z/OS
2.5

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

**SC31-5702-50**

**Note**

Before using this information and the product it supports, read the information in "Notices" on page 119.

This edition applies to Version 2 Release 5 of z/OS® (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2023-08-25

# Contents

# Figures

# Tables

# 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 send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

**Important:** If your comment regards a technical question or problem, see instead "If you have a technical problem" on page xiii.

Submit your feedback by using the appropriate method for your type of comment or question:

**Feedback on z/OS function**
If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

**Feedback on IBM® Documentation function**
If your comment or question is about the IBM Documentation functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Documentation Support at ibmdocs@us.ibm.com.

**Feedback on the z/OS product documentation and content**
If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: zDNN Programming Guide and Reference, SC31-5702-50
- The section title of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive authority to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

# If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

# 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 (www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/ibm-zos-doc-update-policy?OpenDocument).

## Summary of changes for z/OS 2.5

The following content is new, changed, or no longer included in z/OS 2.5.

### New

The following content is new.

**August 2023 refresh**

- Information about the IBM Z Artificial Intelligence Optimization (zAIO) library is added. (APAR OA64842)

### Changed

The following content is changed.

**January 2023 refresh**

- The Format is updated in "base10Cluster" on page 109. (APARs OA63952 (z/OS 2.5) and OA63951 (z/OS 2.4))
- The Format and Parameters are updated in "ibm-data2vec" on page 110. (APARs OA63952 (z/OS 2.5) and OA63951 (z/OS 2.4))

**September 2022 refresh**

- The version information is changed from 1.0.0 to 1.0.1 in "zDNN version information" on page 4.
- Updates are made in "Convolution 2D requirements" on page 57.
- Updates are made in "base10Cluster" on page 109.

### Deleted

The following content was deleted.

- 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.

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.
- IBM Z Integrated Accelerator for AI has complex requirements for the layout of data; these requirements arrange the tensor to enhance the performance characteristics of the operations. zDNN formats the tensor appropriately on behalf of the caller, and it does so using an optimized approach.
- For deep learning operations, the IBM Z Integrated Accelerator for AI requires the use of an internal data type (DLFLOAT16). This is a 2-byte data type, similar in concept to Brain float (BFLOAT16); that is, it is an AI optimized format that is used to speed up training and inference (from 4-byte formats) while minimizing the loss of accuracy at inference time.

The zDNN library provides a set of APIs that an exploiter can use to drive the desired request. zDNN is available on z/OS and Linux® on Z. The inclusion of Linux on 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.

#### IBM Z Integrated Accelerator for AI operation limits

The IBM Z Integrated Accelerator for AI 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 **zdnn_get_nnpa_max_dim_idx_size** function.
- Total number of bytes required for storing a transformed tensor must not exceed the value returned by the **zdnn_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 IBM Z Integrated Accelerator for AI.

### zDNN general

The zDNN deep learning library provides the standard IBM Z software interface to the IBM Z Integrated Accelerator for AI. This IBM-provided C library provides a set of functions that handle the data transformation requirements of the IBM Z Integrated Accelerator for AI 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 9.

# 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 "1.0.1"
#define ZDNN_VERNUM 0x010001 // 0x[major][minor][patch]
#define ZDNN_VER_MAJOR 1
#define ZDNN_VER_MINOR 0
#define ZDNN_VER_PATCH 1
```

**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 functionalities that are compatible with earlier versions are introduced to the API or if any API functionalities are 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 11.

# zDNN tensor

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

```
typedef struct zdnn_ztensor {
  zdnn_tensor_desc
      *pre_transformed_desc; // tensor's shape information before transformation
  zdnn_tensor_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[31];    // not currently used, should contain zeros.
} zdnn_ztensor;
```

*Figure 1. zDNN tensor structure*

## General zTensor requirements

Observe the following general requirements for a zTensor:

- Requirements for the *buffer* field:
  - The **zdnn_init_ztensor_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 **zdnn_getsize_ztensor** 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 **zdnn_init_ztensor** or **zdnn_init_ztensor_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 **zdnn_generate_transformed_desc_concatenated** function with the appropriate concatenation information.

  Do not use **zdnn_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 **zdnn_transform_ztensor** function, as normal.
- Follow the requirements in “General zTensor requirements” on page 5.

# zDNN tensor descriptor

The structure of a zDNN tensor descriptor is defined by the **zdnn_tensor_desc** structure, as shown in Figure 2 on page 6.

```
typedef struct zdnn_tensor_desc {
  zdnn_data_layouts layout; // data layout
  zdnn_data_formats format; // internal use only
  zdnn_data_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
} zdnn_tensor_desc;
```

*Figure 2. zDNN tensor descriptor*

### Programming notes

- Helper methods **`zdnn_init_pre_transformed_desc`** and **`zdnn_generate_transformed_desc`** or **`zdnn_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_NCHW layout expects dimensions such that `dim4 = N`, `dim3 = H`, `dim2 = W`, and `dim1 = C`.
  - A ZDNN_CNNK_HWCK layout expects dimensions such that `dim4 = W`, `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 **`zdnn_data_layouts`** enumeration, as shown in Figure 3 on page 6. 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 AIU data layout format 0
  ZDNN_FORMAT_4DKERNEL,  // tensor in AIU data layout format 1
} 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 7.

```
typedef enum zdnn_data_types {
  ZDNN_DLFLOAT16 = 0,    // 16-bit deep learning format
  BFLOAT = 253,          // Brain floating point format
  FP16 = 254,            // 16-bit IEEE-754 floating point format
  FP32 = 255,            // 32-bit IEEE-754 floating point format
} zdnn_data_types;
```

*Figure 5. zDNN data types*

## zDNN statuses

### Success status

Table 1 on page 7 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 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 IBM Z Integrated Accelerator for AI operation based on the data in the tensors. This usually indicates an overflow of the 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 8 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_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 | IBM Z Integrated Accelerator for AI 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. |
| ZDNN_MISALIGNED_SAVEAREA | 0x000C0015 | Function-specific save area address is not on a 4K-byte boundary. |

Table 5 on page 9 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`<br>`false` | When set to `true`, tensor integrity prechecks are run before issuing NNPA operations.<br>• Enabling precheck can impact performance.<br>• Enable precheck to debug issues that cause hardware exceptions that otherwise would result in abnormal program termination. |

| Variable name | Valid values | Description |
|---|---|---|
| **ZDNN_STATUS_DIAG** | *nnnnnnnn* (decimal)<br>0x*nnnnnnnn* (hexadecimal) | Prints or produces diagnostic information whenever the zDNN status code equals the specified value. Only one status value can be specified. |

Table 6. zDNN runtime environment variables (continued)

## Programming notes

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

# Chapter 2. zDNN API reference

The zDNN APIs are grouped into the following categories:


- "Support functions" on page 11
- "Data transformation" on page 25
- "Operations" on page 28
- "Convenience functions" on page 58


## Support functions

The zDNN support functions are:


- "Initialization (zdnn_init)" on page 11
- "Query functions" on page 12
- "Get size (zdnn_getsize_ztensor)" on page 17
- "Initialize pre-transformed tensor descriptor (zdnn_init_pre_transformed_desc)" on page 17
- "Generate transformed tensor descriptor (zdnn_generate_transformed_desc)" on page 18
- "Generate concatenated transformed tensor descriptor (zdnn_generate_transformed_desc_concatenated)" on page 18
- "Initialize zTensor (zdnn_init_ztensor)" on page 19
- "Initialize zTensor with memory allocate (zdnn_init_ztensor_with_malloc)" on page 20
- "Reset zTensor (zdnn_reset_ztensor)" on page 21
- "Allocate memory for zTensor (zdnn_allochelper_ztensor)" on page 21
- "Deallocate memory for zTensor (zdnn_free_ztensor_buffer)" on page 22
- "Retrieve status message for a status code (zdnn_get_status_message)" on page 22
- "Reshape zTensor (zdnn_reshape_ztensor)" on page 23
- "Check whether version is runnable (zdnn_is_version_runnable)" on page 24
- "Get maximum runnable version (zdnn_get_max_runnable_version)" on page 25


### Initialization (zdnn_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.

# Query functions

The zDNN query functions are:


- "Maximum dimension index size (zdnn_get_nnpa_max_dim_idx_size)" on page 12
- "Maximum tensor size (zdnn_get_nnpa_max_tensor_size)" on page 12
- "NNPA function availability (zdnn_is_nnpa_function_installed)" on page 13
- "Availability of parameter block formats (zdnn_is_nnpa_parmblk_fmt_installed)" on page 14
- "Availability of NNPA data types (zdnn_is_nnpa_datatype_installed)" on page 14
- "Availability of NNPA data layout formats (zdnn_is_nnpa_layout_fmt_installed)" on page 15
- "Library version (zdnn_get_library_version)" on page 16
- "Library version string (zdnn_get_library_version_str)" on page 16
- "Availability of NNPA data type format conversions (zdnn_is_nnpa_conversion_installed)" on page 15
- "In-memory query result (zdnn_refresh_nnpa_query_result)" on page 16


## Maximum dimension index size (zdnn_get_nnpa_max_dim_idx_size)

### Description

Retrieve the maximum dimension index size value 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.

## Maximum tensor size (zdnn_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 zdnn_get_nnpa_max_tensor_size();
```

