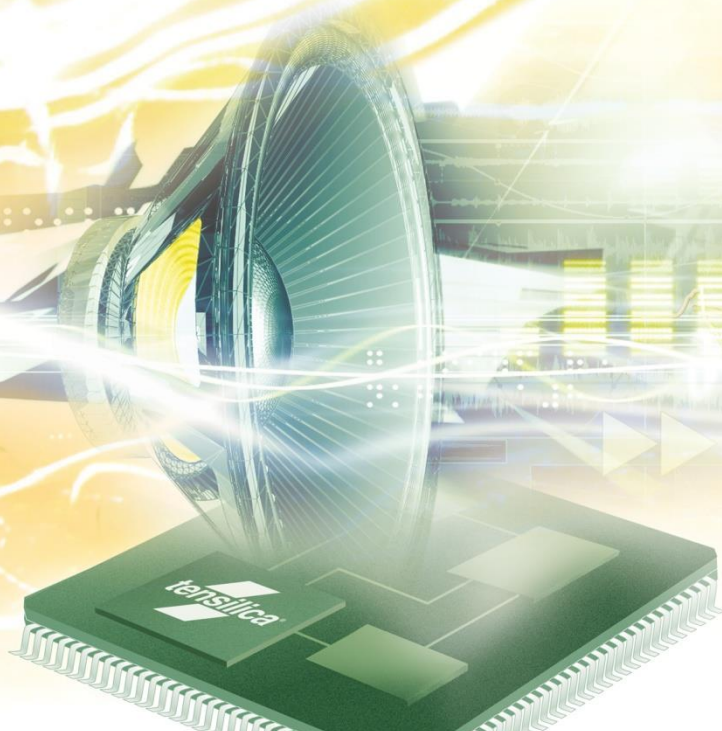




Fusion G3 Neural Network Library

Programmer's Guide - API



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

Copyright © 2024 Cadence Design Systems, Inc.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement;
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration; and
5. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third-party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from the use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Version: 1.1

Last Updated: December 2024

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

Contents

1.	Introduction to the Fusion G3 NN library	1
1.1	Fusion G3 NN Library Specification.....	1
1.1.1	Low-Level Kernels	1
1.1.2	Support for ExecuTorch Operators.....	2
2.	Fusion G3 NN Library – Low-Level Kernels	3
2.1	Activation Kernels	3
2.1.1	Sigmoid.....	3
2.1.2	Tanh.....	4
2.1.3	Softmax	5
2.2	Basic Operations and Miscellaneous Kernels	7
2.2.1	Elementwise Quantize Kernels	7
2.2.2	Elementwise Dequantize Kernels.....	11
2.2.3	Basic Vector math operation Kernels	15
2.2.4	Elementwise Comparison Kernels	19
2.2.5	Basic Kernels with 5D Broadcasting	20
2.2.6	Mean.....	26
2.3	Normalization Kernels.....	29
2.3.1	Layer Normalization Kernel	29
2.4	Reorg Kernels	31
2.4.1	Slice	31
2.4.2	Permute	34
2.4.3	Cat	37
3.	Making the library	39

Nomenclature

Below naming convention is used while defining the APIs for kernels.

Naming convention	Description
f32	Single precision floating point
8	Signed 8-bit
8u	Unsigned 8-bit
16	Signed 16-bit
16u	Unsigned 16-bit
32	Signed 32-bit
32u	Unsigned 32-bit
Sym	Symmetric
Asym	Asymmetric
Axis	Dimension along which the computations will be performed
WORD32	32-bit signed integer
WORD16	16-bit signed half word
WORD8	Signed byte
UWORD32	32-bit unsigned integer
UWORD16	16-bit unsigned half word
UWORD8	Unsigned byte
FLOAT32	Single precision floating point

Document Change History

Version	Changes
1.0	<ul style="list-style-type: none">■ Initial version
1.1	<ul style="list-style-type: none">■ Updated Mean kernel API to fit computation of mean for multiple axis

1. Introduction to the Fusion G3 NN library

The Fusion G3 Neural Network (NN) Library is an optimized implementation of various low-level NN kernels. The low-level NN kernels are the basic building blocks for operators and networks in neural network frameworks with a generic and simple interface.

Note This version of the library supports Fusion G3 DSPs with the SP-VFPU (Single Precision Vector Floating Point Unit).

Note This version of the Fusion G3 NN Library is tested with the xt-clang/xt-clang++ compilers using Xtensa Software Tools from RI-2022.10 release.

The Fusion G3 NN Library package includes the source code containing low-level kernel implementations.

This document covers information related to low level kernel APIs and information required to create the Fusion G3 NN Library. Section 2 provides details of low-level NN kernel APIs. Section 3 provides details of creating the Fusion G3 NN library.

1.1 Fusion G3 NN Library Specification

The current version of the Fusion G3 NN Library provides the following Fusion-optimized low-level kernel implementations.

1.1.1 Low-Level Kernels

- Activation kernels
- Basic operations kernels
 - Quantize and dequantize kernels
 - Basic vector math operators
 - Broadcast kernels
- Normalization kernels
- Reorg kernels

These kernels support fixed point 8-bit, 16-bit, 32-bit, single precision floating point (float32/f32) data types for input and output. Please note that not all the kernels support all the datatypes specified here. The details of what datatypes are supported by each of the kernel is specified in Section 3. float32 is IEEE-754 compliant data types.

1.1.2 Support for Executorch Operators

The Fusion G3 NN Library low-level kernels can be used to implement the following operators of Executorch. 4-bit, 8-bit, 16-bit, 32-bit represents signed or unsigned datatypes. Kernels shown as supporting these datatypes might not support both signed and unsigned representations. Please refer Section 3 for details of the datatypes supported for each of the kernel.

No.	Operator	Float32 Datatype Support	32-bit	16-bit	8-bit	4-bit
1	Add	Yes	Yes	No	No	No
2	sub	Yes	Yes	No	No	No
3	mul	Yes	Yes	No	No	No
4	div	Yes	Yes	No	No	No
5	quantize	No	No	Yes	Yes	yes
6	dequantize	No	No	Yes	Yes	Yes
7	Softmax	Yes	No	No	No	No
8	Layernorm	Yes	No	No	No	No
9	Permute	No	Yes	Yes	Yes	No
10	Exp	Yes	No	No	No	No
11	cat	No	Yes	Yes	Yes	No
12	Slice	No	Yes	Yes	Yes	No
13	Clamp	Yes	No	Yes	Yes	No
14	sigmoid	Yes	No	No	No	No
15	sqrt	Yes	No	No	No	No
16	rsqrt	Yes	No	No	No	No
17	Tanh	Yes	No	No	No	No
18	Mean	Yes	No	No	No	No
19	Where	Yes	No	No	No	No
20	It	Yes	No	No	No	No
21	Transpose	No	Yes	Yes	Yes	No

2. Fusion G3 NN Library – Low-Level Kernels

This section explains the APIs of low-level kernels which will be part of the NN library on Fusion G3 DSP. These kernels will be developed using Executorch as reference implementation. All the low-level kernels have a generic and simple interface.

2.1 Activation Kernels

2.1.1 Sigmoid

Description

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

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

[p]: Input precision in bits

[q]: Output precision in bits

Precision

Type	Description
f32_f32	float32 input, float32 output

Algorithm

$$y_n = \frac{1}{1 + \exp(-x_n)}, \quad n = 0, \dots, \text{vec_length} - 1$$

Prototype

```
WORD32 xa_nn_sigmoid_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,          WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const	p_inp	vec_length	Input vector

Float32 *			
WORD32	vec_length	1	Length of input vector
Output			
Float32 *	p_out	vec_length	Output vector

Returns

0: no error

-1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_inp, p_out	Must not overlap
	Cannot be NULL
vec_length	Greater than 0

2.1.2 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`.

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

[p]: Input precision in bits

[q]: Output precision in bits

Precision

Type	Description
f32_f32	float32 input, float32 output

Algorithm

$$y_n = \tanh(x_n), \quad n = 0, \dots, \text{vec_length} - 1$$

Prototype

```
WORD32 xa_nn_tanh_f32_f32
(Float32 * p_out,          const Float32 * p_inp,          WORD32 vec_length);
```

Arguments

Type	Name	Size	Description
Input			
const	p_inp	vec_length	Input vector

Type	Name	Size	Description
FLOAT32 *			
WORD32	vec_length	1	Length of input vector
Output			
FLOAT32 *	p_out	vec_length	Output vector

Returns

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

Restrictions

Arguments	Restrictions
p_inp, p_out	Must not overlap Cannot be NULL
vec_length	Greater than 0

2.1.3 Softmax

Description

The Softmax kernels compute the softmax (normalized exponential function) of input vector x and give output vector as $y = \text{softmax}(x)$. Both the input and output vectors have the same dimensions and size.

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

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

Precision

Type	Description
f32_f32	float32 input, float32 output

Algorithm

$$y_n = \frac{\exp(x_n - \max)}{\sum_k \exp(x_k - \max)}, \quad \text{axis} = \text{NULL}, \quad n = \prod_{i=0}^{i=\text{num_inp_dims}-1} p_inp_shape[i], \quad k = \prod_{i=0}^{i=\text{num_inp_dims}-1} p_inp_shape[i]$$

When axis is not NULL, below is the algorithm used to compute softmax.

