

z/OS
3.2

*IBM Z Deep Neural Network Library
Programming Guide and Reference*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 173.](#)

This edition applies to IBM® z/OS® 3.2 (5655-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2025-09-30

© **Copyright International Business Machines Corporation 2022, 2025.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

| | |
|---|-------------|
| Figures..... | vii |
| Tables..... | ix |
| About this information..... | xi |
| How to provide feedback to IBM..... | xiii |
| Summary of changes..... | xv |
| Summary of changes for z/OS 3.2..... | xv |
| Summary of changes for z/OS 3.1..... | xv |
| Part 1. IBM Z Deep Neural Network library..... | 1 |
| Chapter 1. Using the IBM Z Deep Neural Network Library..... | 3 |
| zDNN application environment..... | 3 |
| Common data types and structures..... | 4 |
| zDNN version information..... | 4 |
| zDNN tensor..... | 5 |
| zDNN tensor descriptor..... | 6 |
| zDNN data layouts..... | 6 |
| zDNN data formats..... | 7 |
| zDNN data types..... | 7 |
| Quantized transform types..... | 8 |
| zDNN statuses..... | 8 |
| zDNN runtime environment variables..... | 10 |
| Validating the environment at run time..... | 11 |
| Chapter 2. zDNN API reference..... | 13 |
| Support functions..... | 13 |
| Initialization (zdnm_init)..... | 13 |
| Query functions..... | 14 |
| Get size (zdnm_getsize_ztensor)..... | 22 |
| Get range (zdnm_getrange_ztensor)..... | 23 |
| Get maximum limit for a data type (zdnm_get_max_limit)..... | 23 |
| Get minimum limit for a data type (zdnm_get_min_limit)..... | 24 |
| Initialize pre-transformed tensor descriptor (zdnm_init_pre_transformed_desc)..... | 25 |
| Generate transformed tensor descriptor (zdnm_generate_transformed_desc)..... | 26 |
| Generate quantized transformed tensor descriptor (zdnm_generate_quantized_transformed_desc)..... | 26 |
| Generate concatenated transformed tensor descriptor (zdnm_generate_transformed_desc_concatenated)..... | 28 |
| Initialize zTensor (zdnm_init_ztensor)..... | 29 |
| Initialize zTensor with memory allocate (zdnm_init_ztensor_with_malloc)..... | 29 |
| Initialize quantized zTensor (zdnm_init_quantized_ztensor)..... | 30 |
| Initialize quantized zTensor with memory allocate (zdnm_init_quantized_ztensor_with_malloc)..... | 31 |
| Determine quantized zTensor (zdnm_is_quantized_ztensor)..... | 32 |
| Reset zTensor (zdnm_reset_ztensor)..... | 33 |

| | |
|---|------------|
| Allocate memory for zTensor (zdnn_allochelper_ztensor)..... | 34 |
| Deallocate memory for zTensor (zdnn_free_ztensor_buffer)..... | 34 |
| Retrieve status message for a status code (zdnn_get_status_message)..... | 35 |
| Reshape zTensor (zdnn_reshape_ztensor)..... | 36 |
| Check whether version is runnable (zdnn_is_version_runnable)..... | 37 |
| Get maximum runnable version (zdnn_get_max_runnable_version)..... | 38 |
| Data transformation..... | 39 |
| Transform to zTensor (zdnn_transform_ztensor)..... | 39 |
| Transform to zTensor with saturation (zdnn_transform_ztensor_with_saturation)..... | 41 |
| Transform to quantized zTensor (zdnn_transform_quantized_ztensor)..... | 42 |
| Transform to original (zdnn_transform_origtensor)..... | 44 |
| Operations..... | 45 |
| Element-wise operations..... | 45 |
| Activation operations..... | 55 |
| Normalization operations..... | 63 |
| Matmul and matmul with broadcast..... | 72 |
| LSTM (zdnn_lstm)..... | 82 |
| GRU (zdnn_gru)..... | 86 |
| Average pool 2D (zdnn_avgpool2d)..... | 90 |
| Max pool 2D (zdnn_maxpool2d)..... | 93 |
| Convolution 2D (zdnn_conv2d)..... | 95 |
| Convenience functions..... | 98 |
| Chapter 3. zDNN usage examples..... | 99 |
| Part 2. IBM Z Artificial Intelligence Optimization library..... | 129 |
| Chapter 4. Using the IBM Z Artificial Intelligence Optimization Library..... | 131 |
| IBM Z Artificial Intelligence Optimization Library environment..... | 132 |
| IBM Z Artificial Intelligence Optimization code development..... | 132 |
| IBM Z Artificial Intelligence Optimization execution..... | 132 |
| zAIO API return status..... | 133 |
| Chapter 5. IBM Z Artificial Intelligence Optimization Library API reference..... | 135 |
| zAIO initialization (zaio_Init)..... | 135 |
| Check availability of the IBM Z Integrated Accelerator for AI (zaio_zaiuReady)..... | 136 |
| Check CBLAS availability (zaio_cblasReady)..... | 136 |
| Get library version (zaio_getVersion)..... | 136 |
| Copy vector to new location (zaio_vectorCopy)..... | 137 |
| Average vector (zaio_averageVector)..... | 137 |
| Semantic average (zaio_semanticAverage)..... | 138 |
| Dot product (zaio_dotProduct)..... | 139 |
| Cosine distance (zaio_cosineDistance)..... | 139 |
| Vector normalization (zaio_normalize)..... | 140 |
| Vector denormalization (zaio_denormalize)..... | 141 |
| Vector absolute (zaio_absolute)..... | 141 |
| Vector scale (zaio_vectorScale)..... | 142 |
| Matrix-vector multiplication (zaio_matrixVector)..... | 143 |
| Matrix-matrix multiplication (zaio_matrixMatrix)..... | 143 |
| Matrix transpose (zaio_transpose)..... | 144 |
| Semantic similarity (zaio_semanticSimilarity)..... | 144 |
| Semantic clustering (zaio_semanticClustering)..... | 145 |
| Semantic analogy (zaio_semanticAnalogy)..... | 146 |
| Prefetching initialize (zaio_preFetching_Initialize)..... | 147 |
| Prefetching execute (zaio_preFetching_Execute)..... | 148 |
| Prefetching clear (zaio_preFetching_Clear)..... | 149 |

| | |
|---|------------|
| Chapter 6. Examples of using the IBM Z Artificial Intelligence Optimization Library APIs..... | 151 |
| Part 3. IBM Z Artificial Intelligence Data Embedding library..... | 155 |
| Chapter 7. Using the IBM Z Artificial Intelligence Data Embedding Library..... | 157 |
| IBM Z Artificial Intelligence Data Embedding Library environment..... | 158 |
| IBM Z Artificial Intelligence Data Embedding Library permissions..... | 158 |
| Chapter 8. IBM Z Artificial Intelligence Data Embedding Library API reference..... | 159 |
| Set functions..... | 159 |
| setNumThreads..... | 159 |
| setModelFormat..... | 159 |
| setVocabFormat..... | 160 |
| Get functions..... | 160 |
| getVersion..... | 160 |
| Operations..... | 161 |
| base10Cluster..... | 161 |
| train..... | 162 |
| Legacy operation functions..... | 163 |
| callBase10Cluster..... | 163 |
| callTrainDb2Vec..... | 163 |
| Chapter 9. Examples of using the IBM Z Artificial Intelligence Data Embedding Library APIs..... | 165 |
| Chapter 10. IBM Z Artificial Intelligence Data Embedding Library log files..... | 167 |
| Chapter 11. Troubleshooting the IBM Z Artificial Intelligence Data Embedding Library..... | 169 |
| Accessibility..... | 171 |
| Notices..... | 173 |
| Terms and conditions for product documentation..... | 174 |
| IBM Online Privacy Statement..... | 175 |
| Policy for unsupported hardware..... | 175 |
| Minimum supported hardware..... | 175 |
| Trademarks..... | 176 |
| Index..... | 177 |

Figures

1. zDNN tensor structure.....5

2. zDNN tensor descriptor..... 6

3. zDNN data layouts..... 7

4. zDNN data formats..... 7

5. zDNN data types..... 8

6. zDNN quantized transform types..... 8

7. IBM Z Artificial Intelligence Optimization Library framework..... 131

Tables

| | |
|---|-----|
| 1. zDNN success status..... | 8 |
| 2. zDNN warning statuses..... | 8 |
| 3. zDNN general failing statuses..... | 9 |
| 4. zDNN hardware statuses..... | 9 |
| 5. zDNN function-specific hardware statuses..... | 10 |
| 6. zDNN runtime environment variables..... | 10 |
| 7. Requirements for pre_transformed_desc and shape for matmul tensors..... | 73 |
| 8. Requirements for pre_transformed_desc and shape for matmul broadcast tensors..... | 74 |
| 9. Requirements for pre_transformed_desc and shape for matmul transpose tensors..... | 77 |
| 10. Requirements for pre_transformed_desc data layout and shape for quantized matmul tensors..... | 80 |
| 11. Requirements for zdnn_quantized_transform_types for quantized matmul tensors..... | 80 |
| 12. Summary of zdnn_lstm parameters..... | 85 |
| 13. Summary of zdnn_gru parameters..... | 89 |
| 14. Summary of requirements for convolution 2D operations..... | 97 |
| 15. zAIO supported hardware accelerations..... | 133 |
| 16. zAIO return code mnemonic constants..... | 133 |

About this information

IBM Z® Deep Neural Network Library (zDNN) provides high-level libraries that enable frameworks and model compilers to use the IBM Z on-chip AI accelerator, known as the IBM Z Integrated Accelerator for AI.

zDNN provides the following benefits to simplify access to the IBM Z Integrated Accelerator for AI:

- A high-level language application programming interface (API) for supported IBM Z Integrated Accelerator for AI primitives
- A unified approach for tensor data layout and type conversion
- Change management and version control mechanisms for IBM Z Integrated Accelerator for AI feature-level compatibility

This document contains information and reference material about the zDNN API, which is a standard C library.

This document also includes information and reference material about the IBM Z Artificial Intelligence Optimization Library and the IBM Z Artificial Intelligence Data Embedding Library.

- The IBM Z Artificial Intelligence Optimization Library provides a set of APIs that offer a seamless interface to various methods of hardware acceleration for programs that use semantic-based mathematical operations designed around vector and matrix operations.
- The IBM Z Artificial Intelligence Data Embedding Library provides a set of APIs to access a collection of packages designed to build and process vector embedding models on z/OS.

Who should read this information

This information is intended for use by application developers who wish to use the zDNN, IBM Z Artificial Intelligence Optimization Library, and IBM Z Artificial Intelligence Data Embedding Library APIs to create AI applications.

Related information

For additional information about z/OS, see [*z/OS Information Roadmap*](#).

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. For more information, see [How to send feedback to IBM](#).

Summary of changes

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line to the left of the change.

Note: IBM z/OS policy for the integration of service information into the z/OS product documentation library is documented on the z/OS Internet Library under [IBM z/OS Product Documentation Update Policy](http://www.ibm.com/docs/en/zos/latest?topic=zos-product-documentation-update-policy) (www.ibm.com/docs/en/zos/latest?topic=zos-product-documentation-update-policy).

Summary of changes for z/OS 3.2

This information contains no technical changes for this release.

Summary of changes for z/OS 3.1

The following content is new, changed, or no longer included in z/OS 3.1.

New

The following content is new.

June 2025 refresh

- The zDNN library is updated to version 1.1.0 and adds support for IBM z17™ with the IBM Telum II™ processor. (APAR OA66863, which also applies to z/OS 2.5)
 - New zDNN APIs are added to exploit additional functionality available with the Telum II processor. See [Chapter 2, “zDNN API reference,”](#) on page 13.
 - Support is added for quantized zTensors.
 - All zDNN APIs now indicate the level of the zDNN library that they require.

May 2024 refresh

- New functions are added in [Chapter 8, “IBM Z Artificial Intelligence Data Embedding Library API reference,”](#) on page 159. (APAR OA66165 for z/OS 3.1; OA66166 for z/OS 2.5)

September 2023 release

- None.

Changed

The following content is changed.

May 2024 refresh

- The library paths are updated for z/OS 3.1 in [“IBM Z Artificial Intelligence Optimization Library environment”](#) on page 132 and [“IBM Z Artificial Intelligence Data Embedding Library environment”](#) on page 158.
- Updates are made throughout [“Part 3. IBM Z Artificial Intelligence Data Embedding library.”](#) (APAR OA66165 for z/OS 3.1; OA66166 for z/OS 2.5)

September 2023 release

- The installation paths are updated in [“IBM Z Artificial Intelligence Data Embedding Library environment”](#) on page 158.

Deleted

The following content was deleted.

September 2023 release

- None.

Part 1. IBM Z Deep Neural Network library

Chapter 1. Using the IBM Z Deep Neural Network Library

IBM Z Deep Neural Network Library (zDNN) provides high-level libraries that enable frameworks and model compilers to use the IBM Z Integrated Accelerator for AI (zAIU) on IBM z16® and later CPCs.

The zDNN deep learning library support is the software enablement technology that IBM provides to meet the following requirements:

- Specialized-function-assist instructions are intended to provide performance improvements for specific operations used in software libraries, utilities, and operating system (OS) services. The facilities and instructions described as specialized-function-assist instructions may be replaced or removed in the future. As such, IBM recommends that a software library or operating system function be used instead of directly accessing the instructions. This is the function provided by zDNN.
- The zAIU has complex requirements for the layout of data; these requirements arrange the tensor to enhance the performance characteristics of the operations. The zDNN library functions format the tensor appropriately on behalf of the caller using an optimized approach.
- For deep learning operations, zAIU requires the use of these internal data types:

DLFLOAT16

A 2-byte data type supported on CPCs based on IBM Telum® processors (such as IBM z16®) which optimizes training and inferencing while minimizing the loss of accuracy at inference time (in contrast to standard 4-byte formats).

INT8

A 1-byte data type supported on CPCs based on IBM Telum II™ processors (such as IBM z17™) which allows tensor quantization features.

The zDNN library provides a set of APIs that an exploiter can use to drive the desired request. zDNN is available on IBM z/OS and Linux® on IBM Z. The inclusion of Linux on IBM Z provides particular benefit, as it enables acceleration in frameworks for z/OS using z/OS Container Extensions (zCX).

zDNN application environment

The overall z/OS environment for zDNN applications is:

- Problem state
- AMODE 64
- XPLINK

Alignment requirements

The following information describes how zDNN and your application align with the requirements and limits of the IBM Z Integrated Accelerator for AI (zAIU).

zAIU operation limits

The zAIU operation (op) limits (which also imply corresponding zDNN limitations) for all ops are:

- Number of elements in any dimension must not exceed the value returned by the **zdn_get_max_for_dim** function.
- Total number of bytes required for storing a transformed tensor must not exceed the value returned by the **zdn_get_nnpa_max_tensor_size** function.

Application interfaces for enterprise neural network inference

The following information describes the interfaces necessary for your application to access the zAIU.

zDNN general

The zDNN deep learning library provides the standard IBM Z software interface to the zAIU. This IBM-provided C library provides a set of functions that handle the data transformation requirements of the zAIU and provides wrapper functions for the Neural Network Processing Assist (NNPA) instruction primitives.

The zDNN functions use the following criteria to determine if the IBM Z Integrated Accelerator for AI can be used to accelerate a deep learning primitive:

- Neural Network Processing Assist (NNPA) facility indicator in the system STFLE output
- Output of the NNPA-QAF (Query Available Functions) request

Using zDNN

Complete this task to use the zDNN C library for the NNPA instruction.

Link or re-link applications to use the zDNN library. The zDNN library is a library file in the z/OS UNIX System Services file system and can be linked statically or dynamically into your applications.

For z/OS (which requires z/OS Language Environment® (LE)):

- The paths for the zDNN 64-bit dynamic library files are:
 - /lib/libzdnn.so
 - /lib/libzdnn.x
- The path for the zDNN header files is /usr/include/.
- The XL C/C++ compiler and Language Environment provide various environment variables to control processing, in addition to the variables provided by the zDNN library itself.
 - Use the **_CEE_RUNOPTS** environment variable to specify invocation LE runtime options. For more information about using the **_CEE_RUNOPTS** environment variable and other C and LE variables, see *z/OS XL C/C++ Programming Guide*.
 - For environment variables that are accepted by the zDNN library, see [“zDNN runtime environment variables” on page 10](#).

Common data types and structures

The zDNN common data types and structures are defined in the `zdnn.h` include file.

zDNN version information

```
#define ZDNN_VERSION "1.1.0"
#define ZDNN_VERNUM 0x010100 // 0x[major][minor][patch]
#define ZDNN_VER_MAJOR 1
#define ZDNN_VER_MINOR 1
#define ZDNN_VER_PATCH 0
```

ZDNN_VER_MAJOR

The zDNN major version is incremented if any changes that are not compatible with earlier versions are introduced to the API. Such a change may also include minor and patch level changes. Minor and patch versions are reset to 0 when the major version is incremented.

ZDNN_VER_MINOR

The zDNN minor version is incremented if new functionality that is compatible with earlier versions is introduced to the API or if any API functionality is marked as deprecated. Such a change may also include patch level changes. The patch version is reset to 0 when the minor version is incremented.

ZDNN_VER_PATCH

The zDNN patch version is incremented if only bug fixes that are compatible with earlier versions are introduced. A *bug fix* is defined as an internal change that fixes incorrect behavior.

Functions for checking version compatibility with the zDNN load library are provided and described in [“Support functions” on page 13](#).

zDNN tensor

The structure of a zDNN tensor (zTensor) is defined by the **zdnntensor** structure, as shown in [Figure 1 on page 5](#).

```
typedef struct zdnntensor {
    zdnntensor_desc
        *pre_transformed_desc; // tensor's shape information before transformation
    zdnntensor_desc *transformed_desc; // transformed tensor's shape information
    uint64_t buffer_size;           // tensor size in bytes
    void *buffer;                   // pointer to the tensor in memory
    bool is_transformed; // indicator if data in buffer has been transformed
    char reserved[3];      // not currently used, should contain zeros.
    float rec_scale;       // the scale factor for quantization, stored as reciprocal
    float offset;          // the offset for quantization
    char reserved2[20];    // not currently used, should contain zeros.
} zdnntensor;
```

Figure 1. zDNN tensor structure

General zTensor requirements

Observe the following general requirements for a zTensor:

- Requirements for the *buffer* field:
 - The **zdnntensor_init_with_malloc** function automatically allocates and sets a valid *buffer* value for a tensor.
 - The *buffer* field must point to storage that is allocated of a sufficient size to contain the transformed tensor data described by its *transformed_desc* field. Call the **zdnntensor_getsize** function with the tensor's *transformed_desc* value to obtain the required size.
 - The start of the buffer must be 4K-aligned. That is, the *buffer* value must point to a storage address that is on a 4K boundary.
- The *reserved* field should contain zeros; otherwise, the program might not operate compatibly in the future. Calling **zdnntensor_init** or **zdnntensor_init_with_malloc** sets *reserved* to zeros.

Concatenated zTensor requirements

Observe the following requirements for a concatenated zTensor.

- For use with weights, biases, hidden-weights, hidden-biases RNN-gates tensors.
- Follow the requirements in “General zTensor requirements” on page 5.
- Use the **zdnntensor_generate_transformed_desc_concatenated** function with the appropriate concatenation information.

Do not use **zdnntensor_generate_transformed_desc** with concatenated tensors.

- The pre-transformed shape dimensions must not include the concatenation.

Thus, the pre-transformed shape should be that of a single gate, not the shape of the combined gates.

- Afterward, transform with the **zdnntensor_transform** function, as normal.
- Follow the requirements in [“General zTensor requirements” on page 5](#).

Quantized zTensor requirements

Observe the following requirements for a quantized zTensor.

The supported *transform_desc* and *pre_transformed_desc* types for the **zdnntensor_transform_quantized** and **zdnntensor_generate_quantized_transformed_desc** functions are:

- ZDNN_FORMAT_4DFEATURE format:
 - ZDNN_DLFLOAT16: FP16, FP32, BFLOAT
 - ZDNN_BINARY_INT8: INT8, FP16, FP32, BFLOAT
- ZDNN_FORMAT_4DWEIGHTS format:
 - ZDNN_BINARY_INT8: INT8

zDNN tensor descriptor

The structure of a zDNN tensor descriptor is defined by the **zdnntensor_desc** structure, as shown in [Figure 2 on page 6](#).

```
typedef struct zdnntensor_desc {
    zdnndata_layouts layout; // data layout
    zdnndata_formats format; // internal use only
    zdnndata_types type;     // data type
    uint32_t dim4;           // number of elements in outermost dimension
    uint32_t dim3;           // ... outer dimension
    uint32_t dim2;           // ... inner dimension
    uint32_t dim1;           // number of elements in innermost dimension
} zdnntensor_desc;
```

Figure 2. zDNN tensor descriptor

Programming notes

- Helper methods **zdnntensor_init_pre_transformed_desc** and **zdnntensor_generate_transformed_desc** or **zdnntensor_generate_transformed_desc_concatenated** will set the correct dimensions based on the layout and format.
- The layout of the tensor descriptor affects the expected order of the dimensions.
 - For tensors with less than four dimensions:
 - Unspecified dimensions in the *pre_transformed_desc* are ignored. For instance, a ZDNN_3D layout expects values in *dim4*, *dim3*, and *dim2*.
 - In the *transformed_desc*, "unused" dimensions must be 1.
 - A ZDNN_NHWC layout expects dimensions such that *dim4*=N, *dim3*=H, *dim2*=W, and *dim1*=C.
 - A ZDNN_NCHW layout expects dimensions such that *dim4*=N, *dim3*=C, *dim2*=H, *dim1*=W.
 - A ZDNN_HWCK layout expects dimensions such that *dim4*=H, *dim3*=W, *dim2*=C, and *dim1*=K.
- The format changes the expected dimension order for ZDNN_4D tensor layouts:
 - ZDNN_FORMAT_4DFEATURE expects dimensions such that *dim4*=N, *dim3*=H, *dim2*=W, and *dim1*=C.
 - ZDNN_FORMAT_4DKERNEL expects dimensions such that *dim4*=H, *dim3*=W, *dim2*=C, and *dim1*=K.

zDNN data layouts

The zDNN data layouts for zTensor descriptors are defined in the **zdnndata_layouts** enumeration, as shown in [Figure 3 on page 7](#). The layouts indicate the number and order of dimensions for the zTensor data.

```
typedef enum zdnn_data_layouts {
    ZDNN_1D,          // 1d tensor
    ZDNN_2D,          // 2d tensor
    ZDNN_2DS,         // represents special 2D tensors required by LSTM/GRU
    ZDNN_3D,          // 3d tensor
    ZDNN_3DS,         // represents special 3D tensors required by
                      // LSTM, GRU, Softmax, and Matmul
    ZDNN_ZRH,         // represents (update, reset, hidden) used by GRU
    ZDNN_4D,          // 4d tensor
    ZDNN_4DS,         // represents special 4D tensors required by LSTM/GRU output
    ZDNN_NHWC,        // 4d feature tensor in NHWC
    ZDNN_NCHW,        // 4d feature tensor in NCHW
    ZDNN_FICO,        // represents (forget, input, cell, output) used by LSTM
    ZDNN_CNNK_HWCK,   // 4d kernel CNN tensor
    ZDNN_BIDIR_ZRH,   // ZRH variant to work with bidirectional LSTM/GRU output
    ZDNN_BIDIR_FICO   // FICO variant to work with bidirectional LSTM/GRU output
} zdnn_data_layouts;
```

Figure 3. zDNN data layouts

Some layouts also indicate special rearrangement of the data during zTensor transformation.

ZDNN_2DS

The outermost dimension of the original shape is promoted to dim4 during transformation. For instance, a shape of (a, b) becomes [a, 1, 1, b] (dim4, dim3, dim2, dim1) in the *transformed_desc*.

ZDNN_3DS

The outermost dimension of the original shape is promoted to dim4 during transformation. For instance, a shape of (a, b, c) becomes [a, 1, b, c] (dim4, dim3, dim2, dim1) in the *transformed_desc*.

ZDNN_4DS

Arrangement for RNN output tensor.

The following layouts are set automatically in *transformed_desc* based on *info* when calling the **zdnn_generate_transformed_desc_concatenated** function:

ZDNN_ZRH / ZDNN_FICO

During transformation, the RNN input gates are concatenated on the innermost dimension. These layouts are supported with **pre_transformed_layout** of ZDNN_2DS or ZDNN_3DS.

ZDNN_BIDIR_ZRH / ZDNN_BIDIR_FICO

These layouts are similar to **ZDNN_ZRH** and **ZDNN_FICO** and are used when transforming RNN input weight gate data and the input tensor for the current RNN layer is a bidirectional RNN output from a previous RNN layer.

zDNN data formats

The zDNN data formats are defined in the **zdnn_data_formats** enumeration, as shown in [Figure 4 on page 7](#).

```
typedef enum zdnn_data_formats {
    ZDNN_FORMAT_4DFEATURE, // tensor in zAIU data layout format 0
    ZDNN_FORMAT_4DKERNEL,  // tensor in zAIU data layout format 1
    ZDNN_FORMAT_4DWEIGHTS, // tensor in zAIU data layout format 2
    ZDNN_FORMAT_4DGENERIC, // tensor in zAIU data layout format 31
} zdnn_data_formats;
```

Figure 4. zDNN data formats

zDNN data types

The zDNN data types are defined in the **zdnn_data_types** enumeration, as shown in [Figure 5 on page 8](#).

```
typedef enum zdnn_data_types {
    ZDNN_DLFLOAT16, // 16-bit deep learning format
    ZDNN_BINARY_FP32, // 32-bit binary-floating-point format
    ZDNN_BINARY_INT8, // 8-bit signed or unsigned binary integer
    ZDNN_BINARY_INT32, // 32-bit signed or unsigned binary integer
    INT8, // 8-bit signed or unsigned binary integer format
    INT32, // 32-bit signed or unsigned binary integer format
    BFLOAT, // Brain floating point format
    FP16, // 16-bit IEEE-754 floating point format
    FP32, // 32-bit IEEE-754 floating point format
} zdnn_data_types;
```

Figure 5. zDNN data types

Quantized transform types

The zDNN quantized transform types are defined in the **zdnn_quantized_transform_types** enumeration, as shown in Figure 6 on page 8.

```
typedef enum zdnn_quantized_transform_types {
    QUANTIZED_DLFLOAT16 = 0, // quantized dlfloating16
    QUANTIZED_INT8 = 1, // quantized int8
    QUANTIZED_WEIGHTS_INT8 = 2 // quantized weights
} zdnn_quantized_transform_types;
```

Figure 6. zDNN quantized transform types

zDNN statuses

Success status

Table 1 on page 8 lists the success status returned from the zDNN library.

Table 1. zDNN success status

| Mnemonic constant | Value | Meaning |
|-------------------|------------|----------|
| ZDNN_OK | 0x00000000 | Success. |

Warning statuses

Table 2 on page 8 lists the warning statuses returned from the zDNN library.

Table 2. zDNN warning statuses

| Mnemonic constant | Value | Meaning |
|------------------------------|------------|---|
| ZDNN_ELEMENT_RANGE_VIOLATION | 0x00020001 | IBM Z Integrated Accelerator for AI (zAIU) operation resulted in data that was out of the normal range. |

The **ZDNN_ELEMENT_RANGE_VIOLATION** status indicates that a *range violation* occurred for the zAIU operation based on the data in the tensors. This usually indicates an overflow of an NNPA internal data type, but it can also be associated with operation-specific errors, such as "divide by zero." See the appropriate edition of *z/Architecture Principles of Operation* for your hardware model for information about the range violation on the operation that encountered the violation.

General failing statuses

Table 3 on page 9 lists the general failing statuses returned from the zDNN library.

Note: Statuses marked with an asterisk (*) indicate that, in certain scenarios, these statuses are returned only if the **ZDNN_ENABLE_PRECHECK** environment variable is enabled. When **ZDNN_ENABLE_PRECHECK** is not enabled, these scenarios will lead to abnormal program termination.

Table 3. zDNN general failing statuses

| Mnemonic constant | Value | Meaning |
|--------------------------------|------------|---|
| ZDNN_INVALID_SHAPE* | 0x00040001 | Invalid shape information in one or more of the input or output tensors. |
| ZDNN_INVALID_LAYOUT | 0x00040002 | Invalid layout information in one or more of the input or output tensors. |
| ZDNN_INVALID_TYPE* | 0x00040003 | Invalid type information in one or more of the input or output tensors. |
| ZDNN_INVALID_FORMAT* | 0x00040004 | Invalid format information in one or more of the input or output tensors. |
| ZDNN_INVALID_DIRECTION | 0x00040005 | Invalid RNN direction. |
| ZDNN_INVALID_CONCAT_INFO | 0x00040006 | Invalid concatenation information. |
| ZDNN_INVALID_STRIDE_PADDING* | 0x00040007 | Invalid padding type parameter for current strides. |
| ZDNN_INVALID_STRIDES* | 0x00040008 | Invalid stride height or width parameter. |
| ZDNN_MISALIGNED_PARMBLOCK* | 0x00040009 | NNPA parameter block is not on a doubleword boundary. |
| ZDNN_INVALID_CLIPPING_VALUE | 0x0004000A | Invalid clipping for the specified operation. |
| ZDNN_INVALID_ADJUSTMENT_FACTOR | 0x0004000B | Invalid adjustment for the specified operation. |
| ZDNN_INVALID_EPSILON | 0x0004000C | Invalid epsilon for the specified operation. |
| ZDNN_INVALID_TRANSFORM_TYPE | 0x0004000D | Invalid transformation type. |
| ZDNN_INVALID_BETA | 0x0004000E | Invalid beta value for the specified operation. |
| ZDNN_INVALID_GAMMA | 0x0004000F | Invalid gamma value for the specified operation. |
| ZDNN_INVALID_BESSEL_CORRECTION | 0x00040010 | Invalid bessel correction value for the specified operation. |
| ZDNN_INVALID_SCALE | 0x00040011 | Invalid scale value for the specified operation. |
| ZDNN_INVALID_OFFSET | 0x00040012 | Invalid offset value for the specified operation. |
| ZDNN_ALLOCATION_FAILURE | 0x00100001 | Cannot allocate storage. |
| ZDNN_INVALID_BUFFER | 0x00100002 | Buffer address is NULL or not on a 4K-byte boundary, or insufficient buffer size. |
| ZDNN_CONVERT_FAILURE | 0x00100003 | Floating point data conversion failure. |
| ZDNN_INVALID_STATE | 0x00100004 | Invalid zTensor state. |
| ZDNN_UNSUPPORTED_AIU_EXCEPTION | 0x00100005 | A zAIU operation returned an unexpected exception. |

Hardware statuses

Table 4 on page 9 lists statuses that are returned from the hardware.

Table 4. zDNN hardware statuses

| Status | Value | Meaning |
|----------------------------|------------|---|
| ZDNN_UNSUPPORTED_PARMBLOCK | 0x000C0001 | NNPA parameter block format is not supported by the model. |
| ZDNN_UNAVAILABLE_FUNCTION | 0x000C0002 | Specified NNPA function is not defined or installed on the machine. |
| ZDNN_UNSUPPORTED_FORMAT | 0x000C0010 | Specified tensor data layout format is not supported. |
| ZDNN_UNSUPPORTED_TYPE | 0x000C0011 | Specified tensor data type is not supported. |
| ZDNN_EXCEEDS_MDIS | 0x000C0012 | Tensor dimension exceeds maximum dimension index size (MDIS). |
| ZDNN_EXCEEDS_MTS | 0x000C0013 | Total number of bytes in tensor exceeds maximum tensor size (MTS). |
| ZDNN_MISALIGNED_TENSOR | 0x000C0014 | Tensor address is not on a 4K-byte boundary. |

Table 4. zDNN hardware statuses (continued)

| Status | Value | Meaning |
|--------------------------|------------|---|
| ZDNN_MISALIGNED_SAVEAREA | 0x000C0015 | Function-specific save area address is not on a 4K-byte boundary. |

Table 5 on page 10 lists hardware statuses whose meanings vary based on operation. See the description of the operation that returned the status for the specific meaning.

Table 5. zDNN function-specific hardware statuses

| Status | Value | Meaning |
|-------------------|------------|---|
| ZDNN_FUNC_RC_F000 | 0x000CF000 | Function-specific response code (F000). |
| ZDNN_FUNC_RC_F001 | 0x000CF001 | Function-specific response code (F001). |
| ZDNN_FUNC_RC_F002 | 0x000CF002 | Function-specific response code (F002). |
| ZDNN_FUNC_RC_F003 | 0x000CF003 | Function-specific response code (F003). |
| ZDNN_FUNC_RC_F004 | 0x000CF004 | Function-specific response code (F004). |
| ZDNN_FUNC_RC_F005 | 0x000CF005 | Function-specific response code (F005). |
| ZDNN_FUNC_RC_F006 | 0x000CF006 | Function-specific response code (F006). |
| ZDNN_FUNC_RC_F007 | 0x000CF007 | Function-specific response code (F007). |
| ZDNN_FUNC_RC_F008 | 0x000CF008 | Function-specific response code (F008). |
| ZDNN_FUNC_RC_F009 | 0x000CF009 | Function-specific response code (F009). |

zDNN runtime environment variables

The following tables list the zDNN runtime environment variables.

Table 6. zDNN runtime environment variables

| Variable name | Valid values | Description |
|-----------------------------|--|---|
| ZDNN_ENABLE_PRECHECK | true false | When set to true, tensor integrity prechecks are run before issuing NNPA operations. <ul style="list-style-type: none"> Enabling precheck can impact performance. Enable precheck to debug issues that cause hardware exceptions that otherwise would result in abnormal program termination. |
| ZDNN_STATUS_DIAG | nnnnnnnn (decimal) 0xnnnnnnnn (hexadecimal) | Prints or produces diagnostic information whenever the zDNN status code equals the specified value. Only one status value can be specified. |

Programming notes

- Environment variable settings are checked during initial library load by the **zdn_init** function.
- To change environment variable settings after the library is loaded, you must manually call **zdn_init** again.

Validating the environment at run time

Most API calls require a minimum level of the zDNN library and hardware for the API to function. There are 3 zDNN APIs for validation of the zDNN runtime environment:

- **Validating the zDNN library version**

The zDNN library version is the version of the libzdnn package that is installed on the host or embedded in the runtime application.

The zDNN library version is independent of the hardware available on the current system. zDNN APIs that are introduced in later versions of the zDNN library will not exist in earlier versions of the library; attempting to call them will result in application crashes.

The zDNN library version is returned by the **zdnn_get_library_version** function.

- **Validating the zDNN API version**

The zDNN API version is the version of zDNN APIs that are compatible on the current system and is separate from the zDNN library version. Calling zDNN APIs while running on a system that does not support that API version will return a hardware status (see “Hardware statuses” on page 9) instead of ZDNN_OK.

The zDNN API version that is available is returned by **zdnn_get_max_runnable_version** and is reflected in the return value of **zdnn_is_version_runnable**.

- zDNN API 1.0.x indicates that the API requires IBM Telum processors or later.
- zDNN API 1.1.x indicates that the API requires IBM Telum II™ processors or later.

- **Validating NNPA availability**

NNPA availability indicates whether the current system has IBM Z Integrated Accelerator for AI (zAIU) hardware present and enabled. It is possible to be running on a system with zAIU hardware, but the feature is unavailable, such as on z/VM® when there is a mix of hardware levels.

This hardware availability is returned by the **zdnn_is_nnpa_installed** function.

Examples

- Given an IBM Telum based system with zDNN 1.1.0 installed:
 - The **zdnn_get_library_version** function returns 0x00010100 indicating that zDNN library 1.1.0 is installed.
 - The **zdnn_is_nnpa_installed** function returns true (unless the zAIU feature is disabled for the system).
 - The **zdnn_get_max_runnable_version** function returns 0x000100FF indicating that zDNN APIs 1.0.x and earlier are available for use on the system.
 - Calling **zdnn_is_version_runnable(0x00010100)** (1.1.0) returns false because only zDNN APIs 1.0.x and earlier are available for use on the system.
 - Calling **zdnn_is_version_runnable(0x00010000)** (1.0.0) returns true because zDNN APIs 1.0.x and earlier are available for use on the system.
- Given an IBM Telum II™ based system with zDNN 1.1.0 installed:
 - The **zdnn_get_library_version** function returns 0x00010100 indicating that zDNN library 1.1.0 is installed.
 - The **zdnn_is_nnpa_installed** function returns true (unless the zAIU feature is disabled for the system).
 - The **zdnn_get_max_runnable_version** function returns 0x000101FF indicating that zDNN APIs 1.1.x and earlier are available for use on the system.
 - Calling **zdnn_is_version_runnable(0x00010100)** (1.1.0) returns true because zDNN APIs 1.1.x and earlier are available for use on the system.

- Calling `zdsn_is_version_runnable(0x00010000)` (1.0.0) returns `true` because zDNN APIs 1.1.x and earlier are available for use on the system.
- Given an IBM Telum II™ based system with zDNN 1.0.0 installed:
 - The **`zdsn_get_library_version`** function returns `0x00010000` indicating that zDNN library 1.0.0 is installed.
 - The **`zdsn_is_nnpa_installed`** function returns `true` (unless the zAIU feature is disabled for the system).
 - The **`zdsn_get_max_runnable_version`** function returns `0x000100FF` indicating that zDNN APIs 1.0.x and earlier are available for use on the system.
 - Calling `zdsn_is_version_runnable(0x00010100)` (1.1.0) returns `false` as only zDNN APIs 1.0.x and earlier are available for use on the system.
 - Calling `zdsn_is_version_runnable(0x00010000)` (1.0.0) returns `true` as zDNN APIs 1.0.x and earlier are available for use on the system.

Chapter 2. zDNN API reference

The zDNN APIs are grouped into the following categories:

- [“Support functions” on page 13](#)
- [“Data transformation” on page 39](#)
- [“Operations” on page 45](#)
- [“Convenience functions” on page 98](#)

Support functions

The zDNN support functions are:

- [“Initialization \(zdn_init\)” on page 13](#)
- [“Query functions” on page 14](#)
- [“Get size \(zdn_getsize_ztensor\)” on page 22](#)
- [“Get range \(zdn_getrange_ztensor\)” on page 23](#)
- [“Get maximum limit for a data type \(zdn_get_max_limit\)” on page 23](#)
- [“Get minimum limit for a data type \(zdn_get_min_limit\)” on page 24](#)
- [“Initialize pre-transformed tensor descriptor \(zdn_init_pre_transformed_desc\)” on page 25](#)
- [“Generate transformed tensor descriptor \(zdn_generate_transformed_desc\)” on page 26](#)
- [“Generate quantized transformed tensor descriptor \(zdn_generate_quantized_transformed_desc\)” on page 26](#)
- [“Generate concatenated transformed tensor descriptor \(zdn_generate_transformed_desc_concatenated\)” on page 28](#)
- [“Initialize zTensor \(zdn_init_ztensor\)” on page 29](#)
- [“Initialize zTensor with memory allocate \(zdn_init_ztensor_with_malloc\)” on page 29](#)
- [“Initialize quantized zTensor \(zdn_init_quantized_ztensor\)” on page 30](#)
- [“Initialize quantized zTensor with memory allocate \(zdn_init_quantized_ztensor_with_malloc\)” on page 31](#)
- [“Determine quantized zTensor \(zdn_is_quantized_ztensor\)” on page 32](#)
- [“Reset zTensor \(zdn_reset_ztensor\)” on page 33](#)
- [“Allocate memory for zTensor \(zdn_allochelper_ztensor\)” on page 34](#)
- [“Deallocate memory for zTensor \(zdn_free_ztensor_buffer\)” on page 34](#)
- [“Retrieve status message for a status code \(zdn_get_status_message\)” on page 35](#)
- [“Reshape zTensor \(zdn_reshape_ztensor\)” on page 36](#)
- [“Check whether version is runnable \(zdn_is_version_runnable\)” on page 37](#)
- [“Get maximum runnable version \(zdn_get_max_runnable_version\)” on page 38](#)

Initialization (zdn_init)

Description

Initialize the zDNN library. This sends an NNPA-QAF to query the NNPA and loads the current environment variable settings.

This must be invoked at least once if the zDNN library is statically-linked. It is automatically invoked if the zDNN library is dynamically loaded.

Format

```
void zdnn_init();
```

Parameters

None.

Returns

None.

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Query functions

The zDNN query functions are:

- [“Get smallest of the maximum index size values across all dimensions \(zdnn_get_nnpa_max_dim_idx_size\)” on page 14](#)
- [“Get maximum index for a given dimension \(zdnn_get_max_for_dim\)” on page 15](#)
- [“Maximum tensor size \(zdnn_get_nnpa_max_tensor_size\)” on page 16](#)
- [“NNPA function availability \(zdnn_is_nnpa_function_installed\)” on page 17](#)
- [“Availability of parameter block formats \(zdnn_is_nnpa_parmblk_fmt_installed\)” on page 18](#)
- [“Availability of NNPA data types \(zdnn_is_nnpa_datatype_installed\)” on page 18](#)
- [“Availability of NNPA data layout formats \(zdnn_is_nnpa_layout_fmt_installed\)” on page 19](#)
- [“Library version \(zdnn_get_library_version\)” on page 20](#)
- [“Library version string \(zdnn_get_library_version_str\)” on page 21](#)
- [“Availability of NNPA data type format conversions \(zdnn_is_nnpa_conversion_installed\)” on page 20](#)
- [“In-memory query result \(zdnn_refresh_nnpa_query_result\)” on page 21](#)

Get smallest of the maximum index size values across all dimensions (zdnn_get_nnpa_max_dim_idx_size)

Description

Retrieve the smallest of the maximum dimension index size values across all dimensions currently supported by the IBM Z Integrated Accelerator for AI from zDNN internal memory.