### Parameters

None.

### Returns

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

## NNPA availability (zdnn_is_nnpa_installed)

### Description

Queries the hardware to determine if the NNPA and NNP-internal data type (DLFLOAT16) conversion instructions are installed.

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

### Format

```
bool zdnn_is_nnpa_installed();
```

### Parameters

None.

### Returns

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

## NNPA function availability (zdnn_is_nnpa_function_installed)

### Description

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

### Format

```
bool zdnn_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_QAF
NNPA_ADD
NNPA_SUB
NNPA_MUL
NNPA_DIV
NNPA_MIN
NNPA_MAX
NNPA_LOG
NNPA_EXP
NNPA_RELU
NNPA_TANH
NNPA_SIGMOID
NNPA_SOFTMAX
NNPA_BATCHNORMALIZATION
NNPA_MAXPOOL2D
NNPA_AVGPOOL2D
NNPA_LSTMACT
NNPA_GRUACT
NNPA_CONVOLUTION
NNPA_MATMUL_OP
NNPA_MATMUL_OP_BCAST23

### Returns

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

## Availability of parameter block formats (zdnn_is_nnpa_parmblk_fmt_installed)

### Description

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

### Format

```
bool zdnn_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

### Returns

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

## Availability of NNPA data types (zdnn_is_nnpa_datatype_installed)

### Description

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

### Format

```
bool zdnn_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 **zdnn_query_datatypes** enumeration.

QUERY_DATATYPE_INTERNAL1

## Returns

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

# Availability of NNPA data layout formats (zdnn_is_nnpa_layout_fmt_installed)

## Description

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

## Format

```
bool zdnn_query_is_nnpa_layout_fmt_installed(uint32_t layout_bitmask);
```

## Parameters

**uint32_t *layout_bitmask***
The OR of the requested layout bit masks as defined in the **zdnn_query_layoutfmts** enumeration.

QUERY_LAYOUTFMT_4DFEATURE
QUERY_LAYOUTFMT_4DKERNEL

## Returns

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

# Availability of NNPA data type format conversions (zdnn_is_nnpa_conversion_installed)

## Description

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

## Format

```
bool zdnn_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 **zdnn_query_bfpfmts** enumeration.

```
QUERY_BFPFMT_TINY (FP16)
QUERY_BFPFMT_SHORT (FP32/BFLOAT)
```

### Returns

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

## Library version (zdnn_get_library_version)

### Description

Retrieve the library version as a 32-bit, hexadecimal value (`0x00`*`MMmmpp`*), where:

***MM***
: The major level

***mm***
: The minor level

***pp***
: The patch level

### Format

```
uint32_t zdnn_get_library_version();
```

### Parameters

None.

### Returns

Returns the library version number in `0x00`*`MMmmpp`* format.

## 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.

## 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

```
zdnn_status zdnn_refresh_nnpa_query_result();
```

### Parameters

None.

### Programming notes

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

### Returns

One of the following zDNN status indicators:

ZDNN_OK
ZDNN_UNAVAILABLE_FUNCTION

## Get size (zdnn_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 (**zdnn_tensor_desc**) with transformed shape information.

### Format

```
uint64_t zdnn_getsize_ztensor(const zdnn_tensor_desc *tfrmd_desc);
```

### Parameters

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

### Returns

The required buffer size, in bytes.

## Initialize pre-transformed tensor descriptor (zdnn_init_pre_transformed_desc)

### Description

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

### Format

```
void zdnn_init_pre_transformed_desc(zdnn_data_layouts layout,
                                    zdnn_data_types type,
                                    zdnn_tensor_desc *pre_tfrmd_desc, ...);
```

## Parameters

**zdnn_data_layouts *layout***
The data layout.

**zdnn_data_types *type***
The data type.

**zdnn_tensor_desc **pre_tfrmd_desc***
The output **zdnn_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.

# Generate transformed tensor descriptor (zdnn_generate_transformed_desc)

## Description

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

## Format

```
zdnn_status zdnn_generate_transformed_desc(
    const zdnn_tensor_desc *pre_tfrmd_desc, zdnn_tensor_desc *tfrmd_desc);
```

## Parameters

**zdnn_tensor_desc **pre_tfrmd_desc***
The input tensor descriptor with pre-transformed shape information.

**zdnn_tensor_desc **tfrmd_desc***
The output **zdnn_tensor_desc** structure.

## Returns

One of the following zDNN status indicators:

| | |
|---|---|
| zDNN_OK | Success. |
| ZDNN_INVALID_LAYOUT | The pre-transformed *layout* value is not recognized or is a layout only used for concatenated tensors. |

# Generate concatenated transformed tensor descriptor (zdnn_generate_transformed_desc_concatenated)

## Description

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

## Format

```
zdnn_status zdnn_generate_transformed_desc_concatenated(
    const zdnn_tensor_desc *pre_tfrmd_desc,
    zdnn_concat_info info, zdnn_tensor_desc *tfrmd_desc);
```

## Parameters

**zdnn_tensor_desc *pre_tfrmd_desc**
The input tensor descriptor with pre-transformed shape information.

**zdnn_concat_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.

**zdnn_tensor_desc *tfrmd_desc**
The output **zdnn_tensor_desc** structure.

## Returns

One of the following zDNN status indicators:

| | |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_LAYOUT | The pre-transformed *layout* value is not recognized or is not supported for concatenated tensors. |
| ZDNN_INVALID_CONCAT_INFO | Invalid concatenation information. |

# Initialize zTensor (zdnn_init_ztensor)

## Description

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

## Format

```
void zdnn_init_ztensor(zdnn_tensor_desc *pre_tfrmd_desc,
                   zdnn_tensor_desc *tfrmd_desc, zdnn_ztensor *output);
```

### Parameters

**zdnn_tensor_desc *pre_tfrmd_desc***
Input tensor descriptor with pre-transformed shape information.

**zdnn_tensor_desc *tfrmd_desc***
Input tensor descriptor with transformed shape information.

**zdnn_ztensor *output***
The **`zdnn_ztensor`** struct being initialized.

### Returns

None.