```

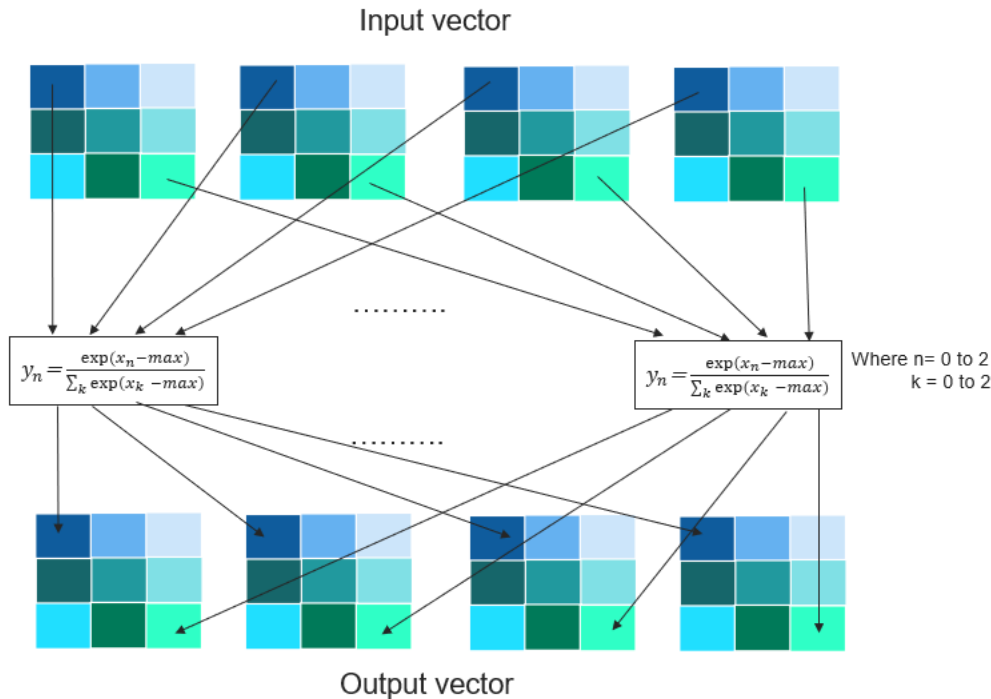
for j=0 to  $\prod_{i=0}^{axis-1} p\_inp\_shape[i]$ 
  outer_stride =  $j * \prod_{i=axis}^{num\_inp\_dims-1} p\_inp\_shape[i]$ 
  for k=0 to  $\prod_{i=axis+1}^{num\_inp\_dims-1} p\_inp\_shape[i]$ 
    max_value = 0;
    accum = 0;
    end =  $\prod_{i=axis}^{num\_inp\_dims-1} p\_inp\_shape[i]$ 
    stride =  $\prod_{i=axis+1}^{num\_inp\_dims-1} p\_inp\_shape[i]$ 
    for m = (outer_stride + k) to end with a stride of stride
      max_value = max(max_value, input[m])

    for m = (outer_stride + k) to end with a stride of stride
      exp[m] = exp(input[m]-max_value)
      accum += exp[m]

    for m = (outer_stride + k) to end with a stride of stride
      out[i] = exp[m]/accum

```

For ex: For an input dimension of [20][12][4][3][3] with the axis set to 2, softmax is computed for groups of 4 elements each, where the dimension along axis 2 is 4. The softmax operation is performed separately for each group of elements in the last three dimensions (4x3x3) as illustrated in the figure below. This process is repeated across the top two dimensions of the input, which are dimensions 20 and 12.



Prototype

```
WORD32 xa_nn_softmax_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,          const WORD32 * p_inp_shape,
WORD32 num_inp_dims,      WORD32 *p_axis);
```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Input vector
const WORD32 *	p_inp_shape	num_inp_dims	Input shape
WORD32	num_inp_dims	1	Number of dimensions in the input
WORD32 *	p_axis	1	A dimension along which Softmax will be computed
Output			
FLOAT32 *	p_out	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Output vector.

Returns

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

Restrictions

Arguments	Restrictions
p_inp, p_out	Must not overlap
	Cannot be NULL
p_inp_shape	Cannot be NULL
	Dimension values greater than 0
num_inp_dims	Greater than 0
*p_axis	[0, num_inp_dims)

2.2 Basic Operations and Miscellaneous Kernels

2.2.1 Elementwise Quantize Kernels

Description

The Elementwise Quantize kernels perform the quantization operation of the input vector elements to get the output vector. The kernels are developed in reference to the Quantize operator implementation in the ExecuTorch. The Quantize kernels support both symmetric and asymmetric quantization for 4-bit (left

justified in byte), 8-bit and 16-bit datatypes (includes signed and unsigned datatypes). Function variants available are `xa_nn_elm_quantize_[p]_[q]`, where:

[p]: Input precision

[q]: Output precision

Precision

Type	Description
f32_asym4	Asymmetric quantization - single precision float input, signed 4-bit output (left justified in byte)
f32_asym8	Asymmetric quantization - single precision float input, signed 8-bit output
f32_asym16	Asymmetric quantization - single precision float input, signed 16-bit output
f32_sym4	Symmetric quantization - single precision float input, signed 4-bit output (left justified in byte)
f32_sym8	Symmetric quantization - single precision float input, signed 8-bit output
f32_sym16	Symmetric quantization - single precision float input, signed 16-bit output
f32_asym4u	Asymmetric quantization - single precision float input, unsigned 4-bit output (left justified in byte)
f32_asym8u	Asymmetric quantization - single precision float input, unsigned 8-bit output
f32_asym16u	Asymmetric quantization - single precision float input, unsigned 16-bit output
f32_sym4u	Symmetric quantization - single precision float input, unsigned 4-bit output (left justified in byte)
f32_sym8u	Symmetric quantization - single precision float input, unsigned 8-bit output
f32_sym16u	Symmetric quantization - single precision float input, unsigned 16-bit output

Algorithm

Tensor quantization

Asymmetric quantization

for $itr = 0:(num_elm-1)$

$out_value = (int)(p_inp[itr] / out_scale) + out_zero_bias ;$

$p_{out[itr]} = \min(\max(out_value, quant_min), quant_max) ;$

Symmetric quantization

for $itr = 0:(num_elm-1)$

$out_value = (int)(p_inp[itr] / out_scale)$

$p_out[itr] = \min(\max(out_value, quant_min), quant_max)$

In both the above cases, for 4-bit quantized output

for $itr = 0:(num_elm-1)$

$p_out[itr] = p_out[itr] \ll 4$

For channel/axis specific quantization – assuming the axis to be 2

Asymmetric quantization

```
for itr = 0:(p_inp_shape[axis])
    out_value = (int)(p_inp[d0][d1][itr][d3][d4] / out_scale[itr]) + out_zero_bias[itr];
    p_out[d0][d1][itr][d3][d4] = min(max(out_value, quant_min), quant_max);
```

Symmetric quantization

```
for itr = 0:(p_inp_shape[axis])
    out_value = (int)(p_inp[d0][d1][itr][d3][d4] / out_scale[itr]) ;
    p_out[d0][d1][itr][d3][d4] = min(max(out_value, quant_min), quant_max);
```

In both the above cases, for 4-bit quantized output

```
p_out[d0][d1][d2][d3][d4] = p_out[d0][d1][d2][d3][d4] << 4;
```

Where

```
d0 = 0 to p_inp_shape[0]-1
d1 = 0 to p_inp_shape[1]-1
d2 = 0 to p_inp_shape[2]-1
d3 = 0 to p_inp_shape[3]-1
d4 = 0 to p_inp_shape[4]-1
```

Both tensor quantization and channel based quantization is performed using the same kernel function.

Prototype

```
WORD32 xa_nn_elm_quantize_f32_asym4
(WORD8 *__restrict__ p_out,      const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 *p_out_zero_bias,      WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_asym8
(WORD8 *__restrict__ p_out,      const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 *p_out_zero_bias,      WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_asym16
(WORD16 *__restrict__ p_out,     const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 *p_out_zero_bias,      WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym4
(WORD8 *__restrict__ p_out,      const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 quant_min,          WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym8
(WORD8 *__restrict__ p_out,      const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 quant_min,          WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym16
(WORD16 *__restrict__ p_out,     const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims,          WORD32 *p_axis,
FLOAT32 *p_out_scale,           WORD32 quant_min,          WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_asym4u
(UWORD8 *__restrict__ p_out,     const FLOAT32 *__restrict__ p_inp,
```