Format

```
uint32_t zdnn_get_nnpa_max_dim_idx_size();
```

Parameters

None.

Returns

Maximum dimension index size supported by the IBM Z Integrated Accelerator for AI across all dimensions.

Since

1.0.0.

Requirements

This function requires that:

- The **zdsn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Get maximum index for a given dimension (zdsn_get_max_for_dim)

Description

Retrieve the maximum dimension index size value currently supported by the IBM Z Integrated Accelerator for AI for a given dimension from zDNN internal memory. These limits relate to zTensor's transformed descriptor values. Special care is required when using layouts with special re-arrangements of data. See [“zDNN data layouts” on page 6](#) for details.

Format

```
uint32_t zdsn_get_max_for_dim(uint8_t dimension);
```

Parameters

uint8_t dimension

Dimension for which to get the maximum index size.

Returns

Maximum dimension index size supported by the IBM Z Integrated Accelerator for AI for a given dimension.

Since

1.1.0

Requirements

This function requires that:

- The **zdsn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Maximum tensor size (zdnng_get_nnpa_max_tensor_size)

Description

Retrieve the maximum tensor size value (number of bytes required for storing a transformed tensor) currently supported by the IBM Z Integrated Accelerator for AI from zDNN internal memory.

Format

```
uint64_t zdnng_get_nnpa_max_tensor_size();
```

Parameters

None.

Returns

Maximum tensor size supported by the IBM Z Integrated Accelerator for AI.

Since

1.0.0

Requirements

This function requires that:

- The **zdnng_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

NNPA availability (zdnng_is_nnpa_installed)

Description

Queries the hardware to determine if the NNPA and associated instructions are installed.

Use this function during application initialization to determine whether the IBM Z Integrated Accelerator for AI hardware is available.

Format

```
bool zdnng_is_nnpa_installed();
```

Parameters

None.

Returns

Returns `true` if NNPA and associated instructions are installed; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **zddn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

NNPA function availability (zddn_is_nnpa_function_installed)

Description

Query, from zDNN internal memory, whether requested NNPA functions are available.

Format

```
bool zddn_is_nnpa_function_installed(int count, ...);
```

Parameters

count

Number of NNPA functions to check.

... (additional arguments)

Function numbers, separated by commas. For instance: `NNPA_MUL, NNPA_MIN_`

The valid NNPA functions are:

| | |
|-------------------------|----------------|
| NNPA_ADD | NNPA_MAXPOOL2D |
| NNPA_AVGPOOL2D | NNPA_MIN |
| NNPA_BATCHNORMALIZATION | NNPA_MOMENTS |
| NNPA_CONVOLUTION | NNPA_MUL |
| NNPA_DIV | NNPA_NORM |
| NNPA_EXP | NNPA_QAF |
| NNPA_GELU | NNPA_REDUCE |
| NNPA_GRUACT | NNPA_RELU |
| NNPA_INVSQRT | NNPA_SIGMOID |
| NNPA_LAYERNORM | NNPA_SOFTMAX |
| NNPA_LOG | NNPA_SQRT |
| NNPA_LSTMACT | NNPA_SUB |
| NNPA_MATMUL_OP_BCAST1 | NNPA_TANH |
| NNPA_MATMUL_OP_BCAST23 | NNPA_TRANSFORM |
| NNPA_MATMUL_OP | NNPA_TRANSPOSE |
| NNPA_MAX | |

Returns

Returns `true` if all queried functions are installed or if *count* is zero; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **zdsn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Availability of parameter block formats (**zdsn_is_nnpa_parmblk_fmt_installed**)

Description

Query, from zDNN internal memory, whether requested parameter block formats are installed.

Format

```
bool zdsn_is_nnpa_parm_blk_fmt_installed(int count, ...);
```

Parameters

count

Number of NNPA parameter block formats to check.

... (additional arguments)

NNPA parameter block formats, separated by commas.

NNPA_PARMBLKFORMAT_0

NNPA_PARMBLKFORMAT_1

Returns

Returns `true` if all queried formats are installed or if *count* is zero; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **zdsn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Availability of NNPA data types (**zdsn_is_nnpa_datatype_installed**)

Description

Query, from zDNN internal memory, whether requested NNPA data types are installed.

Format

```
bool zdsn_is_nnpa_datatype_installed(uint16_t types_bitmask);
```

Parameters

uint16_t types_bitmask

The OR of the requested data type bit masks as defined in the **zdsn_query_datatypes** enumeration.

```
QUERY_DATATYPE_INTERNAL1
QUERY_DATATYPE_BINARY_FP32
QUERY_DATATYPE_BINARY_INT8
QUERY_DATATYPE_BINARY_INT32
```

Returns

Returns `true` if all queried data types are installed; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **`zdnnpa_installed`** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Availability of NNPA data layout formats (`zdnnpa_layout_fmt_installed`)

Description

Query, from zDNN internal memory, whether requested NNPA data layout formats are installed.

Format

```
bool zdnnpa_query_is_nnnpa_layout_fmt_installed(uint32_t layout_bitmask);
```

Parameters

`uint32_t layout_bitmask`

The OR of the requested layout bit masks as defined in the **`zdnnpa_layout_fmts`** enumeration.

```
QUERY_LAYOUTFMT_4DFEATURE
QUERY_LAYOUTFMT_4DKERNEL
QUERY_LAYOUTFMT_4DWEIGHTS
QUERY_LAYOUTFMT_4DGENERIC
```

Returns

Returns `true` if all queried data layouts are installed; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **`zdnnpa_installed`** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Availability of NNPA data type format conversions (`zdnm_is_nnpa_conversion_installed`)

Description

Query, from zDNN internal memory, whether requested NNPA data type to/from BFP format conversions are installed.

Format

```
bool zdnm_is_nnpa_conversion_installed(nnpa_data_type type,  
                                     uint16_t format_bitmask);
```

Parameters

`nnpa_data_type type`

The NNPA data type number as defined in the **`nnpa_data_type`** enumeration.

`NNPA_DATATYPE_1`

`uint16_t format_bitmask`

The OR of the BFP format bit masks as defined in the **`zdnm_query_bfpfmts`** enumeration.

`QUERY_BFPFMT_TINY (FP16)`

`QUERY_BFPFMT_SHORT (FP32/BFLOAT)`

Returns

Returns `true` if all queried conversions are installed; otherwise, returns `false`.

Since

1.0.0

Requirements

This function requires that:

- The **`zdnm_is_nnpa_installed`** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Library version (`zdnm_get_library_version`)

Description

Retrieve the library version as a 32-bit, hexadecimal value (`0x00MMmmp`), where:

`MM`

The major level

`mm`

The minor level

`pp`

The patch level

This is the version of the libzdnm package installed on the system or of zDNN embedded in a runtime application. The zDNN library version is independent of the system on which zDNN is running.

The library version indicates which zDNN APIs exist in that version of the zDNN library. It does *not* indicate whether those APIs are available for use. To check API availability at run time, see [“Validating the environment at run time” on page 11](#).

Format

```
uint32_t zdnn_get_library_version();
```

Parameters

None.

Returns

Returns the library version number in 0x00MMmmp format.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Library version string (zdnn_get_library_version_str)

Description

Retrieve the library version number and build information as a string.

Format

```
char *zdnn_get_library_version_str();
```

Parameters

None.

Returns

Returns the library version number and build information as a string.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

In-memory query result (zdnn_refresh_nnpa_query_result)

Description

Retrieve the zDNN in-memory query result from the IBM Z Integrated Accelerator for AI.

Format

```
zdnm_status zdnm_refresh_nnps_query_result();
```

Parameters

None.

Programming notes

This function is called automatically as part of **zdnm_init** processing and should not need to be called directly. Manually refreshing the query results before making other **zdnm_query_*** calls may noticeably impact performance.

Returns

One of the following zDNN status indicators:

```
ZDNN_OK  
ZDNN_UNAVAILABLE_FUNCTION
```

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Get size (zdnm_getsize_ztensor)

Description

Use this function to determine the buffer size required for the transformed tensor (including concatenated) in zDNN transformed format. Requires the tensor descriptor (**zdnm_tensor_desc**) with transformed shape information.

Format

```
uint64_t zdnm_getsize_ztensor(const zdnm_tensor_desc *tfrmd_desc);
```

Parameters

zdnm_tensor_desc *tfrmd_desc

Contains transformed information about the shape, layout, and data type.

Returns

The required buffer size, in bytes.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11.](#)

Get range (zdnnggetrange_ztensor)

Description

Use this function to determine the minimum negative value and maximum positive value of the passed zTensor and store the results in *min* and *max*.

Format

```
void zdnnggetrange_ztensor(const zdnng_ztensor *ztensor, float *min, float *max);
```

Parameters

const zdnng_ztensor *ztensor

The zTensor for which to return the min and max values.

float *min

Pointer to an output area in which to store the minimum negative value. If all values are positive, the value -0.0 is stored.

float *max

Pointer to an output area in which to store the maximum positive value. If all values are negative, the value 0.0 is stored.

Returns

None.

Since

1.1.0

Requirements

This function requires that:

- The **zdnng_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11.](#)

Get maximum limit for a data type (zdnngget_max_limit)

Description

Returns the maximum representable value between a transformed and pre-transformed zDNN data type.

Format

```
zdnng_status zdnngget_max_limit(zdnng_data_types transformed_type,  
                                zdnng_data_types pre_transformed_type, void *limit);
```

Parameters

zdnng_data_types transformed_type

The input zDNN transformed data type, which must be one of the following transformed data types:

ZDNN_DLFloat16
ZDNN_BINARY_INT8

ZDNN_BINARY_INT32

zdn_data_types pre_transformed_type

The input zDNN pre-transformed data type, which must be one of the following pre-transformed data types:

INT32
INT8
FP32
FP16
BFLOAT

void *limit

Pointer to an output area in which to store the maximum value between *transformed_type* and *pre_transformed_type*, in the data type of *pre_transformed_type*.

Returns

One of the following zDNN status indicators:

| | |
|-------------------|---|
| zDNN_OK | Success. |
| ZDNN_INVALID_TYPE | The data type of <i>transformed_type</i> or <i>pre_transformed_type</i> is not valid. |

Since

1.1.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Get minimum limit for a data type (zdn_get_min_limit)

Description

Returns the minimum representable value between a transformed and pre-transformed zDNN data type.

Format

```
zdn_status zdn_get_min_limit(zdn_data_types transformed_type,  
                             zdn_data_types pre_transformed_type, void *limit);
```

Parameters

zdn_data_types transformed_type

The input zDNN transformed data type, which must be one of the following transformed data types:

ZDNN_DLFLOAT16
ZDNN_BINARY_INT8
ZDNN_BINARY_INT32

zdn_data_types pre_transformed_type

The input zDNN pre-transformed data type, which must be one of the following pre-transformed data types:

INT32

INT8
FP32
FP16
BFLOAT

void *limit

Pointer to an output area in which to store the minimum value between *transformed_type* and *pre_transformed_type*, in the data type of *pre_transformed_type*.

Returns

One of the following zDNN status indicators:

| | |
|-------------------|---|
| zDNN_OK | Success. |
| ZDNN_INVALID_TYPE | The data type of <i>transformed_type</i> or <i>pre_transformed_type</i> is not valid. |

Since

1.1.0

Requirements

This function requires that:

- The **zdnm_is_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Initialize pre-transformed tensor descriptor (zdnm_init_pre_transformed_desc)

Description

Initialize a tensor descriptor (**zdnm_tensor_desc**) struct with pre-transformed (original) shape information.

Format

```
void zdnm_init_pre_transformed_desc(zdnm_data_layouts layout,  
                                   zdnm_data_types type,  
                                   zdnm_tensor_desc *pre_tfrmd_desc, ...);
```

Parameters

zdnm_data_layouts layout

The data layout.

zdnm_data_types type

The data type.

zdnm_tensor_desc *pre_tfrmd_desc

The output **zdnm_tensor_desc** structure.

... (additional arguments)

(Variadic) The number of elements in each dimension in accordance with the layout, in outermost to innermost order.

Returns

None.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time”](#) on page 11.

Generate transformed tensor descriptor (zdnngenerate_transformed_desc)

Description

Generate transformed tensor descriptor information based on a supplied pre-transformed tensor descriptor.

Format

```
zdnngenerate_transformed_desc(  
    const zdnngenerate_desc *pre_desc, zdnngenerate_desc *desc);
```

Parameters

zdnngenerate_desc *pre_desc

The input tensor descriptor with pre-transformed shape information.

zdnngenerate_desc *desc

The output **zdnngenerate_desc** structure.

Returns

One of the following zDNN status indicators:

| | |
|---------------------|--|
| zDNN_OK | Success. |
| ZDNN_INVALID_TYPE | The pre-transformed <i>type</i> value is not recognized or is a type that is only used for quantized zTensors. |
| ZDNN_INVALID_LAYOUT | The pre-transformed <i>layout</i> value is not recognized or is a layout only used for concatenated tensors. |

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time”](#) on page 11.

Generate quantized transformed tensor descriptor (zdnngenerate_quantized_transformed_desc)

Description

Generate quantized transformed tensor descriptor information based on a supplied pre-transformed tensor descriptor and quantized transform type.

Format

```
zdn_status zdn_generate_quantized_transformed_desc(  
    const zdn_tensor_desc *pre_tfrmd_desc,  
    zdn_quantized_transform_types transform_type,  
    zdn_tensor_desc *tfrmd_desc);
```

Parameters

zdn_tensor_desc *pre_tfrmd_desc

The input tensor descriptor with pre-transformed shape information. The following pre-transformed layouts are supported:

ZDNN_1D
ZDNN_2D
ZDNN_2DS
ZDNN_3D
ZDNN_3DS
ZDNN_4D
ZDNN_NHWC

zdn_quantized_transform_types transform_type

One of the following quantized transformation types:

QUANTIZED_DLFLOAT16
QUANTIZED_INT8
QUANTIZED_WEIGHTS_INT8

zdn_tensor_desc *tfrmd_desc

The output **zdn_tensor_desc** structure.

Returns

One of the following zDNN status indicators:

| | |
|-----------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_TYPE | The pre-transformed <i>type</i> value is not recognized or is not supported for quantized zTensors. For more information, see “Quantized zTensor requirements” on page 5. |
| ZDNN_INVALID_LAYOUT | The pre-transformed <i>layout</i> value is not recognized or is not supported for quantized zTensors, or is a layout that is only used for concatenated tensors.. |
| ZDNN_INVALID_TRANSFORM_TYPE | The transformation type is not valid. For more information, see “Quantized zTensor requirements” on page 5. |

Since

1.1.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Generate concatenated transformed tensor descriptor (zdnngenerate_transformed_desc_concatenated)

Description

Generate concatenated transformed tensor descriptor information for RNN input-gates tensors based on a supplied pre-transformed tensor descriptor.

Format

```
zdnngenerate_transformed_desc_concatenated(  
    const zdnntensor_desc *pre_tfrmd_desc,  
    zdnncat_info info, zdnntensor_desc *tfrmd_desc);
```

Parameters

zdnntensor_desc *pre_tfrmd_desc

The input tensor descriptor with pre-transformed shape information.

zdnncat_info info

Information about how the tensors will be concatenated. Consists of the **RNN_TYPE**, **PREV_LAYER**, and **USAGE** flags ORed together.

RNN_TYPE flags:

- **RNN_TYPE_LSTM** - For LSTM.
- **RNN_TYPE_GRU** - For GRU.

PREV_LAYER flags:

- **PREV_LAYER_UNI** - Previous RNN layer is unidirectional.
- **PREV_LAYER_NONE** - Previous layer is not an RNN layer.
- **PREV_LAYER_BIDIR** - Previous RNN layer is bidirectional.

USAGE flags:

- **USAGE_WEIGHTS** - Concatenate as input weights.
- **USAGE_HIDDEN_WEIGHTS** - Concatenate as input hidden-weights.
- **USAGE_BIASES** - Concatenate as input biases.
- **USAGE_HIDDEN_BIASES** - Concatenate as input hidden-biases.

zdnntensor_desc *tfrmd_desc

The output **zdnntensor_desc** structure.

Returns

One of the following zDNN status indicators:

| | |
|--------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_TYPE | The pre-transformed <i>type</i> value is not recognized or is not supported for concatenated tensors. |
| ZDNN_INVALID_LAYOUT | The pre-transformed <i>layout</i> value is not recognized or is not supported for concatenated tensors. |
| ZDNN_INVALID_CONCAT_INFO | Invalid concatenation information. |

Since

1.0.0

Requirements

This function requires that:

- The **zdnnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Initialize zTensor (zdnnpa_init_ztensor)

Description

Initialize a **zdnnpa_ztensor** structure using the pre-transformed and transformed tensor shape information.

Format

```
void zdnnpa_init_ztensor(zdnnpa_tensor_desc *pre_tfrmd_desc,  
                        zdnnpa_tensor_desc *tfrmd_desc, zdnnpa_ztensor *output);
```

Parameters

zdnnpa_tensor_desc *pre_tfrmd_desc

Input tensor descriptor with pre-transformed shape information.

zdnnpa_tensor_desc *tfrmd_desc

Input tensor descriptor with transformed shape information.

zdnnpa_ztensor *output

The **zdnnpa_ztensor** struct being initialized.

Returns

None.

Since

1.0.0

Requirements

This function requires that:

- The **zdnnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Initialize zTensor with memory allocate (zdnnpa_init_ztensor_with_malloc)

Description

Provides the same functionality as **zdnnpa_init_ztensor**, and computes the size required for the tensor in the zDNN transformed format and allocates the storage for it. Sets the **buffer** and **buffer_size** fields within **output**.

Format

```
zdnnpa_status zdnnpa_init_ztensor_with_malloc(zdnnpa_tensor_desc *pre_tfrmd_desc,  
                                              zdnnpa_tensor_desc *tfrmd_desc,  
                                              zdnnpa_ztensor *output);
```

Parameters

zdnntensor_desc *pre_tfrmd_desc

Input tensor descriptor with pre-transformed shape information.

zdnntensor_desc *tfrmd_desc

Input tensor descriptor with transformed shape information.

zdnntensor *output

The **zdnntensor** struct being initialized.

Returns

One of the following **zdnntstatus** indicators:

| Status | Meaning |
|-------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <i>tfrmd_desc->format</i> is not recognized. |
| ZDNN_INVALID_TYPE | <i>tfrmd_desc->type</i> is not recognized or is a pre-transform type. |
| ZDNN_INVALID_SHAPE | If any of the following conditions are true: <ul style="list-style-type: none">• One of the <i>tfrmd_desc->dim</i> dimensions is 0.• One of the <i>tfrmd_desc->dim</i> dimensions is greater than the size returned by zdnntget_max_for_dim. Note: Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 82 or “GRU input / output requirements” on page 86. <ul style="list-style-type: none">• The total number of <i>tfrmd_desc</i> elements is larger than the size returned by zdnntget_nnpa_max_tensor_size. |
| ZDNN_ALLOCATION_FAILURE | Unable to allocate required memory on a 4K boundary. |

Since

1.0.0

Requirements

This function requires that:

- The **zdnntis_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Initialize quantized zTensor (zdnntinit_quantized_ztensor)

Description

Initialize a **zdnnttensor** structure using the pre-transformed and quantized transformed tensor shape information along with scale and offset.

Format

```
void zdnntinit_quantized_ztensor(zdnnttensor_desc *pre_tfrmd_desc,
                                zdnnttensor_desc *tfrmd_desc, float scale,
                                float offset, zdnnttensor *output);
```

Parameters

zdnnttensor_desc *pre_tfrmd_desc

Input tensor descriptor with pre-transformed shape information.

zdnntensor_desc *tfrmd_desc

Input tensor descriptor with quantized, transformed shape information.

float scale

Scale for quantized zTensor, must not be 0.

float offset

Offset for quantized zTensor.

zdnntensor *output

The **zdnntensor** struct being initialized.

Programming notes

- The reciprocal of the *scale* value is stored as *output->rec_scale* and is used within subsequent quantized calls with reduced precision. Therefore, large *scale* values will lead to a *output->rec_scale* that underflows to 0.0 and will result in an error in subsequent quantized calls.
- The *offset* value is stored as *output->offset* and is used within subsequent quantized calls with reduced precision. Therefore, large *offset* values will overflow to infinity and will result in an error in subsequent quantized calls.

Returns

None.

Since

1.1.0

Requirements

This function requires that:

- The **zdnntest_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Initialize quantized zTensor with memory allocate (zdnntest_init_quantized_ztensor_with_malloc)

Description

Initialize a **zdnntest** structure using the pre-transformed and quantized, transformed tensor shape information along with scale and offset. This function provides the same functionality as [“Initialize quantized zTensor \(zdnntest_init_quantized_ztensor\)”](#) on page 30, computes the size required for the tensor in the zDNN transformed format, and allocates the storage for it. The *buffer* and *buffer_size* fields are set within *output*.

Format

```
zdnntest_status zdnntest_init_quantized_ztensor_with_malloc(  
    zdnntest_desc *pre_tfrmd_desc, zdnntest_desc *tfrmd_desc, float scale,  
    float offset, zdnntest *output);
```

Parameters

zdnntest_desc *pre_tfrmd_desc

Input tensor descriptor with pre-transformed shape information.

zdnntensor_desc *tfrmd_desc

Input tensor descriptor with quantized, transformed shape information.

float scale

Scale for quantized zTensor, must not be 0.

float offset

Offset for quantized zTensor.

zdnntensor *output

The **zdnntensor** struct being initialized.

Programming notes

- The reciprocal of the *scale* value is stored as *output->rec_scale* and is used within subsequent quantized calls with reduced precision. Therefore, large *scale* values will lead to a *output->rec_scale* that underflows to 0.0 and will result in an error in subsequent quantized calls.
- The *offset* value is stored as *output->offset* and is used within subsequent quantized calls with reduced precision. Therefore, large *offset* values will overflow to infinity and will result in an error in subsequent quantized calls.

Returns

One of the following **zdnntstatus** indicators:

| Status | Meaning |
|-------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <i>tfrmd_desc->format</i> is not recognized. |
| ZDNN_INVALID_TYPE | <i>tfrmd_desc->type</i> is not recognized or is a pre-transformed type. |
| ZDNN_INVALID_SHAPE | If any of the following conditions are true: <ul style="list-style-type: none">• One of the <i>tfrmd_desc->dim</i> dimensions is 0.• One of the <i>tfrmd_desc->dim</i> dimensions is greater than the size returned by zdnntget_max_for_dim.• The total number of <i>tfrmd_desc</i> elements is larger than the size returned by zdnntget_nnpa_max_tensor_size. |
| ZDNN_ALLOCATION_FAILURE | Unable to allocate required memory on a 4K boundary. |

Since

1.1.0

Requirements

This function requires that:

- The **zdnntis_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Determine quantized zTensor (zdnntis_quantized_ztensor)

Description

Determine whether or not a given **zdnnttensor** struct represents a quantized zTensor.

Format

```
bool zdnn_is_quantized_ztensor(zdnn_ztensor *ztensor);
```

Parameters

zdnn_ztensor *ztensor

The **zdnn_ztensor** struct being checked.

Returns

Returns `true` if the **zdnn_ztensor** struct represents a quantized zTensor; otherwise, returns `false`.

Since

1.1.0

Requirements

This function requires that:

- The **zdnn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Reset zTensor (zdnn_reset_ztensor)

Description

Reset a **zdnn_ztensor** struct for reuse.

Note: This operation does not set or reset the **buffer** and **buffer_size** fields nor free the transformed area storage.

Format

```
void zdnn_reset_ztensor(zdnn_ztensor *ztensor);
```

Parameters

zdnn_ztensor *ztensor

The **zdnn_ztensor** struct to be reset.

Returns

None.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Allocate memory for zTensor (zdnnp_alloc_helper_ztensor)

Description

Calculate the size required for the tensor in the zDNN transformed format and allocate the needed storage, satisfying alignment requirements. Sets the **buffer** and **buffer_size** fields within **ztensor**.

Note: The calling application assumes ownership of this storage and is responsible for freeing it.

Format

```
zdnnp_status zdnnp_alloc_helper_ztensor(zdnnp_ztensor *ztensor);
```

Parameters

zdnnp_ztensor *ztensor

A **zdnnp_ztensor** struct that contains the transformed shape information in the **transformed_desc** field.

Returns

One of the following **zdnnp_status** indicators:

| Status | Meaning |
|-------------------------|--|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <i>ztensor->transformed_desc->format</i> is not recognized. |
| ZDNN_INVALID_TYPE | <i>ztensor->transformed_desc->type</i> is not recognized or is a pre-transformed type. |
| ZDNN_INVALID_LAYOUT | <i>ztensor->transformed_desc->type</i> is not recognized or is not a valid transformed layout. |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none">• One of the <i>ztensor->transformed_desc->dim</i> dimensions is 0.• One of the <i>ztensor->transformed_desc->dim</i> dimensions is greater than the size returned by zdnnp_get_max_for_dim. <p>Note: Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 82 or “GRU input / output requirements” on page 86.</p> <ul style="list-style-type: none">• The total number of <i>transformed_desc</i> elements is larger than the value returned by zdnnp_get_nnpp_max_tensor_size. |
| ZDNN_ALLOCATION_FAILURE | Unable to allocate the required memory on a 4K boundary. |

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Deallocate memory for zTensor (zdnnp_free_ztensor_buffer)

Description

Free the transformed area storage associated with an input **zdnnp_ztensor**.

Note: This function does not free the storage allocated for the **zdnnp_ztensor** structure itself.

Format

```
zdsn_status zdsn_free_ztensor_buffer(const zdsn_ztensor *ztensor);
```

Parameters

zdsn_ztensor *ztensor

A **zdsn_ztensor** struct whose **buffer** field points to the allocated storage.

Returns

One of the following **zdsn_status** indicators:

| Status | Meaning |
|---------------------|--|
| ZDNN_OK | Success. |
| ZDNN_INVALID_BUFFER | The <i>ztensor->buffer</i> value is NULL. |

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Retrieve status message for a status code (zdsn_get_status_message)

Description

Retrieve the status message for a status code.

Format

```
const char *zdsn_get_status_message(zdsn_status status);
```

Parameters

zdsn_status status

The status code for which to retrieve the status message.

Returns

A pointer to the description string, or (Status string is not defined.) if *status* is not defined.

Since

1.0.0

Requirements

Any IBM Z hardware level.

See [“Validating the environment at run time” on page 11](#).

Reshape zTensor (zdnnp_reshape_ztensor)

Description

Reshape and copy the contents from the buffer of a source tensor to the buffer of a destination tensor in accordance with the shape of the destination tensor.

The following conditions must be satisfied:

- The *transformed_desc* field of both the source and destination tensors must be fully initialized.
- The destination tensor's buffer (*dest->buffer*) must be preallocated.
- The source tensor (*src*) must be transformed.
- The destination tensor (*dest*) must not already be transformed.
- The *transformed_desc->layout* field of both the source and destination tensors must be the same, either NHWC or HWCK.
- Both tensors must contain an equal number of elements.

Format

```
zdnnp_status zdnnp_reshape_ztensor(const zdnnp_tensor *src, zdnnp_tensor *dest);
```

Parameters

src

The source tensor from which to copy.

dest

The destination tensor to which to copy.

Programming notes

- If *src* and *dest* have the same *transformed_desc->dim1* dimension size, the transformed data is directly copied to the destination without untransformation.
- If *src* and *dest* have different *transformed_desc->dim1* dimension sizes, reshaping will internally untransform the source and then retransform the values into the destination.

Returns

One of the following **zdnnp_status** indicators:

| Status | Meaning |
|--------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none">• The dimensions (<i>transformed_desc->dim</i>) of <i>src</i> and <i>dest</i> total to different numbers of elements.• One of the <i>dest->transformed_desc->dim</i> dimensions is 0.• One of the <i>dest->transformed_desc->dim</i> dimensions is greater than the size returned by zdnnp_get_max_for_dim. <p>Note: Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 82 or “GRU input / output requirements” on page 86.</p> <ul style="list-style-type: none">• The total number of <i>dest->transformed_desc->dim</i> elements is larger than the value returned by zdnnp_get_nnppa_max_tensor_size. |

| Status | Meaning |
|----------------------|--|
| ZDNN_INVALID_LAYOUT | Any of the following reasons: <ul style="list-style-type: none"> • The value of <i>transformed_desc->layout</i> for <i>src</i> and <i>dest</i> are not the same. • The value of <i>transformed_desc->layout</i> is neither ZDNN_NHWC nor ZDNN_HWCK. • The value of <i>src->pre_transformed_desc->layout</i> is not recognized or is not a valid pre-transformation layout. • The value of <i>dest->pre_transformed_desc->layout</i> is not recognized or is not a valid pre-transformation layout. |
| ZDNN_INVALID_STATE | Any of the following reasons: <ul style="list-style-type: none"> • The source tensor (<i>src</i>) is not already transformed. • The destination tensor (<i>dest</i>) is already transformed. |
| ZDNN_INVALID_FORMAT | The format of the source tensor <i>src->transformed_desc->format</i> is not ZDNN_FORMAT_4DFEATURE. |
| ZDNN_INVALID_TYPE | Any of the following reasons: <ul style="list-style-type: none"> • The value of <i>src->pre_transformed_desc->type</i> is not recognized or is a transformed type. • The value of <i>dest->pre_transformed_desc->type</i> is not recognized or is a transformed type. • The value of <i>dest->transformed_desc->type</i> is not recognized or is a pre-transformation type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons: <ul style="list-style-type: none"> • The <i>src->buffer</i> field is NULL. • The <i>src->buffer</i> storage does not start on a 4K boundary. • The <i>dest->buffer</i> field is NULL. • The <i>dest->buffer</i> storage does not start on a 4K boundary. • The <i>dest->buffer_size</i> value is too small to hold the transformed values. |
| ZDNN_CONVERT_FAILURE | Values failed to untransform or transform. |

Since

1.0.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Check whether version is runnable (zdnns_is_version_runnable)

Description

Check whether an application built for zDNN version *ver_num* can be run on the current IBM Z Integrated Accelerator for AI hardware with the installed zDNN library.

Format

```
bool zdnns_is_version_runnable(uint32_t ver_num);
```

Parameters

`uint32_t ver_num`

The version number of the zDNN library application itself, in `0x00[major][minor][patch]` form. Typically, this is the **ZDNN_VERNUM** that was used to compile the application.

Returns

The function returns `true` or `false`.

Since

1.0.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Get maximum runnable version (zdnns_get_max_runnable_version)

Description

Return the maximum version number associated with the APIs that are supported by the hardware and zDNN software in the current environment. This can be compared to the version that is documented in the "Requirements" section of each API to determine whether an API is supported at run time.

The returned value is a version number in the *major.minor* format. APIs defined at that level or earlier are supported in the current environment.

Format

```
uint32_t zdnns_get_max_runnable_version();
```

Parameters

None.

Returns

A 32-bit zDNN version number in `0x00[major][minor]FF` form.

Since

1.0.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Data transformation

The IBM Z Integrated Accelerator for AI (zAIU) requires that the tensor data be arranged in a format that enhances the performance characteristics of the operations. This documentation refers to that format as the *transformed format*. In addition, data conversions are necessary from the common formats (FP32, FP16, BFLOAT) to formats supported by the zAIU (DLFLOAT16, INT8). The following functions are provided:

zdnnt_transform_ztensor

zdnnt_transform_ztensor_with_saturation

These functions transform the input tensor and convert the input data to the format required by the zAIU. The resulting transformed zTensor can be reused as many times as necessary.

See “Transform to zTensor (zdnnt_transform_ztensor)” on page 39 and “Transform to zTensor with saturation (zdnnt_transform_ztensor_with_saturation)” on page 41 for details and restrictions about transforming an input tensor to the internal format.

zdnnt_transform_quantized_ztensor

Transforms an input tensor to a quantized zTensor.

See “Transform to quantized zTensor (zdnnt_transform_quantized_ztensor)” on page 42 for details.

zdnnt_transform_origtensor

Transforms a zTensor (usually output from an operation or network) to the format and data types that are usable by the application.

See “Transform to original (zdnnt_transform_origtensor)” on page 44 for details about transforming an input zTensor to the original format.

Transform to zTensor (zdnnt_transform_ztensor)

Description

Converts the input tensor to the supported transformed format for execution by zDNN operations. If transformation is successful, the **is_transformed** field within **ztensor** is set to `true`; otherwise, it is set to `false`. Transformation fails if the **is_transformed** value is already set to `true`.

Note: Once it is in transformed format, the tensor layout in memory is dependent on the content of the input tensor's descriptors (**zdnnt_tensor_desc** fields). Once converted, a **zdnnt_ztensor** should only be manipulated by zDNN API functions.

Format

```
zdnnt_status zdnnt_transform_ztensor(zdnnt_ztensor ztensor, ...);
```

Parameters

zdnnt_ztensor *tensor

The input **zdnnt_ztensor** struct. The *pre_transformed_desc* and *transformed_desc* fields must be set, and the *is_transformed* field must be set to `false`. A 4K-aligned tensor storage area must be preallocated by the caller (directly or by calling the zDNN allocation helper function) and the *buffer* field must point to the storage area.

... (additional arguments)

(Variadic) A list of pointers for input data to be transformed, as follows:

- **Non-concatenated:** 1 data pointer
- **LSTM concatenated:** 4 data pointers, one for each input gate in Forget, Input, Cell, Output (FICO) order
- **GRU concatenated:** 3 data pointers, one for each input gate in (Z)update, Reset, Hidden, (ZRH) gate order

Programming notes

The **zdnnc_transform_ztensor** function clears the pre-thread floating-point exception flags at entry, and may set FE_UNDERFLOW, FE_INVALID, FE_INEXACT, or FE_OVERFLOW when it encounters errors during data conversion.

Returns

One of the following **zdnnc_status** indicators:

| Status | Meaning |
|----------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <code>zdnnc_ztensor->transformed_desc->format</code> is not recognized. |
| ZDNN_INVALID_LAYOUT | Any of the following reasons: <ul style="list-style-type: none"><code>zdnnc_ztensor->pre_transformed_desc->layout</code> is not recognized or is not a valid pre-transformed layout.<code>zdnnc_ztensor->transformed_desc->layout</code> is not recognized or is not a valid transformed layout. |
| ZDNN_INVALID_TYPE | Any of the following reasons: <ul style="list-style-type: none"><code>zdnnc_ztensor->pre_transformed_desc->type</code> is not recognized or is a pre-transformed type.<code>zdnnc_ztensor->transformed_desc->type</code> is not recognized or is a transform type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons: <ul style="list-style-type: none">The <i>buffer</i> field is NULL.The start of the <i>buffer</i> storage is not on a 4K boundary.The <i>buffer_size</i> value is too small to hold transformed values. |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none">One of the <code>zdnnc_ztensor->transformed_desc->dim</code> dimensions is 0.One of the <code>zdnnc_ztensor->transformed_desc->dim</code> dimensions is greater than the size returned by zdnnc_get_max_for_dim.Note: Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 82 or “GRU input / output requirements” on page 86.The total number of <i>transformed_desc</i> elements is larger than the value returned by zdnnc_get_nnpa_max_tensor_size. |
| ZDNN_INVALID_STATE | Tensor is already transformed. |
| ZDNN_CONVERT_FAILURE | Values failed to transform. |
| ZDNN_FUNC_RC_F000 | Unsupported hardware transformation function. |

Since

1.0.0

Requirements

This function requires that:

- The **zdnnc_is_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Transform to zTensor with saturation (zdnntensor_transform_ztensor_with_saturation)

Description

Converts the input tensor to the supported transformed format for execution by zDNN operations. If during transformation, an element results in a value that exceeds the smallest or largest value that can be represented by DFLOAT16, the resulting element will contain the smallest or largest value, and no range-violation status will be triggered. If transformation is successful, the **is_transformed** field within **ztensor** is set to **true**; otherwise, it is set to **false**. Transformation fails if the **is_transformed** value is already set to **true**.

Note: Once it is in transformed format, the tensor layout in memory is dependent on the content of the input tensor's descriptors (**zdnntensor_desc** fields). Once converted, a **zdnntensor** should only be manipulated by zDNN API functions.

Format

```
zdnntensor_status zdnntensor_transform_ztensor_with_saturation(zdnntensor ztensor, ...);
```

Parameters

zdnntensor *tensor

The input **zdnntensor** struct. The *pre_transformed_desc* and *transformed_desc* fields must be set, and the *is_transformed* field must be set to **false**. A 4K-aligned tensor storage area must be preallocated by the caller (directly or by calling the zDNN allocation helper function) and the *buffer* field must point to the storage area.

... (additional arguments)

(Variadic) A list of pointers for input data to be transformed, as follows:

- Currently, 1 data pointer is supported.

Returns

One of the following **zdnntensor_status** indicators:

| Status | Meaning |
|---------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <i>zdnntensor->transformed_desc->format</i> is not ZDNN_FORMAT_4DFEATURE. |
| ZDNN_INVALID_LAYOUT | Any of the following reasons: <ul style="list-style-type: none">• <i>zdnntensor->pre_transformed_desc->layout</i> is not recognized or is not a valid pre-transformed layout.• <i>zdnntensor->transformed_desc->layout</i> is not recognized or is not a valid transformed layout. |
| ZDNN_INVALID_TYPE | Any of the following reasons: <ul style="list-style-type: none">• <i>zdnntensor->pre_transformed_desc->type</i> is not recognized or is a pre-transformed type.• <i>zdnntensor->transformed_desc->type</i> is not recognized or is a transform type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons: <ul style="list-style-type: none">• The <i>buffer</i> field is NULL.• The start of the <i>buffer</i> storage is not on a 4K boundary.• The <i>buffer_size</i> value is too small to hold the transformed values. |

| Status | Meaning |
|--------------------|---|
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none"> • One of the <code>zdnntensor->transformed_desc->dim*</code> dimensions is 0. • One of the <code>zdnntensor->transformed_desc->dim*</code> dimensions is greater than the size returned by <code>zdnngget_max_for_dim</code>. • The total number of <code>transformed_desc</code> elements is larger than the value returned by <code>zdnngget_nnpa_max_tensor_size</code>. |
| ZDNN_INVALID_STATE | Tensor is already transformed. |
| ZDNN_FUNC_RC_F000 | Unsupported hardware transformation function. |

Since

1.1.0

Requirements

This function requires that:

- The `zdnngis_nnpa_installed` function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Transform to quantized zTensor (zdnnttransform_quantized_ztensor)

Description

Converts the input tensor to the supported transformed format for execution by zDNN operations. If transformation is successful, the `is_transformed` field within `ztensor` is set to `true`; otherwise, it is set to `false`. Transformation fails if the `is_transformed` value is already set to `true`.

Note: Once it is in transformed format, the tensor layout in memory is dependent on the content of the input tensor's descriptor (`zdnnttensor_desc`) fields. Once converted, a `zdnntztensor` should only be manipulated by zDNN API functions.

Format

```
zdnntstatus zdnnttransform_quantized_ztensor(zdnntztensor ztensor,
                                             bool saturation_control,
                                             int8_t clip_min, int8_t clip_max,
                                             const void *data);
```

Parameters

`zdnntztensor *tensor`

The input `zdnntztensor` struct. The `pre_transformed_desc` and `transformed_desc` fields must be set, and the `is_transformed` field must be set to `false`. A 4K-aligned tensor storage area must be preallocated by the caller (directly or by calling the zDNN allocation helper function) and the `buffer` field must point to the storage area.

The following restrictions apply:

- Only the following pre-transformed layouts are supported:

```
ZDNN_1D
ZDNN_2D
ZDNN_2DS
ZDNN_3D
ZDNN_3DS
```

ZDNN_4D
ZDNN_NHWC

- Only the NHWC transformed layout is supported.
- See [“Quantized zTensor requirements” on page 5](#) for supported transform types.

bool saturation_control

When enabled and an element results in a value that exceeds the smallest or largest value that can be represented by DLFLOAT16, the resulting element will contain the smallest or largest value and no range-violation status will be triggered. This only applies when all of the following conditions are true:

- The input tensor is of the QUANTIZED_INT8 type (**zdnnp_quantized_transform_types**).
- The **pre_transformed_desc** type of the input tensor is FP32.

int8_t clip_min

The minimum clipping value. This only applies when the input tensor is of the QUANTIZED_INT8 type (**zdnnp_quantized_transform_types**). The value must be less than *clip_max*.

int8_t clip_max

The maximum clipping value. This only applies when the input tensor is of the QUANTIZED_INT8 type (**zdnnp_quantized_transform_types**). The value must be greater than *clip_min*.