## Initialize zTensor with memory allocate (zdnn_init_ztensor_with_malloc)

### Description

Provides the same functionality as **`zdnn_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

```
zdnn_status zdnn_init_ztensor_with_malloc(zdnn_tensor_desc *pre_tfrmd_desc,
                                          zdnn_tensor_desc *tfrmd_desc,
                                          zdnn_ztensor *output);
```

### Parameters

**zdnn_tensor_desc *pre_tfrmd_desc***
Input tensor descriptor with pre-transformed shape information.

**zdnn_tensor_desc *tfrmd_desc***
Input tensor descriptor with transformed shape information.

**zdnn_ztensor *output***
The **`zdnn_ztensor`** struct being initialized.

### Returns

One of the following **`zdnn_status`** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | *`tfrmd_desc->format`* is not recognized. |
| ZDNN_INVALID_TYPE | *`tfrmd_desc->type`* is not recognized or is a pre-transform type. |
| ZDNN_INVALID_SHAPE | If any of the following conditions are true:<br>• One of the *`tfrmd_desc->dim`* dimensions is 0.<br>• One of the *`tfrmd_desc->dim`* dimensions is greater than the size returned by **`zdnn_get_nnpa_max_dim_idx_size`**.<br>**Note:** Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 43 or “GRU input / output requirements” on page 47.<br>• The total number of *tfrmd_desc* elements is larger than the size returned by **`zdnn_get_nnpa_max_tensor_size`**. |
| ZDNN_ALLOCATION_FAILURE | Unable to allocate required memory on a 4K boundary. |

## 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_tensor *ztensor**
The **zdnn_ztensor** struct to be reset.

### Returns

None.

## Allocate memory for zTensor (zdnn_allochelper_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

```
zdnn_status zdnn_allochelper_ztensor(zdnn_ztensor *ztensor);
```

### Parameters

**zdnn_tensor *ztensor**
A **zdnn_ztensor** struct that contains the transformed shape information in the **transformed_desc** field.

### Returns

One of the following **zdnn_status** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | *ztensor->transformed_desc->format* is not recognized. |
| ZDNN_INVALID_TYPE | *ztensor->transformed_desc->type* is not recognized or is a pre-transform type. |

| Status | Meaning |
|---|---|
| ZDNN_INVALID_SHAPE | Any of the following reasons: <br>• One of the *ztensor->transformed_desc->dim* dimensions is 0. <br>• One of the *ztensor->transformed_desc->dim* dimensions is greater than the size returned by **zdnn_get_nnpa_max_dim_idx_size**. <br>**Note:** Concatenation dimensions have a smaller maximum size. See "LSTM input / output requirements" on page 43 or "GRU input / output requirements" on page 47. <br>• The total number of *transformed_desc* elements is larger than the value returned by **zdnn_get_nnpa_max_tensor_size**. |
| ZDNN_ALLOCATION_FAILURE | Unable to allocate the required memory on a 4K boundary. |

# Deallocate memory for zTensor (zdnn_free_ztensor_buffer)

## Description

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

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

## Format

```
zdnn_status zdnn_free_ztensor_buffer(const zdnn_ztensor *ztensor);
```

## Parameters

**zdnn_tensor *ztensor**
A **zdnn_ztensor** struct whose **buffer** field points to the allocated storage.

## Returns

One of the following **zdnn_status** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_BUFFER | The *ztensor->buffer* value is NULL. |

# Retrieve status message for a status code (zdnn_get_status_message)

## Description

Retrieve the status message for a status code.

## Format

```
const char *zdnn_get_status_message(zdnn_status status);
```

## Parameters

**zdnn_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.

# Reshape zTensor (zdnn_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

```
zdnn_status zdnn_reshape_ztensor(const zdnn_ztensor *src, zdnn_ztensor *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 **zdnn_status** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | Any of the following reasons:<br>• The dimensions (*transformed_desc->dim*) of *src* and *dest* total to different numbers of elements.<br>• One of the *dest->transformed_desc->dim* dimensions is 0.<br>• One of the *dest->transformed_desc->dim* dimensions is greater than the size returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>**Note:** Concatenation dimensions have a smaller maximum size. See "LSTM input / output requirements" on page 43 or "GRU input / output requirements" on page 47.<br>• The total number of *dest->transformed_desc->dim* elements is larger than the value returned by **zdnn_get_nnpa_max_tensor_size**. |

| Status | Meaning |
| --- | --- |
| ZDNN_INVALID_LAYOUT | Any of the following reasons:<br>• The value of *transformed_desc->layout* for *src* and *dest*) are not the same.<br>• The value of *transformed_desc->layout* is neither ZDNN_NHWC nor ZDNN_HWCK.<br>• The value of *src->pre_transformed_desc->layout* is not recognized or is not a valid pre-transformation layout.<br>• The value of *dest->pre_transformed_desc->layout* is not recognized or is not a valid pre-transformation layout. |
| ZDNN_INVALID_STATE | Any of the following reasons:<br>• The source tensor (*src*) is not already transformed.<br>• The destination tensor (*dest*) is already transformed. |
| ZDNN_INVALID_FORMAT | The format of the source tensor *src->transformed_desc->format* is not ZDNN_FORMAT_4DFEATURE. |
| ZDNN_INVALID_TYPE | Any of the following reasons:<br>• The value of *src->pre_transformed_desc->type* is not recognized or is a transformed type.<br>• The value of *dest->pre_transformed_desc->type* is not recognized or is a transformed type.<br>• The value of *dest->transformed_desc->type* is not recognized or is a pre-transformation type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons:<br>• The *src->buffer* field is NULL.<br>• The *src->buffer*) storage does not start on a 4K boundary.<br>• The *dest->buffer* field is NULL.<br>• The *dest->buffer*) storage does not start on a 4K boundary.<br>• The *dest->buffer_size* value is too small to hold the transformed values. |
| ZDNN_CONVERT_FAILURE | Values failed to untransform or transform. |

# Check whether version is runnable (zdnn_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 zdnn_is_version_runnable(uint32_t ver_num);
```

## Parameters

**uint32_t *ver_num***
: The zDNN version number from the application in `0x00`*[major][minor][patch]* form. Typically, this is **ZDNN_VERNUM** that was used to compile the application.

## Returns

The function returns `true` or `false`.

## Get maximum runnable version (zdnn_get_max_runnable_version)

### Description

Return the maximum zDNN version number that the current hardware and installed zDNN library can run together. The returned value means that the current runtime environment fully supports zDNN APIs of that *[major]*.*[minor]* version and earlier.

### Format

```
uint32_t zdnn_get_max_runnable_version();
```

### Parameters

None.

### Returns

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

# Data transformation

The IBM Z Integrated Accelerator for AI 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 the internal format (DLFLOAT16) supported by the IBM Z Integrated Accelerator for AI. zDNN provides the following two conversion functions:

**`zdnn_transform_ztensor`**
: Transforms the input tensor and converts the input data to the format required by the IBM Z Integrated Accelerator for AI. The resulting transformed zTensor can be reused as many times as necessary.

  See "Transform to zTensor (zdnn_transform_ztensor)" on page 25 for details about transforming an input tensor to the internal format.

**`zdnn_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 (zdnn_transform_origtensor)" on page 27 for details about transforming an input zTensor to the original format.

## Transform to zTensor (zdnn_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 (**`zdnn_tensor_desc`** fields). Once converted, a **`zdnn_ztensor`** should only be manipulated by zDNN API functions.

### Format

```
zdnn_status zdnn_transform_ztensor(zdnn_ztensor ztensor, ...);
```

## Parameters

**zdnn_ztensor *tensor***
The input **`zdnn_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 must be preallocated by the caller (directly or by calling the zDNN allocation helper function) and the *buffer* field must point to the storage.