```

const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 *p_out_zero_bias, WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_asym8u
(UWORD8 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 *p_out_zero_bias, WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_asym16u
(UWORD16 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 *p_out_zero_bias, WORD32 quant_min,
WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym4u
(UWORD8 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 quant_min, WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym8u
(UWORD8 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 quant_min, WORD32 quant_max);
WORD32 xa_nn_elm_quantize_f32_sym16u
(UWORD16 *__restrict__ p_out, const FLOAT32 *__restrict__ p_inp,
const WORD32 *const p_inp_shape, WORD32 num_inp_dims, WORD32 *p_axis,
FLOAT32 *p_out_scale, WORD32 quant_min, WORD32 quant_max);

```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp	$\prod_{i=0}^{num_inp_dims-1} p_inp_shape[i]$	Input vector
Const WORD32 *const	P_inp_shape	num_inp_dims	Shape of the input vector
WORD32	num_inp_dims	1	Number of input dimensions
WORD32 *	p_axis	1	A dimension along which quantization will be computed
FLOAT32 *	p_out_scale	1 Or P_inp_shape[*p_axis]	Scale of output
WORD32 *	p_out_zero_bias	1 Or P_inp_shape[*p_axis]	Zero offset of output.
WORD32	quant_min	1	Minimum value used to limit the output
WORD32	quant_max	1	Maximum value used to limit the output
Output			
WORD8 * WORD16 * UWORD8 * UWORD16 *	p_out	$\prod_{i=0}^{num_inp_dims-1} p_inp_shape[i]$	Output vector

Returns

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

Restrictions:

Arguments	Restrictions
p_inp, p_out	Aligned on (size of one element of datatype)-byte boundary Cannot be NULL Must not overlap
num_inp_dims	Greater than 0 and less than or equal 5
*p_out_scale	Not equal to zero and finite single precision float value
*p_out_zero_bias	{-8,.....,7} for out type signed 4-bit {-128,....., 127} for out type signed 8-bit {-32768,...., 32767} for out type signed 16-bit {0,.....,15} for out type unsigned 4-bit {0,....., 255} for out type unsigned 8-bit {0,...., 65535} for out type unsigned 16-bit
*p_axis	[0, num_inp_dims)
quant_min	Greater or equal to -8 for out type signed 4-bit Greater or equal to -128 for out type signed 8-bit Greater or equal to -32768 for out type signed 16-bit Greater or equal to 0 for out type unsigned 4-bit, unsigned 8-bit, unsigned 16-bit quant_min should be less than quant_max
quant_max	Less or equal to 7 for out type signed 4-bit Less or equal to 127 for out type signed 8-bit Less or equal to 32767 for out type signed 16-bit Less or equal to 15 for out type unsigned 4-bit Less or equal to 255 for out type unsigned 8-bit Less or equal to 65535 for out type unsigned 16-bit Quant_max should be greater or equal to quant_min

2.2.2 Elementwise Dequantize Kernels

Description

The Elementwise Dequantize kernels perform the dequantization operation of the input vector elements to get the output vector. The dequantize kernels support both symmetric and asymmetric dequantization for 4-bit (left justified in byte), 8-bit and 16-bit datatypes (includes signed and unsigned datatypes).

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

[p]: Input precision

[q]: Output precision

Precision

Type	Description
asym4_f32	Asymmetric dequantization - signed 4-bit input (left justified in byte), single precision float output
asym8_f32	Asymmetric dequantization - signed 8-bit input, single precision float output
asym16_f32	Asymmetric dequantization - signed 16-bit input, single precision float output
sym4_f32	Symmetric dequantization - signed 4-bit input (left justified in byte), single precision float output
sym8_f32	Symmetric dequantization - signed 8-bit input, single precision float output
sym16_f32	Symmetric dequantization - signed 16-bit input, single precision float output
asym4u_f32	Asymmetric dequantization - unsigned 4-bit input (left justified in byte), single precision float output
asym8u_f32	Asymmetric dequantization - unsigned 8-bit input, single precision float output
asym16u_f32	Asymmetric dequantization - unsigned 16-bit input, single precision float output
sym4u_f32	Symmetric dequantization - unsigned 4-bit input (left justified in byte), single precision float output
sym8u_f32	Symmetric dequantization - unsigned 8-bit input, single precision float output
sym16u_f32	Symmetric dequantization - unsigned 16-bit input, single precision float output

Algorithm

Tensor dequantization

In both the below cases, for 4-bit quantized input

```
for itr = 0:(num_elm-1)
    p_inp[itr] = p_inp[itr] >> 4;
```

Asymmetric dequantization

```
for itr = 0:(num_elm-1)
    p_out[itr] = (float)((p_inp[itr] - inp_zero_bias) * inp_scale))
```

Symmetric dequantization

```
for itr = 0:(num_elm-1)
    p_out[itr] = (float)((p_inp[itr]) * inp_scale))
```

For channel/axis specific dequantization – assuming the axis to be 2

In both the below cases, for 4-bit quantized input

```
p_inp[d0][d1][d2][d3][d4] = p_inp[d0][d1][d2][d3][d4] >> 4;
```

Asymmetric quantization

```
for itr = 0:(p_inp_shape[axis])
```

$$p_out[d0][d1][itr][d3][d4] = (float)((p_inp[d0][d1][itr][d3][d4] - inp_zero_bias[itr]) * inp_scale[itr]);$$

Symmetric quantization

for itr = 0:(p_inp_shape[axis])

$$p_out[d0][d1][itr][d3][d4] = (float)(p_inp[d0][d1][itr][d3][d4] * inp_scale[itr]);$$

Where

d0 = 0 to p_inp_shape[0]-1

d1 = 0 to p_inp_shape[1]-1

d2 = 0 to p_inp_shape[2]-1

d3 = 0 to p_inp_shape[3]-1

d4 = 0 to p_inp_shape[4]-1

Note: Both tensor quantization and channel based quantization is performed using the same kernel function.

Prototype

```
WORD32 xa_nn_elm_dequantize_asym4_f32
(FLOAT32 *__restrict__ p_out,      const WORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_asym8_f32
(FLOAT32 *__restrict__ p_out,      const WORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_asym16_f32
(FLOAT32 *__restrict__ p_out,      const WORD16 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_sym4_f32
(FLOAT32 *__restrict__ p_out,      const WORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_sym8_f32
(FLOAT32 *__restrict__ p_out,      const WORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_sym16_f32
(FLOAT32 *__restrict__ p_out,      const WORD16 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_asym4u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_asym8u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_asym16u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD16 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
WORD32 *p_inp_zero_bias,          FLOAT32 *p_inp_scale);

WORD32 xa_nn_elm_dequantize_sym4u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
```

