default=4)
`-kernel_height`	Kernel height (Default=3)
`-kernel_width`	Kernel width (Default=3)
`-x_stride`	Stride in width dimension (Default=2)
`-y_stride`	Stride in height dimension (Default=2)
`-x_padding`	Left padding in width dimension (Default=2)
`-y_padding`	Top padding in height dimension (Default=2)
`-out_height`	Output height (Default=16)
`-out_width`	Output width (Default=16)
`-acc_shift`	Accumulator left shift (Default=-7)
`-inp_precision`	8, 16, -1(single precision float), -3(asym8); (Default=16)
`-out_precision`	8, 16, -1(single precision float), -3(asym8); (Default=16)
`-frames`	Positive number (Default=2)
`-kernel_name`	avgpool, maxpool (Default= avgpool)
`-write_file`	set to 1 to write input and output vectors to file; (Default=0)
`-read_inp_file_name`	Full filename for reading inputs (order - inp)
`-read_ref_file_name`	Full filename for reading reference output
`-write_inp_file_name`	Full filename for writing inputs (order - inp)
`-write_out_file_name`	Full filename for writing output
`-verify`	Verify output against provided reference; 0: Disable, 1: Bit exact match (Default=1)
`--help, -help, -h`	Prints help

If no command line arguments are given, the Pooling Kernels sample testbench runs with default values from the paramfile (paramfilesimple_pool.txt).

5.7   Sample Testbench for Basic Operations Kernels

The NN library basic kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)

  - `xa_nn_basic_testbench.c`

5.7.1 Usage

The NN library basic kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_basic_test [options]
```

The following options are available:

Option	Description
-io_length	Input/output vector length; Default=1024
-num_inp_dims	Number of input dimensions(Default =4)
-num_axis_dims	Number of axis dimensions(Default =4)
-num_output_dims	Number of output dimensions(Default =4)
-inp_precision	8, 16, -3 (asym8u), -1 (single prec float), -4(asym8s), -7(asym16s) , -8(sym16s), 1(bool); Default=-1
-out_precision	8, 16, -3 (asym8u), -1 (single prec float), -4(asym8s) , -7(asym16s) , -8(sym16s), 1(bool), -10(asym32s); Default=-1
-vec_count	Number of input vectors; Default=1
-frames	Positive number; Default=2
-kernel_name	elm_add, elm_sub, elm_mul, elm_floor, dot_prod, elm_min and elm_max, elm_equal, elm_notequal, elm_greater, elm_greaterequal, elm_less, elm_lessequal, elm_logicaland, elm_logicalor, elm_logicalnot, reduce_max_4D, reduce_mean_4D, elm_min_4D_Bcast, elm_max_4D_Bcast, elm_sine, elm_cosine, elm_logn, elm_abs, elm_ceil, elm_round, elm_neg, elm_square, elm_sqrt, elm_rsqrt, broadcast, elm_requantize, elm_dequantize, elm_quantize, memmove,memset, elm_add_broadcast_4D, elm_sub_broadcast_4D, elm_mul_broadcast_4D, elm_squared_diff_broadcast_4D; Default=elm_add
-write_file	Set to 1 to write input and output vectors to file; Default=0
-read_inp1_file_name	Full filename for reading inputs (order - inp)
-read_inp2_file_name	Full filename for reading inputs (order - inp)
-read_ref_file_name	Full filename for reading reference output
-write_inp1_file_name	Full filename for writing inputs (order - inp)
-write_inp2_file_name	Full filename for writing inputs (order - inp)
-write_out_file_name	Full filename for writing output
-verify	Verify output against provided reference; 0: Disable, 1: Bit exact match; Default=1
-read_inp_shape_str	Takes the input shape dimensions(space ' ' separated) as a string
-read_inp1_shape_str	Takes the input1 shape dimensions(space ' ' separated) as a string
-read_inp2_shape_str	Takes the input2 shape dimensions(space ' ' separated) as a string
-read_out_shape_str	Takes the output shape dimensions(space ' ' separated) as a string

Option	Description
-read_axis_data_str	Takes the axis data (space ' ' separated) as a string
Broadcast specific parameters	
-input1_numElements	Number of elements in input (order - inp)
-input2_numElements	Number of elements in input(order – inp)
-input1_strides	Input strides (order – inp)
-input2_strides	Input strides (order – inp)
Quantized data types specific parameters	
-output_zero_bias	Output zero bias; Default=127
-output_left_shift	Output_left_shift; Default=0
-output_multiplier	Output_multiplier; Default=0x7fff
-output_activation_min	Output_activation_min; Default=0
-output_activation_max	Output_activation_max; Default = 225
-input1_zero_bias	Input1 zero bias; Default=-127
-input1_left_shift	Input1 left shift; Default=0
-input1_multiplier	Input1 multiplier; Default=0x7fff
-input2_zero_bias	Input2 zero bias; Default=-127
-input2_left_shift	Input2 left shift; Default=0
-input2_multiplier	Input2 multiplier; Default=0x7fff
-left_shift	Global left shift; Default=0
-input1_scale	Input scale; Default=0.5
-val_memset	input_memset(Float value. Needed in memset operation); Default=0.0
--help, -help, -h	Prints help

If no command line arguments are given, the Basic Kernels sample testbench runs with default values from the paramfile (paramfilesimple_basic.txt).

5.8   Sample Testbench for Normalization Kernels

The NN library Normalization Kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (test/src)
  - xa_nn_norm_testbench.c

5.8.1 Usage

The NN library Normalization Kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_norm_test [options]
```