***... (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 **`zdnn_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 **`zdnn_status`** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_FORMAT | `zdnn_ztensor->`*`transformed_desc`*`->`*`format`* is not recognized. |
| ZDNN_INVALID_LAYOUT | Any of the following reasons:<br>• `zdnn_ztensor->`*`pre_transformed_desc`*`->`*`layout`* is not recognized or is not a valid pre-transform layout.<br>• `zdnn_ztensor->`*`transformed_desc`*`->`*`layout`* is not recognized or is not a valid transformed layout. |
| ZDNN_INVALID_TYPE | Any of the following reasons:<br>• `zdnn_ztensor->`*`pre_transformed_desc`*`->`*`type`* is not recognized or is a transformed type.<br>• `zdnn_ztensor->`*`transformed_desc`*`->`*`type`* is not recognized or is a pre-transform type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons:<br>• The *buffer* field is `NULL`.<br>• The start of the *buffer* storage is not on a 4K boundary.<br>• The *buffer_size* value is too small to hold transformed values. |
| ZDNN_INVALID_SHAPE | Any of the following reasons:<br>• One of the `zdnn_ztensor->`*`transformed_desc`*`->`*`dim`* dimensions is 0.<br>• One of the `zdnn_ztensor->`*`transformed_desc`*`->`*`dim`* dimensions is greater than the size returned by **`zdnn_get_nnpa_max_dim_idx_size`**.<br>**Note:** Concatenation dimensions have a smaller maximum size. See “LSTM input / output requirements” on page 43 or “GRU input / output requirements” on page 47.<br>• The total number of *transformed_desc* elements is larger than the value returned by **`zdnn_get_nnpa_max_tensor_size`**. |
| ZDNN_INVALID_STATE | Tensor is already transformed. |
| ZDNN_CONVERT_FAILURE | Values failed to transform. |

# Transform to original (zdnn_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

```
zdnn_status zdnn_transform_origtensor(const zdnn_ztensor *ztensor, void *out_buf);
```

## Parameters

**zdnn_ztensor **ztensor***
: The input **`zdnn_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 **`zdnn_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 **`zdnn_status`** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_UNSUPPORTED_TYPE | (SIM only) |
| ZDNN_INVALID_FORMAT | The *`ztensor->transformed_desc->format`* value is not `ZDNN_FORMAT_4DFEATURE`. |
| ZDNN_INVALID_LAYOUT | Any of the following reasons:<br>• The `zdnn_ztensor->`*`pre_transformed_desc->layout`* value is not recognized or is not a valid pre-transform layout.<br>• The `zdnn_ztensor->`*`transformed_desc->layout`* value is not recognized or is not a valid transformed layout required by this function. |
| ZDNN_INVALID_TYPE | Any of the following reasons:<br>• The *`ztensor->pre_transformed_desc->type`* value is not recognized or is a transformed type.<br>• The *`ztensor->transformed_desc->type`* value is not recognized or is a pre-transform type. |
| ZDNN_INVALID_BUFFER | Any of the following reasons:<br>• The *`ztensor->buffer`* field is `NULL`.<br>• The start of the *`ztensor->buffer`* storage is not on a 4K boundary. |
| ZDNN_INVALID_STATE | The *ztensor* is not transformed. |
| ZDNN_CONVERT_FAILURE | Values failed to untransform. |

# Operations

The zDNN operations are organized into the following categories:



## Element-wise operations

The zDNN element-wise operations are:



### 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 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

## Framework examples

- TensorFlow addition (https://www.tensorflow.org/api_docs/python/tf/math/add)
- ONNX addition (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Add)

# Subtraction (zdnn_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

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

## Parameters

**zdnn_ztensor *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.

**zdnn_ztensor *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.

**zdnn_ztensor *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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

## Framework examples

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

- ONNX subtraction (https://github.com/onnx/onnx/blob/master/docs/Operators.md#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 7):

ZDNN_OK
“Warning statuses” on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
“Hardware statuses” on page 8

### Framework examples

- TensorFlow multiplication (https://www.tensorflow.org/api_docs/python/tf/math/multiply)
- ONNX multiplication (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Mul)

## Division (zdnn_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

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

## Parameters

**zdnn_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.

**zdnn_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.

**zdnn_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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

## Framework examples

- TensorFlow division (https://www.tensorflow.org/api_docs/python/tf/math/divide)
- ONNX division (https://github.com/onnx/onnx/blob/master/docs/Operators.md#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 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow minimum (https://www.tensorflow.org/api_docs/python/tf/math/minimum)
- ONNX minimum (https://github.com/onnx/onnx/blob/master/docs/Operators.md#min)

## Maximum (zdnn_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

```
zdnn_status zdnn_max(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 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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT

"Hardware statuses" on page 8

### Framework examples

- TensorFlow maximum (https://www.tensorflow.org/api_docs/python/tf/math/maximum)
- ONNX maximum (https://github.com/onnx/onnx/blob/master/docs/Operators.md#max)

## Natural logarithm (zdnn_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

```
zdnn_status zdnn_log(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 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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow natural logarithm (https://www.tensorflow.org/api_docs/python/tf/math/log)
- ONNX natural logarithm (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Log)

## Exponential (zdnn_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

```
zdnn_status zdnn_exp(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 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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow exponential (https://www.tensorflow.org/api_docs/python/tf/math/exp)
- ONNX exponential (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Exp)

# Activation operations

The zDNN activation operations are:



## 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, clipping_value)`.

### Format

```
zdnn_status zdnn_relu(const zdnn_ztensor *input, const void *clipping_value,
                      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.

**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.

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

### Returns

One of the following **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
ZDNN_INVALID_CLIPPING_VALUE
"Hardware statuses" on page 8

### Framework examples

- TensorFlow rectified linear (https://www.tensorflow.org/api_docs/python/tf/math/relu)
- ONNX rectified linear (https://github.com/onnx/onnx/blob/master/docs/Operators.md#relu)

## 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 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow hyperbolic tangent (https://www.tensorflow.org/api_docs/python/tf/math/tanh)
- ONNX hyperbolic tangent (https://github.com/onnx/onnx/blob/master/docs/Operators.md#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

```
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 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 **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow sigmoid (https://www.tensorflow.org/api_docs/python/tf/math/sigmoid)
- ONNX sigmoid (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Sigmoid)

## Softmax (zdnn_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

```
zdnn_status zdnn_softmax(const zdnn_ztensor *input, void *save_area,
                         zdnn_softmax_act act_func, zdnn_ztensor *output);
```

## Parameters

**zdnn_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.

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

**zdnn_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 **`zdnn_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 1x1x16 and use the same original data array prior to the **`zdnn_transform_ztensor`** call. After transformation, such a tensor would be valid for the **`zdnn_softmax`** operation.

  – As a further example, the 4D 2x2x2x2 tensor could be processed as 2 batches of 8 vectors using a ZDNN_3DS layout *pre_transformed_desc* with dimensions 1x2x8.

## Returns

One of the following **`zdnn_status`** indicators (as described in "zDNN statuses" on page 7):

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 8
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.

## Framework examples

- TensorFlow softmax (https://www.tensorflow.org/api_docs/python/tf/math/softmax)
- ONNX softmax (https://github.com/onnx/onnx/blob/master/docs/Operators.md#Softmax)

# Normalization operations

The zDNN normalization operations are:

- "Mean reduce (zdnn_meanreduce2d)" on page 38
- "Batch norm (zdnn_batchnorm)" on page 39

## 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 7):

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 8
ZDNN_FUNC_RC_F001 — The input tensor has a *Height* or *Width* dimension greater than allowed for **zdnn_meanreduce2d**.

### Framework examples

- TensorFlow 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://github.com/onnx/onnx/blob/master/docs/Operators.md#ReduceMean)

## Batch norm (zdnn_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:

```
ouptut = input_b × input_a + input_c
```

where *input_b* is a precomputed elementwise divide of scale and variance tensors, and *input_c* is a precomputed elementwise multiply of `(-1) * mean` and `input_b + input bias` tensors.

### Format

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

### Parameters

**zdnn_ztensor *input_a**
: Must be a 4D, ZDNN_NHWC input tensor. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_ztensor *input_b**
: Must be a 1D tensor. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_ztensor *input_c**
: Must be a 1D tensor. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_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 39. 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 7):

ZDNN_OK
"Warning statuses" on page 8
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

### Framework examples

- TensorFlow Batchnorm (https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization)
- ONNX Batchnorm (https://github.com/onnx/onnx/blob/master/docs/Operators.md#BatchNormalization)

# Matmul and matmul with broadcast

The zDNN matmul and matmul with broadcast functions are:

- "zdnn_matmul_op" on page 40

- "zdnn_matmul_bcast_op" on page 41

## zdnn_matmul_op

### Description

Given three input zDNN tensors, (*input_a*, *input_b*, and *input_c*, determine the matrix multiplication of `input_a × 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.

Operations:

- 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