```

FLOAT32 *p_inp_scale);
WORD32 xa_nn_elm_dequantize_sym8u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD8 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
FLOAT32 *p_inp_scale);
WORD32 xa_nn_elm_dequantize_sym16u_f32
(FLOAT32 *__restrict__ p_out,      const UWORD16 *__restrict__ p_inp,
const WORD32 *const p_inp_shape,   WORD32 num_inp_dims,                WORD32 *p_axis,
FLOAT32 *p_inp_scale);

```

Arguments

Type	Name	Size	Description
Input			
const WORD8 * WORD16 * UWORD8 * UWORD16 *	p_inp	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Input vector
Const WORD32 *contest	p_inp_shape	num_inp_dims	Shape of the input vector
WORD32	num_inp_dims	1	Number of input dimensions
WORD32 *	p_inp_zero_bias	1 Or P_inp_shape[*p_axis]	Zero offset of input
FLOAT32 *	p_inp_scale	1 Or P_inp_shape[*p_axis]	Input scale
WORD32 *	P_axis	1	A dimension along which dequantization will be computed
Output			
FLOAT32 *	p_out	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	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
	Must not overlap
num_inp_dims	Greater than 0 and less than or equal 5
*p_inp_zero_bias	{-8,.....,7} for out type signed 4-bit {-128,....., 127} for out type signed 8-bit {-32768,....., 32767} for out type signed 16-bit {0,.....,15} for out type unsigned 4-bit {0,....., 255} for out type unsigned 8-bit

	{0...., 65535} for out type unsigned 16-bit
*p_axis	[0, num_inp_dims)
*p_inp_scale	Finite single precision float value

2.2.3 Basic Vector math operation 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, div, exp, clamp, where, square-root and inverse square-root. The supported precisions are: float32, int32, int16, int8, uint8.

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

[o]: Operations: add, add_scalar, sub, sub_scalar, mul, mul_scalar, div, div_scalar, exp, clamp, clamp_scalar, where, sqrt, 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
8_8	signed 8-bit input, signed 8-bit output
8u_8u	unsigned 8-bit input, unsigned 8-bit output
16_16	signed 16-bit input, signed 16-bit output
32x32_32	2 32-bit input, 32-bit output

Algorithm

elm_add	:	$z_n = x_n + \alpha * y_n,$	$n = 0 \dots, num_elm - 1$	
elm_add_scalar:		$z_n = x_n + \alpha * y,$	$n = 0 \dots, num_elm - 1$	
elm_sub	:	$z_n = x_n - \alpha * y_n,$	$n = 0 \dots, num_elm - 1$	
elm_sub_scalar:		$z_n = x_n - \alpha * y,$	$n = 0 \dots, num_elm - 1$	
elm_mul	:	$z_n = x_n * y_n,$	$n = 0 \dots, num_elm - 1$	
elm_mul_scalar:		$z_n = x_n * y,$	$n = 0 \dots, num_elm - 1$	
elm_div	:	$z_n = x_n / y_n,$	$n = 0 \dots, num_elm - 1$	mode=0
	:	$z_n = truncate(x_n / y_n),$	$n = 0 \dots, num_elm - 1,$	mode=1
	:	$z_n = floor(x_n / y_n),$	$n = 0 \dots, num_elm - 1,$	mode=2
elm_div_scalar:		$z_n = x_n / y,$	$n = 0 \dots, num_elm - 1,$	mode=0
	:	$z_n = truncate(x_n / y),$	$n = 0 \dots, num_elm - 1,$	mode=1
	:	$z_n = floor(x_n / y),$	$n = 0 \dots, num_elm - 1,$	mode=2
elm_exp:		$z_n = exp(x_n),$	$n = 0 \dots, num_elm - 1$	
elm_clamp:		$z_n = \min(max(x_n, xmin_n), xmax_n),$	$n = 0 \dots, num_elm - 1$	
elm_clamp_scalar:		$z_n = \min(max(x_n, xmin), xmax)$	$n = 0 \dots, num_elm - 1$	
elm_where:		$z_n = (a_n ? x_n : y_n),$	$n = 0 \dots, num_elm - 1$	

elm_sqrt: $z_n = \sqrt{x_n}, \quad n = 0 \dots, num_elm - 1$
elm_rsqrt: $z_n = 1 \div \sqrt{x_n}, \quad n = 0 \dots, num_elm - 1$

x_n represents first input, y_n represents second input, α represents a scale value, a_n represents the condition on which any one of the inputs is selected in “where” operator, $xmin_n$ and $xmax_n$ represents array of minimum and maximum values used to limit the input in “clamp” operator. All the variables without a subscript “n” has the same meaning as above but they are scalar values.

z_n represents output.

Prototype

```
WORD32 xa_nn_elm_add_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 * p_inp2,
FLOAT32 alpha,            WORD32 num_elm);
WORD32 xa_nn_elm_add_scalar_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 inp2,
FLOAT32 alpha,            WORD32 num_elm);
WORD32 xa_nn_elm_add_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 * p_inp2,
WORD32 alpha,             WORD32 num_elm);
WORD32 xa_nn_elm_add_scalar_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 inp2,
WORD32 alpha,             WORD32 num_elm);

WORD32 xa_nn_elm_sub_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 * p_inp2,
FLOAT32 alpha,            WORD32 num_elm);
WORD32 xa_nn_elm_sub_scalar_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 inp2,
FLOAT32 alpha,            WORD32 num_elm);
WORD32 xa_nn_elm_sub_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 * p_inp2,
WORD32 alpha,             WORD32 num_elm);
WORD32 xa_nn_elm_sub_scalar_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 inp2,
WORD32 alpha,             WORD32 num_elm);

WORD32 xa_nn_elm_mul_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 * p_inp2,
WORD32 num_elm);
WORD32 xa_nn_elm_mul_scalar_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 inp2,
WORD32 num_elm);
WORD32 xa_nn_elm_mul_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 * p_inp2,
WORD32 num_elm);
WORD32 xa_nn_elm_mul_scalar_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 inp2,
WORD32 num_elm);

WORD32 xa_nn_elm_div_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 * p_inp2,
WORD32 mode,              WORD32 num_elm);
WORD32 xa_nn_elm_div_32x32_32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 * p_inp2,
WORD32 mode,              WORD32 num_elm);
WORD32 xa_nn_elm_div_32x32_f32
(WORD32 * p_out,           const WORD32 * p_inpl,       const WORD32 * p_inp2,
WORD32 num_elm);

WORD32 xa_nn_elm_div_scalar_f32xf32_f32
```

```

(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 inp2,
WORD32 mode,              WORD32 num_elm);
WORD32 xa_nn_elm_div_scalar_32x32_32
(WORD32 * p_out,          const WORD32 * p_inpl,      const WORD32 inp2,
WORD32 mode,              WORD32 num_elm);
WORD32 xa_nn_elm_div_scalar_32x32_f32
(WORD32 * p_out,          const WORD32 * p_inpl,      const WORD32 inp2,
WORD32 num_elm);

WORD32 xa_nn_elm_exp_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,      WORD32 num_elm);

WORD32 xa_nn_elm_clamp_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,      const FLOAT32 * p_min,
const FLOAT32 * p_max,      WORD32 num_elm);
WORD32 xa_nn_elm_clamp_scalar_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,      const FLOAT32 min,
const FLOAT32 max,          WORD32 num_elm);
WORD32 xa_nn_elm_clamp_16_16
(WORD16 * p_out,          const WORD16 * p_inp,      const WORD16 * p_min,
const WORD16 * p_max,      WORD32 num_elm);
WORD32 xa_nn_elm_clamp_scalar_16_16
(WORD16 * p_out,          const WORD16 * p_inp,      const WORD16 min,
const WORD16 max,          WORD32 num_elm);
WORD32 xa_nn_elm_clamp_8_8
(WORD8 * p_out,           const WORD8 * p_inp,      const WORD8 * p_min,
const WORD8 * p_max,      WORD8 num_elm);
WORD32 xa_nn_elm_clamp_scalar_8_8
(WORD8 * p_out,           const WORD8 * p_inp,      const WORD8 min,
const WORD8 max,          WORD8 num_elm);
WORD32 xa_nn_elm_clamp_8u_8u
(UWORD8 * p_out,          const UWORD8 * p_inp,      const UWORD8 * p_min,
const UWORD8 * p_max,      WORD8 num_elm);
WORD32 xa_nn_elm_clamp_scalar_8u_8u
(UWORD8 * p_out,          const UWORD8 * p_inp,      const UWORD8 min,
const UWORD8 max,          WORD8 num_elm);

WORD32 xa_nn_elm_where_f32xf32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inpl,      const FLOAT32 * p_inp2,
const UWORD8 * p_cond,      WORD32 num_elm);

WORD32 xa_nn_elm_sqrt_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,      WORD32 num_elm);

WORD32 xa_nn_elm_rsqrt_f32_f32
(FLOAT32 * p_out,          const FLOAT32 * p_inp,      WORD32 num_elm);

```

Arguments

Type	Name	Size	Description
Input			
const UWORD8 * WORD8 * WORD16 * WORD32 * FLOAT32 *	p_inpl, p_inp,	num_elm	First input vector
const WORD32 * FLOAT32 *	p_inp2	num_elm	Second input vector
WORD32 FLOAT32	inp2	1	Second input which is scalar
WORD32	num_elm	1	Number of elements
WORD32	alpha	1	Scale for the second

Type	Name	Size	Description
Float32			operand in add and sub operators
UWORD8 * WORD8 * WORD16 * Float32 *	p_min, min	num_elm	Minimum values vector
UWORD8 WORD8 WORD16 Float32	min	1	Minimum value which is scalar
UWORD8 * WORD8 * WORD16 * Float32 *	p_max	num_elm	Max values vector
UWORD8 WORD8 WORD16 Float32	max	1	Maximum value which is scalar
UWORD8 *	p_cond	num_elm	Condition on which one of the input is selected in where operator
WORD32	mode	1	Type of division 0 – normal a/b 1 – truncate the result after division 2 – floor of the result after division
Output			
UWORD8 * WORD8 * WORD16 * WORD32 * 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_min, p_max, p_cond, p_out	Aligned on (size of one element)-byte boundary Cannot be NULL
p_out	Must not overlap with the input pointers
num_elm	Greater than 0
mode	0, 1, 2

2.2.4 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 . Currently, the supported operation is: less than ($<$). The output for the comparison kernels is a Boolean value that requires 1-byte space. The supported precisions are: f32.

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

[o]: Operations: less

[p]: Input Precision in bits- input1input2

[q]: output - bool

Precision

Type	Description
f32xf32_bool	2 float32 inputs, Boolean(1-byte) output

Algorithm

elm_less: $z_n = (x_n < y_n), \quad n = 0 \dots, num_elm - 1$
 elm_less_scalar: $z_n = (x_n < y), \quad n = 0 \dots, num_elm - 1$

x_n represents first input, y_n or y represents second input.

z_n represents output.

Prototype