Programming notes

The **zdnnp_transform_quantized_ztensor** function clears the pre-thread floating-point exception flags at entry, and may set FE_UNDERFLOW, FE_INVALID, FE_INEXACT, or FE_OVERFLOW when it encounters errors during data conversion.

Returns

One of the following **zdnnp_status** indicators:

| Status | Meaning |
|---------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | <code>zdnnp_ztensor->transformed_desc->format</code> is not recognized. |
| ZDNN_INVALID_LAYOUT | Any of the following reasons: <ul style="list-style-type: none">• <code>zdnnp_ztensor->pre_transformed_desc->layout</code> is not recognized or is not a valid pre-transformed layout.• <code>zdnnp_ztensor->transformed_desc->layout</code> is not recognized or is not a valid transformed layout. |
| ZDNN_INVALID_TYPE | Any of the following reasons: <ul style="list-style-type: none">• <code>zdnnp_ztensor->pre_transformed_desc->type</code> is not recognized or is a pre-transformed type.• <code>zdnnp_ztensor->transformed_desc->type</code> is not recognized or is a transform type. See “Quantized zTensor requirements” on page 5 . |
| ZDNN_INVALID_BUFFER | Any of the following reasons: <ul style="list-style-type: none">• The <i>buffer</i> field is NULL.• The start of the <i>buffer</i> storage is not on a 4K boundary.• The <i>buffer_size</i> value is too small to hold the transformed values. |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none">• One of the <code>zdnnp_ztensor->transformed_desc->dim*</code> dimensions is 0.• One of the <code>zdnnp_ztensor->transformed_desc->dim*</code> dimensions is greater than the size returned by zdnnp_get_max_for_dim.• The total number of <i>transformed_desc</i> elements is larger than the value returned by zdnnp_get_nnpa_max_tensor_size. |
| ZDNN_INVALID_STATE | Tensor is already transformed. |

| Status | Meaning |
|-----------------------------|---|
| ZDNN_INVALID_CLIPPING_VALUE | The <i>clip_min</i> value is not less than the <i>clip_max</i> value. |
| ZDNN_FUNC_RC_F000 | Unsupported transformation function. |
| ZDNN_FUNC_RC_F001 | Either scale or offset is non-numeric or the scale value is zero. |

Since

1.1.0

Requirements

This function requires that:

- The **zdnns_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Transform to original (zdnns_transform_origtensor)

Description

Converts the input tensor from the zDNN transformed format back to a standard non-transformed layout. The **is_transformed** field within the input tensor must be set to `true`.

All stick-format tensors are supported, except:

- Kernel tensors
- Concatenated RNN input-gates tensors

Format

```
zdnns_status zdnns_transform_origtensor(const zdnns_ztensor *ztensor, void *out_buf);
```

Parameters

zdnns_ztensor *ztensor

The input **zdnns_ztensor** struct. The *pre_transformed_desc*, *transformed_desc*, and *buffer* fields must be set, and the *is_transformed* field must be set to `true`.

void *out_buf

The buffer for storing the standard non-transformed tensor data. Must be preallocated by the caller.

Programming notes

The **zdnns_transform_origtensor** function clears the pre-thread floating-point exception flags at entry, and may set `FE_UNDERFLOW`, `FE_INVALID`, `FE_INEXACT`, or `FE_OVERFLOW` when it encounters errors during data conversion.

Returns

One of the following **zdnns_status** indicators:

| Status | Meaning |
|-----------------------|--|
| ZDNN_OK | Success. |
| ZDNN_UNSUPPORTED_TYPE | (SIM only) |
| ZDNN_INVALID_FORMAT | The <i>ztensor->transformed_desc->format</i> value is not <code>ZDNN_FORMAT_4DFEATURE</code> . |

| Status | Meaning |
|----------------------|--|
| ZDNN_INVALID_LAYOUT | Any of the following reasons: <ul style="list-style-type: none"> The <code>zdnntensor->pre_transformed_desc->layout</code> value is not recognized or is not a valid pre-transformed layout. The <code>zdnntensor->transformed_desc->layout</code> value is not recognized or is not a valid transformed layout required by this function. |
| ZDNN_INVALID_TYPE | Any of the following reasons: <ul style="list-style-type: none"> The <code>zdnntensor->pre_transformed_desc->type</code> value is not recognized or is a pre-transformed type. The <code>zdnntensor->transformed_desc->type</code> value is not recognized or is a transformed type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons: <ul style="list-style-type: none"> The <code>zdnntensor->buffer</code> field is NULL. The start of the <code>zdnntensor->buffer</code> storage is not on a 4K boundary. |
| ZDNN_INVALID_STATE | The <code>zdnntensor</code> is not transformed. |
| ZDNN_CONVERT_FAILURE | Values failed to untransform. |

Since

1.0.0

Requirements

This function requires that:

- The **`zdnntest_is_nnpa_installed`** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Operations

The zDNN operations are organized into the following categories:

- [“Element-wise operations” on page 45](#)
- [“Activation operations” on page 55](#)
- [“Normalization operations” on page 63](#)
- [“Matmul and matmul with broadcast” on page 72](#)
- [“LSTM \(`zdnntest_lstm`\)” on page 82](#)
- [“GRU \(`zdnntest_gru`\)” on page 86](#)
- [“Average pool 2D \(`zdnntest_avgpool2d`\)” on page 90](#)
- [“Max pool 2D \(`zdnntest_maxpool2d`\)” on page 93](#)
- [“Convolution 2D \(`zdnntest_conv2d`\)” on page 95](#)

Element-wise operations

The zDNN element-wise operations are:

- [“Addition \(`zdnntest_add`\)” on page 46](#)
- [“Subtraction \(`zdnntest_sub`\)” on page 47](#)
- [“Multiplication \(`zdnntest_mul`\)” on page 48](#)
- [“Division \(`zdnntest_div`\)” on page 49](#)

- [“Minimum \(zdnn_min\)” on page 50](#)
- [“Maximum \(zdnn_max\)” on page 51](#)
- [“Natural logarithm \(zdnn_log\)” on page 52](#)
- [“Exponential \(zdnn_exp\)” on page 52](#)
- [“Square root \(zdnn_sqrt\)” on page 53](#)
- [“Inverse square root \(zdnn_invsqrt\)” on page 54](#)

Addition (zdnn_add)

Description

Given two input tensors in zDNN transformed format, perform element-wise addition and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnn_status zdnn_add(const zdnn_ztensor *input_a, const zdnn_ztensor *input_b,
                    zdnn_ztensor *output);
```

Parameters

zdnn_ztensor *input_a

An input tensor with addends to add to the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *input_b

An input tensor with addends to add to the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *output

An output tensor to hold the result of the addition. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK
[“Warning statuses” on page 8](#)
 ZDNN_INVALID_SHAPE
 ZDNN_INVALID_TYPE
 ZDNN_INVALID_FORMAT
[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow addition](https://www.tensorflow.org/api_docs/python/tf/math/add) (https://www.tensorflow.org/api_docs/python/tf/math/add)
- [ONNX addition](https://onnx.ai/onnx/operators/onnx__Add.html#l-onnx-doc-add) (https://onnx.ai/onnx/operators/onnx__Add.html#l-onnx-doc-add)

Subtraction (zdnns_sub)

Description

Given two input tensors in zDNN transformed format, perform element-wise subtraction and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnns_status zdnns_sub(const zdnns_tensor *input_a, const zdnns_tensor *input_b,  
                      zdnns_tensor *output);
```

Parameters

zdnns_tensor *input_a

An input tensor with minuends to be subtracted by the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnns_tensor *input_b

An input tensor with subtrahends to subtract from the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnns_tensor *output

An output tensor to hold the result of the subtraction. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnns_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK
[“Warning statuses” on page 8](#)
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow subtraction](https://www.tensorflow.org/api_docs/python/tf/math/subtract) (https://www.tensorflow.org/api_docs/python/tf/math/subtract)

- [ONNX subtraction](https://onnx.ai/onnx/operators/onnx__Sub.html#l-onnx-doc-sub) (https://onnx.ai/onnx/operators/onnx__Sub.html#l-onnx-doc-sub)

Multiplication (zdnn_mul)

Description

Given two input tensors in zDNN transformed format, perform element-wise multiplication and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnn_status zdnn_mul(const zdnn_ztensor *input_a, const zdnn_ztensor *input_b,
                    zdnn_ztensor *output);
```

Parameters

zdnn_ztensor *input_a

An input tensor with multiplicands to be multiplied by the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnn_ztensor *input_b

An input tensor with multipliers for the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnn_ztensor *output

An output tensor to hold the result of the multiplication. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK
[“Warning statuses”](#) on page 8
 ZDNN_INVALID_SHAPE
 ZDNN_INVALID_TYPE
 ZDNN_INVALID_FORMAT
[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- [TensorFlow multiplication](https://www.tensorflow.org/api_docs/python/tf/math/multiply) (https://www.tensorflow.org/api_docs/python/tf/math/multiply)
- [ONNX multiplication](https://onnx.ai/onnx/operators/onnx__Mul.html#l-onnx-doc-mul) (https://onnx.ai/onnx/operators/onnx__Mul.html#l-onnx-doc-mul)

Division (zdnv_div)

Description

Given two input tensors in zDNN transformed format, perform element-wise division and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnv_status zdnv_div(const zdnv_ztensor *input_a, const zdnv_ztensor *input_b,  
                    zdnv_ztensor *output);
```

Parameters

zdnv_ztensor *input_a

An input tensor with dividends to be divided by the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnv_ztensor *input_b

An input tensor with divisors for the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnv_ztensor *output

An output tensor to hold the result of the division. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnv_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnv_is_nnva_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow division](https://www.tensorflow.org/api_docs/python/tf/math/divide) (https://www.tensorflow.org/api_docs/python/tf/math/divide)
- [ONNX division](https://onnx.ai/onnx/operators/onnx__Div.html#l-onnx-doc-div) (https://onnx.ai/onnx/operators/onnx__Div.html#l-onnx-doc-div)

Minimum (zdnn_min)

Description

Given two input tensors in zDNN transformed format, compute the element-wise minimum and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnn_status zdnn_min(const zdnn_ztensor *input_a, const zdnn_ztensor *input_b,  
                    zdnn_ztensor *output);
```

Parameters

zdnn_ztensor *input_a

An input tensor with values to be compared with the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *input_b

An input tensor with values to be compared with the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *output

An output tensor to hold the smaller values from each comparison of the inputs. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow minimum](https://www.tensorflow.org/api_docs/python/tf/math/minimum) (https://www.tensorflow.org/api_docs/python/tf/math/minimum)
- [ONNX minimum](https://onnx.ai/onnx/operators/onnx__Min.html#l-onnx-doc-min) (https://onnx.ai/onnx/operators/onnx__Min.html#l-onnx-doc-min)

Maximum (zdnm_max)

Description

Given two input tensors in zDNN transformed format, compute the element-wise maximum and store the result into the provided output zDNN tensor.

Note: For zDNN use, broadcasting of the input tensors must be done by the caller. As such, the input tensors must be of the same shape.

Format

```
zdnm_status zdnm_max(const zdnm_ztensor *input_a, const zdnm_ztensor *input_b,  
                    zdnm_ztensor *output);
```

Parameters

zdnm_ztensor *input_a

An input tensor with values to be compared with the *input_b* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnm_ztensor *input_b

An input tensor with values to be compared with the *input_a* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnm_ztensor *output

An output tensor to hold the larger value from each comparison of the inputs. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnm_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnm_is_nnps_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow maximum](https://www.tensorflow.org/api_docs/python/tf/math/maximum) (https://www.tensorflow.org/api_docs/python/tf/math/maximum)
- [ONNX maximum](https://onnx.ai/onnx/operators/onnx__Max.html#l-onnx-doc-max) (https://onnx.ai/onnx/operators/onnx__Max.html#l-onnx-doc-max)

Natural logarithm (zdnng_log)

Description

Given an input tensor in zDNN transformed format, compute the natural logarithm element-wise and store the result into the provided output zDNN tensor.

Format

```
zdnng_status zdnng_log(const zdnng_ztensor *input, zdnng_ztensor *output);
```

Parameters

zdnng_ztensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnng_ztensor *output

An output tensor to hold the calculated natural logarithm of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdnng_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zdnng_is_nnpg_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow natural logarithm (https://www.tensorflow.org/api_docs/python/tf/math/log)
- ONNX natural logarithm (https://onnx.ai/onnx/operators/onnx__Log.html#l-onnx-doc-log)

Exponential (zdnng_exp)

Description

Given an input tensor in zDNN transformed format, compute the exponential element-wise and store the result into the provided output zDNN tensor.

Format

```
zdnm_status zdnm_exp(const zdnm_ztensor *input, zdnm_ztensor *output);
```

Parameters

zdnm_ztensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnm_ztensor *output

An output tensor to hold the calculated exponential of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdnm_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zdnm_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- [TensorFlow exponential](https://www.tensorflow.org/api_docs/python/tf/math/exp) (https://www.tensorflow.org/api_docs/python/tf/math/exp)
- [ONNX exponential](https://onnx.ai/onnx/operators/onnx__Exp.html#l-onnx-doc-exp) (https://onnx.ai/onnx/operators/onnx__Exp.html#l-onnx-doc-exp)

Square root (zdnm_sqrt)

Description

Given an input tensor in zDNN transformed format, compute the square root element-wise and store the result into the provided output zDNN tensor.

Format

```
zdnm_status zdnm_sqrt(const zdnm_ztensor *input, zdnm_ztensor *output);
```

Parameters

zdnm_ztensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnntensor *output

An output tensor to hold the calculated square root of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdnntstatus** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.1.0

Requirements

This function requires that:

- The **zdnntis_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow square root (https://www.tensorflow.org/api_docs/python/tf/math/sqrt)
- ONNX square root (https://onnx.ai/onnx/operators/onnx__Sqrt.html#l-onnx-doc-sqrt)

Inverse square root (zdnntinvsqrt)

Description

Given an input tensor in zDNN transformed format, compute the inverse square root element-wise and store the result into the provided output zDNN tensor.

Format

```
zdnntstatus zdnntinvsqrt(const zdnnttensor *input, float epsilon, zdnnttensor *output);
```

Parameters

zdnnttensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

float epsilon

A float value added to the *input* tensor prior to computation.

zdnnttensor *output

An output tensor to hold the calculated inverse square root of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Programming notes

On some models, if either an element, epsilon, or both are very large, the addition of the two may result in a nonnumeric value, the inverse square root of which will also be nonnumeric. This can occur even though the inverse square root of an unconstrained sum would easily fit in the data type of an output-tensor element.

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_INVALID_EPSILON

[“Hardware statuses” on page 9](#)

Since

1.1.0

Requirements

This function requires that:

- The **zdnn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow reciprocal square root](https://www.tensorflow.org/api_docs/python/tf/math/rsqrt) (https://www.tensorflow.org/api_docs/python/tf/math/rsqrt)

Activation operations

The zDNN activation operations are:

- [“Rectified linear \(zdnn_relu\)” on page 55](#)
- [“Leaky rectified linear \(zdnn_leaky_relu\)” on page 56](#)
- [“Hyperbolic tangent \(zdnn_tanh\)” on page 58](#)
- [“Sigmoid \(zdnn_sigmoid\)” on page 58](#)
- [“Softmax \(zdnn_softmax\)” on page 59](#)
- [“Softmax with mask \(zdnn_softmax_mask\)” on page 61](#)
- [“Gaussian error linear unit \(zdnn_gelu\)” on page 63](#)

Rectified linear (zdnn_relu)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the rectified linear function, $y = \max(0, x)$, is applied to the input tensor element-wise. If an optional *clipping_value* is provided, clipping is performed against the intermediate output, where $z = \min(y, \text{clipping_value})$.

Format

```
zdn_status zdn_relu(const zdn_tensor *input, const void *clipping_value,
                    zdn_tensor *output);
```

Parameters

zdn_tensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

void *clipping_value

A pointer to an FP32 value used to clip the elements of the input tensor. If set to NULL or 0, no clipping occurs. The clipping value must not be negative.

zdn_tensor *output

An output tensor to hold the rectified linear function result of each value from the *input* tensor.

Returns

One of the following **zdn_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_INVALID_CLIPPING_VALUE

[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns **true**.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- [TensorFlow rectified linear](https://www.tensorflow.org/api_docs/python/tf/math/relu) (https://www.tensorflow.org/api_docs/python/tf/math/relu)
- [ONNX rectified linear](https://onnx.ai/onnx/operators/onnx__Relu.html#l-onnx-doc-relu) (https://onnx.ai/onnx/operators/onnx__Relu.html#l-onnx-doc-relu)

Leaky rectified linear (zdn_leaky_relu)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the leaky rectified linear function is applied to the input tensor element-wise. The calculation used depends on the input element. When negative, $y = a * x$, where a is the adjustment factor. When 0 or positive, $y = x$. If an optional *clipping_value* is provided, clipping is performed against the intermediate output, where $z = \min(y, clipping_value)$.

Format

```
zdn_status zdn_leaky_relu(const zdn_tensor *input,
                          const void *clipping_value,
                          float adjustment_factor, zdn_tensor *output);
```

Parameters

zdn_tensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

void *clipping_value

A pointer to an FP32 value used to clip the elements of the input tensor. If set to NULL or 0, no clipping occurs. The clipping value must not be negative.

float adjustment_factor

A float value multiplied with negative elements from the *input* tensor. The value must not be negative nor greater than 1.

zdn_tensor *output

An output tensor to hold the rectified linear function result of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdn_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_INVALID_CLIPPING_VALUE

ZDNN_INVALID_ADJUSTMENT_FACTOR

[“Hardware statuses”](#) on page 9

Since

1.1.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow leaky rectified linear (https://www.tensorflow.org/api_docs/python/tf/nn/leaky_relu)
- ONNX leaky rectified linear (https://onnx.ai/onnx/operators/onnx__LeakyRelu.html#l-onnx-doc-leakyrelu)

Hyperbolic tangent (zdnn_tanh)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the hyperbolic tangent function is applied to the input tensor element-wise.

Format

```
zdnn_status zdnn_tanh(const zdnn_ztensor *input, zdnn_ztensor *output);
```

Parameters

zdnn_ztensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *output

An output tensor to hold the hyperbolic tangent result of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses” on page 9](#)

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- TensorFlow hyperbolic tangent (https://www.tensorflow.org/api_docs/python/tf/math/tanh)
- ONNX hyperbolic tangent (https://onnx.ai/onnx/operators/onnx__Tanh.html#l-onnx-doc-tanh)

Sigmoid (zdnn_sigmoid)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the sigmoid function is applied to the input element-wise.

Format

```
zdn_status zdn_tanh(const zdn_ztensor *input, zdn_ztensor *output);
```

Parameters

zdn_ztensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdn_ztensor *output

An output tensor to hold the sigmoid result of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdn_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow [sigmoid](https://www.tensorflow.org/api_docs/python/tf/math/sigmoid) (https://www.tensorflow.org/api_docs/python/tf/math/sigmoid)
- ONNX [sigmoid](https://onnx.ai/onnx/operators/onnx__Sigmoid.html#l-onnx-doc-sigmoid) (https://onnx.ai/onnx/operators/onnx__Sigmoid.html#l-onnx-doc-sigmoid)

Softmax (zdn_softmax)

Description

Given an input tensor in zDNN transformed format, compute the softmax (normalized exponential) for each vector formed in dimension 1, then if *act_func* is not `SOFTMAX_ACT_NONE`, apply the activation function to the results, and store the results into the provided output zDNN tensor.

Note: Other parameters, such as **axis**, are not supported.

Format

```
zdn_status zdn_softmax(const zdn_ztensor *input, void *save_area,
                      zdn_softmax_act act_func, zdn_ztensor *output);
```

Parameters

zdsn_ztensor *input

An input tensor with a ZDNN_3DS layout and pre-transformed shape of [batch size, batch size, vector dimension size] or output from another operation that is of the correct shape. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

void *save_area

The address of preallocated memory to use for temporary storage during internal processing. The preallocated memory must be at least 8K bytes in size and aligned on a 4K boundary. If set to NULL, the operation determines, allocates, and frees storage automatically.

zdsn_softmax_act act_func

The activation function to apply to the results. Valid values are SOFTMAX_ACT_NONE or SOFTMAX_ACT_LOG.

zdsn_ztensor *output

An output tensor with a ZDNN_3DS layout and the same shape as the *input* tensor to hold the softmax result of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Programming notes

- If all elements of a dimension 1 vector are the largest-magnitude negative number possible for the transformed data type, accuracy may be reduced.
- A ZDNN_3DS tensor is expected, where the *transformed_desc* dimension 1 (*dim1*) describes the vector, and *dim2* and *dim4* are used to batch multiple vector requests together. The *dim3* dimension must always be 1. The **zdsn_softmax** operation is performed against the vector in *dim1*, repeating for each *dim1* vector in the *dim4* and *dim2* dimensions.
- Tensors that cannot be processed as vectors in dimension 1 or as batches of dimension 1 vectors must be coerced or reshaped by the caller.

When the entire tensor is to be processed by softmax, it can be coerced by creating an alternate descriptor prior to zDNN transformation. For example:

- A 4D tensor with *pre_transformed_desc* dimensions 2x2x2x2 and a data array of 16 FP32 entries could have an alternate ZDNN_3DS layout *pre_transformed_desc* using dimensions 8x1x2 and use the same original data array prior to the **zdsn_transform_ztensor** call. After transformation, such a tensor would be valid for the **zdsn_softmax** operation.
- As a further example, the 4D 2x2x2x2 tensor could be processed as 8 batches of 2 vectors using a ZDNN_3DS layout *pre_transformed_desc* with dimensions 1x8x2.

Returns

One of the following **zdsn_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_ALLOCATION_FAILURE — A preallocated save area was not specified and internal allocation for the required memory failed.

[“Hardware statuses”](#) on page 9

ZDNN_FUNC_RC_F000 — Dimension 3 of the input tensor (*input->transformed_desc->dim3*) is not 1.

ZDNN_FUNC_RC_F001 — Invalid *act_func* value.

Since

1.0.0

Requirements

This function requires that:

- The **zdsn_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow softmax (https://www.tensorflow.org/api_docs/python/tf/nn/softmax)
- ONNX softmax (https://onnx.ai/onnx/operators/onnx__Softmax.html#l-onnx-doc-softmax)

Softmax with mask (zdsn_softmax_mask)

Description

Given an input tensor in zDNN transformed format, compute the softmax (normalized exponential) for each vector formed in dimension 1 (from element zero to mask – 1), then if *act_func* is not `SOFTMAX_ACT_NONE`, apply the activation function to the results, and store the results into the provided output zDNN tensor.

Note: Other parameters, such as **axis**, are not supported.

Format

```
zdsn_status zdsn_softmax_mask(const zdsn_ztensor *input, void *save_area,
                             zdsn_softmax_act act_func, uint32_t softmax_mask,
                             zdsn_ztensor *output);
```

Parameters

zdsn_ztensor *input

An input tensor with a ZDNN_3DS layout and pre-transformed shape of [*batch size*, *batch size*, *vector dimension size*] or output from another operation that is of the correct shape. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

void *save_area

The address of preallocated memory to use for temporary storage during internal processing. The preallocated memory must be at least 8K bytes in size and aligned on a 4K boundary. If set to `NULL`, the operation determines, allocates, and frees storage automatically.

zdsn_softmax_act act_func

The activation function to apply to the results. Valid values are `SOFTMAX_ACT_NONE` or `SOFTMAX_ACT_LOG`.

uint32_t softmax_mask

A 32-bit unsigned binary integer that specifies a count of dimension-1 elements to be processed. If 0, the behavior matches that of the **zdsn_softmax** function. The value must not exceed dimension 1 of the *input* tensor.

zdsn_ztensor *output

An output tensor with a ZDNN_3DS layout and the same shape as the *input* tensor to hold the softmax result of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Programming notes

- If all elements of a dimension 1 vector are the largest-magnitude negative number possible for the transformed data type, accuracy may be reduced.
- A ZDNN_3DS tensor is expected, where the *transformed_desc* dimension 1 (*dim1*) describes the vector, and *dim2* and *dim4* are used to batch multiple vector requests together. The *dim3* dimension must always be 1. The **zdnns_softmax_mask** operation is performed against the vector in *dim1*, repeating for each *dim1* vector in the *dim4* and *dim2* dimensions.
- Tensors that cannot be processed as vectors in dimension 1 or as batches of dimension 1 vectors must be coerced or reshaped by the caller.

When the entire tensor is to be processed by softmax, it can be coerced by creating an alternate descriptor prior to zDNN transformation. For example:

- A 4D tensor with *pre_transformed_desc* dimensions 2x2x2x2 and a data array of 16 FP32 entries could have an alternate ZDNN_3DS layout *pre_transformed_desc* using dimensions 8x1x2 and use the same original data array prior to the **zdnns_transform_ztensor** call. After transformation, such a tensor would be valid for the **zdnns_softmax_mask** operation.
- As a further example, the 4D 2x2x2x2 tensor could be processed as 8 batches of 2 vectors using a ZDNN_3DS layout *pre_transformed_desc* with dimensions 1x8x2.

Returns

One of the following **zdnns_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_ALLOCATION_FAILURE — A preallocated save area was not specified and internal allocation for the required memory failed.

[“Hardware statuses”](#) on page 9

ZDNN_FUNC_RC_F000 — Dimension 3 of the input tensor (*input->transformed_desc->dim3*) is not 1.

ZDNN_FUNC_RC_F001 — Invalid *act_func* value.

ZDNN_FUNC_RC_F002 — The value of *softmax_mask* exceeds dimension 1 of the input tensor.

Since

1.1.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- [TensorFlow softmax](https://www.tensorflow.org/api_docs/python/tf/nn/softmax) (https://www.tensorflow.org/api_docs/python/tf/nn/softmax)
- [ONNX softmax](https://onnx.ai/onnx/operators/onnx__Softmax.html#l-onnx-doc-softmax) (https://onnx.ai/onnx/operators/onnx__Softmax.html#l-onnx-doc-softmax)

Gaussian error linear unit (zdnngelu)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the following Gaussian error linear unit activation function is applied to the input tensor element-wise:

$$y = 0.5 * x * (1 + \tanh(x * 0.7978845608 * (1 + 0.044715 * x * x)))$$

Format

```
zdnngelu(const zdnngtensor *input, zdnngtensor *output);
```

Parameters

zdnngtensor *input

An input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnngtensor *output

An output tensor to hold the Gaussian error linear unit results of each value from the *input* tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Programming note

The range of certain input element values may result in an error of greater than 1% in the output element; however, the accuracy of properly conditioned models is not significantly degraded.

Returns

One of the following **zdnngstatus** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.1.0

Requirements

This function requires that:

- The **zdnng_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow Gaussian error linear unit (https://www.tensorflow.org/api_docs/python/tf/nn/gelu)
- ONNX Gaussian error linear unit (https://onnx.ai/onnx/operators/onnx__Gelu.html#l-onnx-doc-gelu)

Normalization operations

The zDNN normalization operations are:

- [“Mean reduce \(zdnn_meanreduce2d\)” on page 64](#)
- [“Batch norm \(zdnn_batchnorm\)” on page 65](#)
- [“Normalization \(zdnn_norm\)” on page 66](#)
- [“Moments \(zdnn_moments\)” on page 67](#)
- [“Layer normalization \(zdnn_layernorm\)” on page 68](#)
- [“Reduce \(zdnn_reduce\)” on page 70](#)

Mean reduce (zdnn_meanreduce2d)

Description

Given an input tensor in zDNN transformed format, produce a downsampled tensor reducing the middle dimensions to a size of 1 based on the mean of the original values and store the result into the provided output zDNN tensor.

Format

```
zdnn_status zdnn_meanreduce2d(const zdnn_ztensor *input, zdnn_ztensor *output);
```

Parameters

zdnn_ztensor *input

A ZDNN_NHWC tensor with a pre-transformed of [*batch_Num*, *Height*, *Width*, *Channel*]. The *Height* and *Width* dimensions must be less than or equal to 1024. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnn_ztensor *output

The output tensor to hold the result of the pooling operation in its buffer with shape as follows:

- The *output* dimensions *batch_Num* and *Channel* must be the same as the respective *input* dimensions.
- The *output* dimensions *Height* and *Width* must be 1.

The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Returns

One of the following **zdnn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

ZDNN_INVALID_SHAPE — The shape of the input or output tensor is invalid based on the given kernel and stride parameters.

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses” on page 9](#)

ZDNN_FUNC_RC_F001 — The input tensor has a *Height* or *Width* dimension greater than allowed for **zdnn_meanreduce2d**.

Since

1.0.0

Requirements

This function requires that:

- The **zdnn_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow [reduce mean](https://www.tensorflow.org/api_docs/python/tf/math/reduce_mean) with **axis** set for the Height and Width axes, and **keepdims** set to `true` (https://www.tensorflow.org/api_docs/python/tf/math/reduce_mean)
- ONNX [reduce mean](https://onnx.ai/onnx/operators/onnx__ReduceMean.html#l-onnx-doc-reducemean) (https://onnx.ai/onnx/operators/onnx__ReduceMean.html#l-onnx-doc-reducemean)

Batch norm (zdnm_batchnorm)

Description

Given 3 input zDNN tensors, *input_a*, *input_b*, and *input_c*, compute the batch-normalized result for each vector formed in dimension 1 according to the following formula:

$$output = input_b \times input_a + input_c$$

where *input_b* is a precomputed element-wise divide of scale and variance tensors, and *input_c* is a precomputed element-wise multiplication of $(-1) * \text{mean}$ and *input_b* + input bias tensors.

Format

```
zdnm_status zdnm_batchnorm(const zdnm_ztensor *input_a,
                           const zdnm_ztensor *input_b,
                           const zdnm_ztensor *input_c, zdnm_ztensor *output);
```

Parameters

zdnm_ztensor *input_a

Must be a 4D, ZDNN_NHWC input tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnm_ztensor *input_b

Must be a 1D tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnm_ztensor *input_c

Must be a 1D tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnm_ztensor *output

An output tensor of the same size as *input_a* to hold the computed value of the formula stated in [“Description”](#) on page 65. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Returns

One of the following **zdnm_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK

[“Warning statuses”](#) on page 8

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

[“Hardware statuses”](#) on page 9

Since

1.0.0

Requirements

This function requires that:

- The **zddn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- TensorFlow Batchnorm (https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization)
- ONNX Batchnorm (https://onnx.ai/onnx/operators/onnx__BatchNormalization.html#l-onnx-doc-batchnormalization)

Normalization (zddn_norm)

Description

Given 2 input tensors, *input_a* and *input_b*, in zDNN transformed format, produce the norm of the difference of vectors, calculated as follows:

1. Each element in dimension 1 of *input_b* is subtracted by the corresponding element of *input_a*.
2. The differences are squared.
3. The sum of the squared differences for dimension 1 is computed.
4. The square root of the sum is placed in the first element of dimension 1 of the output tensor.

Format

```
zddn_status zddn_norm(const zddn_ztensor *input_a, zddn_ztensor *input_b, zddn_ztensor *output);
```

Parameters

zddn_ztensor *input_a

Input tensor with values to evaluate. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

zddn_ztensor *input_b

Input tensor with values to evaluate. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

zddn_ztensor *output

Output tensor that contains the result of the normalization operation. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

Returns

One of the following **zddn_status** indicators (as described in “zDNN statuses” on page 8):

ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
“Hardware statuses” on page 9

Since

1.1.0

Requirements

This function requires that:

- The **zdnnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework example

- [TensorFlow normalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization)

Moments (zdnnpa_moments)

Description

Given an input tensor in zDNN transformed format and a *bessel correction type*, produce 2 output tensors containing the mean and variance, respectively, for the input tensor.

Format

```
zdnnpa_status zdnnpa_moments(const zdnnpa_tensor *input,
                             zdnnpa_moments_bessel_bessel_correction_type,
                             zdnnpa_tensor *output_a, zdnnpa_tensor *output_b);
```

Parameters

zdnnpa_tensor *input

Must be a 4D, ZDNN_NHWC input tensor. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnnpa_moments_bessel_bessel_correction_type

Bessel correction type to perform the moments operation, one of the following values:

MOMENTS_BESSEL_POPULATION
MOMENTS_BESSEL_SAMPLE

zdnnpa_tensor *output_a

The output tensor that will hold the mean. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnnpa_tensor *output_b

The output tensor that will hold the variance. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

Programming notes

- The **zdnnpa_moments** operation may be used in combination with the **zdnnpa_layernorm** operation. See [“Layer normalization \(zdnnpa_layernorm\)”](#) on page 68 for further guidance.
- When MOMENTS_BESSEL_SAMPLE is specified as the *bessel correction type*, all provided input dimensions of the input tensor must not be equal to 1.

Returns

One of the following **zdnnpa_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK
[“Warning statuses”](#) on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT
ZDNN_INVALID_BESSEL_CORRECTION
[“Hardware statuses” on page 9](#)

Since

1.1.0

Requirements

This function requires that:

- The **zdnns_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework example

- [TensorFlow moments](https://www.tensorflow.org/api_docs/python/tf/nn/moments) (https://www.tensorflow.org/api_docs/python/tf/nn/moments)

Layer normalization (zdnns_layernorm)

Description

Given 3 input tensors, *input_a*, *input_b*, and *input_c*, in zDNN transformed format, produce the layer normalization of the the given tensors, calculated as follows:

1. Each element in dimension 1 of *input_b* is subtracted by the corresponding element of *input_a*.
2. The corresponding element of *input_c* is added to epsilon.
3. The square root of the sum from step 2 is computed.
4. The difference from step 1 is divided by the result of step 3.
5. The quotient from step 4 is multiplied by gamma.
6. The product from step 5 is added to beta.
7. The result is stored in the corresponding element of the output tensor.

This calculation can be expressed as follows:

$$\text{layernorm}(a) = (a - b) / \text{sqrt}(c + \epsilon) * \gamma + \beta$$

Format

```
zdnns_status zdnns_layernorm(const
    zdnns_ztensor *input_a,
    const zdnns_ztensor *input_b,
    const zdnns_ztensor *input_c,
    float beta, float gamma, float epsilon,
    zdnns_ztensor *output);
```

Parameters

zdnns_ztensor *input_a

Must be a 4D, ZDNN_NHWC input tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnns_ztensor *input_b

Must be a 4D, ZDNN_NHWC input tensor that contains arithmetic means. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

zdnnp_ztensor *input_b

Must be a 4D, ZDNN_NHWC input tensor that contains arithmetic variances. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

float beta

Final result adjustment addend.

float gamma

Final result adjustment multiplier.

float epsilon

Intermediate variance adjustment.

zdnnp_ztensor *output

Output tensor that contains the result of the layer normalization operation. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

Programming notes

- The **zdnnp_layernorm** operation is intended to be used in combination with the **zdnnp_moments** normalization operation. The **zdnnp_moments** operation produces 2 output tensors containing the means and variances, respectively, of the dimension-4 index elements of the input tensor. The original input tensor to **zdnnp_moments** is intended to be used as the input tensor 1 to **zdnnp_layernorm**. The output tensors 1 and 2 of **zdnnp_moments** are intended to be used as input, as input tensor 2 and input tensor 3 of the **zdnnp_layernorm** operation.
-

Returns

One of the following **zdnnp_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_INVALID_BETA

ZDNN_INVALID_GAMMA

ZDNN_INVALID_EPSILON

[“Hardware statuses” on page 9](#)

Since

1.1.0

Requirements

This function requires that:

- The **zdnnp_is_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow layer normalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LayerNormalization) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LayerNormalization)
- [ONNX layer normalization](https://onnx.ai/onnx/operators/onnx__LayerNormalization.html#l-onnx-doc-layernormalization) (https://onnx.ai/onnx/operators/onnx__LayerNormalization.html#l-onnx-doc-layernormalization)

Reduce (zdn_reduce)

Description

Given an input tensor in zDNN transformed format, produce an output tensor where the specified reduction operation is performed.

Format

```
zdn_status zdn_reduce(const zdn_ztensor *input, void *save_area,  
                     zdn_reduce_ops op_type, zdn_ztensor *output);
```

Parameters

zdn_ztensor *input

The input tensor with values to evaluate. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

void *save_area

A preallocated memory address to use for temporary storage during internal operation processing. The preallocated memory must be at least 8K bytes in size, aligned on a 4K boundary.

If set to NULL, the operation determines, allocates, and frees storage automatically.

zdn_reduce_ops op_type

One of the following reduction operations to perform on the input tensor:

```
REDUCE_OP_MINIMUM  
REDUCE_OP_MINIMUM_IDX  
REDUCE_OP_MAXIMUM  
REDUCE_OP_MAXIMUM_IDX`
```

zdn_ztensor *output

The output tensor that holds the result of the reduction operation of each value from the input tensor. Output dimension 1 must be 1. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

The data type must be as follows:

FP32, FP16, or BFLOAT

When **op_type** is REDUCE_OP_MINIMUM or REDUCE_OP_MAXIMUM.

INT32

When **op_type** is REDUCE_OP_MINIMUM_IDX or REDUCE_OP_MAXIMUM_IDX.

The output when **op_type** is REDUCE_OP_MINIMUM or REDUCE_OP_MAXIMUM can be initialized using the following coding:

```
zdn_data_layouts input_layout = ZDNN_3DS;  
zdn_data_types input_type = FP32;  
  
uint32_t dim4 = 4;  
uint32_t dim2 = 5;  
uint32_t dim1 = 6;  
  
zdn_tensor_desc input_pre_transformed_desc;  
zdn_init_pre_transformed_desc(input_layout, input_type,  
                             &input_pre_transformed_desc, dim4, dim2, dim1);  
  
zdn_tensor_desc output_pre_transformed_desc;  
zdn_init_pre_transformed_desc(input_layout, input_type,  
                             &output_pre_transformed_desc, dim4, dim2, 1);
```

The output when **op_type** is REDUCE_OP_MINIMUM_IDX or REDUCE_OP_MAXIMUM_IDX can be initialized using the following coding:

```
zdn_data_layouts input_layout = ZDNN_3DS;
zdn_data_types input_type = FP32;

uint32_t dim4 = 4;
uint32_t dim2 = 5;
uint32_t dim1 = 6;

zdn_tensor_desc input_pre_transformed_desc;

zdn_init_pre_transformed_desc(input_layout, input_type,
                              &input_pre_transformed_desc, dim4, dim2, dim1);

zdn_data_types output_type = INT32;

zdn_tensor_desc output_pre_transformed_desc;

zdn_init_pre_transformed_desc(input_layout, output_type,
                              &output_pre_transformed_desc, dim4, dim2, 1);
```

Programming notes

- If a non-numeric element is encountered in a dimension-1 vector of the input tensor, then the resulting element in dimension 1 of the output tensor is unpredictable, and the range-violation status will be returned.
- When the reduction operation is REDUCE_OP_MINIMUM_IDX, the index of the first min value, from left-to-right, is returned when there are multiple elements with the same min value.
- When the reduction operation is REDUCE_OP_MAXIMUM_IDX, the index of the first max value, from left-to-right, is returned when there are multiple elements with the same max value.

Returns

One of the following **zdn_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK

[“Warning statuses” on page 8](#)

ZDNN_INVALID_SHAPE

ZDNN_INVALID_TYPE

ZDNN_INVALID_FORMAT

ZDNN_UNAVAILABLE_FUNCTION

ZDNN_ALLOCATION_FAILURE — A preallocated save area was not specified, and internal allocation for the required memory failed.

[“Hardware statuses” on page 9](#)

ZDNN_FUNC_RC_F000 — An invalid reduction operation (**op_type**) was specified.

Since

1.1.0

Requirements

This function requires that:

- The **zdn_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- TensorFlow reduce min and max
 - TensorFlow reduce minimum (https://www.tensorflow.org/api_docs/python/tf/math/reduce_min)
 - TensorFlow reduce maximum (https://www.tensorflow.org/api_docs/python/tf/math/reduce_max)
- ONNX reduce min and max
 - ONNX reduce minimum (https://onnx.ai/onnx/operators/onnx__ReduceMin.html#l-onnx-doc-reducemin)
 - ONNX reduce maximum (https://onnx.ai/onnx/operators/onnx__ReduceMax.html#l-onnx-doc-reducemax)

Matmul and matmul with broadcast

The zDNN matmul and matmul with broadcast functions are:

- “[zdnn_matmul_op](#)” on page 72
- “[zdnn_matmul_bcast_op](#)” on page 74
- “[zdnn_matmul_transpose_op](#)” on page 76
- “[zdnn_quantized_matmul_op](#)” on page 78

zdnn_matmul_op

Description

Given three input zDNN tensors, *input_a*, *input_b*, and *input_c*, determine the matrix multiplication of *input_a* \times *input_b*, then perform one of the following operations using *input_c* against the dot product, and store the result into the specified *output* zDNN tensor.

The operations are:

- Addition
- Compare — if dot product is greater than element
- Compare — if dot product is greater than or equal to element
- Compare — if dot product is equal to element
- Compare — if dot product is not equal to element
- Compare — if dot product is less than or equal to element
- Compare — if dot product is less than element

For an operation type of addition, *input_c* is added to the intermediate dot product.

For operation types of comparison, the intermediate dot product is compared to *input_c* and if the comparison is true, the result is set to 1; otherwise, it is set to 0.

The outermost dimension can optionally indicate that the inputs are stacks of matrices. The results for each matrix stack are independent of other stacks, but all stacks are calculated in a single call.

Format