```
zdnn_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 40 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 | *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) |

### 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 "Input and output requirements for matmul tensors" on page 40.

**zdnn_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 matmul tensors" on page 40.

**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 "Input and output requirements for matmul tensors" on page 40.

**zdnn_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
```

**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 "Input and output requirements for matmul tensors" on page 40.

## Returns

One of the following **`zdnn_status`** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8
ZDNN_FUNC_RC_F000 — Invalid *op_type* value

## Framework examples

- TensorFlow matmul (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/mat-mul)
- ONNX matmul (https://github.com/onnx/onnx/blob/master/docs/Operators.md#MatMul)

# zdnn_matmul_bcast_op

## Description

Given three input zDNN tensors, *input_a*, *input_b*, and *input_c*, determine the matrix multiplication of *input_a × 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.

Operations:

- Addition

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

```
zdnn_status zdnn_matmul_bcast_op(const zdnn_ztensor *input_a,
                                 const zdnn_ztensor *input_b,
                                 const zdnn_ztensor *input_c,
                                 zdnn_matmul_bcast_ops op_type, zdnn_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 42 lists the requirements for *pre_transformed_desc* and shape for stacked and unstacked tensors.

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

| ***input_a*** | ***input_b*** | ***input_c*** | ***output*** |
|---|---|---|---|
| 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 8 on page 42.

**zdnn_ztensor **input_b***
: An input tensor with the second matrix for multiplication. The same single *input_b* matrix is broadcast and used as the multiplier for each stack dimension of *input_a*. The pre-transformed shape and layout must be as described in Table 8 on page 42.

**zdnn_ztensor **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 42.

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

    `MATMUL_BCAST_OP_ADDITION`

**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 8 on page 42.

## Programming notes

The **`zdnn_matmul_bcast_op`** function only supports a value of `MATMUL_BCAST_OP_ADDITION` for *op_type*. Any other values are ignored and may not operate compatibly in the future.

## Returns

One of the following **`zdnn_status`** indicators (as described in "zDNN statuses" on page 7):

ZDNN_OK
ZDNN_INVALID_SHAPE
ZDNN_INVALID_TYPE
ZDNN_INVALID_FORMAT
"Hardware statuses" on page 8

## Framework examples

- TensorFlow matmul (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/mat-mul)
- ONNX matmul (https://github.com/onnx/onnx/blob/master/docs/Operators.md#MatMul)

# LSTM (zdnn_lstm)

## Description

Implements the Long-Short Term Memory[1] (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 60.

## LSTM input / output requirements

Any *num_hidden* dimension must be less than or equal to 8192 elements.

## Parameters

**zdnn_ztensor *input***
: An input tensor with shape (num_timesteps, num_batches, 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 (num_dirs, num_batches, 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 43.

**zdnn_ztensor *c0***
: An input tensor that contains the initial cell state with shape (num_dirs, num_batches, 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 43.

**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 (num_dirs, num_features, 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 43.

**zdnn_ztensor **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 **`zdnn_concat_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 43.

**zdnn_ztensor **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 **`zdnn_concat_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 43.

**zdnn_ztensor **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 **`zdnn_concat_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 43.

**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 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.

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

**zdnn_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 “LSTM input / output requirements” on page 43.

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.

**zdnn_ztensor **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 43.

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 9. Summary of zdnn_lstm parameters*

| Parameter | Pre-transformed layout | Pre-transformed shape | To create transformed descriptor |
|---|---|---|---|
| **input** | ZDNN_3DS | (num_timesteps, num_batches, num_features) | **zdnn_generate_transformed_desc** |
| **h0** | ZDNN_3DS | (num_dirs, num_batches, num_hidden) | **zdnn_generate_transformed_desc** |
| **c0** | ZDNN_3DS | (num_dirs, num_batches, num_hidden) | **zdnn_generate_transformed_desc** |
| **weights** | ZDNN_3DS | (num_dirs, num_features, num_hidden) | **zdnn_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) | **zdnn_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) | **zdnn_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) | **zdnn_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) | **zdnn_generate_transformed_desc** |
| **cf_output** | ZDNN_4DS | (1, num_dirs, num_batches, num_hidden) | **zdnn_generate_transformed_desc** |

## Returns

One of the following **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

| 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:<br>• The *hn_output* timesteps dimension must be 1 or the same size as the *input* timestep dimension.<br>• All tensors with a direction dimension have the same direction dimension size.<br>• The *input* timestep dimension must be greater than or equal to 1.<br>• Other general shape violations (such as, exceeds MDIS, and others) |
| ZDNN_INVALID_DIRECTION | The specified *direction* parameter is not a recognized **lstm_gru_direction** direction. |
| ZDNN_ALLOCATION _FAILURE | A preallocated *work_area* was not specified and internal allocation for the required memory failed. |

| Status | Meaning |
|---|---|
| Hardware statuses | See "Hardware statuses" on page 8. |

### Framework examples

- TensorFlow LSTM (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell)
- ONNX LSTM (https://github.com/onnx/onnx/blob/master/docs/Operators.md#LSTM)

## GRU (zdnn_gru)

### Description

Implements Gated Recurrent Unit[2] (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

```
zdnn_status zdnn_gru(const zdnn_ztensor *input, const zdnn_ztensor *h0,
                     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);
```

For an example of the calling syntax, see "Example: Calling the zdnn_gru API (forward)" on page 73.

### GRU input / output requirements

Any *num_hidden* dimension must be less than or equal to 10,880 elements.

### Parameters

**zdnn_ztensor *input**
: An input tensor with shape (num_timesteps, num_batches, 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 (num_dirs, num_batches, 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 "GRU input / output requirements" on page 47.

**zdnn_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 **zdnn_concat_info** field must have the following flags turned on:

[2] Kyunghyun Cho, 2014.

- 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 47.

**zdnn_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 **zdnn_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 47.

**zdnn_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 **zdnn_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 47.

**zdnn_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 **zdnn_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 47.

**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 num_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.

```
zdnn_tensor_desc desc;
desc.dim4 = (3 * timestep) + 5;
desc.dim3 = 1;
desc.dim2 = batch;
desc.dim1 = hidden_state_size;
uint64_t work_area_size = zdnn_getsize_ztensor(&desc);
```

**zdnn_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 47.

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 10. Summary of zdnn_gru parameters*

| Parameter | Pre-transformed layout | Pre-transformed shape | To create transformed descriptor |
|---|---|---|---|
| **input** | ZDNN_3DS | (num_timesteps, num_batches, num_features) | **zdnn_generate_transformed_desc** |
| **h0** | ZDNN_3DS | (num_dirs, num_batches, num_hidden) | **zdnn_generate_transformed_desc** |
| **c0** | ZDNN_3DS | (num_dirs, num_batches, num_hidden) | **zdnn_generate_transformed_desc** |
| **weights** | ZDNN_3DS | (num_dirs, num_features, num_hidden) | **zdnn_generate_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) | **zdnn_generate_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) | **zdnn_generate_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) | **zdnn_generate_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) | **zdnn_generate_transformed_desc** |

## Returns

One of the following **zdnn_status** indicators (as described in "zDNN statuses" on page 7):

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_TYPE | See "General failing statuses" on page 8. |
| ZDNN_INVALID_FORMAT | See "General failing statuses" on page 8. |
| ZDNN_INVALID_SHAPE | Any of the following reasons: • The *hn_output* timesteps dimension must be 1 or the same size as the *input* timestep dimension. • All tensors with a direction dimension have the same direction dimension size. • The *input* 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 *direction* parameter is not a recognized **lstm_gru_direction** direction. |
| ZDNN_ALLOCATION _FAILURE | A preallocated *work_area* was not specified and internal allocation for the required memory failed. |
| Hardware statuses | See "Hardware statuses" on page 8. |

## Framework examples

- TensorFlow GRU (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRUCell)
- ONNX GRU (https://github.com/onnx/onnx/blob/master/docs/Operators.md#GRU)

## Average pool 2D (zdnn_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

```
zdnn_status zdnn_avgpool2d(const zdnn_ztensor *input,
                           zdnn_pool_padding padding_type,
                           uint32_t kernel_height, uint32_t kernel_width,
                           uint32_t stride_height, uint32_t stride_width,
                           zdnn_ztensor *output);
```

### Parameters

**zdnn_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 52 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 52 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).