```
WORD32 xa_nn_elm_less_f32xf32_bool
(WORD8 * p_out,          const FLOAT32 * p_inp1,    const FLOAT32 * p_inp2,
 WORD32  num_elm);

WORD32 xa_nn_elm_less_scalar_f32xf32_bool
(WORD8 * p_out,          const FLOAT32 * p_inp1,    const FLOAT32 inp2,
 WORD32  num_elm);
```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp1	num_elm	First input vector
const FLOAT32 *	p_inp2	num_elm	Second input vector or scalar
FLOAT32	inp2	1	Second input as scalar
WORD32	num_elm	1	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_inp2, p_out	Aligned on (size of one element)-byte boundary
	Cannot be NULL
P_out	Must not overlap with the input pointers
num_elm	Greater than 0

2.2.5 Basic Kernels with 5D Broadcasting

Description

The Basic kernels with 5D broadcasting perform a broadcast operation and apply an operator on the inputs. The supported operators are: elementwise add, sub, mul, div, less, clamp and where.

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

These kernels support upto 5-dimensional input/output tensors. The 1/2/3/4-dimensional inputs can be scaled up to 5D within the kernel. Both inputs and output must have the same number of 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_elm_[op]_broadcast_5D_[p]_[q]`, where:

- [op]: Operation: add, sub, mul, div, less, where, clamp
- [p]: Input Precision in bits- input1xinput2 or input1
- [q]: Output Precision in bits

Precision

Type	Description
f32_f32	Float32 input, float32 output
8_8	Signed 8-bit input, signed 8-bit output
8u_8u	Unsigned 8-bit input, unsigned 8-bit output
16_16	Signed 16-bit input, signed 16-bit output

32x32_32	2 32-bit input, 32-bit output
f32xf32_bool	2 float32 inputs, Boolean(1-byte) output
f32xf32_f32	2 f32 inputs, f32 output

Algorithm

$$p\text{-out}[i_0][i_1] \dots [i_4] = [op](p_inp1[i1_0][i1_1] \dots [i1_4], p_inp2[i2_0][i2_1] \dots [i2_4])$$

Where,

$$\blacksquare \quad i_n = \max(i1_n, i2_n); n = [0, 4]$$

Ops are:

elm_add:	$z_n = x_i + \alpha * y_j$	
elm_sub:	$z_n = x_i - \alpha * y_j$	
elm_mul:	$z_n = x_i * y_j$	
elm_div:	$z_n = x_i / y_j,$	mode = 0
:	$z_n = \text{truncate}(x_i / y_j),$	mode = 1
:	$z_n = \text{floor}(x_i / y_j),$	mode = 2
elm_less:	$z_n = x_i < y_j$	
elm_where:	$z_n = (a_i ? x_j : y_k)$	
elm_clamp:	$z_n = \min(\max(x_i, x_{min_j}), x_{max_k})$	

Prototypes

```
WORD32 xa_nn_elm_add_broadcast_5D_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 num_inp_dims,
 FLOAT32 alpha
);
WORD32 xa_nn_elm_add_broadcast_5D_32x32_32
(WORD32 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 const WORD32 * __restrict__ p_inp1,
 const WORD32 *const p_inp1_shape,
 const WORD32 * __restrict__ p_inp2,
 const WORD32 *const p_inp2_shape,
 WORD32 num_inp_dims,
 WORD32 alpha
);
WORD32 xa_nn_elm_sub_broadcast_5D_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      num_inp_dims,
FLOAT32 alpha
);
WORD32 xa_nn_elm_sub_broadcast_5D_32x32_32
(WORD32 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const WORD32 * __restrict__ p_inp1,
const WORD32 *const p_inp1_shape,
const WORD32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape,
WORD32      num_inp_dims,
WORD32 alpha
);
WORD32 xa_nn_elm_mul_broadcast_5D_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      num_inp_dims
);
WORD32 xa_nn_elm_mul_broadcast_5D_32x32_32
(WORD32 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const WORD32 * __restrict__ p_inp1,
const WORD32 *const p_inp1_shape,
const WORD32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape,
WORD32      num_inp_dims
);
WORD32 xa_nn_elm_div_broadcast_5D_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      mode,
WORD32      num_inp_dims
);
WORD32 xa_nn_elm_div_broadcast_5D_32x32_32
(WORD32 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const WORD32 * __restrict__ p_inp1,
const WORD32 *const p_inp1_shape,
const WORD32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape,
WORD32      mode,
WORD32      num_inp_dims
);
WORD32 xa_nn_elm_div_broadcast_5D_32x32_f32
(WORD32 * __restrict__ p_out,

```

```

const WORD32 *const p_out_shape,
const WORD32 * __restrict__ p_inpl,
const WORD32 *const p_inpl_shape,
const WORD32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape
WORD32      num_inp_dims
);

WORD32 xa_nn_elm_less_broadcast_5D_f32xf32_bool
(WORD8 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const FLOAT32 * __restrict__ p_inpl,
const WORD32 *const p_inpl_shape,
const FLOAT32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape,
WORD32      num_inp_dims
);

WORD32 xa_nn_elm_where_broadcast_5D_f32xf32_f32
(FLOAT32 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const FLOAT32 * __restrict__ p_inpl,
const WORD32 *const p_inpl_shape,
const FLOAT32 * __restrict__ p_inp2,
const WORD32 *const p_inp2_shape,
const UWORD8 * p_cond,
const WORD32 *const p_cond_shape,
WORD32      num_inp_dims
);

WORD32 xa_nn_elm_clamp_broadcast_5D_f32_f32
(FLOAT32 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const FLOAT32 * __restrict__ p_inp,
const WORD32 *const p_inpl_shape,
const FLOAT32 * __restrict__ p_min,
const WORD32 *const p_min_shape,
const FLOAT32 * __restrict__ p_max,
const WORD32 *const p_max_shape,
WORD32      num_inp_dims
);

WORD32 xa_nn_elm_clamp_broadcast_5D_16_16
(WORD16 * __restrict__ p_out,
const WORD32 *const p_out_shape,
const WORD16 * __restrict__ p_inp,
const WORD32 *const p_inpl_shape,
const WORD16 * __restrict__ p_min,
const WORD32 *const p_min_shape,
const WORD16 * p_max,
const WORD32 *const p_max_shape,
WORD32      num_inp_dims
);

```

```

WORD32 xa_nn_elm_clamp_broadcast_5D_8_8
(WORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 const WORD8 * __restrict__ p_inp,
 const WORD32 *const p_inpl_shape,
 const WORD8 * __restrict__ p_min,
 const WORD32 *const p_min_shape,
 const WORD8 * p_max,
 const WORD32 *const p_max_shape,
 WORD32      num_inp_dims
);