```
zdn_status zdnn_matmul_op(const zdnn_ztensor *input_a,
                          const zdnn_ztensor *input_b,
                          const zdnn_ztensor *input_c,
                          zdnn_matmul_ops op_type, zdnn_ztensor *output);
```

Input and output requirements for matmul tensors

- All tensors must either be stacked or unstacked.
- The tensor must satisfy the requirements in “[General zTensor requirements](#)” on page 5.

- [Table 7 on page 73](#) lists the requirements for *pre_transformed_desc* and shape requirements for each tensor.

Table 7. Requirements for *pre_transformed_desc* and shape for matmul tensors

| Type | <i>input_a</i> | <i>input_b</i> | <i>input_c</i> | <i>output</i> |
|-----------|--------------------|--------------------|-----------------|--------------------|
| Unstacked | ZDNN_2D (m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_2D (m, p) |
| Stacked | ZDNN_3DS (s, m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |

Parameters

zdnnp_tensor *input_a

An input tensor with the first matrix for multiplication. The pre-transformed shape and layout must be as described in [“Input and output requirements for matmul tensors” on page 72](#).

zdnnp_tensor *input_b

An input tensor with the second matrix for multiplication. The pre-transformed shape and layout must be as described in [“Input and output requirements for matmul tensors” on page 72](#).

zdnnp_tensor *input_c

An input tensor that will have the requested operation performed against the intermediate dot product of *input_a* and *input_b*. The pre-transformed shape and layout must be as described in [“Input and output requirements for matmul tensors” on page 72](#).

zdnnp_matmul_ops op_type

The operation to perform on the dot product. Valid values are:

```
MATMUL_OP_ADDITION
MATMUL_OP_GREATER
MATMUL_OP_GREATER_EQUAL
MATMUL_OP_EQUAL
MATMUL_OP_NOT_EQUAL
MATMUL_OP_LESSER_EQUAL
MATMUL_OP_LESSER
```

zdnnp_tensor *output

The output tensor that will hold the result of the operation in its buffer. The pre-transformed shape and layout must be as described in [“Input and output requirements for matmul tensors” on page 72](#).

Returns

One of the following **zdnnp_status** indicators (as described in [“zDNN statuses” on page 8](#)):

```
ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
“Hardware statuses” on page 9
ZDNN_FUNC_RC_F000 — Invalid op_type value
```

Since

1.0.0

Requirements

This function requires that:

- The **zdnnp_is_nnpa_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow matmul (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/mat-mul)
- ONNX matmul (https://onnx.ai/onnx/operators/onnx__MatMul.html#l-onnx-doc-matmul)

zdsn_matmul_bcast_op

Description

Given three input zDNN tensors, *input_a*, *input_b*, and *input_c*, determine the matrix multiplication of $input_a \times input_b + bias$, then perform one of the following operations using *input_c* against the dot product, and store the result into the specified *output* zDNN tensor.

The operations are:

- Addition
- Compare — if dot product is greater than element
- Compare — if dot product is greater or equal to element
- Compare — if dot product is equal to element
- Compare — if dot product is not equal to element
- Compare — if dot product is less than or equal to element
- Compare — if dot product is less than element

When an input has the **ZDNN_3DS** layout, the outermost dimension for that input can optionally indicate that the input is a stack of matrices. Likewise, when an input has the **ZDNN_2DS** layout, the outermost dimension for that input can optionally indicate that the input is a stack of vectors.

For instance, if *input_a* has the **ZDNN_3DS** layout, each stack of *input_a* is multiplied by the same *input_b* matrix and *input_c* vector which are broadcast over each stack of *input_a*. Results for each stack are returned in the corresponding stack index of *output*.

Similarly, if *input_b* has the **ZDNN_3DS** layout and *input_c* has the **ZDNN_2DS** layout, each stack of *input_b* is multiplied by the same *input_a* matrix which is broadcast over each stack of *input_b* and *input_c*. Results for each stack are returned in the corresponding stack index of *output*.

The outermost dimension for *input_a* can optionally indicate that the input is a stack of matrices. Each stack of *input_a* is multiplied by the same *input_b* matrix and *input_c* which are broadcast over each stack of *input_a*. The results for each stack are returned in the corresponding stack index of *output*.

Format

```
zdsn_status zdsn_matmul_bcast_op(const zdsn_ztensor *input_a,
                                const zdsn_ztensor *input_b,
                                const zdsn_ztensor *input_c,
                                zdsn_matmul_bcast_ops op_type,
                                zdsn_ztensor *output);
```

Input and output requirements for matmul broadcast tensors

- The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.
- Table 8 on page 74 lists the *pre_transformed_desc* and shape requirements for each tensor.

Table 8. Requirements for *pre_transformed_desc* and shape for matmul broadcast tensors

| Type | <i>input_a</i> | <i>input_b</i> | <i>input_c</i> | <i>output</i> |
|-----------|--------------------|--------------------|-----------------|--------------------|
| unstacked | ZDNN_2D (m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_2D (m, p) |
| stacked | ZDNN_3DS (s, m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |

Table 8. Requirements for *pre_transformed_desc* and shape for *matmul broadcast tensors* (continued)

| Type | <i>input_a</i> | <i>input_b</i> | <i>input_c</i> | <i>output</i> |
|---------|--------------------|--------------------|-----------------|--------------------|
| bcast1 | ZDNN_2D (m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |
| bcast23 | ZDNN_3DS (s, m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_3DS (s, m, p) |

Parameters

zdnntensor *input_a

An input tensor with the first matrix for multiplication. The pre-transformed shape and layout must be as described in [Table 8 on page 74](#).

zdnntensor *input_b

An input tensor with the second matrix for multiplication. The pre-transformed shape and layout must be as described in [Table 8 on page 74](#).

zdnntensor *input_c

An input tensor that will have the requested operation performed against the intermediate dot product for each *m* dimension in *output*. The pre-transformed shape and layout must be as described in [Table 8 on page 74](#).

zdnntensor *matmul_bcast_ops op_type

The operation to perform on the dot product. Valid values are:

MATMUL_BCAST_OP_ADDITION
MATMUL_BCAST_OP_GREATER
MATMUL_BCAST_OP_GREATER_EQUAL
MATMUL_BCAST_OP_EQUAL
MATMUL_BCAST_OP_NOT_EQUAL
MATMUL_BCAST_OP_LESSER_EQUAL
MATMUL_BCAST_OP_LESSER

zdnntensor *output

The output tensor that will hold the result of the operation in its buffer. The pre-transformed shape and layout must be as described in [Table 8 on page 74](#).

Programming notes

- When the NNPA_PARMBLKFORMAT_1 parameter block format is not installed, **zdnntensor *matmul_bcast_ops** only supports the MATMUL_BCAST_OP_ADDITION operation (*op_type*). If any other value is specified for *op_type*, the ZDNN_UNAVAILABLE_FUNCTION status is returned.
- BCAST1 is not supported when the NNPA_PARMBLKFORMAT_1 parameter block format is not installed and returns the ZDNN_UNAVAILABLE_FUNCTION status.
- Carefully compare values for equality or inequality because the order of operations and rounding may produce what appear to be slightly different values when they are essentially the same value.

Returns

One of the following **zdnntensor *status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
ZDNN_UNAVAILABLE_FUNCTION
[“Hardware statuses” on page 9](#)
ZDNN_FUNC_RC_F000 — Invalid operation (*op_type*).
ZDNN_FUNC_RC_F001 — Invalid input/output type or format combination.

Since

1.0.0

Requirements

This function requires that:

- The **zdsn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time for the following cases:
 - The `bcast1` data layout.
 - The following operations (*op_type*):

MATMUL_BCAST_OP_GREATER
MATMUL_BCAST_OP_GREATER_EQUAL
MATMUL_BCAST_OP_EQUAL
MATMUL_BCAST_OP_NOT_EQUAL
MATMUL_BCAST_OP_LESSER_EQUAL
MATMUL_BCAST_OP_LESSER

- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- TensorFlow `matmul` (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/mat-mul)
- ONNX `matmul` (https://onnx.ai/onnx/operators/onnx__MatMul.html#l-onnx-doc-matmul)

zdsn_matmul_transpose_op

Description

Given three input zDNN tensors, *input_a*, *input_b*, and *input_c*, determine the matrix multiplication of $input_a \times input_b$, then perform one of the following operations using *input_c* against the dot product, and store the result into the specified *output* zDNN tensor.

The operations are:

- Addition
- Compare — if dot product is greater than element
- Compare — if dot product is greater or equal to element
- Compare — if dot product is equal to element
- Compare — if dot product is not equal to element
- Compare — if dot product is less than or equal to element
- Compare — if dot product is less than element

For the addition operation type, the *input_c* tensor is added to the intermediate dot product. For the comparison operation types, the intermediate dot product is compared to *input_c* and if the comparison is true, the result is set to 1; otherwise, it is set to 0.

The outermost dimension can optionally indicate that the inputs are stacks of matrices. The results for each matrix stack is independent of other stacks, but all stacks are calculated in a single call.

Format

```
zdsn_status zdsn_matmul_transpose_op(const zdsn_ztensor *input_a,  
                                     const zdsn_ztensor *input_b,  
                                     const zdsn_ztensor *input_c,
```

```
bool transpose_a, bool transpose_b,
zdnn_matmul_ops op_type,
zdnn_ztensor *output);
```

Input and output tensor requirements for matmul transpose

- The tensor must satisfy the requirements in “General zTensor requirements” on page 5.
- All tensors must either be stacked or unstacked.
- Table 9 on page 77 lists the *pre_transformed_desc* and shape requirements for each tensor.

Table 9. Requirements for *pre_transformed_desc* and shape for matmul transpose tensors

| Type | <i>input_a</i> | <i>input_b</i> | <i>input_c</i> | <i>output</i> |
|-----------|--------------------|--------------------|-----------------|--------------------|
| unstacked | ZDNN_2D (m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_2D (m, p) |
| stacked | ZDNN_3DS (s, m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |
| bcast1 | ZDNN_2D (m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |
| bcast23 | ZDNN_3DS (s, m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_3DS (s, m, p) |

Parameters

zdnn_ztensor *input_a

An input tensor with the first matrix for multiplication. The pre-transformed shape and layout must be as described in Table 9 on page 77.

zdnn_ztensor *input_b

An input tensor with the second matrix for multiplication. The pre-transformed shape and layout must be as described in Table 9 on page 77.

zdnn_ztensor *input_c

An input tensor that will have the requested operation performed against the intermediate dot product of *input_a* and *input_b*. The pre-transformed shape and layout must be as described in Table 9 on page 77.

bool transpose_a

Indicates whether to transpose *input_a* prior to the dot product. If `true`, *input_a* must have the unstacked dimensions (n, m) or stacked dimensions (s, n, m).

bool transpose_b

Indicates whether to transpose *input_b* prior to the dot product. If `true`, *input_b* must have the unstacked dimensions (p, n) or stacked dimensions (s, p, n).

zdnn_matmul_ops op_type

The operation to perform on the dot product. Valid values are:

```
MATMUL_BCAST_OP_ADDITION
MATMUL_BCAST_OP_GREATER
MATMUL_BCAST_OP_GREATER_EQUAL
MATMUL_BCAST_OP_EQUAL
MATMUL_BCAST_OP_NOT_EQUAL
MATMUL_BCAST_OP_LESSER_EQUAL
MATMUL_BCAST_OP_LESSER
```

zdnn_ztensor *output

The output tensor that will hold the result of the operation in its buffer. The pre-transformed shape and layout must be as described in Table 9 on page 77.

Programming notes

- The **zdnn_matmul_transpose_op** operation is not supported when the NNPA_PARMBLKFORMAT_1 parameter block format is not installed and will return the ZDNN_UNAVAILABLE_FUNCTION status.

- Carefully compare values for equality or inequality because the order of operations and rounding may produce what appear to be slightly different values when they are essentially the same value.

Returns

One of the following **zdnnc_status** indicators (as described in [“zDNN statuses”](#) on page 8):

ZDNN_OK
 ZDNN_INVALID_SHAPE
 ZDNN_INVALID_TYPE
 ZDNN_INVALID_FORMAT
 ZDNN_UNAVAILABLE_FUNCTION
[“Hardware statuses”](#) on page 9
 ZDNN_FUNC_RC_F000 — Invalid operation (*op_type*).
 ZDNN_FUNC_RC_F001 — Invalid input/output type or format combination.

Since

1.1.0

Requirements

This function requires that:

- The **zdnnc_is_nnapi_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.
- For matmul transpose, the underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time”](#) on page 11.

Framework examples

- TensorFlow matmul (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/mat-mul)
- ONNX matmul (https://onnx.ai/onnx/operators/onnx__MatMul.html#l-onnx-doc-matmul)

zdnnc_quantized_matmul_op

Description

Given three input zDNN tensors, *input_a*, *input_b*, and *input_c*, determine the matrix multiplication of *input_a* \times *input_b*, then perform one of the following operations using *input_c* against the dot product, and store the result into the specified *output* zDNN tensor.

The operations are:

- Addition
- Compare — if dot product is greater than element
- Compare — if dot product is greater than or equal to element
- Compare — if dot product is equal to element
- Compare — if dot product is not equal to element
- Compare — if dot product is less than or equal to element
- Compare — if dot product is less than element

For the addition operation type, *input_c* is added to the intermediate dot product.

For the compare operation types, the intermediate dot product is compared to *input_c*. If the comparison is true, the result is set to 1; otherwise, it is set to 0.

The outermost dimension can optionally indicate that the inputs are stacks of matrices. The results for each matrix stack are independent of other stacks, but all stacks are calculated in a single call.

When the *dequantize* parameter is true, the output is dequantized after computation.

When the *pre_computed* parameter is true, the pre-computed value of *input_c* for the addition operation can be obtained as follows:

```
Za = input_a->offset;
Sa = 1 / input_a->rec_scale;

Zb = input_b->offset;
Sb = 1 / input_b->rec_scale;

Zc = input_c->offset;
Sc = 1 / input_c->rec_scale;

Zy = output->offset;
Sy = 1 / output->rec_scale;

N = input_b->pre_transformed_desc->dim2;

pre_computed = Zy - (Sc/Sy) * Zc - (Sc/Sy) * input_c + ((Sa * Sb) / Sy) * NZaZb;
```

When the *pre_computed* parameter is true, the pre-computed value of *input_c* for the compare operations can be obtained as follows:

```
Za = input_a->offset;
Sa = 1 / input_a->rec_scale;

Zb = input_b->offset;
Sb = 1 / input_b->rec_scale;

Zc = input_c->offset;
Sc = 1 / input_c->rec_scale;

pre_computed = Sc / (Sa * Sb) * (input_c - Zc) + Za * sum(input_b, axis=-2)
```

Format

```
zdn_status zdn_quantized_matmul_op(const zdn_ztensor *input_a,
                                   const zdn_ztensor *input_b,
                                   const zdn_ztensor *input_c,
                                   zdn_matmul_ops op_type,
                                   const int8_t clip_min,
                                   const int8_t clip_max,
                                   const bool disable_clipping,
                                   const bool dequantize,
                                   const bool pre_computed,
                                   void *work_area,
                                   zdn_ztensor *output);
```

Input and output requirements for quantized matmul tensors

- All tensors must either be stacked or unstacked.
- The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.
- [Table 10 on page 80](#) lists the requirements for *pre_transformed_desc* data layout and shape for each tensor.
- All tensors must use **zdn_generate_quantized_transformed_desc** when generating transformed descriptors and pass the appropriate **zdn_quantized_transform_types** value.
- All quantized tensors must be initialized with **zdn_init_quantized_ztensor** or **zdn_init_quantized_ztensor_with_malloc** and passing the *scale* and *offset* quantization parameters.

scale

Must be in the range $(-DLFLT_MAX \leq scale \leq DLFLT_MAX)$, and *scale* must not be 0. (See [“Get maximum limit for a data type \(zdn_get_max_limit\)”](#) on page 23.)

offset

Must be in the range $(-DLFLT_MAX \leq offset \leq DLFLT_MAX)$. (See [“Get maximum limit for a data type \(zdnng_get_max_limit\)”](#) on page 23.)

- All quantized input tensors must use **zdnng_transform_quantized_ztensor** to transform, and pass the *clip_min* and *clip_max* quantization parameters.

Table 10. Requirements for *pre_transformed_desc* data layout and shape for quantized matmul tensors

| Type | input_a | input_b | input_c | output |
|-----------|--------------------|--------------------|-----------------|--------------------|
| Unstacked | ZDNN_2D (m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_2D (m, p) |
| Stacked | ZDNN_3DS (s, m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |
| bcast1 | ZDNN_2D (m, n) | ZDNN_3DS (s, n, p) | ZDNN_2DS (s, p) | ZDNN_3DS (s, m, p) |
| bcast23 | ZDNN_3DS (s, m, n) | ZDNN_2D (n, p) | ZDNN_1D (p) | ZDNN_3DS (s, m, p) |

Table 11. Requirements for *zdnng_quantized_transform_types* for quantized matmul tensors

| Type | input_a | input_b | input_c | output |
|------------|---------------------|------------------------|----------------|---------------------|
| Normal | QUANTIZED_INT8 | QUANTIZED_WEIGHTS_INT8 | QUANTIZED_INT8 | QUANTIZED_DLFLOAT16 |
| On-the-fly | QUANTIZED_DLFLOAT16 | QUANTIZED_WEIGHTS_INT8 | QUANTIZED_INT8 | QUANTIZED_DLFLOAT16 |

Parameters**zdnng_ztensor *input_a**

An input tensor with the first matrix for multiplication. The pre-transformed shape and layout must be as described in [“Input and output requirements for quantized matmul tensors”](#) on page 79.

zdnng_ztensor *input_b

An input tensor with the second matrix for multiplication. The pre-transformed shape and layout must be as described in [“Input and output requirements for quantized matmul tensors”](#) on page 79.

zdnng_ztensor *input_c

An input tensor that will have the requested operation performed against the intermediate dot product of *input_a* and *input_b*. The pre-transformed shape and layout must be as described in [“Input and output requirements for quantized matmul tensors”](#) on page 79.

int8_t clip_min

The minimum quantized value for *input_a* prior to dot product. The value must be less than *clip_max* and only applies when performing on-the-fly quantization.

int8_t clip_max

The maximum quantized value for *input_a* prior to dot product. The value must be greater than *clip_min* and only applies when performing on-the-fly quantization.

bool disable_clipping

When true, disables clipping and rounding.

bool dequantize

Indicates whether to dequantize the returned zTensor.

bool pre_computed

Indicates whether bias is already computed.

void *work_area

A preallocated memory address to use for temporary storage during internal operation processing. If set to NULL, the operation determines, allocates, and frees storage automatically. The amount of required storage is the same as *input_c->buffer_size*. The start of the storage buffer must be 4K-aligned

zdnng_matmul_ops op_type

The operation to perform on the dot product. Valid values are:

MATMUL_OP_ADDITION
MATMUL_OP_GREATER
MATMUL_OP_GREATER_EQUAL
MATMUL_OP_EQUAL
MATMUL_OP_NOT_EQUAL
MATMUL_OP_LESSER_EQUAL
MATMUL_OP_LESSER

zdnnp_tensor *output

The output tensor that will hold the result of the operation in its buffer. The pre-transformed shape and layout must be as described in [“Input and output requirements for quantized matmul tensors” on page 79](#).

Programming notes

- The **zdnnp_quantized_matmul_op** operation is not supported when NNPA_PARMBLKFORMAT_1 is not installed and will return the ZDNN_UNAVAILABLE_FUNCTION status.
- Carefully compare values for equality or inequality because the order of operations and rounding may produce what appear to be slightly different values when they are essentially the same value.

Returns

One of the following **zdnnp_status** indicators (as described in [“zDNN statuses” on page 8](#)):

ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
ZDNN_INVALID_SCALE
ZDNN_INVALID_OFFSET
ZDNN_INVALID_CLIPPING_VALUE
ZDNN_UNAVAILABLE_FUNCTION
[“Hardware statuses” on page 9](#)
ZDNN_FUNC_RC_F000 — Invalid *op_type* value
ZDNN_FUNC_RC_F001 — Invalid input/output type or format combination
ZDNN_FUNC_RC_F002 — Invalid input/output scale.

Since

1.1.0

Requirements

This function requires that:

- The **zdnnp_is_nnnp_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.1.x or later at run time.

See [“Validating the environment at run time” on page 11](#).

Framework examples

- [TensorFlow quantized matmul](https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/quantized-mat-mul) (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/quantized-mat-mul)
- [ONNX quantize linear](https://onnx.ai/onnx/operators/onnx__QuantizeLinear.html#l-onnx-doc-quantizelinear) (https://onnx.ai/onnx/operators/onnx__QuantizeLinear.html#l-onnx-doc-quantizelinear)

LSTM (zdnn_lstm)

Description

Implements the Long-Short Term Memory¹ (LSTM) layer. The following formula is computed for the input tensor, t , for all time steps. The defaults are: f = Sigmoid, g = Tanh, h = Tanh.

```
it = f(Xt*(Wi^T) + Ht-1*(Ri^T) + Wbi + Rbi)
ft = f(Xt*(Wf^T) + Ht-1*(Rf^T) + Wbf + Rbf)
ct = g(Xt*(Wc^T) + Ht-1*(Rc^T) + Wbc + Rbc)
Ct = ft (.) Ct-1 + it (.) ct
ot = f(Xt*(Wo^T) + Ht-1*(Ro^T) + Wbo + Rbo)
Ht = ot (.) h(Ct)
```

Format

```
zdnn_status zdnn_lstm(const zdnn_ztensor *input, const zdnn_ztensor *h0,
                      const zdnn_ztensor *c0, const zdnn_ztensor *weights,
                      const zdnn_ztensor *biases,
                      const zdnn_ztensor *hidden_weights,
                      const zdnn_ztensor *hidden_biases,
                      lstm_gru_direction direction, void *work_area,
                      zdnn_ztensor *hn_output, zdnn_ztensor *cf_output);
```

For an example of the calling syntax, see [“Example: Calling the zdnn_lstm API \(forward\)”](#) on page 110.

LSTM input / output requirements

Any *num_hidden* dimension must be less than or equal to $(\text{zdnn_get_max_for_dim}(2) / 4)$ elements.

Parameters

zdnn_ztensor *input

An input tensor with shape $(\text{num_timesteps}, \text{num_batches}, \text{num_features})$ prior to transformation by the **zdnn_transform_ztensor** function. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5.

zdnn_ztensor *h0

An input tensor that contains the initial hidden state with shape $(\text{num_dirs}, \text{num_batches}, \text{num_hidden})$ prior to transformation by the **zdnn_transform_ztensor** function. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5. The tensor must also satisfy the requirements in [“LSTM input / output requirements”](#) on page 82.

zdnn_ztensor *c0

An input tensor that contains the initial cell state with shape $(\text{num_dirs}, \text{num_batches}, \text{num_hidden})$ prior to transformation by the **zdnn_transform_ztensor** function. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5. The tensor must also satisfy the requirements in [“LSTM input / output requirements”](#) on page 82.

zdnn_ztensor *weights

An input tensor that contains the concatenated input connection weights in Forget, Input, Cell, Output (FICO) order. Prior to transformation, each gate must be transposed to shape $(\text{num_dirs}, \text{num_features}, \text{num_hidden})$ by the caller. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The **zdnn_concat_info** field must have the following flags turned on:

¹ Hochreiter, 1997.

- RNN_TYPE_LSTM
- USAGE_WEIGHTS
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in [“Concatenated zTensor requirements” on page 5](#)) and [“LSTM input / output requirements” on page 82](#).

zdnntensor *biases

An input tensor that contains the concatenated input connection bias in FICO order. Prior to transformation, each gate must be of shape (num_dirs, num_hidden) . The value of *pre_transformed_desc->layout* must be ZDNN_2DS. The **zdnncat_info** field must have the following flags turned on:

- RNN_TYPE_LSTM
- USAGE_BIASES
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in [“Concatenated zTensor requirements” on page 5](#)) and [“LSTM input / output requirements” on page 82](#).

zdnntensor *hidden_weights

An input tensor that contains the concatenated hidden connection weights in FICO order. Prior to transformation, each gate must be transposed to shape $(num_dirs, num_hidden, num_hidden)$ by the caller. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The **zdnncat_info** field must have the following flags turned on:

- RNN_TYPE_LSTM
- USAGE_HIDDEN_WEIGHTS
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in [“Concatenated zTensor requirements” on page 5](#)) and [“LSTM input / output requirements” on page 82](#).

zdnntensor *hidden_biases

An input tensor that contains the concatenated hidden connection bias in FICO order. Prior to transformation, each gate must be of shape (num_dirs, num_hidden) . The value of *pre_transformed_desc->layout* must be ZDNN_2DS. The **zdnncat_info** field must have the following flags turned on:

- RNN_TYPE_LSTM
- USAGE_HIDDEN_BIASES
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in [“Concatenated zTensor requirements”](#) on page 5) and [“LSTM input / output requirements”](#) on page 82.

lstm_gru_direction direction

A direction indicator of **lstm_gru_direction direction** type. Valid values are:

- FWD (forward)
- BWD (backward)
- BIDIR (bidirectional)

For input and output shapes, the *num_dirs* dimension should be:

- 1 for unidirectional calls, such as FWD or BWD
- 2 for bidirectional calls, such that:
 - Dimension 0 contains FWD values.
 - Dimension 1 contains BWD values.

void *work_area

The address of preallocated memory to use for temporary storage during internal operation processing. If set to NULL, the operation determines, allocates, and frees storage automatically. The amount of required storage can be determined given the LSTM *num_timesteps*, *num_batches*, and *num_hidden* values. For bidirectional operations, twice the amount of contiguous storage is required. The start of the storage buffer must be 4K-aligned.

The following sample code creates a zTensor descriptor that is an equivalent size of the required *work_area*. To use this sample code, replace the *num_timesteps*, *num_batches*, and *num_hidden* variables with your own values.

```
zdnntensor_desc desc;  
desc.dim4 = (4 * num_timesteps) + 6;  
desc.dim3 = 1;  
desc.dim2 = num_batches;  
desc.dim1 = num_hidden;  
uint64_t work_area_size = zdnntensor_getsize_ztensor(&desc);
```

zdnntensor *hn_output

An output tensor to hold the results of the hidden states. The value of *pre_transformed_desc->layout* must be ZDNN_4DS.

The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5. The tensor must also satisfy the requirements in [“LSTM input / output requirements”](#) on page 82.

Output pre-transformed shapes:

- All timesteps: (*num_timesteps*, *num_dirs*, *num_batches*, *num_hidden*)
- Final timestep only: (1, *num_dirs*, *num_batches*, *num_hidden*)

For bidirectional (BIDIR) output:

- Forward and backward results are concatenated on the innermost dimension.
- Can be used directly as input for subsequent RNN layers without needing untransformation. Cannot be used directly as input for other non-RNN zDNN operations.
- Untransformation is supported.

Note that for BWD and the backward component of BIDIR directions, the output order matches the order of the input, not the processing order. For instance, the first input time step is the last to be processed, and its result is the first time step of the output.

zdnntensor *cf_output

An output tensor to hold the results of the cell state for the last processed time step. The value of **pre_transformed_desc->layout** must be ZDNN_4DS. The tensor must satisfy the requirements in [“General zTensor requirements”](#) on page 5. The tensor must also satisfy the requirements in [“LSTM input / output requirements”](#) on page 82.

Output pre-transformed shapes: $(1, num_dirs, num_batches, num_hidden)$

For bidirectional (BIDIR) output:

- Forward and backward results are concatenated on the innermost dimension.
- Cannot be used directly as input for other non-RNN zDNN operations.
- Untransformation of output is supported.

Summary

Table 12. Summary of `zdn_lstm` parameters

| Parameter | Pre-transformed layout | Pre-transformed shape | To create transformed descriptor |
|-----------------------|------------------------|---|---|
| input | ZDNN_3DS | $(num_timesteps, num_batches, num_features)$ | zdn_generate_transformed_desc |
| h0 | ZDNN_3DS | $(num_dirs, num_batches, num_hidden)$ | zdn_generate_transformed_desc |
| c0 | ZDNN_3DS | $(num_dirs, num_batches, num_hidden)$ | zdn_generate_transformed_desc |
| weights | ZDNN_3DS | $(num_dirs, num_features, num_hidden)$ | zdn_generate_transformed_desc_concatenated RNN_TYPE_LSTM + USAGE_WEIGHTS + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| biases | ZDNN_2DS | (num_dirs, num_hidden) | zdn_generate_transformed_desc_concatenated RNN_TYPE_LSTM + USAGE_BIASES + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hidden_weights | ZDNN_3DS | $(num_dirs, num_hidden, num_hidden)$ | zdn_generate_transformed_desc_concatenated RNN_TYPE_LSTM + USAGE_HIDDEN_WEIGHTS + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hidden_biases | ZDNN_2DS | (num_dirs, num_hidden) | zdn_generate_transformed_desc_concatenated RNN_TYPE_LSTM + USAGE_HIDDEN_BIASES + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hn_output | ZDNN_4DS | $(num_timesteps, num_dirs, num_batches, num_hidden)$ (Last time step only when $num_timesteps = 1$) | zdn_generate_transformed_desc |
| cf_output | ZDNN_4DS | $(1, num_dirs, num_batches, num_hidden)$ | zdn_generate_transformed_desc |

Returns

One of the following **zdn_status** indicators (as described in “zDNN statuses” on page 8):

| Status | Meaning |
|-------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_TYPE | See “General failing statuses” on page 8. |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none">• The <i>hn_output</i> timesteps dimension must be 1 or the same size as the <i>input</i> timestep dimension.• All tensors with a direction dimension have the same direction dimension size.• The <i>input</i> timestep dimension must be greater than or equal to 1.• Other general shape violations (such as, exceeds MDIS, and others) |
| ZDNN_INVALID_DIRECTION | The specified <i>direction</i> parameter is not a recognized lstm_gru_direction direction. |
| ZDNN_ALLOCATION_FAILURE | A preallocated <i>work_area</i> was not specified and internal allocation for the required memory failed. |

| Status | Meaning |
|-------------------|------------------------------------|
| Hardware statuses | See “Hardware statuses” on page 9. |

Since

1.0.0

Requirements

This function requires that:

- The **zddn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- TensorFlow LSTM (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell)
- ONNX LSTM (https://onnx.ai/onnx/operators/onnx__LSTM.html#l-onnx-doc-lstm)

GRU (zddn_gru)

Description

Implements Gated Recurrent Unit² (GRU), and only supports reset after linear.

The following formula is computed for the input tensor, *input(t)*, for all time steps. The defaults are: f = Sigmoid, g = Tanh.

```

zt = f(Xt*(Wz^T) + Ht-1*(Rz^T) + Wbz + Rbz)
rt = f(Xt*(Wr^T) + Ht-1*(Rr^T) + Wbr + Rbr)
ht = g(Xt*(Wh^T) + (rt (.) (Ht-1*(Rh^T) + Rbh)) + Wbh)
Ht = (1 - zt) (.) ht + zt (.) Ht-1

```

Format

```

zddn_status zddn_gru(const zddn_ztensor *input, const zddn_ztensor *h0,
                    const zddn_ztensor *weights, const zddn_ztensor *biases,
                    const zddn_ztensor *hidden_weights,
                    const zddn_ztensor *hidden_biases,
                    lstm_gru_direction direction, void *work_area,
                    zddn_ztensor *hn_output);

```

For an example of the calling syntax, see “Example: Calling the zddn_gru API (forward)” on page 122.

GRU input / output requirements

Any *num_hidden* dimension must be less than or equal to (zddn_get_max_for_dim(2) / 3) elements.

Parameters

zddn_ztensor *input

An input tensor with shape (*num_timesteps*, *num_batches*, *num_features*) prior to transformation by the **zddn_transform_ztensor** function. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

² Kyunghyun Cho, 2014.

zdnns_ztensor *h0

An input tensor that contains the initial hidden state with shape (*num_dirs*, *num_batches*, *num_hidden*) prior to transformation by the **zdnns_transform_ztensor** function. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The tensor must satisfy the requirements in “General zTensor requirements” on page 5. The tensor must also satisfy the requirements in “GRU input / output requirements” on page 86.

zdnns_ztensor *weights

An input tensor that contains the concatenated input connection weights in (Z)update, Reset, Hidden (ZRH) order. Prior to transformation, each gate must be transposed to shape (*num_dirs*, *num_features*, *num_hidden*) by the caller. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The **zdnns_concat_info** field must have the following flags turned on:

- RNN_TYPE_GRU
- USAGE_WEIGHTS
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in “Concatenated zTensor requirements” on page 5 and “GRU input / output requirements” on page 86.

zdnns_ztensor *biases

An input tensor that contains the concatenated input connection bias in ZRH order. Prior to transformation, each gate must be of shape (*num_dirs*, *num_hidden*). The value of *pre_transformed_desc->layout* must be ZDNN_2DS. The **zdnns_concat_info** field must have the following flags turned on:

- RNN_TYPE_GRU
- USAGE_BIASES
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in “Concatenated zTensor requirements” on page 5 and “GRU input / output requirements” on page 86.

zdnns_ztensor *hidden_weights

An input tensor that contains the concatenated hidden connection weights in ZRH order. Prior to transformation, each gate must be transposed to shape (*num_dirs*, *num_hidden*, *num_hidden*) by the caller. The value of *pre_transformed_desc->layout* must be ZDNN_3DS. The **zdnns_concat_info** field must have the following flags turned on:

- RNN_TYPE_GRU
- USAGE_HIDDEN_WEIGHTS
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in “Concatenated zTensor requirements” on page 5 and “GRU input / output requirements” on page 86.

zdnns_ztensor *hidden_biases

An input tensor that contains the concatenated hidden connection bias in ZRH order. Prior to transformation, each gate must be of shape (*num_dirs*, *num_hidden*). The value of

pre_transformed_desc->layout must be ZDNN_2DS. The **zdn_concat_info** field must have the following flags turned on:

- RNN_TYPE_GRU
- USAGE_HIDDEN_BIASES
- Appropriate **PREV_LAYER** flag:
 - PREV_LAYER_NONE, if *input* tensor is not from a previous RNN layer.
 - PREV_LAYER_UNI, if *input* tensor is unidirectional output from a previous RNN layer.
 - PREV_LAYER_BIDIR, if *input* tensor is bidirectional output from a previous RNN layer.

The tensor must satisfy the requirements in [“Concatenated zTensor requirements” on page 5](#) and [“GRU input / output requirements” on page 86](#).

lstm_gru_direction direction

A direction indicator of **lstm_gru_direction direction** type. Valid values are:

- FWD (forward)
- BWD (backward)
- BIDIR (bidirectional)

For input shapes, the *num_dirs* dimension should be:

- 1 for unidirectional calls, such as FWD or BWD
- 2 for bidirectional calls, such that:
 - Dimension 0 contains FWD values.
 - Dimension 1 contains BWD values.

void work_area

The address of preallocated memory to use for temporary storage during internal operation processing. If set to NULL, the operation determines, allocates, and frees storage automatically. The amount of required storage can be determined from the GRU timestep, batch, and hidden values. For bidirectional operations, twice the amount of contiguous storage is required. The start of the buffer storage must be 4K-aligned.

The following sample code creates a zTensor descriptor that is an equivalent size of the required *work_area*. To use this sample code, replace the *num_timesteps*, *num_batches*, and *num_hidden* variables with your own values.