The following options are available:

Option	Description
-num_elms	Number of elements; Default=256
-inp_precision	-4(asym8s) and -1(float32); Default=16
-out_precision	-4(asym8s) and -1(float32); Default=16
-frames	Positive number; Default=2
-kernel_name	L2_norm; Default=l2_norm
-zero_point	Input Zero point; Default = 0
-write_file	Set to 1 to write input and output vectors to file; Default=0
-read_inp_file_name	Full filename for reading inputs (order - inp)
-read_ref_file_name	Full filename for reading reference output
-write_inp_file_name	Full filename for writing inputs (order - inp)
-write_out_file_name	Full filename for writing output
-verify	Verify output against provided reference; 0: Disable, 1: Bit exact match; Default=1
--help, -help, -h	Prints help

If no command line arguments are given, the Normalization Kernels sample testbench runs with default values from the paramfile (paramfilesimple_norm.txt).

5.9 Sample Testbench for Reorg Kernels

The NN library reorg kernels are provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (test/src)
  - xa_nn_reorg_testbench.c

5.9.1 Usage

The NN library reorg kernels executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_reorg_test [options]
```

The following options are available:

Option	Description
`-inp_data_format`	Data format of input and output, 0 for nhwc; Default=0
`-num_inp_dims`	Number of input dimensions; Default=4
`-num_pad_dims`	Number of pad dimensions; Default=2
`-num_out_dims`	Number of output dimensions; Default=4
`-pad_value`	Input to be padded with this pad value; Default=0
`-permute_vec`	Permutation values of dimensions for transpose
`-input_height`	Input height; Default=16
`-input_width`	Input width; Default=16
`-input_channels`	Input channels; Default=16
`-block_size`	Block size; Default=2
`-out_height`	Output height; Default=16
`-out_width`	Output width; Default=16
`-out_channels`	Output channels; Default=4
`-inp_precision`	8, 16, 32; Default=8
`-out_precision`	8, 16, 32; Default=8
`-frames`	Positive number; Default=2
`-kernel_name`	depth_to_space, space_to_depth, pad, batch_to_space_nd, space_to_batch_nd, strided_slice, transpose; Default=depth_to_space
`-write_file`	Set to 1 to write input and output vectors to file; Default=0
`-read_inp_file_name`	Full filename for reading inputs (order - inp)
`-read_ref_file_name`	Full filename for reading reference output
`-write_inp_file_name`	Full filename for writing inputs (order - inp)
`-write_out_file_name`	Full filename for writing output
`-verify`	Verify output against provided reference; 0
`-inp_shape`	Takes the input shape dimensions (num_inp_dims values space ' ' separated)
`-pad_shape`	Takes the pad shape dimensions (num_pad_dims values space ' ' separated)
`-out_shape`	Takes the output shape dimensions (num_out_dims values space ' ' separated)
`-pad_values`	Takes the pad values(prod(pad_shape) values space ' ' separated)
`-block_sizes`	Takes the block sizes ((num_inp_dims-2) values space ' ' separated) for batch_to_space_nd and space_to_batch_nd kernels
`-crop_or_pad_sizes`	Takes the crop sizes for batch_to_space_nd or pad sizes for space_to_batch_nd (2*(num_inp_dims-2) values space ' ' separated)
`--help, -help, -h`	Prints help.

If no command line arguments are given, the Reorg Kernels sample testbench runs with default values from the paramfile (paramfilesimple_reorg.txt).

5.10 Sample Testbench for GRU Layer

The NN library GRU layer is provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)

    - `xa_nn_gru_testbench.c`

5.10.1 Usage

The NN library GRU executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_gru_test [options]
```

The following options are available:

Option	Description	Additional Information
`--in_feats`	Input length (Default=256)	Range: 4-2048 **Note:** Input length must be multiple of 4
`--out_feats`	Output length (Default=256)	Range: 4-2048 **Note:** Output length must be multiple of 4
`--membank_padding`	Memory bank padding (Default=1)	Must be 0 or 1
`--split bias`	Split Bias option (Default=0)	Must be 0 or 1
`--mat_prec`	Coefficient precision (Default=16)	Must be 8 or 16
`--vec_prec`	Input precision (Default=16)	Must be 16
`--verify`	Verify output against ref output (Default=1)	Supported values: 0: Disable, 1: Enable
`--input_file`	Input file name	
`--filter_path`	Path where file containing filter are stored	
`--output_file`	File to which output is written	
`--prev_h_file`	File containing context data	
`--ref_file`	File which has ref output	
`--help, -help, -h`	Prints help	

If no command line arguments are given, the GRU sample testbench runs with default values from the paramfile (paramfilesimple_gru.txt).