WORD32 xa_nn_elm_clamp_broadcast_5D_8u_8u
(UWORD8 * __restrict__ p_out,
 const WORD32 *const p_out_shape,
 const UWORD8 * __restrict__ p_inp,
 const WORD32 *const p_inpl_shape,
 const UWORD8 * __restrict__ p_min,
 const WORD32 *const p_min_shape,
 const UWORD8 * p_max,
 const WORD32 *const p_max_shape,
 WORD32      num_inp_dims
);

```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *, const UWORD8 *, const WORD16 *, const FLOAT32 *, const WORD32 *	p_inpl, p_inp	$\prod_{i=0}^{i=num_inp_dims-1} p_inp1_shape[i]$	First input tensor
const FLOAT32 *, const WORD32 *	p_inp2	$\prod_{i=0}^{i=num_inp_dims-1} p_inp2_shape[i]$	Second input tensor
const WORD8 *	p_cond	$\prod_{i=0}^{i=num_inp_dims-1} p_cond_shape[i]$	Input tensor holding conditions used in where operator
const WORD8 *, const UWORD8 *, const WORD16 *, const FLOAT32 *, const WORD32 *	p_max	$\prod_{i=0}^{i=num_inp_dims-1} p_max_shape[i]$	Input tensor holding max values for Clamp operator
const WORD8 *, const	p_min	$\prod_{i=0}^{i=num_inp_dims-1} p_min_shape[i]$	Input tensor holding min values for Clamp

Type	Name	Size	Description
UWORD8 *, const WORD16 *, const FLOAT32 *,			operator
const WORD32 *const	p_out_shape	num_inp_dims	Shape of output (first dimension is outer most)
const WORD32 *const	p_inp1_shape p_inp_shape	num_inp_dims	Shape of first input (first dimension is outer most)
const WORD32 *const	p_inp2_shape	num_inp_dims	Shape of second input (first dimension is outer most)
const WORD32 *const	p_cond_shape	num_inp_dims	Shape of condition tensor input (first dimension is outer most)
const WORD32 *const	p_max_shape	num_inp_dims	Shape of max tensor input (first dimension is outer most)
const WORD32 *const	p_min_shape	num_inp_dims	Shape of min tensor input (first dimension is outer most)
WORD32	num_inp_dims	1	Number of input dimensions
WORD32	mode	1	Type of division 0 – normal a/b 1 – truncate the result after division 2 – floor of the result after division
Output			
UWORD8 * WORD8 * FLOAT32 * WORD16 * WORD32 *	p_out	$\prod_{i=0}^{i=num_inp_dims-1} p_out_shape[i]$	Output tensor

Returns

0: no error

-1: error, invalid parameters

Restrictions

Arguments	Restrictions
p_inp1, p_inp, p_inp2, p_cond, p_max, p_min p_out	Aligned on (size of one element)-byte boundary Cannot be NULL
p_out	Must not overlap with the input pointers
p_out_shape, p_inp1_shape,	Cannot be NULL

Arguments	Restrictions
p_inp2_shape, p_cond_shape, p_max_shape, p_min_shape	Aligned on 4-byte boundary
	Shapes must be broadcast compatible, that is, p_out_shape[i] must be max(ith shape of all the inputs) p_inp1_shape[i] must be either equal to ith shape of other inputs or 1 p_inp2_shape[i] must be either equal to ith shape of other inputs or 1 p_cond_shape[i] must be either equal to ith shape of other inputs or 1 p_max_shape[i] must be either equal to ith shape of other inputs or 1 p_min_shape[i] must be either equal to ith shape of other inputs or 1

2.2.6 Mean

Description

The Mean kernel computes the mean of input vector x and gives output vector as $y = \text{mean}(x)$. The output vector dimension will be reduced while performing the operation.

The number of input dimensions must be less than or equal to 5. The 1/2/3/4-dimensional inputs can be scaled up to 5D.

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

[p]: Input precision in bits

[q]: Output precision in bits

Precision

There is single variant available:

Type	Description
f32_f32	float32 input, float32 output

Algorithm

$$y = \frac{\sum_k x_n}{K}, \quad * p_axis = NULL, \quad k = \prod_{i=0}^{i=\text{num_inp_dims}-1} p_inp_shape[i]$$

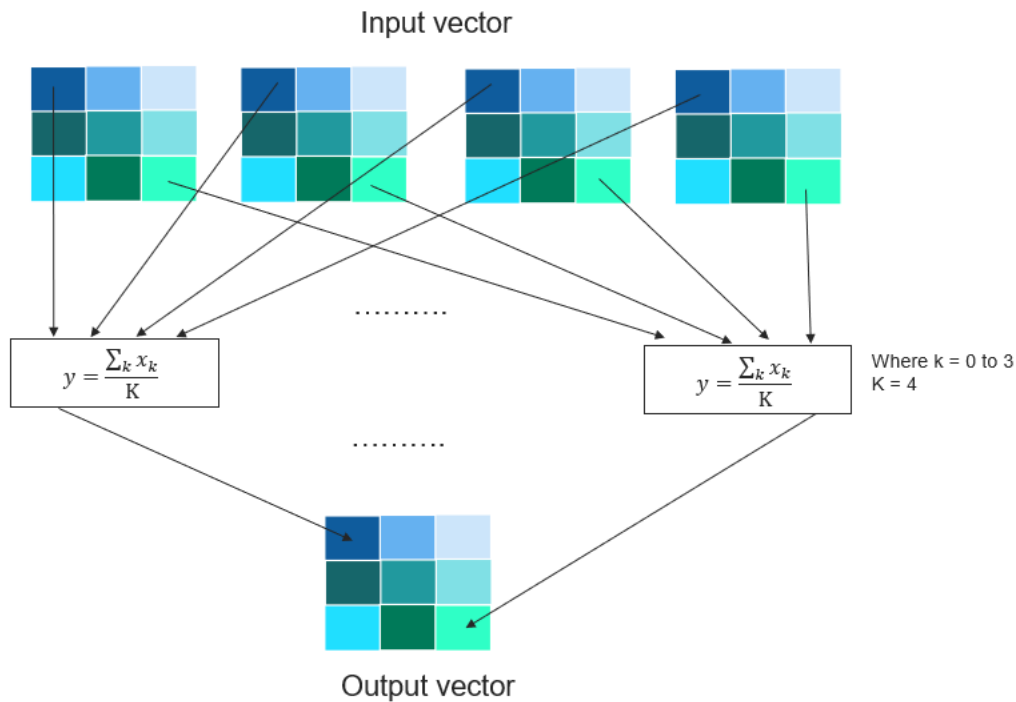
When axis is not NULL, below is the algorithm used to compute mean around one dimension in axis. The code will be repeated for all the listed dimensions in axis pointer.

`out_idx = 0;`

```

for j=0 to  $\prod_{i=0}^{axis-1} p\_inp\_shape[i]$ 
  outer_stride =  $j * \prod_{i=axis}^{num\_inp\_dims-1} p\_inp\_shape[i]$ ;
  for k=0 to  $\prod_{i=axis+1}^{num\_inp\_dims-1} p\_inp\_shape[i]$ 
    accum = 0;
    end =  $\prod_{i=axis}^{num\_inp\_dims-1} p\_inp\_shape[i]$  ;
    stride =  $\prod_{i=axis+1}^{num\_inp\_dims-1} p\_inp\_shape[i]$ ;
    for m = (outer_stride + k) to end with a stride of stride
      accum = accum + input[m];
    out[out_idx] = accum / p_inp_shape[axis];
    out_idx = out_idx + 1;
  
```

For ex: For an input dimension of [20][12][4][3][3] with the axis set to 2, mean is computed for groups of 4 elements each, where the dimension along axis 2 is 4. The mean is computed on a group of 4 elements each from the last 3 dimensions (4x3x3) as illustrated in the figure below. This process is repeated across the top two dimensions of the input, which are dimensions 20 and 12.



Prototype