```
zdn_tensor_desc desc;  
desc.dim4 = (3 * num_timesteps) + 5;  
desc.dim3 = 1;  
desc.dim2 = num_batches;  
desc.dim1 = num_hidden;  
uint64_t work_area_size = zdn_getsize_ztensor(&desc);
```

zdn_ztensor *hn_output

An output tensor to hold the results of the hidden states. The value of *pre_transformed_desc->layout* must be ZDNN_4DS.

The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#). The tensor must also satisfy the requirements in [“GRU input / output requirements” on page 86](#).

Output pre-transformed shapes:

- All timesteps: (*num_timesteps*, *num_dirs*, *num_batches*, *num_hidden*)
- Final timestep only: (1, *num_dirs*, *num_batches*, *num_hidden*)

For bidirectional (BIDIR) output:

- Forward and backward results are concatenated on the innermost dimension.
- Can be used directly as input for subsequent RNN layers without needing untransformation.
- Untransformation of output is supported.

Note that for BWD and the backward component of BIDIR directions, the output order matches the order of the input, not the processing order. For instance, the first input time step is the last to be processed, and its result is the first time step of the output.

Summary

Table 13. Summary of `zdnngpu` parameters

| Parameter | Pre-transformed layout | Pre-transformed shape | To create transformed descriptor |
|-----------------------|------------------------|--|--|
| input | ZDNN_3DS | $(num_timesteps, num_batches, num_features)$ | zdnngenerate_transformed_desc |
| h0 | ZDNN_3DS | $(num_dirs, num_batches, num_hidden)$ | zdnngenerate_transformed_desc |
| c0 | ZDNN_3DS | $(num_dirs, num_batches, num_hidden)$ | zdnngenerate_transformed_desc |
| weights | ZDNN_3DS | $(num_dirs, num_features, num_hidden)$ | zdnngenerate_transformed_desc_concatenated RNN_TYPE_GRU + USAGE_WEIGHTS + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| biases | ZDNN_2DS | (num_dirs, num_hidden) | zdnngenerate_transformed_desc_concatenated RNN_TYPE_GRU + USAGE_BIASES + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hidden_weights | ZDNN_3DS | $(num_dirs, num_hidden, num_hidden)$ | zdnngenerate_transformed_desc_concatenated RNN_TYPE_GRU + USAGE_HIDDEN_WEIGHTS + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hidden_biases | ZDNN_2DS | (num_dirs, num_hidden) | zdnngenerate_transformed_desc_concatenated RNN_TYPE_GRU + USAGE_HIDDEN_BIASES + one of: PREV_LAYER_NONE / PREV_LAYER_UNI / PREV_LAYER_BIDIR |
| hn_output | ZDNN_4DS | $(num_timesteps, num_dirs, num_batches, num_hidden)$ (Last timestep only when $num_timesteps = 1$) | zdnngenerate_transformed_desc |

Returns

One of the following **zdnngpu_status** indicators (as described in “zDNN statuses” on page 8):

| Status | Meaning |
|-------------------------|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_TYPE | See “General failing statuses” on page 8. |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_SHAPE | Any of the following reasons: <ul style="list-style-type: none"> The <i>hn_output</i> timesteps dimension must be 1 or the same size as the <i>input</i> timestep dimension. All tensors with a direction dimension have the same direction dimension size. The <i>input</i> timestep dimension must be greater than or equal to 1. Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_DIRECTION | The specified <i>direction</i> parameter is not a recognized lstm_gpu_direction direction. |
| ZDNN_ALLOCATION_FAILURE | A preallocated <i>work_area</i> was not specified and internal allocation for the required memory failed. |
| Hardware statuses | See “Hardware statuses” on page 9. |

Since

1.0.0

Requirements

This function requires that:

- The **zddn_is_nnpa_installed** function returns **true**.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- TensorFlow GRU (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRUCell)
- ONNX GRU (https://onnx.ai/onnx/operators/onnx__GRU.html#l-onnx-doc-gru)

Average pool 2D (zddn_avgpool2d)

Description

Given an input tensor in zDNN transformed format, padding type, kernel size, and kernel stride, produce a downsampled tensor, reducing the middle dimensions based on the mean values within the kernel window at each step, and store the results into the provided output zDNN tensor.

Format

```
zddn_status zddn_avgpool2d(const zddn_ztensor *input,
                           zddn_pool_padding padding_type,
                           uint32_t kernel_height, uint32_t kernel_width,
                           uint32_t stride_height, uint32_t stride_width,
                           zddn_ztensor *output);
```

Parameters

zddn_ztensor *input

A tensor with original values to be downsampled in the output tensor. This must be a ZDNN_NHWC tensor with a pre-transformed shape of [batch_Num, Height, Width, Channel]. See “AvgPool2D parameter restrictions” on page 91 for information about the expected shape of the input tensor. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

padding_type

The type of padding to use for the pooling operations. Valid values are SAME_PADDING or VALID_PADDING. See “AvgPool2D parameter restrictions” on page 91 for information about the expected *padding_type* value. For information about *same* and *valid* padding, see [What is the difference between 'SAME' and 'VALID' padding in tf.nn.max_pool of tensorflow?](https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow/) (<https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow/>).

kernel_height

The size of the kernel window that passes over the height dimension of the input tensor. See “AvgPool2D parameter restrictions” on page 91 for information about the expected *kernel_height* value.

kernel_width

The size of the kernel window that passes over the width dimension of the input tensor. See “AvgPool2D parameter restrictions” on page 91 for information about the expected *kernel_width* value.

stride_height

The number of positions the kernel moves over the height dimension of the input tensor at each step.

- If *stride_height* is 0, *stride_width* must also be 0.
- If strides are greater than 0, *stride_height* must be less than or equal to 30.

stride_width

The number of positions the kernel moves over the width dimension of the input tensor at each step.

- If *stride_height* is 0, *stride_width* must also be 0.
- If strides are greater than 0, *stride_width* must be less than or equal to 30.

zdnntensor *output

The output tensor that will hold the result of the pooling operation in its buffer. This must be a ZDNN_NHWC tensor with a pre-transformed shape of [batch_Num, Height, Width, Channel]. See [“AvgPool2D parameter restrictions” on page 91](#) for information about the expected shape of the output tensor. The tensor must satisfy the requirements in [“General zTensor requirements” on page 5](#).

AvgPool2D parameter restrictions

Parameter restrictions may vary based on the provided strides and padding type.

- The batch_Num and channel dimensions of the input tensor must always match the respective dimensions of the output tensor.
- If strides are 0:
 - The height dimension of the input tensor and the *kernel_height* value must match and be less than or equal to 1024.
 - The width dimension of the input tensor and the *kernel_width* value must match and be less than or equal to 1024.
 - The height and width dimensions of the output tensor must be 1.
 - The *padding_type* value must be VALID_PADDING.
- If strides are greater than 0:
 - The *kernel_width* and *kernel_height* values must be less than or equal to 64.
 - The height and weight dimensions of the input tensor must not be greater than 1024.
 - If the *padding_type* value is SAME_PADDING:
 - The height dimension of the output tensor must equal the following value:

```
ceil((float)input-height ÷ stride_height)
```

- The width dimension of the output tensor must equal the following value:

```
ceil((float)input-width ÷ stride_width)
```

- If the *padding_type* value is VALID_PADDING:
 - The height dimension of the output tensor must equal the following value:
- The width dimension of the output tensor must equal the following value:

```
ceil((float)(input-height - kernel_height + 1) ÷ stride_height)
```

```
ceil((float)(input-width - kernel_width + 1) ÷ stride_width)
```

Programming notes

If the magnitude of difference between elements of *input_tensor* is large (greater than 10), accuracy may be reduced.

Returns

One of the following **zdnntstatus** indicators:

| Status | Meaning |
|-----------------------------|--|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | <ul style="list-style-type: none"> Shape of the input or output tensor is invalid based on the given kernel and stride parameters. Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | |
| ZDNN_INVALID_FORMAT | See “General failing statuses” on page 8. |
| ZDNN_INVALID_STRIDE_PADDING | |
| ZDNN_INVALID_STRIDES | One stride was non-zero, but not the other. |
| Hardware statuses | <p>See “Hardware statuses” on page 9.</p> <p>In addition, the ZDNN_EXCEEDS_MDIS hardware status also occurs if any of the following conditions occur:</p> <ul style="list-style-type: none"> The <i>stride_height</i> value is larger than the value returned by <code>zdn_get_max_for_dim(3)</code>. The <i>stride_width</i> value is larger than the value returned by <code>zdn_get_max_for_dim(2)</code>. The <i>kernel_height</i> value is 0 or is larger than the value returned by <code>zdn_get_max_for_dim(3)</code>. The <i>kernel_width</i> value is 0 or is larger than the value returned by <code>zdn_get_max_for_dim(2)</code>. |
| ZDNN_FUNC_RC_F000 | Invalid <i>padding_type</i> value. |
| ZDNN_FUNC_RC_F001 | <i>stride_height</i> = 0 and <i>stride_width</i> = 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the <i>kernel_height</i> and <i>kernel_width</i> parameters. |
| ZDNN_FUNC_RC_F002 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the <i>kernel_height</i> and <i>kernel_width</i> parameters. |
| ZDNN_FUNC_RC_F003 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but a stride parameter is greater than allowed. See the earlier descriptions of the <i>stride_height</i> and <i>stride_width</i> parameters. |
| ZDNN_FUNC_RC_F004 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but either the height or weight dimension of the input tensor is greater than 1024. |

Since

1.0.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- TensorFlow AvgPool (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/avg-pool)
- ONNX AvgPool (https://onnx.ai/onnx/operators/onnx__AveragePool.html#l-onnx-doc-averagepool)

Max pool 2D (zdnnpool2d)

Description

Given an input tensor in zDNN transformed format, padding type, kernel size, and kernel stride, produce a downsampled tensor, reducing the middle dimensions based on the maximum values within the kernel window at each step, and store the results into the provided output zDNN tensor.

Format

```
zdnnpool_status zdnnpool2d(const zdnnpool_tensor *input,
                           zdnnpool_pooling_type padding_type,
                           uint32_t kernel_height, uint32_t kernel_width,
                           uint32_t stride_height, uint32_t stride_width,
                           zdnnpool_tensor *output);
```

Parameters

zdnnpool_tensor *input

An input tensor with original values to be downsampled in the output tensor. This must be a zDNN_NHWC tensor with a pre-transformed shape of [batch_Num, Height, Width, Channel]. See “MaxPool2D parameter restrictions” on page 93 for information about the expected shape of the input tensor. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

padding_type

The type of padding to use for the pooling operations. Valid values are SAME_PADDING or VALID_PADDING. See “MaxPool2D parameter restrictions” on page 93 for information about the expected value of *padding_type*. For information about "same" and "valid" padding, see [What is the difference between 'SAME' and 'VALID' padding in tf.nn.max_pool of tensorflow?](https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow/) (<https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow/>).

kernel_height

Size of the kernel window that passes over the height dimension of the input tensor. See “MaxPool2D parameter restrictions” on page 93 for information about the expected value of the *kernel_height*.

kernel_width

The size of the kernel window that passes over the width dimension of the input tensor. See “MaxPool2D parameter restrictions” on page 93 for information about the expected value of *kernel_width*.

stride_height

The number of positions the kernel moves over the height dimension of the input tensor at each step.

- If *stride_height* is 0, *stride_width* must also be 0.
- If strides are greater than 0, *stride_height* must be less than or equal to 30.

stride_width

Number of positions the kernel moves over the width dimension of the input tensor at each step.

- If *stride_height* is 0, *stride_width* must also be 0.
- If strides are greater than 0, *stride_width* must be less than or equal to 30.

zdnnpool_tensor *output

The output tensor that will hold the result of the pooling operation in its buffer. This must be a zDNN_NHWC tensor with a pre-transformed shape of [batch_Num, Height, Width, Channel]. See “MaxPool2D parameter restrictions” on page 93 for information about the expected shape of the output tensor. The tensor must satisfy the requirements in “General zTensor requirements” on page 5.

MaxPool2D parameter restrictions

Parameter restrictions may vary based on the provided strides and padding type.

- The `batch_Num` and `Channel` dimensions of the input tensor must always match the respective dimensions of the output tensor.
- If strides are 0:
 - The `Height` dimension of the input tensor and the `kernel_height` value must match and must be less than or equal to 1024.
 - The `Width` dimension of the input tensor and the `kernel_width` value must match and must be less than or equal to 1024.
 - The height and width dimensions of the output tensor must be 1.
 - The `padding_type` value must be `VALID_PADDING`.
- If strides are greater than 0:
 - The `kernel_width` and `kernel_height` values must be less than or equal to 64.
 - The height and weight dimensions of the input tensor must not be greater than 1024.
 - If the `padding_type` value is `SAME_PADDING`:
 - The height dimension of the output tensor must equal the following value:


```
ceil((float)input-height ÷ stride_height)
```
 - The width dimension of the output tensor must equal the following value:


```
ceil((float)input-width ÷ stride_width)
```
 - If the `padding_type` value is `VALID_PADDING`:
 - The height dimension of the output tensor must equal the following value:


```
ceil((float)(input-height - kernel_height + 1) ÷ stride_height)
```
 - The width dimension of the output tensor must equal the following value:


```
ceil((float)(input-width - kernel_width + 1) ÷ stride_width)
```

Programming notes

- If the magnitude of difference between elements of `input` is large (greater than 10), accuracy may be reduced.

Returns

One of the following **zdnn_status** indicators:

| Status | Meaning |
|-----------------------------|--|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | <ul style="list-style-type: none"> • Shape of the input or output tensor is invalid based on the given kernel and stride parameters. • Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | |
| ZDNN_INVALID_FORMAT | See “General failing statuses” on page 8. |
| ZDNN_INVALID_STRIDE_PADDING | |
| ZDNN_INVALID_STRIDES | One stride is non-zero, but not the other. |

| Status | Meaning |
|-------------------|---|
| Hardware statuses | See “Hardware statuses” on page 9. In addition, the ZDNN_EXCEEDS_MDIS hardware status also occurs if any of the following conditions occur: <ul style="list-style-type: none"> • The <i>stride_height</i> value is larger than the value returned by <code>zdn_get_max_for_dim(3)</code>. • The <i>stride_width</i> value is larger than the value returned by <code>zdn_get_max_for_dim(2)</code>. • The <i>kernel_height</i> value is 0 or is larger than the value returned by <code>zdn_get_max_for_dim(3)</code>. • The <i>kernel_width</i> value is 0 or is larger than the value returned by <code>zdn_get_max_for_dim(2)</code>. |
| ZDNN_FUNC_RC_F000 | Invalid <i>padding_type</i> value. |
| ZDNN_FUNC_RC_F001 | <i>stride_height</i> = 0 and <i>stride_width</i> = 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the <i>kernel_height</i> and <i>kernel_width</i> parameters. |
| ZDNN_FUNC_RC_F002 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the <i>kernel_height</i> and <i>kernel_width</i> parameters. |
| ZDNN_FUNC_RC_F003 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but a stride parameter is greater than allowed. See the earlier descriptions of the <i>stride_height</i> and <i>stride_width</i> parameters. |
| ZDNN_FUNC_RC_F004 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but either the height or weight dimension of the input tensor is greater than 1024. |

Since

1.0.0

Requirements

This function requires that:

- The **zdn_is_nnpa_installed** function returns `true`.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- TensorFlow MaxPool (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/max-pool)
- ONNX MaxPool (https://onnx.ai/onnx/operators/onnx__MaxPool.html#l-onnx-doc-maxpool)

Convolution 2D (zdn_conv2d)

Description

Perform 2D convolution over an input tensor in zDNN transformed format, as follows:

1. The *input* tensor is convolved with the *kernel* tensor.
2. The *bias* tensor is added to the results.
3. If the activation function, *act_func*, is not CONV2D_ACT_NONE, the activation function is applied to the results.
4. If *act_func* is set to CONV2D_ACT_RELU and *clipping_value* is not NULL or 0, clipping is performed against the intermediate result (*z*), where $z = \min(\text{intermediate_result}, \text{clipping_value})$.
5. The result is stored into the provided output zDNN tensor.

Format

```
zdn_status zdn_fusedconv2d(const zdn_tensor *input,
                           const zdn_tensor *kernel,
                           const zdn_tensor *bias,
                           zdn_pool_padding padding_type,
                           uint32_t stride_height, uint32_t stride_width,
                           zdn_conv2d_act act_func,
                           const void *clipping_value, zdn_tensor *output);
```

Parameters

zdn_tensor *input

An input tensor with original values to be downsampled in the output tensor. See “[Convolution 2D requirements](#)” on page 96 for requirements. The tensor must satisfy the requirements in “[General zTensor requirements](#)” on page 5.

zdn_tensor *kernel

The kernel tensor to convolve with the *input* tensor. This must be a ZDNN_CNNK_HWCK tensor with a pre-transformed shape of [kernel_height, kernel_width, channels_in, channels_out]. See “[Convolution 2D requirements](#)” on page 96 for additional requirements. The tensor must satisfy the requirements in “[General zTensor requirements](#)” on page 5.

zdn_tensor *bias

The bias tensor to add to the convolved results. This must be a ZDNN_1D tensor with a pre-transformed shape of [channels_out]. See “[Convolution 2D requirements](#)” on page 96 for additional requirements. The tensor must satisfy the requirements in “[General zTensor requirements](#)” on page 5.

zdn_pool_padding padding_type

The type of padding to use for the pooling operations. Valid values are SAME_PADDING or VALID_PADDING. For information about "same" and "valid" padding, see [What is the difference between 'SAME' and 'VALID' padding in tf.nn.max_pool of tensorflow?](https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow?) ([https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow](https://www.pico.net/kb/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-tensorflow?)).

uint32_t stride_height

Number of positions the kernel moves over the input's dim3 dimension at each step. See “[Convolution 2D requirements](#)” on page 96 for requirements.

uint32_t stride_width

Number of positions the kernel moves over the input's dim2 dimension at each step. See “[Convolution 2D requirements](#)” on page 96 for requirements.

zdn_conv2d_act act_func

The activation function to apply to the results. Valid values are CONV2D_ACT_NONE or CONV2D_ACT_RELU.

void *clipping_value

A pointer to an FP32 value used to clip the elements of the input tensor. This value must not be negative. If this value is set to NULL or 0, no clipping occurs. This value is ignored if *act_func* is not set to CONV2D_ACT_RELU.

zdn_tensor *output

The output tensor that will hold the result of the operation. This must be a ZDNN_NHWC tensor with a pre-transformed shape of [num_batches, height_out, width_out, channels_out]. See “[Convolution 2D requirements](#)” on page 96 for additional requirements. The tensor must satisfy the requirements in “[General zTensor requirements](#)” on page 5.

Convolution 2D requirements

Table 14 on page 97 summarizes the requirements for the *input*, *input_kernel*, *input_bias*, and *output* tensors for convolution 2D operations based on the specified strides and padding.

Table 14. Summary of requirements for convolution 2D operations

| Strides and padding | input (num_batches, height_in, width_in, channels_in) | kernel (kernel_height, kernel_width, channels_in, channels_out) | bias (channels_out) | output (num_batches, height_out, width_out, channels_out) |
|--|--|--|---------------------|--|
| Strides > 0 and ≤ 13 and SAME padding | – | Both <i>kernel_height</i> and <i>kernel_width</i> must be ≤ 64. | – | $height_out = \text{ceil}(height_in \div stride_height)$ $width_out = \text{ceil}(width_in \div stride_width)$ |
| Strides > 0 and ≤ 13 and VALID padding | <i>height_in</i> must be ≥ <i>kernel_height</i> . <i>width_in</i> must be ≥ <i>kernel_width</i> . | Both <i>kernel_height</i> and <i>kernel_width</i> must be ≤ 64. | – | $height_out = \text{ceil}((height_in - kernel_height + 1) \div stride_height)$ $width_out = \text{ceil}((width_in - kernel_width + 1) \div stride_width)$ |
| Strides = 0 and VALID padding | <i>height_in</i> must be = <i>kernel_height</i> . <i>width_in</i> must be = <i>kernel_width</i> | Both <i>kernel_height</i> and <i>kernel_width</i> must be ≤ 448. | – | Both <i>height_out</i> and <i>width_out</i> must be 1. |

Returns

One of the following **zdnnp_status** indicators:

| Status | Meaning |
|-----------------------------|--|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | <ul style="list-style-type: none"> Shape of the input or output tensor is invalid based on the given kernel and stride parameters. Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_STRIDE_PADDING | See “General failing statuses” on page 8. |
| ZDNN_INVALID_STRIDES | |
| ZDNN_INVALID_CLIPPING_VALUE | |
| Hardware statuses | See “Hardware statuses” on page 9. |
| ZDNN_FUNC_RC_F000 | Invalid <i>padding_type</i> value. |
| ZDNN_FUNC_RC_F001 | Invalid <i>act_func</i> value. |
| ZDNN_FUNC_RC_F002 | <i>stride_height</i> = 0 and <i>stride_width</i> = 0, but either the <i>kernel_height</i> or <i>kernel_width</i> value is greater than 448. |
| ZDNN_FUNC_RC_F003 | <i>stride_height</i> > 0 and <i>stride_width</i> > 0, but either the <i>kernel_height</i> or <i>kernel_width</i> value is greater than 64. |
| ZDNN_FUNC_RC_F004 | Either the <i>stride_height</i> or <i>stride_width</i> value is greater than 13. |

Since

1.0.0

Requirements

This function requires that:

- The **zdnnp_is_nnapi_installed** function returns true.
- The underlying hardware supports zDNN APIs 1.0.x or later at run time.

See “Validating the environment at run time” on page 11.

Framework examples

- [TensorFlow Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D)
- [ONNX Conv2D](https://onnx.ai/onnx/operators/onnx__Conv.html#l-onnx-doc-conv) (https://onnx.ai/onnx/operators/onnx__Conv.html#l-onnx-doc-conv)

Convenience functions

None.

Chapter 3. zDNN usage examples

The following examples illustrate how to code and use some of the zDNN APIs to develop applications.

- [“Example: Flow of an application calling the zDNN APIs” on page 99](#)
- [“Example: Calling the zdnnp_quantized_matmul_op API \(normal\)” on page 100](#)
- [“Example: Calling the zdnnp_quantized_matmul_op API \(on-the-fly\)” on page 102](#)
- [“Example: Calling the zdnnp_quantized_matmul_op API with pre_computed=true \(normal\)” on page 105](#)
- [“Example: Calling the zdnnp_quantized_matmul_op API with pre_computed=true \(on-the-fly\)” on page 107](#)
- [“Example: Calling the zdnnp_lstm API \(forward\)” on page 110](#)
- [“Example: Calling the zdnnp_lstm API \(bi-directional\)” on page 114](#)
- [“Example: Calling the zdnnp_lstm API \(multi-layer, bi-directional\)” on page 118](#)
- [“Example: Calling the zdnnp_gru API \(forward\)” on page 122](#)
- [“Example: Creating a quantized zTensor” on page 126](#)

Example: Flow of an application calling the zDNN APIs

```
#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnnp.h"

// *****
// Sample:
//
// Create 2 zTensors a and b, and add them together via zdnnp_add()
// *****
int main(int argc, char *argv[]) {
    zdnnp_tensor_desc pre_tfmd_desc, tfmd_desc;
    zdnnp_ztensor ztensor_a;
    zdnnp_ztensor ztensor_b;
    zdnnp_ztensor ztensor_out;
    zdnnp_status status;

    uint32_t dim_n = 1, dim_h = 32, dim_w = 32, dim_c = 3;
    zdnnp_data_types type = FP32;
    short element_size = 4; // size of each element in bytes
    uint64_t num_elements = dim_n * dim_h * dim_w * dim_c;

    // allocate tensor data storage
    void *data1 = malloc(num_elements * element_size);
    void *data2 = malloc(num_elements * element_size);
    void *data_out = malloc(num_elements * element_size);

    // read input_data

    // check status for zAIU availability, supported ops, etc. here
    // status = zdnnp_query();

    // set input tensor data to 0 to 127 sequentially and repeat
    for (uint64_t i = 0; i < num_elements; i++) {
        ((float *)data1)[i] = (float)(i & 0x7f);
        ((float *)data2)[i] = (float)(i & 0x7f);
    }

    zdnnp_init_pre_transformed_desc(ZDNN_NHWC, type, &pre_tfmd_desc, dim_n, dim_h,
                                   dim_w, dim_c);
    // generate transformed shape information
    status = zdnnp_generate_transformed_desc(&pre_tfmd_desc, &tfmd_desc);
    assert(status == ZDNN_OK);

    // initialize zTensors and allocate 4k-aligned storage via helper function
```

```

status =
    zdnn_init_ztensor_with_malloc(&pre_tfirmd_desc, &tfirmd_desc, &ztensor_a);
assert(status == ZDNN_OK);
status =
    zdnn_init_ztensor_with_malloc(&pre_tfirmd_desc, &tfirmd_desc, &ztensor_b);
assert(status == ZDNN_OK);
status =
    zdnn_init_ztensor_with_malloc(&pre_tfirmd_desc, &tfirmd_desc, &ztensor_out);
assert(status == ZDNN_OK);

// transform the feature tensor
status = zdnn_transform_ztensor(&ztensor_a, data1);
assert(status == ZDNN_OK);
status = zdnn_transform_ztensor(&ztensor_b, data2);
assert(status == ZDNN_OK);

// perform element-wise add between the two input tensors
status = zdnn_add(&ztensor_a, &ztensor_b, &ztensor_out);
assert(status == ZDNN_OK);

// transform resultant zTensor back to original data format
status = zdnn_transform_origtensor(&ztensor_out, data_out);
assert(status == ZDNN_OK);

for (uint64_t i = 0; i < num_elements; i++) {
    printf("out element %" PRIu64 " %f\n", i, ((float *)data_out)[i]);
}

// Free zTensors
status = zdnn_free_ztensor_buffer(&ztensor_a);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&ztensor_b);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&ztensor_out);
assert(status == ZDNN_OK);

free(data1);
free(data2);
free(data_out);
}

```

Example: Calling the `zdnn_quantized_matmul_op` API (normal)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021, 2024
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: Quantized Matmul
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
     *
     * Quantized Matmul:
     */

```

```

* INPUTS -----
* input          | ZDNN_3DS | (s, m, n)
* weights        | ZDNN_3DS | (s, n, p)
* input_biases   | ZDNN_2DS | (s, p)
*
* OUTPUTS -----
* output         | ZDNN_3DS | (s, m, p)
*****/
uint32_t s = 2;
uint32_t m = 3;
uint32_t n = 4;
uint32_t p = 5;

short int8_size = 1; // size of each int8 element in bytes
short float_size = 4; // size of each float element in bytes

/*****
* Create input zTensor
*****/
zdnntensor_desc input_pre_tfmd_desc, input_tfmd_desc;
zdnntensor input;

zdnntensor_init_pre_transformed_desc(ZDNN_3DS, FP32, &input_pre_tfmd_desc,
                                     s, m, n);

status = zdnntensor_generate_quantized_transformed_desc(
    &input_pre_tfmd_desc, QUANTIZED_INT8, &input_tfmd_desc);
assert(status == ZDNN_OK);

float input_scale = 1.f;
float input_offset = 0.f;

status = zdnntensor_init_quantized_ztensor_with_malloc(&input_pre_tfmd_desc,
    &input_tfmd_desc,
    input_scale, input_offset,
    &input);

assert(status == ZDNN_OK);

uint64_t input_data_size = s * m * n * float_size;
void *input_data = malloc(input_data_size);

status = zdnntensor_transform_quantized_ztensor(&input, false, INT8_MIN, INT8_MAX,
    input_data);
assert(status == ZDNN_OK);

/*****
* Create weights zTensor
*****/
zdnntensor_desc weights_pre_tfmd_desc, weights_tfmd_desc;
zdnntensor weights;

zdnntensor_init_pre_transformed_desc(ZDNN_3DS, INT8, &weights_pre_tfmd_desc,
                                     s, n, p);

status = zdnntensor_generate_quantized_transformed_desc(
    &weights_pre_tfmd_desc, QUANTIZED_WEIGHTS_INT8, &weights_tfmd_desc);
assert(status == ZDNN_OK);

float weights_scale = 1.f;
float weights_offset = 0.f;

status = zdnntensor_init_quantized_ztensor_with_malloc(&weights_pre_tfmd_desc,
    &weights_tfmd_desc,
    weights_scale,
    weights_offset, &weights);

assert(status == ZDNN_OK);

uint64_t weights_data_size = s * n * p * int8_size;
void *weights_data = malloc(weights_data_size);

status = zdnntensor_transform_quantized_ztensor(&weights, false, INT8_MIN, INT8_MAX,
    weights_data);
assert(status == ZDNN_OK);

/*****
* Create biases zTensor
*****/
zdnntensor_desc biases_pre_tfmd_desc, biases_tfmd_desc;
zdnntensor biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, FP32, &biases_pre_tfmd_desc,
                                     s, p);

```

```

status = zdnn_generate_quantized_transformed_desc(
    &biases_pre_tfrmd_desc, QUANTIZED_INT8, &biases_tfrmd_desc);
assert(status == ZDNN_OK);

float biases_scale = 1.f;
float biases_offset = 0.f;

status = zdnn_init_quantized_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                                &biases_tfrmd_desc,
                                                biases_scale, biases_offset,
                                                &biases);

assert(status == ZDNN_OK);

uint64_t biases_data_size = s * p * float_size;
void *biases_data = malloc(biases_data_size);

status = zdnn_transform_quantized_ztensor(&biases, false, INT8_MIN, INT8_MAX,
                                          biases_data);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/
zdnn_tensor_desc output_pre_tfrmd_desc, output_tfrmd_desc;
zdnn_ztensor output;

zdnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &output_pre_tfrmd_desc,
                              s, m, p);

status = zdnn_generate_quantized_transformed_desc(
    &output_pre_tfrmd_desc, QUANTIZED_DLFLOAT16, &output_tfrmd_desc);
assert(status == ZDNN_OK);

float output_scale = 1.f;
float output_offset = 0.f;

status = zdnn_init_quantized_ztensor_with_malloc(&output_pre_tfrmd_desc,
                                                &output_tfrmd_desc,
                                                output_scale, output_offset,
                                                &output);

assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/
status = zdnn_quantized_matmul_op(&input, &weights, &biases,
                                MATMUL_OP_ADDITION, INT8_MIN, INT8_MAX,
                                NULL, &output);

assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/
uint64_t output_data_size = s * m * p * float_size;
void *output_data = malloc(output_data_size);

status = zdnn_transform_origtensor(&output, output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&output);
assert(status == ZDNN_OK);

free(input_data);
free(weights_data);
free(biases_data);
free(output_data);
}

```

Example: Calling the `zdnn_quantized_matmul_op` API (on-the-fly)

```

// SPDX-License-Identifier: Apache-2.0
/*

```

```

* Copyright IBM Corp. 2021, 2024
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: Quantized Matmul on-the-fly
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
    *
    * Quantized Matmul on-the-fly:
    *
    * INPUTS -----
    * input          | ZDNN_3DS | (s, m, n)
    * weights        | ZDNN_3DS | (s, n, p)
    * input_biases   | ZDNN_2DS | (s, p)
    *
    * OUTPUTS -----
    * output         | ZDNN_3DS | (s, m, p)
    *****/
    uint32_t s = 2;
    uint32_t m = 3;
    uint32_t n = 4;
    uint32_t p = 5;

    short int8_size = 1; // size of each int8 element in bytes
    short float_size = 4; // size of each float element in bytes

    /*****
    * Create input zTensor
    *****/
    zdnn_tensor_desc input_pre_tfrmd_desc, input_tfrmd_desc;
    zdnn_ztensor input;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &input_pre_tfrmd_desc,
                                   s, m, n);

    status = zdnn_generate_quantized_transformed_desc(
        &input_pre_tfrmd_desc, QUANTIZED_DLFLLOAT16, &input_tfrmd_desc);
    assert(status == ZDNN_OK);

    float input_scale = 1.f;
    float input_offset = 0.f;

    status = zdnn_init_quantized_ztensor_with_malloc(&input_pre_tfrmd_desc,
                                                    &input_tfrmd_desc,
                                                    input_scale, input_offset,
                                                    &input);

    assert(status == ZDNN_OK);

    uint64_t input_data_size = s * m * n * float_size;
    void *input_data = malloc(input_data_size);

    status = zdnn_transform_ztensor(&input, input_data);
    assert(status == ZDNN_OK);

    /*****
    * Create weights zTensor
    *****/
    zdnn_tensor_desc weights_pre_tfrmd_desc, weights_tfrmd_desc;

```

```

zdnns_ztensor weights;

zdnns_init_pre_transformed_desc(ZDNN_3DS, INT8, &weights_pre_tfrmd_desc,
                                s, n, p);

status = zdnns_generate_quantized_transformed_desc(
    &weights_pre_tfrmd_desc, QUANTIZED_WEIGHTS_INT8, &weights_tfrmd_desc);
assert(status == ZDNN_OK);

float weights_scale = 1.f;
float weights_offset = 0.f;

status = zdnns_init_quantized_ztensor_with_malloc(&weights_pre_tfrmd_desc,
                                                  &weights_tfrmd_desc,
                                                  weights_scale,
                                                  weights_offset, &weights);

assert(status == ZDNN_OK);

uint64_t weights_data_size = s * n * p * int8_size;
void *weights_data = malloc(weights_data_size);

status = zdnns_transform_quantized_ztensor(&weights, false, INT8_MIN, INT8_MAX,
                                           weights_data);
assert(status == ZDNN_OK);

/*****
 * Create biases zTensor
 *****/
zdnns_tensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnns_ztensor biases;

zdnns_init_pre_transformed_desc(ZDNN_2DS, FP32, &biases_pre_tfrmd_desc,
                                s, p);

status = zdnns_generate_quantized_transformed_desc(
    &biases_pre_tfrmd_desc, QUANTIZED_INT8, &biases_tfrmd_desc);
assert(status == ZDNN_OK);

float biases_scale = 1.f;
float biases_offset = 0.f;

status = zdnns_init_quantized_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                                  &biases_tfrmd_desc,
                                                  biases_scale, biases_offset,
                                                  &biases);

assert(status == ZDNN_OK);

uint64_t biases_data_size = s * p * float_size;
void *biases_data = malloc(biases_data_size);

status = zdnns_transform_quantized_ztensor(&biases, false, INT8_MIN, INT8_MAX,
                                           biases_data);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/
zdnns_tensor_desc output_pre_tfrmd_desc, output_tfrmd_desc;
zdnns_ztensor output;

zdnns_init_pre_transformed_desc(ZDNN_3DS, FP32, &output_pre_tfrmd_desc,
                                s, m, p);

status = zdnns_generate_quantized_transformed_desc(
    &output_pre_tfrmd_desc, QUANTIZED_DFLOAT16, &output_tfrmd_desc);
assert(status == ZDNN_OK);

float output_scale = 1.f;
float output_offset = 0.f;

status = zdnns_init_quantized_ztensor_with_malloc(&output_pre_tfrmd_desc,
                                                  &output_tfrmd_desc,
                                                  output_scale, output_offset,
                                                  &output);

assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/
status = zdnns_quantized_matmul_op(&input, &weights, &biases,
                                   MATMUL_OP_ADDITION, INT8_MIN, INT8_MAX,
                                   NULL, &output);

```



```

assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/
uint64_t output_data_size = s * m * p * float_size;
void *output_data = malloc(output_data_size);

status = zdnn_transform_ortensor(&output, output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&output);
assert(status == ZDNN_OK);

free(input_data);
free(weights_data);
free(biases_data);
free(output_data);
}

```

Example: Calling the `zdnn_quantized_matmul_op` API with `pre_computed=true` (normal)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021, 2024
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: Quantized Matmul Pre-Computed
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
     *
     * Quantized Matmul Pre-Computed:
     *
     * INPUTS -----
     * input          | ZDNN_3DS | (s, m, n)
     * weights        | ZDNN_3DS | (s, n, p)
     * input_biases   | ZDNN_2DS | (s, p)
     *
     * OUTPUTS -----
     * output         | ZDNN_3DS | (s, m, p)
     *****/
    uint32_t s = 2;
    uint32_t m = 3;
    uint32_t n = 4;
    uint32_t p = 5;

```

```

short int8_size = 1; // size of each int8 element in bytes
short float_size = 4; // size of each float element in bytes

/*****
 * Create input zTensor
 *****/
znnn_tensor_desc input_pre_tfmd_desc, input_tfmd_desc;
znnn_ztensor input;

znnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &input_pre_tfmd_desc,
                               s, m, n);

status = znnn_generate_quantized_transformed_desc(
    &input_pre_tfmd_desc, QUANTIZED_INT8, &input_tfmd_desc);
assert(status == ZDNN_OK);

float input_scale = 1.f;
float input_offset = 0.f;

status = znnn_init_quantized_ztensor_with_malloc(&input_pre_tfmd_desc,
                                                  &input_tfmd_desc,
                                                  input_scale, input_offset,
                                                  &input);

assert(status == ZDNN_OK);

uint64_t input_data_size = s * m * n * float_size;
void *input_data = malloc(input_data_size);

status = znnn_transform_quantized_ztensor(&input, false, INT8_MIN, INT8_MAX,
                                           input_data);
assert(status == ZDNN_OK);

/*****
 * Create weights zTensor
 *****/
znnn_tensor_desc weights_pre_tfmd_desc, weights_tfmd_desc;
znnn_ztensor weights;

znnn_init_pre_transformed_desc(ZDNN_3DS, INT8, &weights_pre_tfmd_desc,
                               s, n, p);

status = znnn_generate_quantized_transformed_desc(
    &weights_pre_tfmd_desc, QUANTIZED_WEIGHTS_INT8, &weights_tfmd_desc);
assert(status == ZDNN_OK);

float weights_scale = 1.f;
float weights_offset = 0.f;

status = znnn_init_quantized_ztensor_with_malloc(&weights_pre_tfmd_desc,
                                                  &weights_tfmd_desc,
                                                  weights_scale,
                                                  weights_offset, &weights);

assert(status == ZDNN_OK);

uint64_t weights_data_size = s * n * p * int8_size;
void *weights_data = malloc(weights_data_size);

status = znnn_transform_quantized_ztensor(&weights, false, INT8_MIN, INT8_MAX,
                                           weights_data);
assert(status == ZDNN_OK);

/*****
 * Create pre-computed biases zTensor
 *****/
znnn_tensor_desc biases_pre_tfmd_desc, biases_tfmd_desc;
znnn_ztensor biases;

znnn_init_pre_transformed_desc(ZDNN_2DS, FP32, &biases_pre_tfmd_desc,
                               s, p);

status = znnn_generate_quantized_transformed_desc(
    &biases_pre_tfmd_desc, QUANTIZED_DLFL0AT16, &biases_tfmd_desc);
assert(status == ZDNN_OK);

float biases_scale = 1.f;
float biases_offset = 0.f;

status = znnn_init_quantized_ztensor_with_malloc(&biases_pre_tfmd_desc,
                                                  &biases_tfmd_desc,
                                                  biases_scale, biases_offset,
                                                  &biases);

assert(status == ZDNN_OK);

```

```

uint64_t biases_data_size = s * p * float_size;
void *biases_data = malloc(biases_data_size);

status = zdnn_transform_ztensor(&biases, biases_data);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/
zdnn_tensor_desc output_pre_tfirmd_desc, output_tfirmd_desc;
zdnn_ztensor output;

zdnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &output_pre_tfirmd_desc,
                               s, m, p);

status = zdnn_generate_quantized_transformed_desc(
    &output_pre_tfirmd_desc, QUANTIZED_DLFloat16, &output_tfirmd_desc);
assert(status == ZDNN_OK);

float output_scale = 1.f;
float output_offset = 0.f;

status = zdnn_init_quantized_ztensor_with_malloc(&output_pre_tfirmd_desc,
                                                  &output_tfirmd_desc,
                                                  output_scale, output_offset,
                                                  &output);

assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/
status = zdnn_quantized_matmul_op(&input, &weights, &biases,
                                  MATMUL_OP_ADDITION, INT8_MIN,
                                  INT8_MAX, false, true, NULL, &output);

assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/
uint64_t output_data_size = s * m * p * float_size;
void *output_data = malloc(output_data_size);

status = zdnn_transform_origtensor(&output, output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&output);
assert(status == ZDNN_OK);

free(input_data);
free(weights_data);
free(biases_data);
free(output_data);
}

```

Example: Calling the `zdnn_quantized_matmul_op` API with `pre_computed=true` (on-the-fly)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021, 2024
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and

```

```

* limitations under the License.
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: Quantized Matmul Pre-Computed on-the-fly
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
    *
    * Quantized Matmul Pre-Computed on-the-fly:
    *
    * INPUTS -----
    * input          | ZDNN_3DS | (s, m, n)
    * weights        | ZDNN_3DS | (s, n, p)
    * input_biases   | ZDNN_2DS | (s, p)
    *
    * OUTPUTS -----
    * output         | ZDNN_3DS | (s, m, p)
    *****/
    uint32_t s = 2;
    uint32_t m = 3;
    uint32_t n = 4;
    uint32_t p = 5;

    short int8_size = 1; // size of each int8 element in bytes
    short float_size = 4; // size of each float element in bytes

    /*****
    * Create input zTensor
    *****/
    zdnn_tensor_desc input_pre_tfrmd_desc, input_tfrmd_desc;
    zdnn_ztensor input;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &input_pre_tfrmd_desc,
                                   s, m, n);

    status = zdnn_generate_quantized_transformed_desc(
        &input_pre_tfrmd_desc, QUANTIZED_DLFLOAT16, &input_tfrmd_desc);
    assert(status == ZDNN_OK);

    float input_scale = 1.f;
    float input_offset = 0.f;

    status = zdnn_init_quantized_ztensor_with_malloc(&input_pre_tfrmd_desc,
                                                    &input_tfrmd_desc,
                                                    input_scale, input_offset,
                                                    &input);

    assert(status == ZDNN_OK);

    uint64_t input_data_size = s * m * n * float_size;
    void *input_data = malloc(input_data_size);

    status = zdnn_transform_ztensor(&input, input_data);
    assert(status == ZDNN_OK);

    /*****
    * Create weights zTensor
    *****/
    zdnn_tensor_desc weights_pre_tfrmd_desc, weights_tfrmd_desc;
    zdnn_ztensor weights;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, INT8, &weights_pre_tfrmd_desc,
                                   s, n, p);

    status = zdnn_generate_quantized_transformed_desc(
        &weights_pre_tfrmd_desc, QUANTIZED_WEIGHTS_INT8, &weights_tfrmd_desc);
    assert(status == ZDNN_OK);

    float weights_scale = 1.f;
    float weights_offset = 0.f;

```

```

status = zdnn_init_quantized_ztensor_with_malloc(&weights_pre_tfrmd_desc,
                                                &weights_tfrmd_desc,
                                                weights_scale,
                                                weights_offset, &weights);

assert(status == ZDNN_OK);

uint64_t weights_data_size = s * n * p * int8_size;
void *weights_data = malloc(weights_data_size);

status = zdnn_transform_quantized_ztensor(&weights, false, INT8_MIN, INT8_MAX,
                                         weights_data);
assert(status == ZDNN_OK);

/*****
 * Create pre-computed biases zTensor
 *****/
zdnn_tensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnn_ztensor biases;

zdnn_init_pre_transformed_desc(ZDNN_2DS, FP32, &biases_pre_tfrmd_desc,
                              s, p);

status = zdnn_generate_quantized_transformed_desc(
    &biases_pre_tfrmd_desc, QUANTIZED_DLFLOAT16, &biases_tfrmd_desc);
assert(status == ZDNN_OK);

float biases_scale = 1.f;
float biases_offset = 0.f;

status = zdnn_init_quantized_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                                &biases_tfrmd_desc,
                                                biases_scale, biases_offset,
                                                &biases);

assert(status == ZDNN_OK);

uint64_t biases_data_size = s * p * float_size;
void *biases_data = malloc(biases_data_size);

status = zdnn_transform_ztensor(&biases, biases_data);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/
zdnn_tensor_desc output_pre_tfrmd_desc, output_tfrmd_desc;
zdnn_ztensor output;

zdnn_init_pre_transformed_desc(ZDNN_3DS, FP32, &output_pre_tfrmd_desc,
                              s, m, p);

status = zdnn_generate_quantized_transformed_desc(
    &output_pre_tfrmd_desc, QUANTIZED_DLFLOAT16, &output_tfrmd_desc);
assert(status == ZDNN_OK);

float output_scale = 1.f;
float output_offset = 0.f;

status = zdnn_init_quantized_ztensor_with_malloc(&output_pre_tfrmd_desc,
                                                &output_tfrmd_desc,
                                                output_scale, output_offset,
                                                &output);

assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/
status = zdnn_quantized_matmul_op(&input, &weights, &biases,
                                MATMUL_OP_ADDITION, INT8_MIN,
                                INT8_MAX, false, true, NULL, &output);

assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/
uint64_t output_data_size = s * m * p * float_size;
void *output_data = malloc(output_data_size);

status = zdnn_transform_origtensor(&output, output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);

```

```

status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&output);
assert(status == ZDNN_OK);

free(input_data);
free(weights_data);
free(biases_data);
free(output_data);
}

```

Example: Calling the zdnn_lstm API (forward)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: LSTM
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
     *
     * LSTM (FWD/BWD):
     *
     * INPUTS -----
     * input          | ZDNN_3DS | (num_timesteps, num_batches, num_features)
     * h0              | ZDNN_3DS | (1, num_batches, num_hidden)
     * c0              | ZDNN_3DS | (1, num_batches, num_hidden)
     * weights         | ZDNN_3DS | (1, num_features, num_hidden)
     * biases          | ZDNN_2DS | (1, num_hidden)
     * hidden_weights  | ZDNN_3DS | (1, num_hidden, num_hidden)
     * hidden_biases   | ZDNN_2DS | (1, num_hidden)
     *
     * OUTPUTS -----
     * hn_output       | ZDNN_4DS | (num_timesteps, 1, num_batches, num_hidden)
     *                 |         | or (1, 1, num_batches, num_hidden)
     * cf_output       | ZDNN_4DS | (1, 1, num_batches, num_hidden)
     *****/

    /*****
     * Create input zTensor
     *****/

    zdnn_tensor_desc input_pre_tfrmd_desc, input_tfrmd_desc;
    zdnn_ztensor input;

    uint32_t num_timesteps = 5;
    uint32_t num_batches = 3;
    uint32_t num_features = 32;
    uint32_t num_hidden = 5;

    zdnn_data_types type = FP32;

```

```

short element_size = 4; // size of each element in bytes

lstm_gru_direction dir = FWD;
uint8_t num_dirs = 1;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &input_pre_tfird_desc,
                             num_timesteps, num_batches, num_features);
status =
    zdn_generate_transformed_desc(&input_pre_tfird_desc, &input_tfird_desc);
assert(status == ZDNN_OK);
status = zdn_init_ztensor_with_malloc(&input_pre_tfird_desc,
                                     &input_tfird_desc, &input);
assert(status == ZDNN_OK);

uint64_t input_data_size =
    num_timesteps * num_batches * num_features * element_size;
void *input_data = malloc(input_data_size);

status = zdn_transform_ztensor(&input, input_data);
assert(status == ZDNN_OK);