***kernel_height***

The size of the kernel window that passes over the height dimension of the input tensor. See "AvgPool2D parameter restrictions" on page 52 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 52 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.

**zdnn_ztensor *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 52 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:

      ```
      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_tensor* 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 | • Shape of the input or output tensor is invalid based on the given kernel and stride parameters.<br>• Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | See "General failing statuses" on page 8. |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_STRIDE_PADDING | |
| ZDNN_INVALID_STRIDES | One stride was non-zero, but not the other. |

| Status | Meaning |
|---|---|
| Hardware statuses | See “Hardware statuses” on page 8.<br>In addition, the ZDNN_EXCEEDS_MDIS hardware status also occurs if any of the following conditions occur:<br>• The *stride_height* value is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *stride_width* value is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *kernel_height* value is 0 or is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *kernel_width* value is 0 or is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**. |
| ZDNN_FUNC_RC_F000 | Invalid *padding_type* value. |
| ZDNN_FUNC_RC_F001 | *stride_height* = 0 and *stride_width* = 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the *kernel_height* and *kernel_width* parameters. |
| ZDNN_FUNC_RC_F002 | *stride_height* > 0 and *stride_width* > 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the *kernel_height* and *kernel_width* parameters. |
| ZDNN_FUNC_RC_F003 | *stride_height* > 0 and *stride_width* > 0, but a stride parameter is greater than allowed. See the earlier descriptions of the *stride_height* and *stride_width* parameters. |
| ZDNN_FUNC_RC_F004 | *stride_height* > 0 and *stride_width* > 0, but either the height or weight dimension of the input tensor is greater than 1024. |

### Framework examples

- TensorFlow AvgPool (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/avg-pool)
- ONNX AvgPool (https://github.com/onnx/onnx/blob/master/docs/Operators.md#AveragePool)

## Max pool 2D (zdnn_maxpool2d)

### 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

```
zdnn_status zdnn_maxpool2d(const zdnn_ztensor *input,
                           zdnn_pool_padding padding_type,
                           uint32_t kernel_height, uint32_t kernel_width,
                           uint32_t stride_height, uint32_t stride_width,
                           zdnn_ztensor *output);
```

### Parameters

**zdnn_ztensor *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 54 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 54 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).

***kernel_height***
Size of the kernel window that passes over the height dimension of the input tensor. See "MaxPool2D parameter restrictions" on page 54 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 54 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.

**zdnn_ztensor **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 54 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 | • Shape of the input or output tensor is invalid based on the given kernel and stride parameters.<br>• Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | See "General failing statuses" on page 8. |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_STRIDE_PADDING | |
| ZDNN_INVALID_STRIDES | One stride is non-zero, but not the other. |
| Hardware statuses | See "Hardware statuses" on page 8.<br>In addition, the ZDNN_EXCEEDS_MDIS hardware status also occurs if any of the following conditions occur:<br>• The *stride_height* value is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *stride_width* value is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *kernel_height* value is 0 or is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**.<br>• The *kernel_width* value is 0 or is larger than the value returned by **zdnn_get_nnpa_max_dim_idx_size**. |
| ZDNN_FUNC_RC_F000 | Invalid *padding_type* value. |
| ZDNN_FUNC_RC_F001 | *stride_height* = 0 and *stride_width* = 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the *kernel_height* and *kernel_width* parameters. |
| ZDNN_FUNC_RC_F002 | *stride_height* > 0 and *stride_width* > 0, but a kernel parameter is greater than allowed. See the earlier descriptions of the *kernel_height* and *kernel_width* parameters. |
| ZDNN_FUNC_RC_F003 | *stride_height* > 0 and *stride_width* > 0, but a stride parameter is greater than allowed. See the earlier descriptions of the *stride_height* and *stride_width* parameters. |
| ZDNN_FUNC_RC_F004 | *stride_height* > 0 and *stride_width* > 0, but either the height or weight dimension of the input tensor is greater than 1024. |

## Framework examples

- TensorFlow MaxPool (https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/max-pool)
- ONNX MaxPool (https://github.com/onnx/onnx/blob/master/docs/Operators.md#MaxPool)

# Convolution 2D (zdnn_conv2d)

## Description

Perform 2D convolution over an input tensor in zDNN transformed format, as follows:

1. The *input* tensor in 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(intermediate_result, clipping_value)`.
5. The result is stored into the provided output zDNN tensor.

## Format

```
zdnn_status zdnn_fusedconv2d(const zdnn_ztensor *input,
                             const zdnn_ztensor *kernel,
                             const zdnn_ztensor *bias,
                             zdnn_pool_padding padding_type,
                             uint32_t stride_height, uint32_t stride_width,
                             zdnn_conv2d_act act_func,
                             const void *clipping_value, zdnn_ztensor *output);
```

## Parameters

**zdnn_ztensor *input***
: An input tensor with original values to be downsampled in the output tensor. See "Convolution 2D requirements" on page 57 for requirements. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_ztensor *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 57 for additional requirements. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_ztensor *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 57 for additional requirements. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

**zdnn_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).

**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 57 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 57 for requirements.

**zdnn_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`.

**zdnn_ztensor *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 57 for additional requirements. The tensor must satisfy the requirements in "General zTensor requirements" on page 5.

## Convolution 2D requirements

Table 11 on page 57 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 11. 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 *kernel_height* and *kernel_width* must be ≤ 64. | – | *height_out* = ceil(*height_in* ÷ *stride_height*)<br>*width_out* = ceil(*width_in* ÷ *stride_width*) |
| Strides > 0 and ≤ 13 and VALID padding | *height_in* must be ≥ *kernel_height*.<br>*width_in* must be ≥ *kernel_width*. | Both *kernel_height* and *kernel_width* must be ≤ 64. | – | *height_out* = ceil((*height_in* − *kernel_height* + 1) ÷ *stride_height*)<br>*width_out* = ceil((*width_in* − *kernel_width* + 1) ÷ *stride_width*) |
| Strides = 0 and VALID padding | *height_in* must be = *kernel_height*.<br>width_in must be = kernel_width | Both *kernel_height* and *kernel_width* must be ≤ 448. | – | Both *height_out* and *width_out* must be 1. |

## Returns

One of the following **`zdnn_status`** indicators:

| Status | Meaning |
|---|---|
| ZDNN_OK | Success. |
| ZDNN_INVALID_SHAPE | • Shape of the input or output tensor is invalid based on the given kernel and stride parameters.<br>• Other general shape violations (such as, exceeds MDIS, and others). |
| ZDNN_INVALID_TYPE | See "General failing statuses" on page 8. |
| ZDNN_INVALID_FORMAT | |
| ZDNN_INVALID_STRIDE_PADDING | |
| ZDNN_INVALID_STRIDES | |
| ZDNN_INVALID_CLIPPING_VALUE | |
| Hardware statuses | See "Hardware statuses" on page 8. |
| ZDNN_FUNC_RC_F000 | Invalid *padding_type* value. |
| ZDNN_FUNC_RC_F001 | Invalid *act_func* value. |
| ZDNN_FUNC_RC_F002 | *stride_height* = 0 and *stride_width* = 0, but either the *kernel_height* or *kernel_width* value is greater than 448. |
| ZDNN_FUNC_RC_F003 | *stride_height* > 0 and *stride_width* > 0, but either the *kernel_height* or *kernel_width* value is greater than 64. |
| ZDNN_FUNC_RC_F004 | Either the *stride_height* or *stride_width* value is greater than 13. |