5.11 Sample Testbench for LSTM Layer

The NN library LSTM layer is provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)
  - `xa_nn_lstm_testbench.c`

5.11.1 Usage

The NN library LSTM executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_lstm_test [options]
```

The following options are available:

Option	Description	Additional Information
`--in_feats`	Input length (Default=256)	Range: 4-2048 **Note:** Input length must be multiple of 4
`--out_feats`	Output length (Default=256)	Range: 4-2048 **Note:** Output length must be multiple of 4
`--membank_padding`	Memory bank padding (Default=1)	Must be 0 or 1
`--mat_prec`	Coefficient precision (Default=16)	Must be 8 or 16
`--vec_prec`	Input precision (Default=16)	Must be 16
`--verify`	Verify output against ref output (Default=1)	Supported values: 0: Disable, 1: Enable
`--input_file`	File containing input shape	
`--filter_path`	Path where file containing filter are stored	
`--output_file`	File to which output is written	
`--output_cell_file`	File to which cell output is written	
`--prev_h_file`	File containing context (previous output) data	
`--prev_c_file`	File containing context (previous cell state) data	
`--ref_file`	File which has ref output	
`--ref_cell_file`	File which has ref cell output	
`--help, -help, -h`	Prints help	

If no command line arguments are given, the LSTM sample testbench runs with default values from the paramfile (paramfilesimple_lstm.txt).

5.12 Sample Testbench for CNN Layer

The NN library CNN layer is provided with a sample testbench application. The supplied testbench consists of the following files:

- Testbench source files (`test/src`)

  - `xa_nn_cnn_testbench.c`

5.12.1 Usage

The NN Library CNN executable can be run with command-line options as follows.

```
$ xt-run [--mem_model] [--turbo] xa_nn_cnn_test [options]
```

The following options are available:

Option	Description
`-input_height`	Input height (Default=16)
`-input_width`	Input width (Default=16)
`-input_channels`	Input channels (Default=4)
`-kernel_height`	Kernel height (Default=3)
`-kernel_width`	Kernel width (Default=3)
`-out_channels`	Out channels (Default=4)
`-channels_multiplier`	Channel Multiplier (Default=1)
`-x_stride`	Stride in width dimension (Default=2)
`-y_stride`	Stride in height dimension (Default=2)
`-x_padding`	Left padding in width dimension (Default=2)
`-y_padding`	Top padding in height dimension (Default=2)
`-out_height`	Output height (Default=16)
`-out_width`	Output width (Default=16)
`-bias_shift`	Bias shift (Default=7)
`-acc_shift`	Accumulator shift (Default=-7)
`-out_data_format`	Output data format, 0 (SHAPE_CUBE_DWH_T), 1 (SHAPE_CUBE_WHD_T); (Default=0)
`-inp_precision`	8, 16, -1(single precision float); (Default=16)
`-kernel_precision`	8, 16, -1(single precision float); (Default=8)
`-out_precision`	8, 16, -1(single precision float); (Default=16)
`-bias_precision`	8, 16, -1(single precision float); (Default=16)
`-frames`	Positive number; (Default=2)
`-kernel_name`	conv2d_std, conv2d_depth, conv1d_std; (Default= conv2d_std)

Option	Description
`-write_file`	Set to 1 to write input and output vectors to file; (Default=0)
`-read_inp_file_name`	Full filename for reading inputs (order - input, kernel, bias, (pointwise kernel, pointwise bias for depth separable))
`-read_ref_file_name`	Full filename for reading reference output
`-write_inp_file_name`	Full filename for writing inputs (order - input, kernel, bias, (pointwise kernel, pointwise bias for depth separable))
`-write_out_file_name`	Full filename for writing output
`-verify`	Verify output against provided reference; 0: Disable, 1: Bit exact match; Default=1
`--help, -help, -h`	Prints help

If no command line arguments are given, the CNN sample testbench runs with default values from the paramfile (paramfilesimple_cnn.txt).

6.References

[1]   Reference Wiki page for GRU. https://en.wikipedia.org/wiki/Gated_recurrent_unit

[2]   TF Micro Lite speech recognition example: https://github.com/tensorflow/tensorflow/tree/r2.3/tensorflow/lite/micro/examples/micro_speech

[3]   TensorFlow Lite for Microcontrollers

[4]   TensorFlow XLA Documentation: https://www.tensorflow.org/xla/broadcasting
NumPy Theory: https://numpy.org/devdocs/user/basics.broadcasting.html
General Broadcasting syntax: https://www.tensorflow.org/guide/tensor#broadcasting

[5]   'strides' as defined in the structure 'NDArrayDesc' at https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/common.h