/*****
 * Create initial hidden and cell state zTensors
 *****/

zdn_tensor_desc h0c0_pre_tfird_desc, h0c0_tfird_desc;
zdn_ztensor h0, c0;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &h0c0_pre_tfird_desc, num_dirs,
                             num_batches, num_hidden);
status =
    zdn_generate_transformed_desc(&h0c0_pre_tfird_desc, &h0c0_tfird_desc);
assert(status == ZDNN_OK);

status = zdn_init_ztensor_with_malloc(&h0c0_pre_tfird_desc, &h0c0_tfird_desc,
                                     &h0);
assert(status == ZDNN_OK);
status = zdn_init_ztensor_with_malloc(&h0c0_pre_tfird_desc, &h0c0_tfird_desc,
                                     &c0);
assert(status == ZDNN_OK);

uint64_t h0c0_data_size = num_batches * num_hidden * element_size;
void *hidden_state_data = malloc(h0c0_data_size);
void *cell_state_data = malloc(h0c0_data_size);

status = zdn_transform_ztensor(&h0, hidden_state_data);
assert(status == ZDNN_OK);
status = zdn_transform_ztensor(&c0, cell_state_data);
assert(status == ZDNN_OK);

/*****
 * Create input weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdn_tensor_desc weights_pre_tfird_desc, weights_tfird_desc;
zdn_ztensor weights;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &weights_pre_tfird_desc,
                             num_dirs, num_features, num_hidden);
status = zdn_generate_transformed_desc_concatenated(
    &weights_pre_tfird_desc, RNN_TYPE_LSTM | USAGE_WEIGHTS | PREV_LAYER_NONE,
    &weights_tfird_desc);
assert(status == ZDNN_OK);

status = zdn_init_ztensor_with_malloc(&weights_pre_tfird_desc,
                                     &weights_tfird_desc, &weights);
assert(status == ZDNN_OK);

uint64_t weights_data_size = num_features * num_hidden * element_size;
void *weights_data_f = malloc(weights_data_size);
void *weights_data_i = malloc(weights_data_size);
void *weights_data_c = malloc(weights_data_size);
void *weights_data_o = malloc(weights_data_size);

status = zdn_transform_ztensor(&weights, weights_data_f, weights_data_i,
                             weights_data_c, weights_data_o);
assert(status == ZDNN_OK);

/*****
 * Create biases zTensors
 * Resultant zTensors are concatenated
 *****/

```

```

*****/
zdnntensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnntensor biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, type, &biases_pre_tfrmd_desc,
                                     num_dirs, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &biases_pre_tfrmd_desc, RNN_TYPE_LSTM | USAGE_BIASES | PREV_LAYER_NONE,
    &biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                             &biases_tfrmd_desc, &biases);
assert(status == ZDNN_OK);

uint64_t biases_data_size = num_hidden * element_size;
void *biases_data_f = malloc(biases_data_size);
void *biases_data_i = malloc(biases_data_size);
void *biases_data_c = malloc(biases_data_size);
void *biases_data_o = malloc(biases_data_size);

status = zdnntensor_transform_ztensor(&biases, biases_data_f, biases_data_i,
                                       biases_data_c, biases_data_o);
assert(status == ZDNN_OK);

/*****
 * Create hidden weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdnntensor_desc hidden_weights_pre_tfrmd_desc, hidden_weights_tfrmd_desc;
zdnntensor hidden_weights;

zdnntensor_init_pre_transformed_desc(ZDNN_3DS, type, &hidden_weights_pre_tfrmd_desc,
                                     num_dirs, num_hidden, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_weights_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_WEIGHTS | PREV_LAYER_NONE,
    &hidden_weights_tfrmd_desc);
assert(status == ZDNN_OK);
status = zdnntensor_init_ztensor_with_malloc(&hidden_weights_pre_tfrmd_desc,
                                             &hidden_weights_tfrmd_desc,
                                             &hidden_weights);

assert(status == ZDNN_OK);

uint64_t hidden_weights_data_size = num_hidden * num_hidden * element_size;
void *hidden_weights_data_f = malloc(hidden_weights_data_size);
void *hidden_weights_data_i = malloc(hidden_weights_data_size);
void *hidden_weights_data_c = malloc(hidden_weights_data_size);
void *hidden_weights_data_o = malloc(hidden_weights_data_size);

status = zdnntensor_transform_ztensor(&hidden_weights, hidden_weights_data_f,
                                       hidden_weights_data_i, hidden_weights_data_c,
                                       hidden_weights_data_o);
assert(status == ZDNN_OK);

/*****
 * Create hidden biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnntensor_desc hidden_biases_pre_tfrmd_desc, hidden_biases_tfrmd_desc;
zdnntensor hidden_biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, type, &hidden_biases_pre_tfrmd_desc,
                                     num_dirs, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_biases_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_BIASES | PREV_LAYER_NONE,
    &hidden_biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(
    &hidden_biases_pre_tfrmd_desc, &hidden_biases_tfrmd_desc, &hidden_biases);
assert(status == ZDNN_OK);

uint64_t hidden_biases_data_size = num_hidden * element_size;

void *hidden_biases_data_f = malloc(hidden_biases_data_size);
void *hidden_biases_data_i = malloc(hidden_biases_data_size);
void *hidden_biases_data_c = malloc(hidden_biases_data_size);

```



```

void *hidden_biases_data_o = malloc(hidden_biases_data_size);

status = zdnn_transform_ztensor(&hidden_biases, hidden_biases_data_f,
                                hidden_biases_data_i, hidden_biases_data_c,
                                hidden_biases_data_o);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/

// get only the last timestep, thus hn and cf can share descriptor
zdnn_tensor_desc hncf_pre_tfrmd_desc, hncf_tfrmd_desc;

zdnn_ztensor hn_output_ztensor, cf_output_ztensor;

zdnn_init_pre_transformed_desc(ZDNN_4DS, type, &hncf_pre_tfrmd_desc, 1, 1,
                                num_batches, num_hidden);
status =
    zdnn_generate_transformed_desc(&hncf_pre_tfrmd_desc, &hncf_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&hncf_pre_tfrmd_desc, &hncf_tfrmd_desc,
                                        &hn_output_ztensor);
assert(status == ZDNN_OK);
status = zdnn_init_ztensor_with_malloc(&hncf_pre_tfrmd_desc, &hncf_tfrmd_desc,
                                        &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/

void *work_area = NULL;

status = zdnn_lstm(&input, &h0, &c0, &weights, &biases, &hidden_weights,
                  &hidden_biases, dir, work_area, &hn_output_ztensor,
                  &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/

uint64_t hncf_data_size = num_batches * num_hidden * element_size;
void *hn_output_data = malloc(hncf_data_size);
void *cf_output_data = malloc(hncf_data_size);

status = zdnn_transform_origtensor(&hn_output_ztensor, hn_output_data);
assert(status == ZDNN_OK);
status = zdnn_transform_origtensor(&cf_output_ztensor, cf_output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&h0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&c0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hn_output_ztensor);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&cf_output_ztensor);
assert(status == ZDNN_OK);

free(input_data);
free(hidden_state_data);
free(cell_state_data);
free(weights_data_f);
free(weights_data_i);
free(weights_data_c);
free(weights_data_o);
free(hidden_weights_data_f);
free(hidden_weights_data_i);
free(hidden_weights_data_c);

```

```

free(hidden_weights_data_o);
free(biases_data_f);
free(biases_data_i);
free(biases_data_c);
free(biases_data_o);
free(hidden_biases_data_f);
free(hidden_biases_data_i);
free(hidden_biases_data_c);
free(hidden_biases_data_o);
free(hn_output_data);
free(cf_output_data);
}

```

Example: Calling the zdnn_lstm API (bi-directional)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: LSTM BI-DIR
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
     *
     * LSTM (BI-DIR):
     *
     * INPUTS -----
     * input          | ZDNN_3DS | (num_timesteps, num_batches, num_features)
     * h0              | ZDNN_3DS | (2, num_batches, num_hidden)
     * c0              | ZDNN_3DS | (2, num_batches, num_hidden)
     * weights         | ZDNN_3DS | (2, num_features, num_hidden)
     * biases          | ZDNN_2DS | (2, num_hidden)
     * hidden_weights  | ZDNN_3DS | (2, num_hidden, num_hidden)
     * hidden_biases   | ZDNN_2DS | (2, num_hidden)
     *
     * OUTPUTS -----
     * hn_output       | ZDNN_4DS | (num_timesteps, 2, num_batches, num_hidden)
     *                 |         | or (1, 2, num_batches, num_hidden)
     * cf_output       | ZDNN_4DS | (1, 2, num_batches, num_hidden)
     *****/

    /*****
     * Create input zTensor
     *****/

    zdnn_tensor_desc input_pre_tfrmd_desc, input_tfrmd_desc;
    zdnn_ztensor input;

    uint32_t num_timesteps = 5;
    uint32_t num_batches = 3;
    uint32_t num_features = 32;
    uint32_t num_hidden = 5;

    zdnn_data_types type = FP32;

```

```

short element_size = 4; // size of each element in bytes

lstm_gru_direction dir = BIDIR;
uint8_t num_dirs = 2;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &input_pre_tfrmd_desc,
                              num_timesteps, num_batches, num_features);
status =
    zdn_generate_transformed_desc(&input_pre_tfrmd_desc, &input_tfrmd_desc);
assert(status == ZDNN_OK);
status = zdn_init_ztensor_with_malloc(&input_pre_tfrmd_desc,
                                      &input_tfrmd_desc, &input);
assert(status == ZDNN_OK);

uint64_t input_data_size =
    num_timesteps * num_batches * num_features * element_size;
void *input_data = malloc(input_data_size);

status = zdn_transform_ztensor(&input, input_data);
assert(status == ZDNN_OK);

/*****
 * Create initial hidden and cell state zTensors
 *****/

zdn_tensor_desc h0c0_pre_tfrmd_desc, h0c0_tfrmd_desc;
zdn_ztensor h0, c0;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &h0c0_pre_tfrmd_desc, num_dirs,
                              num_batches, num_hidden);
status =
    zdn_generate_transformed_desc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdn_init_ztensor_with_malloc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc,
                                      &h0);
assert(status == ZDNN_OK);
status = zdn_init_ztensor_with_malloc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc,
                                      &c0);
assert(status == ZDNN_OK);

uint64_t h0c0_data_size = num_batches * num_hidden * element_size;
void *hidden_state_data = malloc(h0c0_data_size);
void *cell_state_data = malloc(h0c0_data_size);

status = zdn_transform_ztensor(&h0, hidden_state_data);
assert(status == ZDNN_OK);
status = zdn_transform_ztensor(&c0, cell_state_data);
assert(status == ZDNN_OK);

/*****
 * Create input weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdn_tensor_desc weights_pre_tfrmd_desc, weights_tfrmd_desc;
zdn_ztensor weights;

zdn_init_pre_transformed_desc(ZDNN_3DS, type, &weights_pre_tfrmd_desc,
                              num_dirs, num_features, num_hidden);
status = zdn_generate_transformed_desc_concatenated(
    &weights_pre_tfrmd_desc, RNN_TYPE_LSTM | USAGE_WEIGHTS | PREV_LAYER_NONE,
    &weights_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdn_init_ztensor_with_malloc(&weights_pre_tfrmd_desc,
                                      &weights_tfrmd_desc, &weights);
assert(status == ZDNN_OK);

uint64_t weights_data_size = num_features * num_hidden * element_size;
void *weights_data_f = malloc(weights_data_size);
void *weights_data_i = malloc(weights_data_size);
void *weights_data_c = malloc(weights_data_size);
void *weights_data_o = malloc(weights_data_size);

status = zdn_transform_ztensor(&weights, weights_data_f, weights_data_i,
                              weights_data_c, weights_data_o);
assert(status == ZDNN_OK);

/*****
 * Create biases zTensors
 * Resultant zTensors are concatenated
 *****/

```

```

*****/
zdnntensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnntensor biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, type, &biases_pre_tfrmd_desc,
                                     num_dirs, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &biases_pre_tfrmd_desc, RNN_TYPE_LSTM | USAGE_BIASES | PREV_LAYER_NONE,
    &biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                             &biases_tfrmd_desc, &biases);
assert(status == ZDNN_OK);

uint64_t biases_data_size = num_hidden * element_size;
void *biases_data_f = malloc(biases_data_size);
void *biases_data_i = malloc(biases_data_size);
void *biases_data_c = malloc(biases_data_size);
void *biases_data_o = malloc(biases_data_size);

status = zdnntensor_transform_ztensor(&biases, biases_data_f, biases_data_i,
                                       biases_data_c, biases_data_o);
assert(status == ZDNN_OK);

/*****
 * Create hidden weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdnntensor_desc hidden_weights_pre_tfrmd_desc, hidden_weights_tfrmd_desc;
zdnntensor hidden_weights;

zdnntensor_init_pre_transformed_desc(ZDNN_3DS, type, &hidden_weights_pre_tfrmd_desc,
                                     num_dirs, num_hidden, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_weights_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_WEIGHTS | PREV_LAYER_NONE,
    &hidden_weights_tfrmd_desc);
assert(status == ZDNN_OK);
status = zdnntensor_init_ztensor_with_malloc(&hidden_weights_pre_tfrmd_desc,
                                             &hidden_weights_tfrmd_desc,
                                             &hidden_weights);

assert(status == ZDNN_OK);

uint64_t hidden_weights_data_size = num_hidden * num_hidden * element_size;
void *hidden_weights_data_f = malloc(hidden_weights_data_size);
void *hidden_weights_data_i = malloc(hidden_weights_data_size);
void *hidden_weights_data_c = malloc(hidden_weights_data_size);
void *hidden_weights_data_o = malloc(hidden_weights_data_size);

status = zdnntensor_transform_ztensor(&hidden_weights, hidden_weights_data_f,
                                       hidden_weights_data_i, hidden_weights_data_c,
                                       hidden_weights_data_o);
assert(status == ZDNN_OK);

/*****
 * Create hidden biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnntensor_desc hidden_biases_pre_tfrmd_desc, hidden_biases_tfrmd_desc;
zdnntensor hidden_biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, type, &hidden_biases_pre_tfrmd_desc,
                                     num_dirs, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_biases_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_BIASES | PREV_LAYER_NONE,
    &hidden_biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(
    &hidden_biases_pre_tfrmd_desc, &hidden_biases_tfrmd_desc, &hidden_biases);
assert(status == ZDNN_OK);

uint64_t hidden_biases_data_size = num_hidden * element_size;

void *hidden_biases_data_f = malloc(hidden_biases_data_size);
void *hidden_biases_data_i = malloc(hidden_biases_data_size);
void *hidden_biases_data_c = malloc(hidden_biases_data_size);

```

```

void *hidden_biases_data_o = malloc(hidden_biases_data_size);

status = zdnn_transform_ztensor(&hidden_biases, hidden_biases_data_f,
                                hidden_biases_data_i, hidden_biases_data_c,
                                hidden_biases_data_o);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/

zdnn_tensor_desc hn_pre_tfrmd_desc, hn_tfrmd_desc, cf_pre_tfrmd_desc,
                  cf_tfrmd_desc;

zdnn_ztensor hn_output_ztensor, cf_output_ztensor;

zdnn_init_pre_transformed_desc(ZDNN_4DS, type, &hn_pre_tfrmd_desc,
                                num_timesteps, 2, num_batches, num_hidden);
status = zdnn_generate_transformed_desc(&hn_pre_tfrmd_desc, &hn_tfrmd_desc);
assert(status == ZDNN_OK);

zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &cf_pre_tfrmd_desc, 1, 2,
                                num_batches, num_hidden);
status = zdnn_generate_transformed_desc(&cf_pre_tfrmd_desc, &cf_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&hn_pre_tfrmd_desc, &hn_tfrmd_desc,
                                        &hn_output_ztensor);
assert(status == ZDNN_OK);
status = zdnn_init_ztensor_with_malloc(&cf_pre_tfrmd_desc, &cf_tfrmd_desc,
                                        &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/

void *work_area = NULL;

status = zdnn_lstm(&input, &h0, &c0, &weights, &biases, &hidden_weights,
                  &hidden_biases, dir, work_area, &hn_output_ztensor,
                  &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/

uint64_t hn_data_size =
    num_timesteps * 2 * num_batches * num_hidden * element_size;
uint64_t cf_data_size = 2 * num_batches * num_hidden * element_size;
void *hn_output_data = malloc(hn_data_size);
void *cf_output_data = malloc(cf_data_size);

status = zdnn_transform_origtensor(&hn_output_ztensor, hn_output_data);
assert(status == ZDNN_OK);
status = zdnn_transform_origtensor(&cf_output_ztensor, cf_output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&h0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&c0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hn_output_ztensor);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&cf_output_ztensor);
assert(status == ZDNN_OK);

free(input_data);
free(hidden_state_data);
free(cell_state_data);
free(weights_data_f);

```

```

free(weights_data_i);
free(weights_data_c);
free(weights_data_o);
free(hidden_weights_data_f);
free(hidden_weights_data_i);
free(hidden_weights_data_c);
free(hidden_weights_data_o);
free(biases_data_f);
free(biases_data_i);
free(biases_data_c);
free(biases_data_o);
free(hidden_biases_data_f);
free(hidden_biases_data_i);
free(hidden_biases_data_c);
free(hidden_biases_data_o);
free(hn_output_data);
free(cf_output_data);
}

```

Example: Calling the zdnn_lstm API (multi-layer, bi-directional)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

void do_bidir_layer(zdnn_ztensor *input, uint32_t num_hidden,
                  zdnn_ztensor *hn_output, bool is_prev_layer_bidir) {

    zdnn_status status;

    uint32_t num_batches = input->pre_transformed_desc->dim2;

    // if input is bidir output from previous layer then number of features for
    // this layer is 2x of hidden-state size (dim1) of the previous layer
    uint32_t num_features =
        input->pre_transformed_desc->dim1 * (is_prev_layer_bidir ? 2 : 1);

    zdnn_data_types type = FP32;
    short element_size = 4; // size of each element in bytes

    lstm_gru_direction dir = BIDIR;
    uint8_t num_dirs = 2;

    /*****
     * Create initial hidden and cell state zTensors
     *****/

    zdnn_tensor_desc h0c0_pre_tfrmd_desc, h0c0_tfrmd_desc;
    zdnn_ztensor h0, c0;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &h0c0_pre_tfrmd_desc, num_dirs,
                                   num_batches, num_hidden);
    status =
        zdnn_generate_transformed_desc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc);
    assert(status == ZDNN_OK);

    status = zdnn_init_ztensor_with_malloc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc,
                                           &h0);
    assert(status == ZDNN_OK);

```

```

status = zdnn_init_ztensor_with_malloc(&h0c0_pre_tfrmd_desc, &h0c0_tfrmd_desc,
                                      &c0);
assert(status == ZDNN_OK);

uint64_t h0c0_data_size = num_batches * num_hidden * element_size;
void *hidden_state_data = malloc(h0c0_data_size);
void *cell_state_data = malloc(h0c0_data_size);

status = zdnn_transform_ztensor(&h0, hidden_state_data);
assert(status == ZDNN_OK);
status = zdnn_transform_ztensor(&c0, cell_state_data);
assert(status == ZDNN_OK);

/*****
 * Create input weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdnn_tensor_desc weights_pre_tfrmd_desc, weights_tfrmd_desc;
zdnn_ztensor weights;

// if using previous layer bidir output as input then number of features of
// this layer is
zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &weights_pre_tfrmd_desc,
                              num_dirs, num_features, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &weights_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_WEIGHTS |
    (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
    &weights_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&weights_pre_tfrmd_desc,
                                      &weights_tfrmd_desc, &weights);
assert(status == ZDNN_OK);

uint64_t weights_data_size = num_features * num_hidden * element_size;
void *weights_data_f = malloc(weights_data_size);
void *weights_data_i = malloc(weights_data_size);
void *weights_data_c = malloc(weights_data_size);
void *weights_data_o = malloc(weights_data_size);

status = zdnn_transform_ztensor(&weights, weights_data_f, weights_data_i,
                              weights_data_c, weights_data_o);
assert(status == ZDNN_OK);

/*****
 * Create biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnn_tensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnn_ztensor biases;

zdnn_init_pre_transformed_desc(ZDNN_2DS, type, &biases_pre_tfrmd_desc,
                              num_dirs, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &biases_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_BIASES |
    (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
    &biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                      &biases_tfrmd_desc, &biases);
assert(status == ZDNN_OK);

uint64_t biases_data_size = num_hidden * element_size;
void *biases_data_f = malloc(biases_data_size);
void *biases_data_i = malloc(biases_data_size);
void *biases_data_c = malloc(biases_data_size);
void *biases_data_o = malloc(biases_data_size);

status = zdnn_transform_ztensor(&biases, biases_data_f, biases_data_i,
                              biases_data_c, biases_data_o);
assert(status == ZDNN_OK);

/*****
 * Create hidden weights zTensor
 * Resultant zTensor is concatenated
 *****/

```

```

zdnntensor_desc hidden_weights_pre_tfrmd_desc, hidden_weights_tfrmd_desc;
zdnntensor hidden_weights;

zdnntensor_init_pre_transformed_desc(ZDNN_3DS, type, &hidden_weights_pre_tfrmd_desc,
                                     num_dirs, num_hidden, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_weights_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_WEIGHTS |
    (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
    &hidden_weights_tfrmd_desc);
assert(status == ZDNN_OK);
status = zdnntensor_init_ztensor_with_malloc(&hidden_weights_pre_tfrmd_desc,
                                             &hidden_weights_tfrmd_desc,
                                             &hidden_weights);

assert(status == ZDNN_OK);

uint64_t hidden_weights_data_size = num_hidden * num_hidden * element_size;
void *hidden_weights_data_f = malloc(hidden_weights_data_size);
void *hidden_weights_data_i = malloc(hidden_weights_data_size);
void *hidden_weights_data_c = malloc(hidden_weights_data_size);
void *hidden_weights_data_o = malloc(hidden_weights_data_size);

status = zdnntensor_transform_ztensor(&hidden_weights, hidden_weights_data_f,
                                       hidden_weights_data_i, hidden_weights_data_c,
                                       hidden_weights_data_o);

assert(status == ZDNN_OK);

/*****
 * Create hidden biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnntensor_desc hidden_biases_pre_tfrmd_desc, hidden_biases_tfrmd_desc;
zdnntensor hidden_biases;

zdnntensor_init_pre_transformed_desc(ZDNN_2DS, type, &hidden_biases_pre_tfrmd_desc,
                                     num_dirs, num_hidden);
status = zdnntensor_generate_transformed_desc_concatenated(
    &hidden_biases_pre_tfrmd_desc,
    RNN_TYPE_LSTM | USAGE_HIDDEN_BIASES |
    (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
    &hidden_biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(
    &hidden_biases_pre_tfrmd_desc, &hidden_biases_tfrmd_desc, &hidden_biases);
assert(status == ZDNN_OK);

uint64_t hidden_biases_data_size = num_hidden * element_size;

void *hidden_biases_data_f = malloc(hidden_biases_data_size);
void *hidden_biases_data_i = malloc(hidden_biases_data_size);
void *hidden_biases_data_c = malloc(hidden_biases_data_size);
void *hidden_biases_data_o = malloc(hidden_biases_data_size);

status = zdnntensor_transform_ztensor(&hidden_biases, hidden_biases_data_f,
                                       hidden_biases_data_i, hidden_biases_data_c,
                                       hidden_biases_data_o);

assert(status == ZDNN_OK);

/*****
 * Create cf output zTensor
 *****/

zdnntensor_desc cf_pre_tfrmd_desc, cf_tfrmd_desc;

zdnntensor cf_output_ztensor;

zdnntensor_init_pre_transformed_desc(ZDNN_4DS, type, &cf_pre_tfrmd_desc, 1, 2,
                                     num_batches, num_hidden);
status = zdnntensor_generate_transformed_desc(&cf_pre_tfrmd_desc, &cf_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnntensor_init_ztensor_with_malloc(&cf_pre_tfrmd_desc, &cf_tfrmd_desc,
                                             &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/

void *work_area = NULL;

```



```

status =
    zdnn_lstm(input, &h0, &c0, &weights, &biases, &hidden_weights,
               &hidden_biases, dir, work_area, hn_output, &cf_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Cleanup and Return
 *****/

status = zdnn_free_ztensor_buffer(&h0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&c0);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&cf_output_ztensor);
assert(status == ZDNN_OK);

free(hidden_state_data);
free(cell_state_data);
free(weights_data_f);
free(weights_data_i);
free(weights_data_c);
free(weights_data_o);
free(hidden_weights_data_f);
free(hidden_weights_data_i);
free(hidden_weights_data_c);
free(hidden_weights_data_o);
free(biases_data_f);
free(biases_data_i);
free(biases_data_c);
free(biases_data_o);
free(hidden_biases_data_f);
free(hidden_biases_data_i);
free(hidden_biases_data_c);
free(hidden_biases_data_o);
}

// Sample: LSTM multi-layer BIDIR
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    uint32_t num_hidden[2] = {5, 4};

    /*****
     * Create input zTensor
     *****/

    zdnn_tensor_desc input_pre_tfrmd_desc, input_tfrmd_desc;
    zdnn_ztensor input;

    uint32_t num_timesteps = 5;
    uint32_t num_batches = 3;
    uint32_t num_features = 32;

    zdnn_data_types type = FP32;
    short element_size = 4; // size of each element in bytes

    zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &input_pre_tfrmd_desc,
                                   num_timesteps, num_batches, num_features);
    status =
        zdnn_generate_transformed_desc(&input_pre_tfrmd_desc, &input_tfrmd_desc);
    assert(status == ZDNN_OK);
    status = zdnn_init_ztensor_with_malloc(&input_pre_tfrmd_desc,
                                           &input_tfrmd_desc, &input);
    assert(status == ZDNN_OK);

    uint64_t input_data_size =
        num_timesteps * num_batches * num_features * element_size;
    void *input_data = malloc(input_data_size);

```

```

status = zdnn_transform_ztensor(&input, input_data);
assert(status == ZDNN_OK);

/*****
 * Create 2 hn output zTensors
 *****/

zdnn_tensor_desc hn_pre_tfrmd_desc[2], hn_tfrmd_desc[2];
zdnn_ztensor hn_output[2];

for (int i = 0; i < 2; i++) {
    zdnn_init_pre_transformed_desc(ZDNN_4DS, type, &hn_pre_tfrmd_desc[i],
                                   num_timesteps, 2, num_batches,
                                   num_hidden[i]);
    status = zdnn_generate_transformed_desc(&hn_pre_tfrmd_desc[i],
                                           &hn_tfrmd_desc[i]);
    assert(status == ZDNN_OK);

    status = zdnn_init_ztensor_with_malloc(&hn_pre_tfrmd_desc[i],
                                           &hn_tfrmd_desc[i], &hn_output[i]);
    assert(status == ZDNN_OK);
}

/*****
 * Do the layers
 *****/

// call the first layer with input, previous layer bidir = false, output goes
// to hn_output[0]
do_bidir_layer(&input, num_hidden[0], &hn_output[0], false);

// call the second layer with hn_output[0] from layer 1, previous layer bidir
// = true, output goes to hn_output[1]
do_bidir_layer(&hn_output[0], num_hidden[1], &hn_output[1], true);

/*****
 * Output and Cleanup
 *****/

void *hn_output_data[2];

for (int i = 0; i < 2; i++) {
    uint64_t hn_output_data_size = (uint64_t)num_timesteps * num_batches *
                                   num_hidden[i] * 2 * element_size;
    hn_output_data[i] = malloc(hn_output_data_size);

    status = zdnn_transform_origtensor(&hn_output[i], hn_output_data[i]);
    assert(status == ZDNN_OK);
}

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hn_output[0]);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hn_output[1]);
assert(status == ZDNN_OK);

free(input_data);
free(hn_output_data[0]);
free(hn_output_data[1]);
}

```

Example: Calling the zdnn_gru API (forward)

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2021
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.

```

```

*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// Sample: GRU
int main(int argc, char *argv[]) {
    zdnn_status status;

#ifdef STATIC_LIB
    zdnn_init();
#endif

    /*****
    *
    * GRU (FWD/BWD):
    *
    * INPUTS -----
    * input          | ZDNN_3DS | (num_timesteps, num_batches, num_features)
    * h0             | ZDNN_3DS | (1, num_batches, num_hidden)
    * weights        | ZDNN_3DS | (1, num_features, num_hidden)
    * input_biases   | ZDNN_2DS | (1, num_hidden)
    * hidden_weights | ZDNN_3DS | (1, num_hidden, num_hidden)
    * hidden_biases  | ZDNN_2DS | (1, num_hidden)
    *
    * OUTPUTS -----
    * hn_output      | ZDNN_4DS | (num_timesteps, 1, num_batches, num_hidden)
    *               |         | or (1, 1, num_batches, num_hidden)
    *****/

    /*****
    * Create input zTensor
    *****/

    zdnn_tensor_desc input_pre_tfird_desc, input_tfird_desc;
    zdnn_ztensor input;

    uint32_t num_timesteps = 5;
    uint32_t num_batches = 3;
    uint32_t num_features = 32;
    uint32_t num_hidden = 5;

    zdnn_data_types type = FP32;
    short element_size = 4; // size of each element in bytes

    lstm_gru_direction dir = FWD;
    uint8_t num_dirs = 1;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &input_pre_tfird_desc,
                                   num_timesteps, num_batches, num_features);
    status =
        zdnn_generate_transformed_desc(&input_pre_tfird_desc, &input_tfird_desc);
    assert(status == ZDNN_OK);
    status = zdnn_init_ztensor_with_malloc(&input_pre_tfird_desc,
                                           &input_tfird_desc, &input);
    assert(status == ZDNN_OK);

    uint64_t input_data_size =
        num_timesteps * num_batches * num_features * element_size;
    void *input_data = malloc(input_data_size);

    status = zdnn_transform_ztensor(&input, input_data);
    assert(status == ZDNN_OK);

    /*****
    * Create initial hidden zTensor
    *****/

    zdnn_tensor_desc h0_pre_tfird_desc, h0_tfird_desc;
    zdnn_ztensor h0;

    zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &h0_pre_tfird_desc, num_dirs,
                                   num_batches, num_hidden);
    status = zdnn_generate_transformed_desc(&h0_pre_tfird_desc, &h0_tfird_desc);
    assert(status == ZDNN_OK);

    status =
        zdnn_init_ztensor_with_malloc(&h0_pre_tfird_desc, &h0_tfird_desc, &h0);

```

```

assert(status == ZDNN_OK);

uint64_t h0_data_size = num_batches * num_hidden * element_size;
void *hidden_state_data = malloc(h0_data_size);

status = zdnn_transform_ztensor(&h0, hidden_state_data);
assert(status == ZDNN_OK);

/*****
 * Create input weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdnn_tensor_desc weights_pre_tfrmd_desc, weights_tfrmd_desc;
zdnn_ztensor weights;

zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &weights_pre_tfrmd_desc,
                               num_dirs, num_features, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &weights_pre_tfrmd_desc, RNN_TYPE_GRU | USAGE_WEIGHTS | PREV_LAYER_NONE,
    &weights_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&weights_pre_tfrmd_desc,
                                       &weights_tfrmd_desc, &weights);
assert(status == ZDNN_OK);

uint64_t weights_data_size = num_features * num_hidden * element_size;
void *weights_data_z = malloc(weights_data_size);
void *weights_data_r = malloc(weights_data_size);
void *weights_data_h = malloc(weights_data_size);

status = zdnn_transform_ztensor(&weights, weights_data_z, weights_data_r,
                                weights_data_h);
assert(status == ZDNN_OK);

/*****
 * Create biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnn_tensor_desc biases_pre_tfrmd_desc, biases_tfrmd_desc;
zdnn_ztensor biases;

zdnn_init_pre_transformed_desc(ZDNN_2DS, type, &biases_pre_tfrmd_desc,
                               num_dirs, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &biases_pre_tfrmd_desc, RNN_TYPE_GRU | USAGE_BIASES | PREV_LAYER_NONE,
    &biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&biases_pre_tfrmd_desc,
                                       &biases_tfrmd_desc, &biases);
assert(status == ZDNN_OK);

uint64_t biases_data_size = num_hidden * element_size;
void *biases_data_z = malloc(biases_data_size);
void *biases_data_r = malloc(biases_data_size);
void *biases_data_h = malloc(biases_data_size);

status = zdnn_transform_ztensor(&biases, biases_data_z, biases_data_r,
                                biases_data_h);
assert(status == ZDNN_OK);

/*****
 * Create hidden weights zTensor
 * Resultant zTensor is concatenated
 *****/

zdnn_tensor_desc hidden_weights_pre_tfrmd_desc, hidden_weights_tfrmd_desc;
zdnn_ztensor hidden_weights;

zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &hidden_weights_pre_tfrmd_desc,
                               num_dirs, num_hidden, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &hidden_weights_pre_tfrmd_desc,
    RNN_TYPE_GRU | USAGE_HIDDEN_WEIGHTS | PREV_LAYER_NONE,
    &hidden_weights_tfrmd_desc);
assert(status == ZDNN_OK);
status = zdnn_init_ztensor_with_malloc(&hidden_weights_pre_tfrmd_desc,
                                       &hidden_weights_tfrmd_desc,
                                       &hidden_weights);

```

```

assert(status == ZDNN_OK);

uint64_t hidden_weights_data_size = num_hidden * num_hidden * element_size;
void *hidden_weights_data_z = malloc(hidden_weights_data_size);
void *hidden_weights_data_r = malloc(hidden_weights_data_size);
void *hidden_weights_data_h = malloc(hidden_weights_data_size);

status = zdnn_transform_ztensor(&hidden_weights, hidden_weights_data_z,
                                hidden_weights_data_r, hidden_weights_data_h);
assert(status == ZDNN_OK);

/*****
 * Create hidden biases zTensors
 * Resultant zTensors are concatenated
 *****/

zdnn_tensor_desc hidden_biases_pre_tfrmd_desc, hidden_biases_tfrmd_desc;
zdnn_ztensor hidden_biases;

zdnn_init_pre_transformed_desc(ZDNN_2DS, type, &hidden_biases_pre_tfrmd_desc,
                                num_dirs, num_hidden);
status = zdnn_generate_transformed_desc_concatenated(
    &hidden_biases_pre_tfrmd_desc,
    RNN_TYPE_GRU | USAGE_HIDDEN_BIASES | PREV_LAYER_NONE,
    &hidden_biases_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(
    &hidden_biases_pre_tfrmd_desc, &hidden_biases_tfrmd_desc, &hidden_biases);
assert(status == ZDNN_OK);

uint64_t hidden_biases_data_size = num_hidden * element_size;
void *hidden_biases_data_z = malloc(hidden_biases_data_size);
void *hidden_biases_data_r = malloc(hidden_biases_data_size);
void *hidden_biases_data_h = malloc(hidden_biases_data_size);

status = zdnn_transform_ztensor(&hidden_biases, hidden_biases_data_z,
                                hidden_biases_data_r, hidden_biases_data_h);
assert(status == ZDNN_OK);

/*****
 * Create output zTensor
 *****/

// get only the last timestep
zdnn_tensor_desc hn_pre_tfrmd_desc, hn_tfrmd_desc;

zdnn_ztensor hn_output_ztensor;

zdnn_init_pre_transformed_desc(ZDNN_4DS, type, &hn_pre_tfrmd_desc, 1, 1,
                                num_batches, num_hidden);
status = zdnn_generate_transformed_desc(&hn_pre_tfrmd_desc, &hn_tfrmd_desc);
assert(status == ZDNN_OK);

status = zdnn_init_ztensor_with_malloc(&hn_pre_tfrmd_desc, &hn_tfrmd_desc,
    &hn_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Call the zAIU
 *****/

void *work_area = NULL;

status = zdnn_gru(&input, &h0, &weights, &biases, &hidden_weights,
    &hidden_biases, dir, work_area, &hn_output_ztensor);
assert(status == ZDNN_OK);

/*****
 * Output and Cleanup
 *****/

uint64_t hn_data_size = num_batches * num_hidden * element_size;
void *hn_output_data = malloc(hn_data_size);

status = zdnn_transform_origtensor(&hn_output_ztensor, hn_output_data);
assert(status == ZDNN_OK);

status = zdnn_free_ztensor_buffer(&input);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&h0);
assert(status == ZDNN_OK);

```

```

status = zdnn_free_ztensor_buffer(&weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_weights);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hidden_biases);
assert(status == ZDNN_OK);
status = zdnn_free_ztensor_buffer(&hn_output_ztensor);
assert(status == ZDNN_OK);

free(input_data);
free(hidden_state_data);
free(weights_data_z);
free(weights_data_r);
free(weights_data_h);
free(hidden_weights_data_z);
free(hidden_weights_data_r);
free(hidden_weights_data_h);
free(biases_data_z);
free(biases_data_r);
free(biases_data_h);
free(hidden_biases_data_z);
free(hidden_biases_data_r);
free(hidden_biases_data_h);
free(hn_output_data);
}

```

Example: Creating a quantized zTensor

```

// SPDX-License-Identifier: Apache-2.0
/*
 * Copyright IBM Corp. 2023
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// *****
// Sample:
//
// Create a quantized zTensors
// *****
int main(int argc, char *argv[]) {
    zdnn_tensor_desc pre_tfirmd_desc, tfirmd_desc;
    zdnn_ztensor ztensor;
    zdnn_status status;

    uint32_t dim_n = 1, dim_h = 32, dim_w = 32, dim_c = 3;
    zdnn_data_types type = FP32;
    short element_size = 4; // size of each element in bytes
    uint64_t num_elements = dim_n * dim_h * dim_w * dim_c;

    // allocate tensor data storage
    void *data1 = malloc(num_elements * element_size);

    // read input_data

    // check status for zAIU availability, supported ops, etc. here
    // status = zdnn_query();

    // set input tensor data to 0 to 127 sequentially and repeat

```

```

for (uint64_t i = 0; i < num_elements; i++) {
    ((float *)data1)[i] = (float)(i & 0x7f);
}

zdn_init_pre_transformed_desc(ZDNN_NHWC, type, &pre_tfmd_desc, dim_n, dim_h,
                             dim_w, dim_c);
float scale = 3;
float offset = 2;

// generate transformed shape information
status = zdn_generate_quantized_transformed_desc(
    &pre_tfmd_desc, QUANTIZED_DLFLOAT16, &tfmd_desc);
assert(status == ZDNN_OK);

// initialize zTensors and allocate 4k-aligned storage via helper function
status = zdn_init_quantized_ztensor_with_malloc(&pre_tfmd_desc, &tfmd_desc,
                                                scale, offset, &ztensor);
assert(status == ZDNN_OK);

// transform the feature tensor
status = zdn_transform_ztensor(&ztensor, data1);
assert(status == ZDNN_OK);

// Free zTensors
status = zdn_free_ztensor_buffer(&ztensor);
assert(status == ZDNN_OK);

free(data1);
}

```

Part 2. IBM Z Artificial Intelligence Optimization library

Chapter 4. Using the IBM Z Artificial Intelligence Optimization Library

The IBM Z Artificial Intelligence Optimization Library (zAIO) provides an interface to core functions used to implement database queries embedded with AI information, extracted from a given database, and expressed as semantic queries. With this library, developers can integrate semantic embedding operations to a database-capable tool, such as IBM Db2®, and SQL-enabled frameworks, such as Spark SQL and Python SQL.

The library is composed of several modules, as shown in [Figure 7 on page 131](#).