### Framework examples

- TensorFlow Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D)
- ONNX Conv2D (https://github.com/onnx/onnx/blob/master/docs/Operators.md)

## 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

```
#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "zdnn.h"

// ***************************************************************************
// Sample:
//
// Create 2 zTensors a and b, and add them together via zdnn_add()
// ***************************************************************************
int main(int argc, char *argv[]) {
  zdnn_tensor_desc pre_tfrmd_desc, tfrmd_desc;
  zdnn_ztensor ztensor_a;
  zdnn_ztensor ztensor_b;
  zdnn_ztensor ztensor_out;
  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);
  void *data2 = malloc(num_elements * element_size);
  void *data_out = malloc(num_elements * element_size);

  // read input_data

  // check status for AIU 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);
    ((float *)data2)[i] = (float)(i & 0x7f);
  }

  zdnn_init_pre_transformed_desc(ZDNN_NHWC, type, &pre_tfrmd_desc, dim_n, dim_h,
                                 dim_w, dim_c);
  // generate transformed shape information
  status = zdnn_generate_transformed_desc(&pre_tfrmd_desc, &tfrmd_desc);
  assert(status == ZDNN_OK);

  // initialize zTensors and allocate 4k-aligned storage via helper function
  status =
      zdnn_init_ztensor_with_malloc(&pre_tfrmd_desc, &tfrmd_desc, &ztensor_a);
  assert(status == ZDNN_OK);
  status =
      zdnn_init_ztensor_with_malloc(&pre_tfrmd_desc, &tfrmd_desc, &ztensor_b);
  assert(status == ZDNN_OK);
  status =
      zdnn_init_ztensor_with_malloc(&pre_tfrmd_desc, &tfrmd_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_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;

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 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;

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 | 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_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 | 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_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
 ***********************************************************************/

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_LSTM | 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_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 = zdnn_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
 ***********************************************************************/

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_LSTM | 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_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 AIU
   ***********************************************************************/

  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;

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 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;

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 | 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_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 | 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_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
 ***********************************************************************/

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_LSTM | 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_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 = zdnn_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
 ***********************************************************************/

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_LSTM | 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_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 AIU
 ***********************************************************************/

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
     ***********************************************************************/

    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_LSTM | USAGE_HIDDEN_WEIGHTS |
            (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
        &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_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 = zdnn_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
     ***********************************************************************/

    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_LSTM | USAGE_HIDDEN_BIASES |
            (is_prev_layer_bidir ? PREV_LAYER_BIDIR : PREV_LAYER_UNI),
        &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_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 cf output zTensor
     ***********************************************************************/

    zdnn_tensor_desc cf_pre_tfrmd_desc, cf_tfrmd_desc;
```

```
  zdnn_ztensor cf_output_ztensor;

  zdnn_init_pre_transformed_desc(ZDNN_4DS, 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(&cf_pre_tfrmd_desc, &cf_tfrmd_desc,
                                         &cf_output_ztensor);
  assert(status == ZDNN_OK);

  /***********************************************************************
   * Call the AIU
   ***********************************************************************/

  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_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;

  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 initial hidden zTensor
 ***********************************************************************/

zdnn_tensor_desc h0_pre_tfrmd_desc, h0_tfrmd_desc;
zdnn_ztensor h0;

zdnn_init_pre_transformed_desc(ZDNN_3DS, type, &h0_pre_tfrmd_desc, num_dirs,
                               num_batches, num_hidden);
status = zdnn_generate_transformed_desc(&h0_pre_tfrmd_desc, &h0_tfrmd_desc);
assert(status == ZDNN_OK);

status =
    zdnn_init_ztensor_with_malloc(&h0_pre_tfrmd_desc, &h0_tfrmd_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 AIU
 **********************************************************************/

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);
}
```

# 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 6 on page 79.

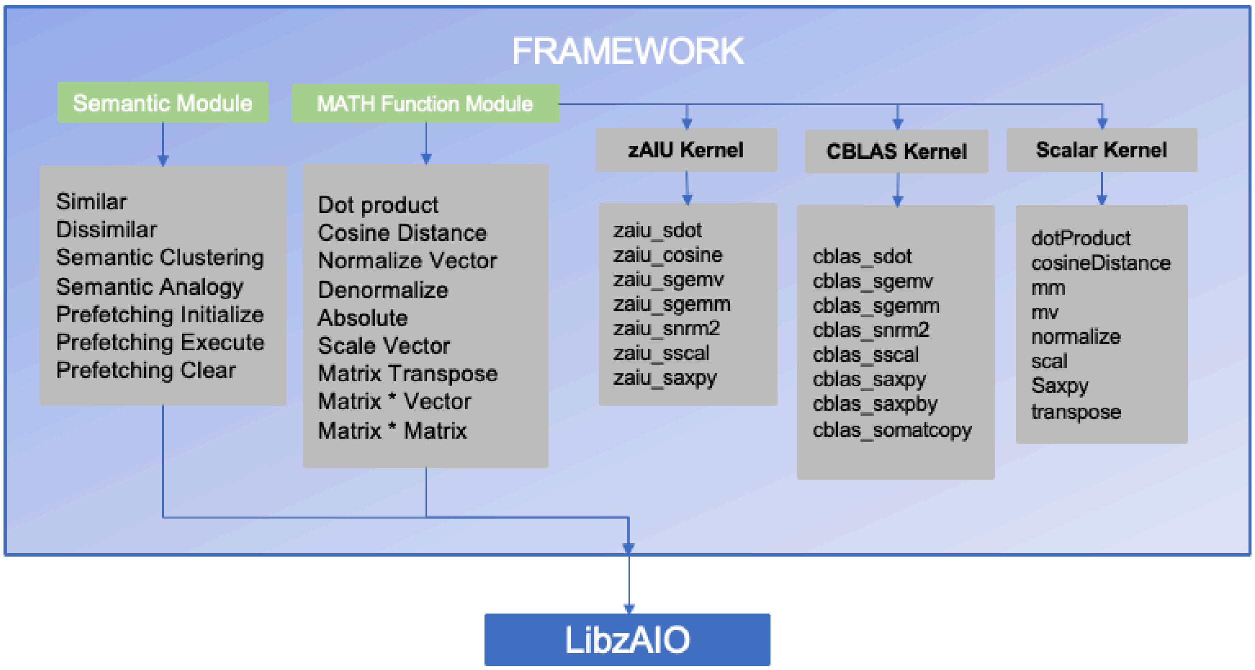


*Figure 6. IBM Z Artificial Intelligence Optimization Library framework*

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 features available on IBM Z systems designed with the new 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(AIEZDNNS) | `/usr/lpp/IBM/aie/zdnn/lib/libzdnn.so` |
| IBM Z Artificial Intelligence Optimization Library | SYS1.SIEALNKE(AIEZAIOS) | `/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/IBM/aie/zdnn"
ZAIO_INSTALL_DIR="/usr/lpp/IBM/aie/zaio"
OPENBLAS_INSTALL_DIR="/usr/lpp/cbclib"
export LIBPATH=${ZDNN_INSTALL_DIR}/lib:${ZAIO_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 AIEZAIOX 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 80.

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 83) 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 12. 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:

```
0xnnnnnnnnnnnnrrrr
```

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 13 on page 81 lists the zAIO return code mnemonic constants and meanings.

*Table 13. 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. |
| 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. |

*Table 13. zAIO return code mnemonic constants (continued)*

| Mnemonic constant | Meaning |
| --- | --- |
| 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)

### 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 _ZAIO_SOURCE 2 prior to the inclusion of the library header.

```
#define _ZAIO_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:

$$\textbf{Average}\ (\boldsymbol{V}_1, \boldsymbol{V}_2, \cdots, \boldsymbol{V}_k) = \frac{\left(V_{1_i} + V_{2_i} + \cdots + V_{k_i}\right)}{\boldsymbol{k}} \{i\colon 1 \to v\_size\}$$

**Note:** This function is enabled with directive _ZAIO_SOURCE 2 prior to the inclusion of the library header.