```

WORD32 xa_nn_mean_f32_f32
(FLOAT32 * p_out,
const FLOAT32 * p_inp,
WORD32 *p_axis,
const WORD32 * p_out_shape,
const WORD32 * p_inp_shape,
WORD32 num_axis_dims,
WORD32 num_out_dims,
WORD32 num_inp_dims,
void *p_scratch_in);

```


Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Input vector
const WORD32 *	p_inp_shape	num_inp_dims	Input shape
const WORD32 *	p_out_shape	num_out_dims	Output shape
WORD32	num_inp_dims	1	Number of dimensions in the input
WORD32	num_out_dims	1	Number of dimensions in the output
WORD32	p_axis	1	A one or more dimension values along which mean will be computed
WORD32	num_axis_dims	1	Number of dimension along which mean has to be calculated
WORD32 *	p_scratch_in	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Scratch pointer to be allocated by application and passed to the kernel. This is used for intermediate calculations.
Output			
FLOAT32 *	p_out	$\prod_{i=0}^{i=num_out_dims-1} p_out_shape[i]$	Output vector.

Returns

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

Restrictions

Arguments	Restrictions
p_inp, p_out, p_scratch_in	Aligned on (size of one element)-byte boundary must not overlap Cannot be NULL
p_inp_shape, p_out_shape	The dimension values should be greater than 0 Aligned on 4-byte boundary
num_inp_dims num_out_dims	Greater than 0
Dim	Maximum value is num_inp_dims-1
*p_axis	all the values need to be in the range [0, num_dim)
Num_axis_dims	[0, num_dim)

2.3 Normalization Kernels

2.3.1 Layer Normalization Kernel

Description

The Layer Normalization kernel applies Layer normalization on an input vector x to get output vector z .

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

[p]: Input precision in bits

[q]: Output precision in bits

Precision

Type	Description
float32	float32 input, float32 output

Algorithm

$$\left. \begin{aligned}
 z_i[n] &= \frac{x_i[n] - \mu_i}{\sigma_i + \epsilon} * w[n] + b[n] \\
 \mu_i &= \text{mean}(x_i[n]) \\
 \sigma_i &= \text{std}(x_i[n])
 \end{aligned} \right\} \begin{aligned}
 &\text{for a given axis } k \text{ and dimension size } \text{num_inp_dims} \\
 &i = \prod_{j=0}^{j=k-1} p_inp_shape[j] \\
 &n = \prod_{j=k}^{j=\text{num_inp_dims}-1} p_inp_shape[j]
 \end{aligned}$$

$$\left. \begin{aligned}
 z_n &= \frac{x_n - \mu}{\sigma + \epsilon} * w_n + b_n
 \end{aligned} \right\} \begin{aligned}
 &\text{when axis is } 0 \\
 &n = \prod_{j=0}^{j=\text{num_inp_dims}-1} p_inp_shape[j]
 \end{aligned}$$

x_n and $x_i[n]$ represents input vector.

w_n and $w[n]$ represents weight vector.

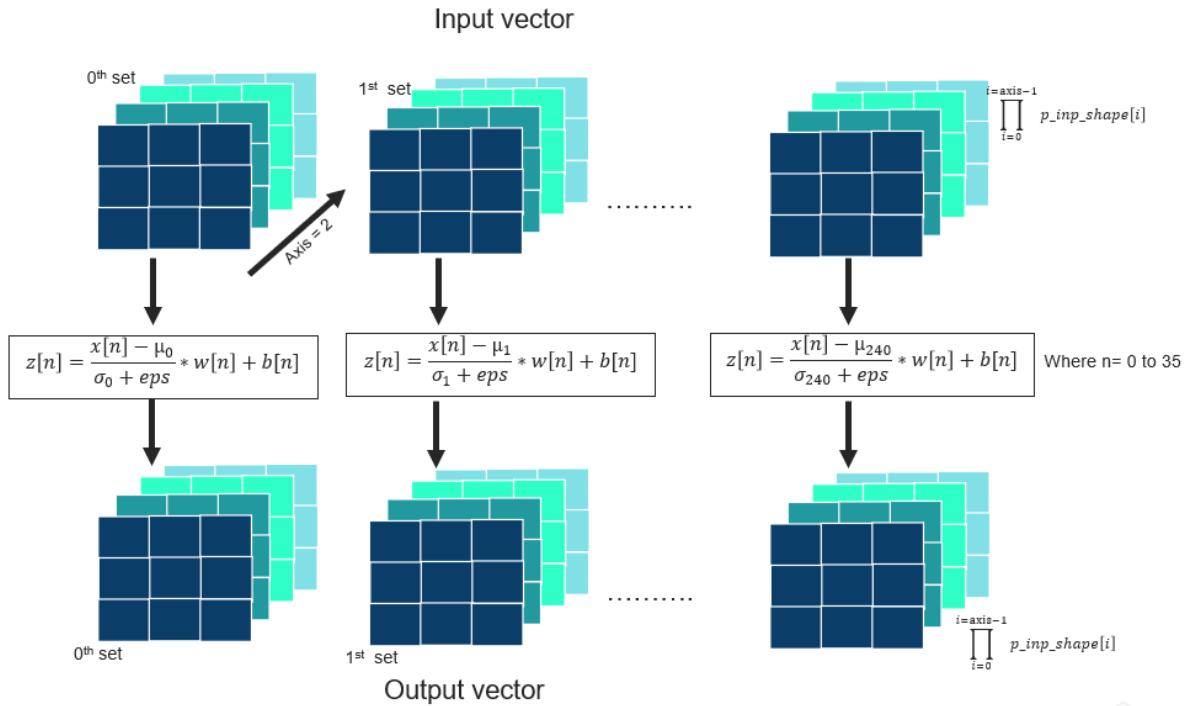
b_n and $b[n]$ represents bias vector.

μ and μ_i represents mean.

σ and σ_i represents standard deviation.

z_n and $z_i[n]$ represents output vector.

For ex: For an input dimension of [20][12][4][3][3] and with the axis set to 2, layer normalization is applied to groups of 36 elements, where each group consists of a 4x3x3 block of elements. This operation is performed independently for each group, as illustrated in the figure below. The process is repeated across the top two dimensions of the input, namely dimensions 20 and 12..



Prototype

```
WORD32 xa_nn_native_layer_norm_f32_f32
(FLOAT32 * p_out,          FLOAT32 * p_mean,          FLOAT32 * p_rstd,
 const FLOAT32 * p_inp,    const WORD32 *const p_inp_shape, WORD32 num_inp_dims,
 WORD32 axis,              const FLOAT32 * p_weight,    const FLOAT32 * p_bias,
 FLOAT32 eps);
```

Arguments

Type	Name	Size	Description
Input			
const FLOAT32 *	p_inp	$i = \text{num_inp_dims} - 1$ $\prod_{i=0} p_inp_shape[i]$	Input vector
const FLOAT32 *	p_weight	$i = \text{num_inp_dims} - 1$ $\prod_{i=p_axis} p_inp_shape[i]$	Weight vector
const FLOAT32 *	p_bias		Bias vector
Const	P_inp_shape	num_inp_dims	Input shape

WORD32 *			
const			
WORD32	num_inp_dims	1	Number of dimensions in the input
WORD32	axis	1	Dimension number for which layer normalization will be calculated
FLOAT32	eps	1	Value used in division operations to avoid divide by zero error
Output			
FLOAT32 *	p_out	$\prod_{i=0}^{i=\text{num_inp_dims}-1} p_inp_shape[i]$	Output vector
FLOAT32 *	P_mean	$\prod_{i=\text{axis}-1}^{i=\text{num_inp_dims}-1} p_inp_shape[i]$	Mean vector
FLOAT32 *	P_rstd	$\prod_{i=0}^{i=\text{axis}-1} p_inp_shape[i]$ (OR) 1	Inverse of standard deviation vector

Returns

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

Restrictions

Arguments	Restrictions
p_inp, p_out, p_weight, p_bias, p_mean, p_rstd, p_inp_shape	Aligned on (size of one element)-byte boundary Cannot be NULL
P_out, p_mean, p_std	Must not overlap with inputs
eps, num_inp_dims	Greater than 0
axis	[0, num_inp_dims)

2.4 Reorg Kernels

2.4.1 Slice

Description

The Slice kernels process the input data based on the parameters: axis, start, stop, and stride. The axis parameter indicates the dimension along which the slicing will occur. The operation begins at the position specified by the start parameter and selects elements according to the stride value, continuing until it reaches the stop point within that dimension.

Function variants available are `xa_nn_slice`.

Precision

The kernel is designed to manage all the specified variants with a single implementation. It requires an input parameter, "elm_size" which specifies the size of the data type. By utilizing this parameter, the kernel is capable of supporting all the precision variants listed below.

Type	Description
8_8	8-bit input, 8-bit output
16_16	16-bit input, 16-bit output
32_32	32-bit input, 32-bit output
8u_8u	Unsigned 8-bit input, 8-bit output
16u_16u	Unsigned 16-bit input, 16-bit output
32u_32u	Unsigned 32-bit input, 32-bit output

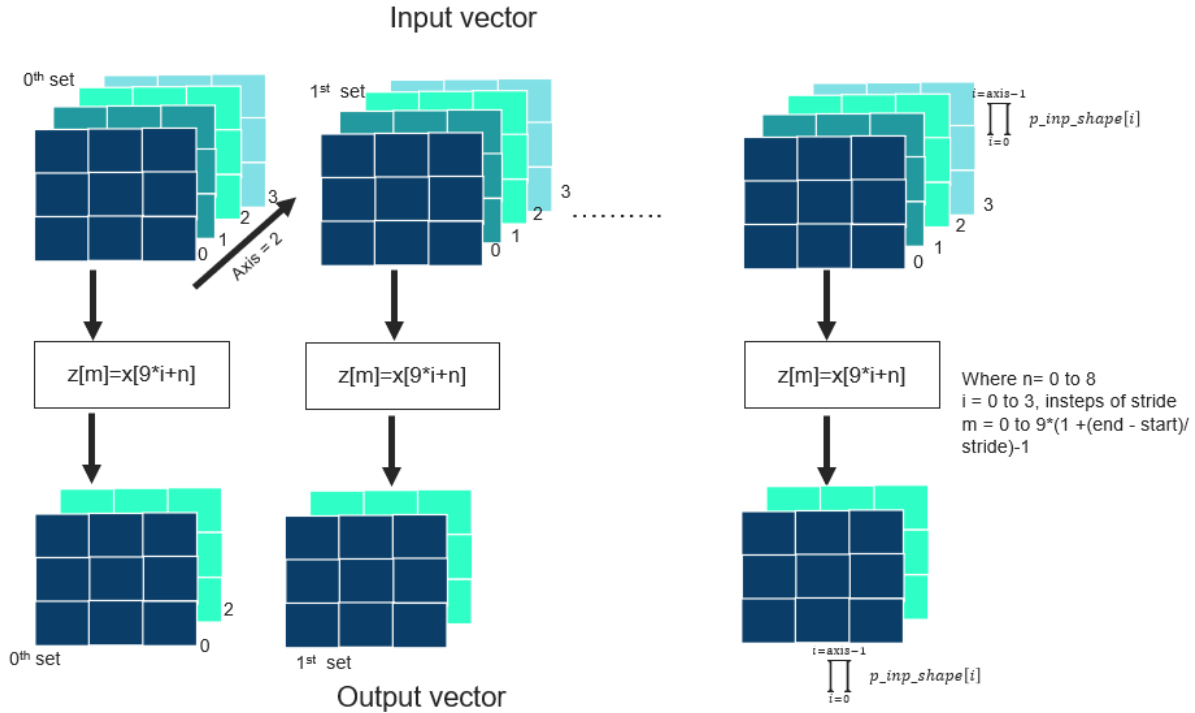
Algorithm