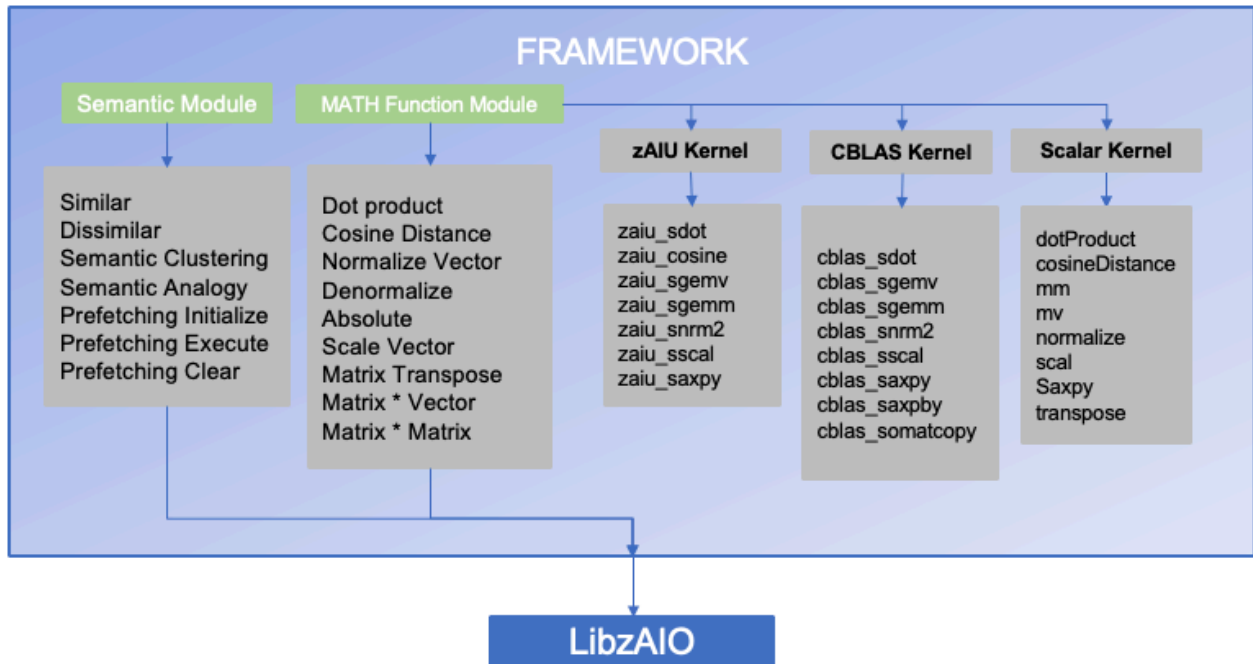


Figure 7. IBM Z Artificial Intelligence Optimization Library framework

The first module, known as the *semantic module*, implements atomic semantic operations on elements or words of a dictionary. The dictionary represents a model composed of unique words and multi-dimensional floating-point vectors. The dictionary or model is the output of a training operation performed on a translated view of a database. See [ibm-data2vec documentation](#) for specific details regarding data input, data processing, training, and output model generation. The semantic operations implement specific mathematical operations as described in the paper, [Demonstrating Semantic SQL Queries over Relational Data using the AI-Powered Database](#) ([aidb-workshop.github.io/aidb2019-proceeding/6-neves.pdf](#)). The semantic operations can be executed in a word-by-word mode or in batch where multiple words are selected for a given semantic operation. For instance, dot product can be a one-to-one operation (similar) or a one-to-many operation (prefetching).

The second module, known as the *mathematical module*, exposes a set of functions designed to operate on vectors and matrices as required by the semantic operations, such as the dot product or cosine distance calculations between two vectors as it is used by semantic similarity. This module is organized around three *kernels* that implement several linear algebra operations in three different ways to explore hardware acceleration features, as available by specific hardware and selected by the user-specified acceleration mode. The first kernel, *scalar*, implements the operations in a hardware agnostic way and thus is the slowest mode for most operations. The second kernel, *CBLAS*, implements the same operations using a linear algebra software package, such as OpenBLAS, which is designed to exploit hardware vector instructions on targeted hardware architectures, such as IBM Z and IBM Power®. The third kernel, *zAIU_Math*, implements the same functions targeting the use of the IBM Z Integrated Accelerator for AI (zAIU) features available on IBM Z systems designed with the IBM Telum processor.

The topics that follow describe in detail how to integrate the library to a C®/C++ application as well as the syntax descriptions of all APIs. It also describes several acceleration capabilities to speed up most of the functions, including how to exploit hardware acceleration with the latest AI acceleration unit on the IBM z16® CPU. A C program example that shows how to use all the APIs is also included.

IBM Z Artificial Intelligence Optimization Library environment

The IBM Z Artificial Intelligence Optimization Library relies on the IBM Z Deep Neural Network and IBM z/OS OpenBLAS libraries to implement the core numerical and semantic computations. When installed, the required libraries can be found by default in the following locations:

| Library | MVS data set | z/OS UNIX file system path |
|--|-------------------------|---|
| IBM Z Deep Neural Network Library | SYS1.SIEALNKE(AIEZDNN) | /usr/lpp/aie/zdnn/lib/libzdnn.so (Prior to z/OS 3.1, the path is /usr/lpp/IBM/aie/zdnn/lib/libzdnn.so.) |
| IBM Z Artificial Intelligence Optimization Library | SYS1.SIEALNKE(AIEZAIOS) | /usr/lpp/aie/zaio/lib/libzaio.so (Prior to z/OS 3.1, the path is /usr/lpp/IBM/aie/zaio/lib/libzaio.so.) |
| IBM z/OS OpenBLAS library | CEE.SCEERUN2(CRTEQOBL) | /usr/lpp/cbclib/lib/libopenblas.dll |

The following example shows how to configure your environment for full use of the IBM Z Artificial Intelligence Optimization libraries residing in the z/OS UNIX file system:

```
ZDNN_INSTALL_DIR="/usr/lpp/aie/zdnn"
ZAI0_INSTALL_DIR="/usr/lpp/aie/zaio"
OPENBLAS_INSTALL_DIR="/usr/lpp/cbclib"
export LIBPATH=${ZDNN_INSTALL_DIR}/lib:${ZAI0_INSTALL_DIR}/lib:${OPENBLAS_INSTALL_DIR}/lib:${LIBPATH}
```

IBM Z Artificial Intelligence Optimization code development

As a developer, you have access to all the functions in the IBM Z Artificial Intelligence Optimization Library by performing the following steps:

1. Include the `zaio.h` file.

```
#include "zaio.h"
```

2. Compile the `.c` or `.cpp` code with the `-I/dir_location_zaio` option.
3. If desired, add to the bind command either the `libzaio.x` z/OS UNIX file or the `AIEZAI0X` MVS member.

Example:

```
obj/sotest.o: test/sotest.cpp
$(CC) -c $(CFLAGS) test/sotest.cpp -o
$@ >
$@.lst sotest: obj/sotest.o $(CC) -o $@ libzaio.x $(LDFLAGS) $< >
$@.lnk.lst
```

Note that **\$(CC)**, **\$(CFLAGS)**, and **\$(LDFLAGS)** are generic variables to be set with your desired parameters for code compilation and binding.

IBM Z Artificial Intelligence Optimization execution

Follow the guidelines described in [“IBM Z Artificial Intelligence Optimization Library environment” on page 132](#).

The `zAIO` library attempts to load the `z/OS OpenBLAS` library and the `zDNN` library during initialization phase in the following order:

1. From the MVS load library (CRTEQOBL and AIEZDNNS)
2. From the z/OS UNIX file system (libopenblas.dll and libzdnn.so), only if the host application is not using Preinitialized Language Environment for Authorized Programs

For details about setting up the **LIBPATH** environment variable and the MVS load library search order, see [Loading DLLs in z/OS XL C/C++ Programming Guide](#).

The setup of the library paths only guarantees that the algebraic functions to be accelerated by hardware are available. The actual use of a given function depends on user input (as described in [Chapter 5, “IBM Z Artificial Intelligence Optimization Library API reference,”](#) on page 135) and internal checks performed by the IBM Z Artificial Intelligence Optimization library to determine if other libraries are installed in the execution machine as well as if the execution machine has the required hardware resources.

Acceleration mode

Hardware acceleration also depends on the system on which the code is being executed. Specifically, SIMD acceleration with the z/OS OpenBLAS library requires an IBM z14® or later system. IBM Z Integrated Accelerator for AI acceleration with the zDNN library requires an IBM z16 or later system. No hardware acceleration is possible with zAIO for older systems, such as IBM z13® or EC12. In such systems, the APIs can still run but will use a scalar implementation. The following table describes the zAIO supported hardware accelerations and their hardware and software requirements.

Table 15. zAIO supported hardware accelerations

| Hardware acceleration | System requirement | Library requirement |
|-------------------------------------|--------------------|---------------------|
| IBM Z Integrated Accelerator for AI | z16 or later | zDNN library |
| SIMD | z14 or later | OpenBLAS library |
| Scalar | EC12 or later | None |

zAIO API return status

Most of the zAIO functions return a 64-bit `zaio_status_t` unsigned number indicating whether the operation was successful, of which the rightmost 16 bits denote the return code:

```
0xxxxxxxxxxxxxxxxrrrr
```

Only the rightmost 16 bits can be used to compare with the mnemonic constants. Exploiters should use the `ZAIO_STATUS_RC` macro to extract the return code from `zaio_status_t` upon zAIO function return.

For debugging purposes, zAIO requires the full 64-bit `zaio_status_t` value to be logged by the exploiter, as in the following example:

```
zaio_status = zaio_absolute(input_a, &result, SIZE);
if (ZAIO_STATUS_RC(zaio_status) != ZAIO_OK) {
    // log zaio_status and handle error
}
```

zAIO return code mnemonic constants

Table 16 on page 133 lists the zAIO return code mnemonic constants and meanings.

Table 16. zAIO return code mnemonic constants

| Mnemonic constant | Meaning |
|-------------------------|-----------------------------------|
| ZAIO_OK | Success. |
| ZAIO_ALLOCATION_FAILURE | Cannot allocate storage. |
| ZAIO_NO_CONTEXT | Missed passing the context space. |
| ZAIO_INVALID_SIZE | Invalid vector size. |

Table 16. zAIO return code mnemonic constants (continued)

| Mnemonic constant | Meaning |
|--------------------------|--|
| ZAIO_INVALID_PARAMETER | Invalid parameter for selected function. |
| ZAIO_MISSING_PARAMETER | Required parameter to selected function is missing. |
| ZAIO_INVALID_DIMENSIONS | Matrix or vector dimensions are invalid. |
| ZAIO_NO_OPT | Caller did not pass a valid function. |
| ZAIO_UNKNOWN_FUNCTION | Selected function is unknown. |
| ZAIO_ERR_ACCEL_SPECIFIC | Accelerator specific; more information in reserved fields. |

Chapter 5. IBM Z Artificial Intelligence Optimization Library API reference

The IBM Z Artificial Intelligence Optimization Library includes the following application programming interfaces (APIs):

- [“zAIO initialization \(zaio_Init\)” on page 135](#)
- [“Check availability of the IBM Z Integrated Accelerator for AI \(zaio_zaiuReady\)” on page 136](#)
- [“Check CBLAS availability \(zaio_cblasReady\)” on page 136](#)
- [“Get library version \(zaio_getVersion\)” on page 136](#)
- [“Copy vector to new location \(zaio_vectorCopy\)” on page 137](#)
- [“Average vector \(zaio_averageVector\)” on page 137](#)
- [“Semantic average \(zaio_semanticAverage\)” on page 138](#)
- [“Dot product \(zaio_dotProduct\)” on page 139](#)
- [“Cosine distance \(zaio_cosineDistance\)” on page 139](#)
- [“Vector normalization \(zaio_normalize\)” on page 140](#)
- [“Vector denormalization \(zaio_denormalize\)” on page 141](#)
- [“Vector absolute \(zaio_absolute\)” on page 141](#)
- [“Vector scale \(zaio_vectorScale\)” on page 142](#)
- [“Matrix-vector multiplication \(zaio_matrixVector\)” on page 143](#)
- [“Matrix-matrix multiplication \(zaio_matrixMatrix\)” on page 143](#)
- [“Matrix transpose \(zaio_transpose\)” on page 144](#)
- [“Semantic similarity \(zaio_semanticSimilarity\)” on page 144](#)
- [“Semantic clustering \(zaio_semanticClustering\)” on page 145](#)
- [“Semantic analogy \(zaio_semanticAnalogy\)” on page 146](#)
- [“Prefetching initialize \(zaio_preFetching_Initialize\)” on page 147](#)
- [“Prefetching execute \(zaio_preFetching_Execute\)” on page 148](#)
- [“Prefetching clear \(zaio_preFetching_Clear\)” on page 149](#)

zAIO initialization (zaio_Init)

Description

This function is automatically run when the library is loaded by an application. It automatically detects the following features and sets the correct execution mode for hardware acceleration:

- The IBM Z machine on which the application is running
- The installed support libraries

Format

```
void zaio_Init();
```

Parameters

None.

Returns

None.

Check availability of the IBM Z Integrated Accelerator for AI (zaio_zaiuReady)

Description

This function determines if the hardware environment contains the IBM Z Integrated Accelerator for AI. This capability is only available on IBM z16 (and later) systems.

Format

```
bool zaio_zaiuReady();
```

Parameters

None.

Returns

Returns `true` if IBM Z Integrated Accelerator for AI hardware and support is available; otherwise, returns `false`.

Check CBLAS availability (zaio_cblasReady)

Description

This function determines if the hardware environment contains the OpenBLAS library and if the hardware on which the code is running is suitable for that library.

Format

```
bool zaio_cblasReady();
```

Parameters

None.

Returns

Returns `true` if SIMD hardware and software is available; otherwise, returns `false`.

Get library version (zaio_getVersion)

Description

Retrieve the library version number and build information as a string.

Format

```
char *zaio_getVersion();
```

Parameters

None.

Returns

Returns the library version number and build information as a string.

Copy vector to new location (zaio_vectorCopy)

Description

This function copies a given vector to a provided new location as to preserve the data in the original vector.

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```

Format

```
zaio_status_t zaio_vectorCopy(float *v, float *vtmp, int v_size);
```

Parameters

float *v

A pointer to a vector of size `v_size` that will be copied.

float *vtmp

A pointer to a vector of size `v_size` to which `v` will be copied.

int v_size

The dimension of both vectors.

Returns

The result of the operation is stored in `vtmp` while the function returns a status of `zaio_status_t`, of which the rightmost 16 bits denote the return code.

Average vector (zaio_averageVector)

Description

This function returns the average of two vectors. The average is the sum of the individual positions of each vector divided by K , where K is equal to 2 for this function. The caller must allocate space for the return vector of size `v_size`.

The following figure illustrates this equation:

$$\text{Average}(V_1, V_2, \dots, V_k) = \frac{(V_{1i} + V_{2i} + \dots + V_{ki})}{k} \{i: 1 \rightarrow v_size\}$$

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```

Format

```
zaio_status_t zaio_averageVector(float *v1, float *v2, float *avg, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float *v2

A pointer to the second vector of size *v_size*.

float *avg

A pointer to the output vector of size *v_size*.

int v_size

The dimension of the vectors.

Returns

The result of the operation is stored in *avg* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Semantic average (zaio_semanticAverage)

Description

This function returns the average of three vectors. The average is the sum of the individual positions of each vector divided by *K*, where *K* is equal to 3 for this function. This function is a convenience function for semantic clustering where the similarity between one vector is compared to the average of multiple vectors.

The following figure illustrates this equation:

$$\text{Average}(V_1, V_2, \dots, V_k) = \frac{(V_{1i} + V_{2i} + \dots + V_{ki})}{k} \{i: 1 \rightarrow v_size\}$$

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```

Format

```
zaio_status_t zaio_semanticAverage(float *v1, float *v2, float *v3, float *avg, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float *v2

A pointer to the second vector of size *v_size*.

float *v3

A pointer to the third vector of size *v_size*.

float *avg

A pointer to the output vector of size *v_size*.

int v_size

The dimension of the vectors.

Returns

The result of the operation is stored in *avg* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Dot product (zaio_dotProduct)

Description

This function performs the dot product operation between two vectors of the same size, as shown by the following equation.

$$\text{Dot Product } V_1 V_2 = \sum_{i=1}^{v_size} V_{1_i} V_{2_i}$$

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```

Format

```
zaio_status_t zaio_dotProduct(float *v1, float *v2, float *result, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float *v2

A pointer to the second vector of size *v_size*.

float *result

The location where the dot product is stored. The caller passes the address to this location.

int v_size

The dimension of both vectors.

Returns

The result of the operation is stored in *result* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Cosine distance (zaio_cosineDistance)

Description

This function returns the cosine distance between two vectors. The cosine distance is defined as the dot product between two vectors, divided by the product of the length of each vector, as shown by the following equation. If both vectors are normalized beforehand, the cosine distance can be obtained by a single dot product operation making it much faster to calculate. The output of the operation is a value between -1.0 and 1.0. The maximum is 1.0 when both vectors point in the same direction. It is -1.0 when both vectors point in opposite directions, and it is 0 when the vectors are perpendicular to each other.

$$\text{Cosine Similarity } \cos(V_1, V_2) = \frac{V_1 V_2}{\|V_1\| \|V_2\|} = \frac{\sum_{i=1}^{v_size} V_{1_i} V_{2_i}}{\sqrt{\sum_{i=1}^{v_size} V_{1_i}^2} \sqrt{\sum_{i=1}^{v_size} V_{2_i}^2}}$$

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zai0.h>
```

Format

```
zai0_status_t zai0_cosineDistance(float *v1, float *v2, float *distance, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size `v_size`.

float *v2

A pointer to the second vector of size `v_size`.

float *distance

The location where the cosine distance is stored. The caller passes the address to this location.

int v_size

The dimension of both vectors.

Returns

The result of the operation is stored in `distance` while the function returns a status of `zai0_status_t`, of which the rightmost 16 bits denote the return code.

Vector normalization (zai0_normalize)

Description

Vector normalization is defined as making the length of a vector to be 1. This is accomplished by dividing each element of the vector by its norm or L2-norm, which is the square root of the sum of the squares of each element in the vector, as shown in the following equation:

$$\text{Vector Normalization } \|V\| = \frac{V_i}{\sqrt{\sum_{i=1}^{v_size} V_i^2}} \quad \{i: 1 \rightarrow v_size\}$$

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zai0.h>
```

Format

```
zai0_status_t zai0_normalize(float *v1, int v_size);
```

Parameters

float *v1

A pointer to a vector of size *v_size*.

int v_size

The dimension of the vector.

Returns

The result of the operation is the updated vector *v1* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Notes

This is a *destructive* operation. The *v1* vector will return containing the normalized version of the *v1* vector. If the *v1* vector needs to be preserved, use the `zaio_vectorCopy` function to create a copy prior to calling this function.

Vector denormalization (zaio_denormalize)

Description

Vector denormalization is a helper function that you can run to get a new vector. It implies two things: that the vector being passed is a normalized vector, and that the scale passed to the function has been obtained by running `zaio_absolute` on a given vector.

Format

```
zaio_status_t zaio_denormalization(float *v1, float scale, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float scale

Value to change the passing vector.

int v_size

The dimension of the vector.

Returns

The function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Notes

This is a *destructive* operation. The *v1* vector will return containing the normalized version of the *v1* vector. If the *v1* vector needs to be preserved, use the `zaio_vectorCopy` function to create a copy prior to calling this function.

Vector absolute (zaio_absolute)

Description

The absolute value of a vector, also known as the length of a vector, is also known as the L2-norm of a vector and is defined as the square root of the sum of each vector element squared. This value is needed to denormalize a given vector.

Format

```
zaio_status_t zaio_absolute(float *v1, float *result, int v_size);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float *result

The location where the L2-norm is stored. The caller passes the address to this location.

int v_size

The dimension of the vector.

Returns

The result of the operation is stored in *result* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Vector scale (zaio_vectorScale)

Description

This function scales an input vector, *x*, by a constant, *alpha*, and adds the result to another vector, *y*. If scaling is desired, you can initialize vector *y* with zeros. Both vectors must be of the same size.

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```

Format

```
zaio_status_t zaio_vectorScale(float *x, float *y, float alpha, int v_size);
```

Parameters

float *x

A pointer to the first vector of size *v_size*.

float *y

A pointer to the second vector of size *v_size*.

float alpha

The scale constant to apply to *x*.

int v_size

The dimension of both vectors.

Returns

The function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Notes

This is a *destructive* operation. The *y* vector will return containing the results of the scalar and addition operations. If the *y* vector needs to be preserved, use the `zaio_vectorCopy` function to create a copy prior to calling this function.

Matrix-vector multiplication (zaio_matrixVector)

Description

Scales an input vector, X by a constant, α , and adds the result to another vector, Y . If scaling is desired, you can initialize vector Y with zeros. Both vectors must be of the same size.

Format

```
zaio_status_t zaio_matrixVector(float *A, float *v, float *vr, int m, int n, int v_size);
```

Parameters

float *A

A pointer to the matrix of size $[m, v_size]$.

float *v

A pointer to the vector of size v_size .

float *vr

A pointer to the results vector of size m .

int m

The number of rows in the matrix.

int n

The number of columns in the matrix. This value must be equal to v_size .

int v_size

The dimension of the vector.

Returns

The function returns the result of the multiplication of A by the vector, and a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Matrix-matrix multiplication (zaio_matrixMatrix)

Description

Scales an input vector, X , by a constant, α , and adds the result to another vector, Y . If scaling is desired, you can initialize vector Y with zeros. Both vectors must be of the same size.

Format

```
zaio_status_t zaio_matrixMatrix(float *A1, float *A2, float *Mout, int A1_row, int A1_col,
                                int A2_row, int A2_col);
```

Parameters

float *A1

A pointer to the first matrix of size $[A1_row, A1_col]$.

float *A2

A pointer to the second matrix of size $[A2_row, A2_col]$.

float *Mout

A pointer to the results matrix. This matrix must be of size $[A1_row, A1_col]$.

int A1_row

The number of rows in the $A1$ matrix.

int A1_col

The number of columns in the $A1$ matrix.

int A2_row

The number of rows in the A2 matrix.

int A2_col

The number of columns in the A2 matrix.

Returns

The function returns the result of the multiplication of A1 by A2, and a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Matrix transpose (zaio_transpose)

Description

This function implements the transpose of a matrix. If a matrix has a dimension $[M, N]$, where M is the number of rows and N is the number of columns, its transpose will have a dimension of $[N, M]$. The transpose operation implies getting a row from the original matrix and making it a column in the transposed matrix. The first row becomes the first column, the second row becomes the second column, and so on until all the rows are transposed.

Format

```
zaio_status_t zaio_transpose(float *A, float *A_T, int m, int n);
```

Parameters

float *A

A pointer to the first element of matrix A.

float *A_T

A pointer to the first element of the transposed matrix A_T.

int m

The number of rows in matrix A.

int n

The number of columns in matrix A.

Returns

The function returns the result of the transpose of A in A_T, and a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Semantic similarity (zaio_semanticSimilarity)

Description

This function calculates the semantic similarity between two vectors. This is equivalent to calculating the cosine similarity between the two vectors. If the vectors are normalized, you can set the boolean variable *normal* to `true`, and the function will perform a dot product between the vectors instead of cosine distance. The output of the operation is a value between -1.0 and 1.0. The maximum is 1.0 when both vectors point in the same direction. It is -1.0 when both vectors point in opposite directions, and it is 0 when the vectors are perpendicular to each other.

Note: This function is enabled with directive `_ZAI0_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAI0_SOURCE 2
#include <zaio.h>
```


Format

```
zaio_status_t zaio_semanticSimilarity(float *v1, float *v2, float *result,  
                                     int v_size, bool normal);
```

Parameters

float *v1

A pointer to the first vector of size *v_size*.

float *v2

A pointer to the second vector of size *v_size*.

float *result

The location where the similarity between vectors is stored. The user passes the address of this location.

int v_size

The dimension of both vectors.

bool normal

True if the vectors are known to be normalized; otherwise, false.

Returns

The result of the operation is stored in *result* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Semantic clustering (zaio_semanticClustering)

Description

This function calculates the semantic similarity between a vector and the average of 3 vectors. The 3 vectors are constant throughout the operation, while the fourth vector changes. Note that the 3 vectors must be denormalized prior to calculating the average using the *zaio_semanticAverage* function. The average of 3 normalized vectors is different than the average of 3 unnormalized vectors. The 3 vectors can be denormalized by using the *zaio_denormalize* function provided that the length of the original vectors is known. If the fourth vector is normalized, you can set the boolean variable *normal* to true, and the function will normalize the average and perform a dot product between the average and the fourth vector instead of cosine distance. If the variable is set to false, the average is not normalized and a cosine distance is performed between the average and the fourth vector. The output of the operation is a value between -1.0 and 1.0. The maximum is 1.0 when both vectors point in the same direction. It is -1.0 when both vectors point in opposite directions, and it is 0 when the vectors are perpendicular to each other.

Format

```
zaio_status_t zaio_semanticClustering(float *vx, float *vy, float *vq,  
                                     float *vw, float *result, int v_size, bool normal);
```

Parameters

float *vx

A pointer to the first vector of size *v_size*.

float *vy

A pointer to the second vector of size *v_size*.

float *vq

A pointer to the third vector of size *v_size*.

float *vw

A pointer to the fourth vector of size *v_size*.

float *result

The location where the result is to be stored. The caller passes the address of this location.

int v_size

The dimension of all vectors.

bool normal

True if the vectors are known to be normalized; otherwise, false.

Note: The *vx*, *vy*, and *vq* vectors must be unnormalized. The state of the *vw* vector determines how the boolean variable is used: If normalized, the variable should be true; otherwise, false.

Returns

The result of the operation is stored in *result* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Semantic analogy (zaio_semanticAnalogy)

Description

Analogy is a concept that refers to the relationship between a pair of entities for the purpose of illustrating a feature they share. This shared feature is then used to determine if two other entities share a similar feature.

An example of analogy query might be *Lawyer : Client :: Doctor : ?* where the expected answer would be *Patient*. Note that the answer is predicated on the text describing the relationship. In terms of semantics, 3 vectors are constant (*Lawyer*, *Client*, and *Doctor*) and the fourth vector is the one we want to find in the corpus of relevant vectors.

Mathematically, the operation performed is illustrated in the following figure and the similarities between vectors are changed such that the analogy score is non-negative.

$$Analogy = \frac{\cos(V_w, V_q)\cos(V_w, V_y)}{\cos(V_w, V_x) + \epsilon}$$

In this equation, vectors *Vx*, *Vy*, and *Vq* are the constant vectors and *Vw* is the unknown vector that we want to find whose relationship to *Vq* better matches the relationship between *Vx* and *Vy*, that is *Vx : Vy :: Vq : Vw*.

Format

```
zaio_status_t zaio_semanticAnalogy(float *vx, float *vy, float *vq, float *vw,  
                                   float *result, int v_size, bool normal);
```

Parameters

float *vx

A pointer to the first constant vector of size *v_size*.

float *vy

A pointer to the second constant vector of size *v_size*.

float *vq

A pointer to the third constant vector of size *v_size*.

float *vw

A pointer to the variable vector of size *v_size*.

float *result

The location where the result is to be stored. The caller passes the address of this location.

int v_size

The dimension of all vectors.

bool normal

True if the vectors are known to be normalized; otherwise, false.

Returns

The result of the operation is stored in *result* while the function returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Prefetching initialize (zaio_preFetching_Initialize)

Description

Semantic prefetching allows a user to execute semantic operations in matrix mode instead of vector mode or row-by-row as other operations described here are executed. Semantic similarity ([zaio_semanticSimilarity](#)), semantic clustering ([zaio_semanticClustering](#)), and semantic analogy ([zaio_semanticAnalogy](#)) are functions that work on vectors. However, for a given model used by Db2 SQL DI or any other application that explores semantic operations, the model contents does not change when processing an SQL query. As such, calculating vector (V_x) by vector (V_y), where V_x comes from a set of possible vectors and V_y is the comparing vector, or matrix (collection of many V_x by vector (V_y), produces the same results. The advantage of matrix mode is speed calculation and, if the results are temporarily saved, they can be reused. For such purposes, matrix-based APIs are added to the library and a common interface for multi-row operations is also added. This common interface is called prefetching and consists of three steps: [initialize \(zaio_preFetching_Initialize\)](#) described here, [execute \(zaio_preFetching_Execute\)](#) where the semantic operations are executed, and [clear \(zaio_preFetching_Clear\)](#) to deallocate any resources allocated during initialization and used during execution.

By calling this API, you can initialize a context that contains the desired semantic operation from the list of possible operations. Note that the operations listed have individual APIs for vector based operations.

```
typedef enum semantic_functions {
    NO_OPERATION,
    SIMILARITY,
    DISSIMILARITY,
    CLUSTERING,
    ANALOGY
} semanticOpt_t;
```

Further, the context will also have pointers to the data, such as matrices and/or vectors, as well as their respective dimensions. This API checks for several parameter values depending on the chosen semantic operation and returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code. Note that the API checks if the pointers to the data structures are NULL or not but does not check if the allocated space or its contents is valid. The outcome of the API is a pointer to a context with all the relevant members filled up. This data structure for the context follows:

```
struct zaio_preFetching_context {
    semanticOpt_t function;
    int vector_size;
    int inputConstantParameters;
    float *inputConstantArray;
    int numberVectors;
    float *inputVectorMatrix;
    int outputParameters;
    float *outputArray;
    bool clustered;
    char reserved[64];
};
```

The purpose of this strategy is several fold. First, the execution of semantic operations in matrix mode goes through a single interface. As new operations are developed, you only need to specify the operation from the list of available operations and pass it as a parameter to the `zaio_preFetching_Initialize` API.

Another purpose is that it allows you to execute a context many times before it needs to be cleared. This is needed because matrices can be very large and for memory management purposes it may be desirable to execute an operation on subsets of the original matrix instead of passing the full matrix. Also, you can create multiple contexts for different purposes. Each can be executed with the same `zaio_preFetching_Execute` API call.

Format

```
zaio_status_t zaio_preFetching_Initialize(zaio_preFetching_context_t *PFContext,
                                         semanticOpt_t function, int size,
                                         int inputConstantParameters,
                                         float *inputConstantArray, int numberVectors,
                                         float *inputVectorMatrix, int outputParameters,
                                         float *outputArray);
```

Parameters

zaio_preFetching_context_t *PFContext

A pointer to context the user allocated prior to the call to the API.

semanticOpt_t function

The semantic operation selected from the following list:

- NO_OPERATION
- SIMILARITY
- DISSIMILARITY
- CLUSTERING
- ANALOGY

int size

The dimension of all vectors.

int inputConstantParameters

A value indicating whether one or more vectors are passed as constant.

float *inputConstantArray

A pointer to an array of one or more vectors.

int numberVectors

The number of vectors in the matrix. Matrix size is `[numberVectors, size]`.

float *inputVectorMatrix

A pointer to the input matrix containing `numberVectors` vectors.

int outputParameters

The size of the output array containing the results of the operation.

float *outputArray

An array containing the results. The size of the array is `[numberVectors, inputConstantParameters]`

Returns

The result of the operation is the context pointer updated with all the fields. The API also returns a status of `zaio_status_t`, of which the rightmost 16 bits denote the return code.

Prefetching execute (zaio_preFetching_Execute)

Description

This API executes the semantic operation assigned to the input context.

Format

```
zaio_status_t zaio_preFetching_Execute(zaio_preFetching_context_t *PFContext);
```

Parameters

zaio_preFetching_context_t *PFContext

A pointer to the context that you allocated prior to the call to this API.

Returns

The result of the operation is updated in the *outputArray* member of the context. The API also returns a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Prefetching clear (zaio_preFetching_Clear)

Description

This API resets all the members of a context to default values and resets the semantic operation to NO_OPERATION. Default values means assigning NULL to pointers and 0 to all the integer members. It does not deallocate the context because this is allocated by the user of prefetching. Note that all active contexts must be cleared by the user prior to completing the application to avoid memory leaks.

Format

```
zaio_status_t zaio_preFetching_Clear(zaio_preFetching_context_t *PFContext);
```

Parameters

zaio_preFetching_context_t *PFContext

A pointer to the context that you allocated prior to the call to this API.

Returns

The result of the operation is the context with all the members reset to default values and a status of *zaio_status_t*, of which the rightmost 16 bits denote the return code.

Chapter 6. Examples of using the IBM Z Artificial Intelligence Optimization Library APIs

The following examples illustrate how to code and use some of the IBM Z Artificial Intelligence Optimization Library APIs to develop applications.

- “C example: Explicit DLL load” on page 151
- “C example: Implicit DLL load” on page 153

C example: Explicit DLL load

```
/* DLL_TEST

Testcase to validate the call and use of functions in a DLL
It checks if the library exists. Afterwards, it checks and loads the
library functions and calls the functions and print the results

*/

#define _UNIX03_SOURCE 1
#include <dlfcn.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#define _ZAI0_SOURCE 2
#include "zaio.h"

int main() {
    char *name="libzaio.so";
    void *mylib;
    int eret;
    bool (* ai_zaiu)();
    zaio_status_t (* ai_cp)(float *, float *, int);
    zaio_status_t (* ai_dotp)(float *, float *, float *, int);
    zaio_status_t (* ai_avg)(float *, float *, float *, int);
    zaio_status_t (* ai_cosDist)(float *, float *, float *, int);
    zaio_status_t (* ai_norm)(float *, int);
    zaio_status_t (* ai_similar)(float *, float *, float *, int, bool);

    int size = 200;

    // allocate on heap for large allocations
    float *v1 = (float *)malloc(size*sizeof(float));
    float *v2 = (float *)malloc(size*sizeof(float));
    float *v3 = (float *)malloc(size*sizeof(float));
    float *v4 = (float *)malloc(size*sizeof(float));
    float *avg = (float *)malloc(size*sizeof(float));
    float *vnorm1 = (float *)malloc(size*sizeof(float));
    float *vnorm2 = (float *)malloc(size*sizeof(float));
    for(int i = 0; i < size; i++) {
        v1[i] = i + 1;
        v2[i] = i + 2;
        v3[i] = i + 3;
        v4[i] = i + 4;
    }

    // Check if library ZAI0 is installed in the system
    // True if it is. False if it is not and the program finishes
    //
    mylib = dlopen(name, RTLD_LOCAL | RTLD_LAZY);
    if (mylib == NULL) {
        printf("Error: %s dll is not available\n", name);
        perror("failed on dllload");
        exit(-1);
    } else {
        printf("Found dll library: %s\n", name);
    }
}
```

```

    if ((ai_zaiu = (bool (*)(void))dlsym(mylib, "zaio_zaiuReady")) == NULL) {
        printf("Could not find the function zaiuReady in the dll\n");
        exit(-1);
    }
    if ((ai_cp = (zaio_status_t (*)(float *, float *, int))dlsym(mylib, "zaio_vectorCopy")) ==
        NULL) {
        printf("Could not find the function zaio_vectorCopy in the dll\n");
        exit(-1);
    }
    if ((ai_dotp = (zaio_status_t (*)(float *, float *, float *, int))dlsym(mylib,
        "zaio_dotProduct")) == NULL) {
        printf("Could not find the function zaio_dotProduct in the dll\n");
        exit(-1);
    }
    if ((ai_avg = (zaio_status_t (*)(float *, float *, float *, int))dlsym(mylib,
        "zaio_averageVector")) == NULL) {
        printf("Could not find the function zaio_averageVector in the dll\n");
        exit(-1);
    }
    if ((ai_cosDist = (zaio_status_t (*)(float *, float *, float *, int))dlsym(mylib,
        "zaio_cosineDistance")) == NULL) {
        printf("Could not find the function zaio_cosineDistance in the dll\n");
        exit(-1);
    }
    if ((ai_norm = (zaio_status_t (*)(float *, int))dlsym(mylib, "zaio_normalize")) == NULL) {
        printf("Could not find the function in the dll\n");
        exit(-1);
    }
    if ((ai_similar = (zaio_status_t (*)(float *, float *, float *, int, bool))dlsym(mylib,
        "zaio_semanticSimilarity")) == NULL) {
        printf("Could not find the function in the dll\n");
        exit(-1);
    }
}

// Check if machine is ready for IBM Integrated Accelerator for AI acceleration
// A similar check can be performed to determine if the machine is
// ready for openblas by repeating the code below checking for
// zaio_cblasReady()
//

// Check if IBM Integrated Accelerator for AI is present in the machine
bool xZAIU = ai_zaiu();
if ( xZAIU )
    printf("machine is ready for IBM Integrated Accelerator for AI acceleration
(ZAIU_ACCELERATION parm)\n");
else
    printf("No hardware acceleration available, use CBLAS acceleration (CBLAS_ACCELERATION parm)
\n");

zaio_status_t zaio_status;
float xResult;

// Execute Dot product between two vectors
/* call ai_dotproduct */
zaio_status = ai_dotp(v1, v2, &xResult, size);
printf("Dot product of v1 and v2: %f\n", xResult);

// Execute the average between two vectors
/* call ai_averagevector */
zaio_status = ai_avg(v1, v2, avg, size);
printf("Average between v1 and v2: \n");
for(int i = 0; i < size; i++)
    printf("%4.2f ", avg[i]);
printf("\n");

// Calculate the cosine distance between two vectors
/* call ai_cosineDistance */
zaio_status = ai_cosDist(v1, v2, &xResult, size);
printf("cosine distance of v1 to v2: %f\n", xResult);

// Calculate the normal of two vectors
/* Copy vectors */
zaio_status = ai_cp(v1, vnorm1, size);
zaio_status = ai_cp(v2, vnorm2, size);
/* call ai_normalize vector */
zaio_status = ai_norm(vnorm1, size);
zaio_status = ai_norm(vnorm2, size);
// Calculate the semantic similarity between two vectors
zaio_status = ai_similar(vnorm1, vnorm2, size, &xResult, true);
printf("similarity between v1 to v2, dot_product: %f\n", xResult);

free(v1);

```



```

    free(v2);
    free(v3);
    free(v4);
    free(avg);
    free(vnorm1);
    free(vnorm2);

    eret = dlclose(mylib);

    exit(0);
}

```

C example: Implicit DLL load

```

/* SO_TEST

   Testcase to validate the call and use of functions in a shared library.
   There is no need to load the library functions since the reference
   is determined at link time

*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <dll.h>

#define _ZAIQ_SOURCE 2
#include "zaio.h"

int main() {

    int size = 200;
    float xResult = 0.0;
    zaio_status_t zaio_status;

    // allocate on heap for large allocations
    float *v1 = (float *)malloc(size*sizeof(float));
    float *v2 = (float *)malloc(size*sizeof(float));
    float *v3 = (float *)malloc(size*sizeof(float));
    float *v4 = (float *)malloc(size*sizeof(float));
    float *avg = (float *)malloc(size*sizeof(float));
    float *vnorm1 = (float *)malloc(size*sizeof(float));
    float *vnorm2 = (float *)malloc(size*sizeof(float));
    for(int i = 0; i < size; i++) {
        v1[i] = i + 1;
        v2[i] = i + 2;
        v3[i] = i + 3;
        v4[i] = i + 4;
    }

    zaio_status = zaio_dotProduct(v1, v2, &xResult, size);
    printf("Dot product of v1 and v2: %f\n", xResult);

    zaio_status = zaio_averageVector(v1, v2, avg, size);
    printf("Average between v1 and v2: \n");
    for(int i = 0; i < size; i++)
        printf("%4.2f ", avg[i]);
    printf("\n");

    zaio_status = zaio_semanticAverage(v1, v2, v3, avg, size);
    printf("Average between three vectors: \n");
    for(int i = 0; i < size; i++)
        printf("%4.2f ", avg[i]);
    printf("\n");

    zaio_status = zaio_cosineDistance(v1, v2, &xResult, size);
    printf("cosine distance of v1 to v2: %f\n", xResult);

    /* Copy original vectors to temporary structures to
       preserve the original data */
    zaio_vectorCopy(v1, vnorm1, size);
    zaio_vectorCopy(v2, vnorm2, size);

    zaio_status = zaio_normalize(vnorm1, size);
    zaio_status = zaio_normalize(vnorm2, size);

    zaio_status = zaio_semanticSimilarity(vnorm1, vnorm2, &xResult,

```

```

                                size, true);
printf("similarity between v1 to v2, dot_product: %f\n", xResult);
zaio_status = zaio_semanticSimilarity(v1, v2, &xResult,
                                size, false);
printf("similarity between v1 to v2, cosine_distance: %f\n", xResult);

free(v1);
free(v2);
free(v3);
free(v4);
free(avg);
free(vnorm1);
free(vnorm2);

exit(0);
}

```

Part 3. IBM Z Artificial Intelligence Data Embedding library

Chapter 7. Using the IBM Z Artificial Intelligence Data Embedding Library

The IBM Z Artificial Intelligence Data Embedding Library is a collection of packages designed to build and process vector embedding models on z/OS.

The IBM Z Artificial Intelligence Data Embedding Library provides a user-facing ZADE Java™ class. The library is built as a Java application that integrates native packages that can be invoked using this user-facing ZADE Java class. See [Chapter 8, “IBM Z Artificial Intelligence Data Embedding Library API reference,” on page 159](#) for packaging specific invocations. The front-end Java class takes user-provided input parameters as input and passes them to the respective package functions. Internally, the Java class spawns a special manager process, `zade_main`, that controls the execution of the underlying functions.

The IBM Z Artificial Intelligence Data Embedding Library supports all IBM Z models that are supported by z/OS 2.5 and z/OS 3.1.

base10Cluster

The `base10Cluster` package implements a numerical clustering algorithm that aims to group numerically closer items into different buckets, each representing a distinct cluster. The `base10Cluster` algorithm uses 2 steps to process the input numerical data set:

1. **Binning:** After initial processing, each numerical value is assigned to a bin as determined by its base 10 logarithmic value.
2. **Redistribution:** After binning, the bins are redistributed to get a balanced clustering across different buckets.

The `base10Clustering` algorithm is designed to handle out-of-range, EMPTY, and NULL values by assigning them to a special bucket, termed EMPTY. The `base10Cluster` is a single-pass algorithm and can operate on multiple input data sets by using multiple threads. For each input data set, the `base10Cluster` algorithm generates an output file listing the number of buckets and their corresponding minimum values. This information is used by the pre-processing code to assign a string token identifier to a numeric value.

ibm-data2vec

The `ibm-data2vec` package is an implementation of a self-supervised database embedding algorithm. The database embedding takes as input a text file created from a multi-modal relational table and builds a relationship map between text tokens using the relational data model.

The input training document generated from a relational table consists of string tokens representing different relational entities in the original table. The `ibm-data2vec` algorithm views the training document as a set of sentences, each sentence representing a relational table row. However, unlike the traditional natural language processing approaches (such as word embedding), database embedding views each sentence as an unordered bag of tokens (words) where each word is equally related to every other word. In addition, `ibm-data2vec` supports two special tokens: primary key tokens representing a row, and EMPTY tokens for relational NULL values.

After the training is completed, for each token, `ibm-data2vec` generates a vector of pre-defined length (dimension) that encodes the meaning of that token. The inferred meaning captures the collective contributions of neighboring tokens in all rows in which the input token appears. The core numerical computations of the training process are parallelized using multiple threads and accelerated using hardware-accelerated numerical computations. The final, trained model is stored as a binary file using the DB2 zLoad format.

IBM Z Artificial Intelligence Data Embedding Library environment

The IBM Z Artificial Intelligence Data Embedding Library relies on the IBM Z Artificial Intelligence Optimization Library and IBM z/OS OpenBLAS libraries to implement the core numerical computations. When installed, the required libraries can be found by default in the following locations:

| Library | Path |
|--|--|
| IBM Z Artificial Intelligence Data Embedding Library | /usr/lpp/aie/zade (Prior to z/OS 3.1, the path is /usr/lpp/IBM/aie/zade.) |
| IBM Z Artificial Intelligence Optimization Library | /usr/lpp/aie/zaio (Prior to z/OS 3.1, the path is /usr/lpp/IBM/aie/zaio.) |
| IBM z/OS OpenBLAS library | /usr/lpp/cbclib |

During run time, the following environment variables must be set up:

LIBPATH

Specifies the path to search for dynamic link libraries and must include the path to each of the required libraries, as previously described.

CLASSPATH

Specifies the path to search for user-defined classes and packages and must include the path to the `zade.jar` file found under the IBM Z Artificial Intelligence Data Embedding Library directory (that is, `/usr/lpp/aie/zade/lib/zade.jar`).

PATH

Specifies the path to search for executable programs and must include the path to the `zade_main` executable (that is, `/usr/lpp/aie/zade/bin`) and the path to the Java 8 64-bit executable (for instance, `/usr/lpp/java/J8.0_64/bin`).

In addition, 64-bit Java 8 must be used when invoking the available JNI APIs.

The following example shows how to configure your environment for full use of the IBM Z Artificial Intelligence Data Embedding Library:

```
ZADE_INSTALL_DIR="/usr/lpp/aie/zade"
ZAI0_INSTALL_DIR="/usr/lpp/aie/zaio"
OPENBLAS_INSTALL_DIR="/usr/lpp/cbclib"
JAVA_HOME="/usr/lpp/java/J8.0_64/bin"

export LIBPATH=${ZADE_INSTALL_DIR}/lib:${ZAI0_INSTALL_DIR}/lib:${OPENBLAS_INSTALL_DIR}/lib:${LIBPATH}
export CLASSPATH=${ZADE_INSTALL_DIR}/lib/zade.jar:${CLASSPATH}
export PATH=${ZADE_INSTALL_DIR}/bin:${JAVA_HOME}/bin:${PATH}
```

Note: If the z/OS OpenBLAS library path is not added, the zAI0 library will run in the slower scalar mode.

IBM Z Artificial Intelligence Data Embedding Library permissions

The package functions provided by the IBM Z Artificial Intelligence Data Embedding Library might need to access input and write output files. Therefore, the caller must have proper permissions, as follows:

- Read access to input files and their parent directories
- Write access to requested output directories or files
- Write access to the current working directory from which the library is invoked

The IBM Z Artificial Intelligence Data Embedding Library does not change or elevate permissions in any way and retains permissions given by the caller's environment and application.

Chapter 8. IBM Z Artificial Intelligence Data Embedding Library API reference

The IBM Z Artificial Intelligence Data Embedding Library includes the following application programming interfaces (APIs).

- Set functions:
 - [“setNumThreads” on page 159](#)
 - [“setModelFormat” on page 159](#)
 - [“setVocabFormat” on page 160](#)
- Get functions:
 - [“getVersion” on page 160](#)
- Operations:
 - [“base10Cluster” on page 161](#)
 - [“train” on page 162](#)
- Legacy operation functions:
 - [“callBase10Cluster” on page 163](#)
 - [“callTrainDb2Vec” on page 163](#)

Set functions

setNumThreads

Description

Sets the number of threads.

Format

```
public void setNumThreads(int numThreads)
```

Parameters

int numThreads

Number of threads to use for parallelism.

Since

1.3.0

setModelFormat

Description

Sets the ibm-data2vec output model format.

Format

```
public void setModelFormat(ModelFormat modelFormat)
```

Parameters

ModelFormat *modelFormat*

The valid values are:

ModelFormat.BINARYDB2_V110

Db2 zLoad format

ModelFormat.BINARYDB2_LOAD_V110

Db2 load format

Since

1.3.0

setVocabFormat

Description

Sets the ibm-data2vec output vocab file format.

Format

```
public void setVocabFormat(VocabFormat vocabFormat)
```

Parameters

VocabFormat *vocabFormat*

The valid values are:

VocabFormat.NONE

No vocab file is requested.

VocabFormat.VOCAB_FILE_FMT_1

List of all unique tokens and their number of occurrences.

VocabFormat.VOCAB_FILE_FMT_2

All entries in VOCAB_FILE_FMT_1 plus the number of unique tokens count for each column.

Since

1.3.0

Get functions

getVersion

Description

Returns the version of IBM Z Artificial Intelligence Data Embedding Library.

Format

```
public String getVersion()
```

Parameters

None.

Returns

The version of IBM Z Artificial Intelligence Data Embedding Library as a string.

Since

1.3.0

Operations

base10Cluster

Description

Invokes the base10Cluster operation.

Note: You must call `setNumThreads` prior to calling `base10Cluster`.

Format

```
public void base10Cluster(Collection<String> files, String outputPath)
    throws ZadeException, IllegalArgumentException
```

Parameters

Collection<String> files

List of input file names in the CSV format.

String outputPath

Directory where the output files will be stored. This directory must already exist.

Throws

| Exception | Meaning |
|--------------------------|--|
| ZadeException | An error occurred in the zade_main manager process. |
| B10cException | An error occurred during the b10cluster operation in the zade_main manager process. |
| IllegalArgumentException | One of the following reasons: <ul style="list-style-type: none">• The <code>setNumThreads</code> function was never called or was set to a value not greater than 0.• The <code>files</code> parameter does not contain any files or is null.• The <code>files</code> parameter contains files that do not exist or contains a blank string.• The <code>outputPath</code> parameter is a blank string or is null.• Two or more files in <code>files</code> would have generated the same output file name. |

You can retrieve the error message by calling the exception's `getMessage` method.

Output files

When successful, `base10Cluster` produces a `_output_minimums` file for every input file. Each output file indicates the cluster minimum values of the corresponding CSV file, sorted in increasing order. All output files are stored in the `outputPath` directory.

The function produces a log file, `base10cluster-timestamp.log`, in the current working directory, which contains the execution log messages from the function.

Since

1.3.0

train

Description

Invokes the ibm-data2vec operation.

Note: You must call `setNumThreads`, `setModelFormat` (and `setVocabFormat` if a vocab file is requested) prior to calling `train`.

Format

```
public void train(String trainFile, String modelFile, String vocabFile)
    throws ZadeException, IllegalArgumentException
public void train(String trainFile, String modelFile, String vocabFile,
    long preallocSize)
    throws ZadeException, IllegalArgumentException
```

Parameters

String trainfile

Name of the input training document.

String modelFilePrefix

Name of the output trained model.

String vocabFile

Name of the vocab file. This parameter is ignored if `setVocabFormat` was never called or was called with `VocabFormat.NONE`.

long preallocSize

Memory preallocation size (the number of unique words (tokens) in the training document). A value of 0 indicates that no memory preallocation is requested.

Throws

| Exception | Meaning |
|--------------------------|---|
| ZadeException | An error occurred in the zade_main manager process. |
| D2vException | An error occurred during the ibm-data2vec operation in the zade_main manager process. |
| IllegalArgumentException | One of the following reasons: <ul style="list-style-type: none">• The <code>setNumThreads</code> function was never called or was set to a value not greater than 0.• The <code>trainfile</code> parameter is a blank string, or the file does not exist, or is null.• The <code>modelFile</code> parameter is a blank string or is null.• The vocab file format is <code>VocabFormat.VOCAB_FILE_FMT_1</code> or <code>VocabFormat.VOCAB_FILE_FMT_2</code>, but <code>vocabFile</code> is a blank string or is null.• The <code>preallocSize</code> parameter is not a positive number. |

You can retrieve the error message by calling the exception's `getMessage` method.

Output files

When successful, `ibm-data2vec` produces a trained model file using the model format that was specified by the user on the `setModelFormat` call.

The function produces a log file, `ibm_data2vec-timestamp.log`, in the current working directory, which contains the execution log messages from the function.

Since

1.3.0

Legacy operation functions

callBase10Cluster

Description

Invokes the `base10Cluster` operation.

Format

```
@Deprecated
public void callBase10Cluster(int threads, String[] files, String outputPath)
```

Parameters

int threads

Number of threads to use for parallelism. This value must be greater than 0.

String[] files

List of input file names in the CSV format. You must specify at least one input file, and all input files must exist. An error is raised if 2 or more files would have generated the same output file name.

String outputPath

The directory in which the output files will be stored. This directory must already exist.

Output files

When successful, `base10Cluster` produces a `_output_minimums` file for every input file. Each output file indicates the cluster minimum values of the corresponding CSV file, sorted in increasing order. All output files are stored in the `outputPath` directory.

The function produces a log file, `base10cluster-timestamp.log`, in the current working directory, which contains the execution log messages from the function.

Error messages, if any, are sent to STDOUT.

callTrainDb2Vec

Description

Invokes the `ibm-data2vec` operation.

Format

```
@Deprecated
public void callTrainDb2Vec(int threads, String trainFile, String modelFilePrefix,
                           String format)
```

```
@Deprecated
public void callTrainDb2Vec(int threads, String trainFile, String modelFilePrefix,
                           String format, String vocabFile)
```

```
@Deprecated
public void callTrainDb2Vec(int threads, String trainFile, String modelFilePrefix,
                           String format, String vocabFile, int vocabFileFmt)
```

Parameters

int *threads*

Number of threads to use for parallelism. This value must be greater than 0.

String *trainfile*

Name of the input training document. An error is raised if *trainfile* is a blank string, or if the file does not exist or is null.

String *modelFilePrefix*

File name prefix of the output trained model. The eventual file name is *modelFilePrefix_zload.bin*. An error is raised if *modelFilePrefix* is a blank string or is null.

String *format*

Ignored.

String *vocabFile*

An optional parameter that specifies the name of the vocab file.

If *vocabFile* is specified but *vocabFileFmt* is not specified, format 1 of the vocab file is generated.

When *vocabFileFmt* is 1 or 2, an error is raised if *vocabFile* is a blank string or is null.

int *vocabFileFmt*

An optional parameter that specifies the format number of the vocab file. The supported formats are:

0

No vocab file is requested.

1

List of all unique tokens and their number of occurrences.

2

All entries in format 1, plus the number of unique tokens count for each column.

If omitted, format 1 of the vocab file is generated.

An error is raised if *vocabFileFmt* is not 0, 1, or 2.

Memory preallocation

The `ibm-data2vec` operation also reads from a caller-supplied file, *trainfile_prealloc.info* (where *trainfile* is the name of the input training document without the file extension), that resides in the same directory as the training document and contains information for supplemental memory preallocation. This file must contain a number (in plain text) in the first row that indicates the number of unique words (tokens) in the training document.

Output files

When successful, `ibm-data2vec` produces a trained model file, *modelFilePrefix_zload.bin*, using the Db2 zLoad format.

The function produces a log file, *ibm_data2vec-timestamp.log*, in the current working directory, which contains the execution log messages from the function.

Error messages, if any, are sent to STDOUT.

Chapter 9. Examples of using the IBM Z Artificial Intelligence Data Embedding Library APIs

The following examples illustrate how to code and use the IBM Z Artificial Intelligence Data Embedding Library APIs to develop applications.

- [“Example: Using the base10Cluster function” on page 165](#)
- [“Example: Using the ibm-dat2Vec function” on page 165](#)

Example: Using the base10Cluster function

The following 2 examples demonstrate how to invoke the base10Cluster function on a list of input CSV files. Two input files, `./test/resources/csv/V_SMINT.csv` and `./test/resources/csv/DEC150.csv` are processed, and two output files containing the cluster minimum values, `/tmp/DEC150_output_minimums` and `/tmp/V_SMINT_output_minimums` are stored in the `/tmp` directory.

```
import com.ibm.zos.zdnn.zade.jni.*;
import java.util.HashSet;
import java.util.Set;

public class ZadeTest {
    public static void main(String[] args){
        ZADE zade = new ZADE();
        zade.setNumThreads(4);

        Set<String> files = new HashSet<>();
        files.add("./test/resources/csv/V_SMINT.csv");
        files.add("./test/resources/csv/DEC150.csv");

        try {
            zade.base10Cluster(files, "/tmp");
        } catch (B10cException e) {
            // handle B10cException
        } catch (ZadeException e) {
            // handle ZadeException
        } catch (IllegalArgumentException e) {
            // handle IllegalArgumentException
        }
    }
}
```

Using the legacy API:

```
import com.ibm.zos.zdnn.zade.jni.ZADE;

public class ZadeTest {
    public static void main(String[] args){
        ZADE zade = new ZADE();

        String[] files = new String[2];

        files[0] = "./test/resources/csv/V_SMINT.csv";
        files[1] = "./test/resources/csv/DEC150.csv";

        zade.callBase10Cluster(4, files, "/tmp");
    }
}
```

Example: Using the ibm-dat2Vec function

The following two examples demonstrate how to invoke the ibm-data2Vec function on the input training text file. Both examples take as an input a training file, `Textified_Churn_Data.txt`, use 8 threads for

training, and store the model in the `model_zload.bin` file using the Db2 zLoad format. Finally, a list of unique tokens in the trained model (vocabulary) is stored in the `vocab` file using vocab file format 1.

```
import com.ibm.zos.zdnn.zade.jni.*;
import com.ibm.zos.zdnn.zade.jni.ZADE.ModelFormat;
import com.ibm.zos.zdnn.zade.jni.ZADE.VocabFormat;

public class ZadeTest {
    public static void main(String[] args){
        ZADE zade = new ZADE();
        zade.setNumThreads(8);
        zade.setModelFormat(ModelFormat.BINARYDB2_V110);
        zade.setVocabFormat(VocabFormat.VOCAB_FILE_FMT_1);

        try {
            zade.train("Textified_Churn_Data.txt", "model_zload.bin", "vocab");
        } catch (D2vException e) {
            // handle D2vException
        } catch (ZadeException e) {
            // handle ZadeException
        } catch (IllegalArgumentException e) {
            // handle IllegalArgumentException
        }
    }
}
```

Using the legacy API:

```
import com.ibm.zos.zdnn.zade.jni.ZADE;

public class ZadeTest {
    public static void main(String[] args){
        ZADE zade = new ZADE();

        zade.callTrainDb2Vec(8, "Textified_Churn_Data.txt", "model", "", "vocab", 1);
    }
}
```

Chapter 10. IBM Z Artificial Intelligence Data Embedding Library log files

The IBM Z Artificial Intelligence Data Embedding Library APIs use the following log files:

- base10Cluster log
- ibm-data2vec log

base10Cluster log

The base10Cluster function generates a log file, `base10cluster-timestamp.log`, to store the information regarding the clusters generated for every input file. The log file also reports total execution time taken by each thread. For example, the following log file, generated for the example shown in “Example: Using the base10Cluster function” on page 165 first reports the version of the base10cluster function. Then, for each participating thread, a timestamp is reported, and then for each data set being clustered, their cluster properties are reported, such as cluster min, max, median, average values, occupancy ratios, and, if an EMPTY cluster is generated, its details are also reported.

```
IBM base10 clustering zADE 1.3.0
IBM base10 clustering is using 2 threads to process 2 files and storing output in directory: /tmp
PID 131176 | 2024-04-18 00:04:18 base10_cluster() started clustering.
There are 4 clusters with total entries 50 Min value=-9.999124e+14 Max value=9.984965e+14
Cluster 0 has 15 entries Min value=-9.999124e+14 Max value=-5.200197e+14 Centroid=-7.919574e+14
  Median=-8.043848e+14 occupancy=30.000000 %
Cluster 1 has 16 entries Min value=-4.950732e+14 Max value=-1.156923e+13 Centroid=-2.807077e+14
  Median=-2.769354e+14 occupancy=32.000000 %
Cluster 2 has 9 entries Min value=5.178500e+13 Max value=5.272732e+14 Centroid=3.207427e+14
  Median=3.842086e+14 occupancy=18.000000 %
Cluster 3 has 10 entries Min value=5.788983e+14 Max value=9.984965e+14 Centroid=7.394704e+14
  Median=7.451904e+14 occupancy=20.000000 %
Using file /tmp/DEC150_output_minimums for storing the cluster minimums
There are 5 clusters with total entries 50 Min value=-7.200000e+75 Max value=3.276700e+04
Cluster EMPTY has 3 entries Min value=-7.200000e+75 occupancy=6.000000 %
Cluster 1 has 16 entries Min value=-3.276800e+04 Max value=-1.212000e+04 Centroid=-2.210338e+04
  Median=-2.168100e+04 occupancy=32.000000 %
Cluster 2 has 8 entries Min value=-1.201800e+04 Max value=-4.050000e+03 Centroid=-8.573250e+03
  Median=-6.782000e+03 occupancy=16.000000 %
Cluster 3 has 6 entries Min value=1.086000e+03 Max value=8.719000e+03 Centroid=4.911667e+03
  Median=7.625500e+03 occupancy=12.000000 %
Cluster 4 has 17 entries Min value=1.001000e+04 Max value=3.276700e+04 Centroid=2.053506e+04
  Median=2.011800e+04 occupancy=34.000000 %
Using file /tmp/V_SMINT_output_minimums for storing the cluster minimums
PID 131176 | 2024-04-18 00:04:18 base10_cluster() completed clustering in 0.2393 seconds.
Base10Clustering is terminated successfully.
```

ibm-data2vec log

The ibm-data2vec function generates a log file, `ibm_data2vec-timestamp.log`, that provides additional details about the execution of the ibm-data2vec function. For example, the following log file first reports the version of ibm-data2vec and the name of the training file. It then reports if the dependent library, `libzai0.so`, has been found. If the library is found, it verifies whether `libzai0.so` can use the SIMD features for accelerating BLAS computations using a BLAS library.

Important: If the BLAS library is not available, the computations will be done in a substantially slower scalar format.

The log file reports various properties of the input data set, such as total file size, number of words, number of words in the model vocabulary, name of the output model file, and so on. It also provides details about the execution times of three main stages of the training process:

1. Vocabulary building
2. Database model
3. Model storage

The log file also provides information about the number of threads used and reports incremental progress at 5% intervals. Finally, the log file reports memory consumption per stage, and whether there is any memory that is being used even after a particular stage has completed.

```
ibm-data2Vec (1.3.0 for zOS) starting execution using file Textified_Churn_Data.txt
ibm-data2Vec found the required library: libzaio.so. Proceeding with the training..
ibm-data2Vec will use following mode: CBLAS
User has not provided training chunk size. Using 1 GB chunk size for reading training file.
ibm-data2Vec starting execution using file Textified_Churn_Data.txt using 1 epoch.
67240538 | 2024-04-18 00:18:09 | Time elapsed learning vocab from train file = 0.811192s
Processed 154946 words in the training file. There are 7106 unique words in the vocabulary: Primary
Key words 7043 Internal words 63
Model training code will generate vectors for row-identifier (pk_id) or user-specified primary keys
67240538 | 2024-04-18 00:18:09 | Stage 1 completed. Time elapsed during file reading = 0.812383s
preallocation size 0 size 9096960
Training the database embedding (db2Vec) model using 8 CPU thread(s)
Epoch 0 learning rate Alpha=0.024846 Training Progress=5.00%
...
Epoch 0 learning rate Alpha=0.022001 Training Progress=100.00%
67240538 | 2024-04-18 00:19:08 | Stage 2 completed. Time elapsed training model = 59.408624s
Epoch 0 processed 154946 words in 7043 rows (total processed words 154946 total processed rows 7043).
After Epoch 0, unique words in vocab are 7106.
Epoch 0 allocated in total 181.27 MB:: 19.63 MB for pre_processing, 129.76 MB for vocab, and 31.88 MB
for model training

Saved format 1 vocab file: vocab
Saving ibm-data2Vec model to file model_zload.bin.
ibm-data2Vec storing vectors for the tokens in the vocab. Total vocab entries is 7107
ibm-data2Vec number of stored vocab vectors 7105 and number of not-stored vocab vectors 2
ibm-data2Vec stored 21 DB2_GENERATED_CENTER vectors. Number of columns in the schema is 21
Stored ibm-data2Vec model contains 7126 vectors in total (vocab vectors 7105 and CENTER column vectors 21).
ibm-data2Vec model successfully saved.
67240538 | 2024-04-18 00:19:08 | Stage 3 completed. Time elapsed writing output = 0.155985s
67240538 | 2024-04-18 00:19:08 | ibm-data2Vec execution completed. Total time elapsed = 60.388514s
ibm-data2Vec Total allocated memory 181.30 MB total freed memory 181.30 MB In-use memory 0.00 MB
ibm-data2Vec Per-stage memory usage:
(1) Stage DATA_LOADING_STAGE total-allocated=19.63 MB in-use=0.00 MB;
(2) Stage VOCAB_BUILDING_STAGE total-allocated=129.76 MB in-use=0.00 MB;
(3) Stage MODEL_TRAINING_STAGE total-allocated=31.88 MB in-use=0.00 MB;
(4) Stage MODEL_STORING_STAGE total-allocated=0.03 MB in-use=0.00 MB
```

Chapter 11. Troubleshooting the IBM Z Artificial Intelligence Data Embedding Library

Use the following information to troubleshoot potential issues you might encounter with the IBM Z Artificial Intelligence Data Embedding Library.

Could not find or load main class com.ibm.zos.zdnn.zade.jni.ZADE

This issue usually occurs when the **CLASSPATH** environment variable is not set properly. Ensure that `zade.jar` is included in the **CLASSPATH**, as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#).

zade (Not found in java.library.path)

This issue usually occurs for two reasons:

1. A 32-bit Java is used when invoking the available JNI APIs.
Ensure that a 64-bit Java is either included in the **PATH**, as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#), or explicitly called during invocation.
2. The **LIBPATH** environment variable is not setup properly. Ensure that the IBM Z Artificial Intelligence Data Embedding Library path is included in the **LIBPATH**, as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#).

Error on spawning

This issue usually occurs when the **PATH** environment variable is not set properly. Ensure that the `zade_main` path is included in the **PATH**, as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#).

Unable to open required library: libzaio.so

This issue usually occurs when the **LIBPATH** environment variable is not set properly. Ensure that the IBM Z Artificial Intelligence Optimization Library path is included in the **LIBPATH** as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#).

Using BLAS SIMD acceleration for training: False

This issue usually occurs when the **LIBPATH** environment variable is not set properly. Ensure that the IBM zOS OpenBLAS library path is included in the **LIBPATH**, as described in [“IBM Z Artificial Intelligence Data Embedding Library environment” on page 158](#).

Accessibility

Accessible publications for this product are offered through [IBM Documentation for z/OS \(www.ibm.com/docs/en/zos\)](http://www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS documentation see [How to Send Feedback to IBM](#) to leave documentation feedback.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Corporation
Site Counsel
2455 South Road*

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: [IBM Lifecycle Support for z/OS \(www.ibm.com/software/support/systemsz/lifecycle\)](http://www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and Trademark information \(www.ibm.com/legal/copytrade.shtml\)](http://www.ibm.com/legal/copytrade.shtml).

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

A

- accessibility
 - contact IBM [171](#)
- activation operations
 - Gaussian error linear unit [63](#)
 - hyperbolic tangent [58](#)
 - leaky rectified linear [56](#)
 - rectified linear [55](#)
 - sigmoid [58](#)
 - softmax [59](#)
 - softmax with mask [61](#)
- addition [46](#)
- allocate memory for zTensor
 - zdnm_allochelper_ztensor [34](#)
- API reference
 - IBM Z Artificial Intelligence Data Embedding Library [159](#)
 - IBM Z Artificial Intelligence Optimization Library [135](#)
- application environment, zDNN 3
- Artificial Intelligence Data Embedding library, IBM Z, *See* IBM Z Artificial Intelligence Data Embedding Library
- Artificial Intelligence Optimization library, IBM Z, *See* IBM Z Artificial Intelligence Optimization Library
- assistive technologies [171](#)
- average pool 2D [90](#)
- average vector [137](#)

B

- base10Cluster [161](#)
- batch norm [65](#)
- bias add [65](#)

C

- callBase10Cluster [163](#)
- callTrainDb2Vec [163](#)
- check CBLAS availability [136](#)
- check whether version runnable
 - zdnm_is_version_runnable [37](#)
- check zAIU availability [136](#)
- codes, status [8](#), [133](#)
- common
 - descriptor [6](#)
 - zTensor [5](#)
- contact
 - z/OS [171](#)
- convolution 2D [95](#)
- copy vector to new location [137](#)
- cosine distance [139](#)

D

- Data Embedding library, IBM Z Artificial Intelligence, *See* IBM Z Artificial Intelligence Data Embedding Library
- data formats [7](#)

- data layouts [6](#)
- data transformation
 - to original
 - zdnm_transform_origtensor [44](#)
 - to quantized zTensor
 - zdnm_transform_quantized_ztensor [42](#)
 - to zTensor
 - zdnm_transform_ztensor [39](#)
 - to zTensor with saturation
 - zdnm_transform_ztensor_with_saturation [41](#)
- data types [7](#), [8](#)
- data types, common [4](#)
- deallocate memory for zTensor
 - zdnm_free_ztensor_buffer [34](#)
- deep neural network library, IBM Z, *See* IBM Z Deep Neural Network Library
- descriptor, zTensor [6](#)
- Determine quantized zTensor
 - zdnm_is_quantized_ztensor [32](#)
- development
 - IBM Z Artificial Intelligence Optimization Library [132](#)
- division [49](#)
- dot product [139](#)

E

- element-wise operations
 - addition [46](#)
 - division [49](#)
 - exponential [52](#)
 - inverse square root [54](#)
 - maximum [51](#)
 - minimum [50](#)
 - multiplication [48](#)
 - natural logarithm [52](#)
 - square root [53](#)
 - subtraction [47](#)
- environment
 - IBM Z Artificial Intelligence Data Embedding Library [158](#)
 - IBM Z Artificial Intelligence Optimization Library [132](#)
- environment variables, zDNN runtime [10](#)
- environment, validating at run time [11](#)
- environment, zDNN 3
- examples, IBM Z Artificial Intelligence Data Embedding Library API usage [165](#)
- examples, IBM Z Artificial Intelligence Optimization Library API usage [151](#)
- examples, zDNN usage [99](#)
- execution
 - IBM Z Artificial Intelligence Optimization Library [132](#)
- exponential [52](#)

G

- Gated Recurrent Unit [86](#)
- Gaussian error linear unit [63](#)
- generate concatenated transformed tensor descriptor

generate concatenated transformed tensor descriptor (*continued*)

 zdnngenerate_transformed_desc_concatenated [28](#)

generate quantized transformed tensor descriptor

 zdnngenerate_quantized_transformed_desc [26](#)

generate transformed tensor descriptor

 zdnngenerate_transformed_desc [26](#)

get library version [136](#)

get maximum limit

 zdnng_get_max_limit [23](#)

get maximum runnable version

 zdnng_get_max_runnable_version [38](#)

get minimum limit

 zdnng_get_min_limit [24](#)

get size function

 zdnng_getsize_ztensor [22](#), [23](#)

GRU [86](#)

H

hyperbolic tangent [58](#)

I

IBM Telum II processor [3](#), [11](#)

IBM Telum processor [3](#), [11](#)

IBM Z Artificial Intelligence Data Embedding Library

 API reference [159](#)

 API usage examples [165](#)

 environment [158](#)

 overview [157](#)

 permissions [158](#)

IBM Z Artificial Intelligence Optimization Library

 API reference [135](#)

 API usage examples [151](#)

 development [132](#)

 environment [132](#)

 execution [132](#)

 overview [131](#)

IBM Z Deep Neural Network Library

 overview [3](#)

IBM Z Integrated Accelerator for AI [3](#), [11](#)

initialization

 zdnng_init [13](#)

initialize pre-transformed tensor descriptor

 zdnng_init_pre_transformed_desc [25](#)

initialize quantized zTensor

 zdnng_init_quantized_ztensor [30](#)

initialize quantized zTensor with memory allocate

 zdnng_init_quantized_ztensor_with_malloc [31](#)

initialize zTensor

 zdnng_init_ztensor [29](#)

initialize zTensor with memory allocate

 zdnng_init_ztensor_with_malloc [29](#)

inverse square root [54](#)

K

keyboard

 navigation [171](#)

 PF keys [171](#)

 shortcut keys [171](#)

layer normalization [68](#)

leaky rectified linear [56](#)

library, IBM Z deep neural network, See IBM Z Deep Neural Network Library

linear, leaky rectified [56](#)

linear, rectified [55](#)

logarithm, natural [52](#)

LSTM [82](#)

M

matmul [72](#)

matmul with broadcast [72](#)

matrix transpose [144](#)

matrix-matrix multiplication [143](#)

matrix-vector multiplication [143](#)

max pool 2D [93](#)

maximum [51](#)

mean reduce [64](#)

minimum [50](#)

moments [67](#)

multiplication [48](#)

N

natural logarithm [52](#)

navigation

 keyboard [171](#)

normalization [66](#)

normalization operations

 batch norma [65](#)

 bias add [65](#)

 layer normalization [68](#)

 mean reduce [64](#)

 moments [67](#)

 normalization [66](#)

 reduce [70](#)

O

operations

 activation

 Gaussian error linear unit [63](#)

 hyperbolic tangent [58](#)

 leaky rectified linear [56](#)

 rectified linear [55](#)

 sigmoid [58](#)

 softmax [59](#)

 softmax with mask [61](#)

 average pool 2D [90](#)

 convolution 2D [95](#)

 element-wise

 addition [46](#)

 division [49](#)

 exponential [52](#)

 inverse square root [54](#)

 maximum [51](#)

 minimum [50](#)

 multiplication [48](#)

 natural logarithm [52](#)

 square root [53](#)

- operations (*continued*)
 - element-wise (*continued*)
 - subtraction [47](#)
 - GRU [86](#)
 - LSTM [82](#)
 - matmul [72](#)
 - matmul with broadcast [72](#)
 - max pool 2D [93](#)
 - normalization
 - batch norm [65](#)
 - bias add [65](#)
 - layer normalization [68](#)
 - mean reduce [64](#)
 - moments [67](#)
 - normalization [66](#)
 - reduce [70](#)
 - zdnm_matmul_bcst_op [74](#)
 - zdnm_matmul_op [72](#)
 - zdnm_matmul_transpose_op [76](#)
 - zdnm_quantized_matmul_op [78](#)

Optimization library, IBM Z Artificial Intelligence, *See* IBM Z Artificial Intelligence Optimization Library

P

- permissions
 - IBM Z Artificial Intelligence Data Embedding Library [158](#)
- Prefetching clear [149](#)
- Prefetching execute [148](#)
- Prefetching initialize [147](#)

Q

- query functions
 - zdnm_get_library_version [20](#)
 - zdnm_get_library_version_str [21](#)
 - zdnm_get_max_for_dim [15](#)
 - zdnm_get_nnnpa_max_dim_idx_size [14](#)
 - zdnm_get_nnnpa_max_tensor_size [16](#)
 - zdnm_is_nnnpa_conversion_installed [20](#)
 - zdnm_is_nnnpa_datatype_installed [18](#)
 - zdnm_is_nnnpa_function_installed [17](#)
 - zdnm_is_nnnpa_installed [16](#)
 - zdnm_is_nnnpa_layout_fmt_installed [19](#)
 - zdnm_is_nnnpa_parmblk_fmt_installed [18](#)
 - zdnm_refresh_nnnpa_query_result [21](#)

R

- rectified linear [55](#)
- reduce [70](#)
- reference
 - IBM Z Artificial Intelligence Data Embedding Library [159](#)
 - IBM Z Artificial Intelligence Optimization Library [135](#)
- reset zTensor
 - zdnm_reset_ztensor [33](#)
- reshape zTensor
 - zdnm_reshape_ztensor [36](#)
- retrieve status message
 - zdnm_get_status_message [35](#)
- return codes
 - zAIO [133](#)
 - zDNN [8](#)

- run time, validating environment at [11](#)
- runtime environment variables, zDNN [10](#)

S

- semantic analogy [146](#)
- semantic average [138](#)
- semantic clustering [145](#)
- semantic similarity [144](#)
- setModelFormat [159](#)
- setNumThreads [159](#), [160](#)
- setVocabFormat [160](#)
- shortcut keys [171](#)
- sigmoid [58](#)
- softmax [59](#)
- softmax with mask [61](#)
- square root [53](#)
- statuses
 - zAIO [133](#)
 - zDNN [8](#)
- structures, common [4](#)
- subtraction [47](#)
- summary of changes [xv](#)
- support functions
 - allocate memory for zTensor
 - zdnm_allochelper_ztensor [34](#)
 - check whether version runnable
 - zdnm_is_version_runnable [37](#)
 - deallocate memory for zTensor
 - zdnm_free_ztensor_buffer [34](#)
 - Determine quantized zTensor
 - zdnm_is_quantized_ztensor [32](#)
 - generate concatenated transformed tensor descriptor
 - zdnm_generate_transformed_desc_concatenated [28](#)
 - generate quantized transformed tensor descriptor
 - zdnm_generate_quantized_transformed_desc [26](#)
 - generate transformed tensor descriptor
 - zdnm_generate_transformed_desc [26](#)
 - get maximum limit
 - zdnm_get_max_limit [23](#)
 - get maximum runnable version
 - zdnm_get_max_runnable_version [38](#)
 - get minimum limit
 - zdnm_get_min_limit [24](#)
 - get size function
 - zdnm_getsize_ztensor [22](#), [23](#)
 - initialization
 - zdnm_init [13](#)
 - initialize pre-transformed tensor descriptor
 - zdnm_init_pre_transformed_desc [25](#)
 - initialize quantized zTensor
 - zdnm_init_quantized_ztensor [30](#)
 - initialize quantized zTensor with memory allocate
 - zdnm_init_quantized_ztensor_with_malloc [31](#)
 - initialize zTensor
 - zdnm_init_ztensor [29](#)
 - initialize zTensor with memory allocate
 - zdnm_init_ztensor_with_malloc [29](#)
 - query functions
 - zdnm_get_library_version [20](#)
 - zdnm_get_library_version_str [21](#)
 - zdnm_get_max_for_dim [15](#)
 - zdnm_get_nnnpa_max_dim_idx_size [14](#)

support functions (*continued*)

query functions (*continued*)

- zdnnp_get_nnnpa_max_tensor_size [16](#)
- zdnnp_is_nnnpa_conversion_installed [20](#)
- zdnnp_is_nnnpa_datatype_installed [18](#)
- zdnnp_is_nnnpa_function_installed [17](#)
- zdnnp_is_nnnpa_installed [16](#)
- zdnnp_is_nnnpa_layout_fmt_installed [19](#)
- zdnnp_is_nnnpa_parmblk_fmt_installed [18](#)
- zdnnp_refresh_nnnpa_query_result [21](#)

reset zTensor

- zdnnp_reset_ztensor [33](#)

reshape zTensor

- zdnnp_reshape_ztensor [36](#)

retrieve status message

- zdnnp_get_status_message [35](#)

T

tangent, hyperbolic [58](#)

trademarks [176](#)

train [162](#)

transform to original

- zdnnp_transform_origtensor [44](#)

transform to quantized zTensor

- zdnnp_transform_quantized_ztensor [42](#)

transform to zTensor

- zdnnp_transform_ztensor [39](#)

transform to zTensor with saturation

- zdnnp_transform_ztensor_with_saturation [41](#)

transformation, data, *See* data transformation

typedefs [4](#)

types, common [4](#)

U

user interface

- ISPF [171](#)

- TSO/E [171](#)

V

vector absolute [141](#)

vector denormalization [141](#)

vector normalization [140](#)

vector scale [142](#)

version information, zDNN [4](#)

Z

zAIO, *See* IBM Z Artificial Intelligence Optimization Library

zAIO initialization [135](#)

zAIO return codes [133](#)

zaio_absolute [141](#)

zaio_averageVector [137](#)

zaio_cblasReady [136](#)

zaio_cosineDistance [139](#)

zaio_denormalize [141](#)

zaio_dotProduct [139](#)

zaio_getVersion [136](#)

zaio_Init [135](#)

zaio_matrixMatrix [143](#)

zaio_matrixVector [143](#)

zaio_normalize [140](#)

zaio_preFetching_Clear [149](#)

zaio_preFetching_Execute [148](#)

zaio_prefetching_Initialize [147](#)

zaio_semanticAnalogy [146](#)

zaio_semanticAverage [138](#)

zaio_semanticClustering [145](#)

zaio_semanticSimilarity [144](#)

zaio_transpose [144](#)

zaio_vectorCopy [137](#)

zaio_vectorScale [142](#)

zaio_zaiuReady [136](#)

zDNN tensor (zTensor) [5](#)

zDNN usage examples [99](#)

zdnnp_add [46](#)

zdnnp_allochelper_ztensor [34](#)

zdnnp_avgpool2d [90](#)

zdnnp_batchnorm [65](#)

zdnnp_conv2d [95](#)

zdnnp_div [49](#)

zdnnp_exp [52](#)

zdnnp_free_ztensor_buffer [34](#)

zdnnp_gelu [63](#)

zdnnp_generate_quantized_transformed_desc [26](#)

zdnnp_generate_transformed_desc [26](#)

zdnnp_generate_transformed_desc_concatenated [28](#)

zdnnp_get_library_version [20](#)

zdnnp_get_library_version_str [21](#)

zdnnp_get_max_for_dim [15](#)

zdnnp_get_max_limit [23](#)

zdnnp_get_max_runnable_version [38](#)

zdnnp_get_min_limit [24](#)

zdnnp_get_nnnpa_max_dim_idx_size [14](#)

zdnnp_get_nnnpa_max_tensor_size [16](#)

zdnnp_get_status_message [35](#)

zdnnp_getsize_ztensor [22, 23](#)

zdnnp_gru [86](#)

zdnnp_init [13](#)

zdnnp_init_pre_transformed_desc [25](#)

zdnnp_init_quantized_ztensor [30](#)

zdnnp_init_quantized_ztensor_with_malloc [31](#)

zdnnp_init_ztensor [29](#)

zdnnp_init_ztensor_with_malloc [29](#)

zdnnp_invsqrt [54](#)

zdnnp_is_nnnpa_conversion_installed [20](#)

zdnnp_is_nnnpa_datatype_installed [18](#)

zdnnp_is_nnnpa_function_installed [17](#)

zdnnp_is_nnnpa_installed [16](#)

zdnnp_is_nnnpa_layout_fmt_installed [19](#)

zdnnp_is_nnnpa_parmblk_fmt_installed [18](#)

zdnnp_is_quantized_ztensor [32](#)

zdnnp_is_version_runnable [37](#)

zdnnp_layernorm [68](#)

zdnnp_leaky_relu [56](#)

zdnnp_log [52](#)

zdnnp_lstm [82](#)

zdnnp_matmul_bcast_op [74](#)

zdnnp_matmul_op [72](#)

zdnnp_matmul_transpose_op [76](#)

zdnnp_max [51](#)

zdnnp_maxpool2d [93](#)

zdnnp_meanreduce2d [64](#)

zdnnp_min [50](#)

zdnnp_moments [67](#)

zdnm_mul [48](#)
zdnm_norm [66](#)
zdnm_quantized_matmul_op [78](#)
zdnm_reduce [70](#)
zdnm_refresh_nnnpa_query_result [21](#)
zdnm_relu [55](#)
zdnm_reset_ztensor [33](#)
zdnm_reshape_ztensor [36](#)
zdnm_sigmoid [58](#)
zdnm_softmax [59](#)
zdnm_softmax_mask [61](#)
zdnm_sqrt [53](#)
zdnm_sub [47](#)
zdnm_tanh [58](#)
zdnm_transform_origtensor [44](#)
zdnm_transform_quantized_ztensor [42](#)
zdnm_transform_ztensor [39](#)
zdnm_transform_ztensor_with_saturation [41](#)
zTensor [5](#)
zTensor descriptor [6](#)



Product Number: 5655-ZOS

SC31-5702-70