```

bytes_to_copy = inp_shape[axis+1]*inp_shape[axis+2]*...*inp_shape[num_inp_dims-1] *
size_of_element
for I = 0 to inp_shape[0]*inp_shape[1]*...*inp_shape[axis-1]
  for J = start to stop insteps of stride
    memcpy(out, inp, bytes_to_copy);
    inp = inp + stride*bytes_to_copy;
    out = out + bytes_to_copy;
  end
end

```

For ex: For an input vector with dimensions [20][12][4][3][3], and with the following parameters: axis = 2, start = 0, stride = 2, and stop = 3, slices of size 3x3 are extracted according to these parameters. Specifically, slices are taken from the input along the specified axis (axis 2) starting from the index given by the start parameter (0), selecting elements with the specified stride (2), and ending at the stop parameter (3). This slicing process is performed across the top two dimensions of the input, which are dimensions 20 and 12. This is represented in the below figure.



Prototype

```
WORD32 xa_nn_slice(
  WORD8 * __restrict__ p_out,
  const WORD32 *const p_out_shape,
  const WORD8 * __restrict__ p_inp,
  const WORD32 *const p_inp_shape,
  WORD32 num_inp_dims,
  WORD32 start,
  WORD32 end,
  WORD32 step,
  WORD32 axis,
  WORD32 elm_size);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 *	p_inp	$i = \text{num_inp_dims} - 1$ $\prod_{i=0} p_inp_shape[i]$	Input vector
Const WORD32 *const	p_inp_shape	num_inp_dims	Shape of the input vector
Const WORD32 *const	p_out_shape	num_inp_dims	Shape of the output vector
WORD32	num_inp_dims	1	Number of dimensions in the input vector
WORD32	start	1	Starting index along the axis

Type	Name	Size	Description
WORD32	end	1	Ending index along the axis
WORD32	step	1	stride
WORD32	axis	1	Axis along which the vector has to be sliced
WORD32	elm_size	1	Size of the element. If the tensors of type WORD32, UWORD32 – 4bytes WORD16, UWORD16 – 2 bytes WORD8, UWORD8 – 1 byte
Output			
WORD8 *	p_out	$\prod_{i=0}^{axis-1} p_inp_shape[i] * \dots * inp_shape[1+(end-start)/step] * \prod_{i=axis+1}^{num_inp_dims-1} p_inp_shape[i]$	Output vector

Returns

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

Restrictions:

Arguments	Restrictions
p_inp, p_out, p_inp_shape, p_out_shape	Cannot be NULL Aligned on size of element boundary
p_out	Must not overlap with input
axis	[0, num_inp_dims-1]
step	Greater than 0
start	[0, inp_shape[axis]-1]
end	[0, inp_shape[axis]-1], end>=start
elm_size	1, 2, 4

2.4.2 Permute

Description

This kernel performs a permute operation on a N-dimensional input tensor (up to 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]`.

The number of input dimensions must be less than or equal to 5. The 1/2/3/4-dimensional inputs can be scaled up to 5D. The output shape should be conformant with respect to the values in the permute vector.

The naming convention used for the Permute kernel is as follows:

`xa_nn_permute`

Note: Tranpose is a variant of permute operation. So, a tensor transpose can be achieved using the permute kernel by changing the dimension in `permute_vec`.

Precision

The kernel is designed to manage all the specified variants with a single implementation. It requires an input parameter, "elm_size" which specifies the size of the data type. By utilizing this parameter, the kernel is capable of supporting all the precision variants listed below.

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
8u_8u	Unsigned 8-bit input, unsigned 8-bit output.
16u_16u	Unsigned 16-bit input, unsigned 16-bit output.
32u_32u	Unsigned 32-bit input, unsigned 32-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,

$v = 0 \dots dim0 - 1$
 $w = 0 \dots dim1 - 1$
 $x = 0 \dots dim2 - 1$
 $y = 0 \dots dim3 - 1$
 $z = 0 \dots dim4 - 1$

Prototype

```
WORD32 xa_nn_permute
(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_inp_dims,
 WORD32 elm_size);
```

Arguments

Type	Name	Size	Description
------	------	------	-------------

Type	Name	Size	Description
Input			
const WORD32 *	p_out_shape	num_inp_dims	Shape of output
const WORD8 *	p_inp	$\prod_{i=0}^{i=num_inp_dims-1} p_inp_shape[i]$	Input vector. The size of each element is denoted by elm_size parameter.
const WORD32 *	p_inp_shape	num_inp_dims	Shape of input
const WORD32 *	p_permute_vec	num_inp_dims	Permute Vector
WORD32	num_inp_dims	1	Number of input dimensions
WORD32	elm_size	1	Size of the element. If the tensors of type WORD32, UWORD32 – 4bytes WORD16, UWORD16 – 2 bytes WORD8, UWORD8 – 1 byte
Output			
WORD8 *	p_out	$\prod_{i=0}^{i=num_inp_dims-1} p_out_shape[i]$	Output

Returns

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

Restrictions:

Arguments	Restrictions
p_out, p_inp	Aligned on elm_size -byte boundary
	Cannot be NULL
	Must not overlap
p_out_shape, p_inp_shape	Aligned on a 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_inp_dims	Must be in the range [1, 5].
elm_size	1, 2, 4

2.4.3 Cat

Description

The Cat kernel concatenates the given inputs into a single output along the dimension specified by the axis parameter. For example, 2 inputs of shapes (1, 8, 128, 32) and (1, 16, 128, 32) are concatenated into an output of shape (1, 24, 128, 32) with axis as '1'.

Function variants available are `xa_nn_cat`

Precision

The kernel is designed to manage all the specified variants with a single implementation. It requires an input parameter, "elm_size" which specifies the size of the data type. By utilizing this parameter, the kernel is capable of supporting all the precision variants listed below.

Type	Description
8_8	8-bit input, 8-bit output
16_16	16-bit input, 16-bit output
32_32	32-bit input, 32-bit output
8u_8u	Unsigned 8-bit input, 8-bit output
16u_16u	Unsigned 16-bit input, 16-bit output
32u_32u	Unsigned 32-bit input, 32-bit output

Algorithm

```
inp_dims[num_inp][num_dims]
out_dim[num_dims]
```

For axis = 2

i = 0 to num_inp - 1

out(d0, d1, sum(inp_dims[0][2] to inp_dims[i-1][2]) + d2, d3, d4, d5) = inp[i](d0, d1, d2, d3, d4, d5)

d0 = 0 to inp_dims[i][0]

d1 = 0 to inp_dims[i][1]

d2 = 0 to inp_dims[i][2]

d3 = 0 to inp_dims[i][3]

d4 = 0 to inp_dims[i][4]

d5 = 0 to inp_dims[i][5]

if j != axis

inp_dims[i][j] should be equal to out_dim[j]

if j == axis

out_dim[j] == sum(inp_dims[0][j] ... inp_dims[num_inp - 1][j])

Prototype

```
WORD32 xa_nn_cat
(WORD8 * __restrict__ p_out, const WORD32 *const p_out_shape
, const WORD8 **p_inps, const WORD32 *const *pp_inps_shape
, WORD32 num_inp_dims, WORD32 num_inp
```

```
,WORD32 axis, WORD32 elm_size);
```

Arguments

Type	Name	Size	Description
Input			
const WORD8 **	pp_inps		Inputs
const WORD32 *	p_out_shape		Shape of output
const WORD32 **	pp_inps_shape		Shape of Inputs
WORD32	num_inp_dims		Number of input dimensions
WORD32	num_inp		Number of Inputs
WORD32	axis		Dimension to concat
WORD8	elm_size		Size of the element. If the tensors of type WORD32, UWORD32 – 4bytes WORD16, UWORD16 – 2 bytes WORD8, UWORD8 – 1 byte
Output			
WORD8 *	p_out		Output

Returns

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

Restrictions:

Arguments	Restrictions
p_out, pp_inps, p_out_shape, pp_inps_shape	Aligned on (size of one element)-byte boundary
	Cannot be NULL
	Must not overlap
	The size of each dimension in output except at axis must match with the size of the corresponding each input dimension
p_out	Must not overlap with input
num_inp_dims	Greater than 0
num_inp	Greater than 0
axis	Greater than 0 and less than num_out_dims
elm_size	1, 2, 4

3. Making the library

The Fusion G3 NN library will be released as .tgz file for linux/makefile based usage. The detail about building the library is provided below.

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.