```
#define _ZAIO_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:

$$\textbf{Average}\ (\boldsymbol{V_1}, \boldsymbol{V_2}, \cdots, \boldsymbol{V_k}) = \frac{\left(V_{1_i} + V_{2_i} + \cdots + V_{k_i}\right)}{\boldsymbol{k}} \{i\colon 1 \rightarrow v\_size\}$$

**Note:** This function is enabled with directive _ZAIO_SOURCE 2 prior to the inclusion of the library header.

```
#define _ZAIO_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.

$$\textbf{Dot Product} \quad V_1V_2 = \sum_{i=1}^{v\_size} V_{1_i}V_{2_i}$$

**Note:** This function is enabled with directive `_ZAIO_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAIO_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.

**Cosine Similarity** $$cos(V_1, V_2) = \frac{V_1 V_2}{\|V_1\| \|V_2\|} = \frac{\Sigma_{i=1}^{v\_size} V_{1_i} V_{2_i}}{\sqrt[2]{\Sigma_{i=1}^{v\_size} V_{1_i}^2} \sqrt[2]{\Sigma_{i=1}^{v\_size} V_{2_i}^2}}$$

**Note:** This function is enabled with directive `_ZAIO_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAIO_SOURCE 2
#include <zaio.h>
```

## Format

```
zaio_status_t zaio_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 *zaio_status_t*, of which the rightmost 16 bits denote the return code.

# Vector normalization (zaio_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:

**Vector Normalization** $\|V\| = \frac{V_i}{\sqrt[2]{\sum_{i=1}^{v\_size} V_i^2}} \quad \{i: 1 \rightarrow v\_size\}$

**Note:** This function is enabled with directive _ZAIO_SOURCE 2 prior to the inclusion of the library header.

```
#define _ZAIO_SOURCE 2
#include <zaio.h>
```

### Format

```
zaio_status_t zaio_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 `_ZAIO_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAIO_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, *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.

### 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, *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.

## 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 `_ZAIO_SOURCE 2` prior to the inclusion of the library header.

```
#define _ZAIO_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 (*Vx*) by vector(*Vy*), where

*Vx* comes from a set of possible vectors and *Vy* is the comparing vector, or matrix (collection of many *Vx*) by vector (*Vy*), 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 99
- "C example: Implicit DLL load" on page 101


## 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 _ZAIO_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 ZAIO 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 (*)())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 _ZAIO_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 109 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 2.4 later.

## 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:

- IBM Z Artificial Intelligence Data Embedding Library: `/usr/lpp/IBM/aie/zade`
- IBM Z Artificial Intelligence Optimization Library: `/usr/lpp/IBM/aie/zaio`
- IBM z/OS OpenBLAS library: `/usr/lpp/cbclib`

During runtime, 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/IBM/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/IBM/aie/zade/bin`) and the path to the Java 8 64-bit executable (for instance `/usr/lpp/java/J8.0_64/bin`).

In addition, Java 8 64-bit 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/IBM/aie/zade"
ZAIO_INSTALL_DIR="/usr/lpp/IBM/aie/zaio"
OPENBLAS_INSTALL_DIR="/usr/lpp/cbclib"
JAVA_HOME="/usr/lpp/java/J8.0_64/bin"

export LIBPATH=${ZADE_INSTALL_DIR}/lib:${ZAIO_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 zAIO 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.

# IBM Z Artificial Intelligence Data Embedding Library log files

The IBM Z Artificial Intelligence Data Embedding Library APIs use the following log files:

- zade-main log
- base10Cluster log
- ibm-data2vec log

## zade-main log

Every IBM Z Artificial Intelligence Data Embedding Library application first invokes the zade_main binary, which, in turn, invokes package-specific functions. The zade_main program emits a series of messages on the stdout stream. The messages in the following example represent a successful invocation and termination of the zade_main child process, after invoking one of the user-specified ZADE functions.

```
Spawned
zade_main using process 131801
Parent has done waiting for the termination of child process
The pid of terminated child process is 13180
```

## base10Cluster log

The base10Cluster function generates a log file, `base10cluster-`*`local`*`-`*`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 "base10Cluster" on page 109 first reports the version of the base10cluster function. Then, for each participating thread (in this case, only 1 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 v3.4.0 Release Build 19:28:51 Mar 30 2022
PID 131842 | 2022-03-31 00:25:46 base10_cluster() started clustering.
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=7.6 25500e+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 ./V_SMINT_output_minimums for storing the cluster minimums
PID 131842 | 2022-03-31 00:25:46 base10_cluster() completed clustering in 0.910 seconds.
Base10Clustering is terminated successfully.
```

## ibm-data2vec log

The ibm-data2vec function also generates a log file, `ibm-data2vec-local-timestamp.log`, that provides additional details about the execution of the ibm-data2bvec function. For example, the following log file first reports the version of ibm-data2vec and the name of the training file. Then it reports if the dependent library, `libzaio.so`, has been found. If the library is detected, it verifies whether the `libzaio.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, 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 (v1.3.0 Release Build 19:28:56 Mar 30 2022 for zOS) starting execution using file
Textified_Churn_Data.txt
ibm-data2vec found the required library: libzaio.so. Proceeding with the training..
ibm-data2vec using BLAS SIMD acceleration for training: True
Learning vocabulary from the training file: Textified_Churn_Data.txt with size 2899990 bytes
16908848 | 2022-03-30 23:35:25 | Time elapsed learning vocab from train file = 0.773295s
There are 154946 words in the training file. There are 7104 words in the vocabulary: Primary Key words
 7043 Internal words 60
Model training code will generate vectors for row-identifier (pk_id) or user-specified primary keys
16908848 | 2022-03-30 23:35:25 | Stage 1 completed. Time elapsed during file reading = 0.775163s
Training the database embedding (db2Vec) model using 8 CPU thread(s)
Learning rate Alpha=0.023796 Training Progress=5.00%
....
Learning rate Alpha=0.001701 Training Progress=100.00%
16908848 | 2022-03-30 23:35:58 | Stage 2 completed. Time elapsed training model = 32.547968s
16908848 | 2022-03-30 23:35:58 | Stage 3 completed. Time elapsed writing output = 0.125447s
16908848 | 2022-03-30 23:35:58 | ibm-data2vec execution completed. Total time elapsed = 33.481913s
ibm-data2Vec Total allocated memory 176.51 MB In-use memory 0.00 MB
ibm-data2Vec Per-stage memory usage:
    (1) Stage DATA_LOADING_STAGE total-allocated=19.68 MB in-use=0.00 MB;
    (2) Stage VOCAB_BUILDING_STAGE total-allocated=124.96 MB in-use=0.00 MB;
    (3) Stage MODEL_TRAINING_STAGE total-allocated=31.87 MB in-use=0.00 MB;
    (4) Stage MODEL_STORING_STAGE total-allocated=0.00 MB in-use=0.00 MB
```

# 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):


- "base10Cluster" on page 109
- "ibm-data2vec" on page 110


## base10Cluster

### Description

The base10Cluster package implements a numerical clustering algorithm that aims to cluster together numerically closers items in different buckets, each representing a distinct cluster. The base10Cluster algorithm uses two 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 base10Cluster clustering has been 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 set 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 preprocessing code to assign a string token identifier to a numeric value.

### Format

The base10Cluster component can be invoked using the `zade.Base10Cluster` instance function:

```
⋮
ZADE zade = new ZADE();
zade.Base10Cluster(args);
⋮
```

### Parameters

**int *num_threads***
: The number of threads used for parallelization.

**String[] *file_names***
: The list of input file names in CSV format. When submitted via the command line, input file names must be separated by a semicolon (`;`).

**String *output_dir***
: The directory where the output file will be stored.

### Output files

**`minimums`**
: A file, stored in *output_dir*, that lists the cluster minimum values, sorted in increasing order.

**log**
A file, stored in the current working directory, that contains the execution log messages from the function.

# ibm-data2vec

## Description

The ibm-data2vec function 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 function views the training document as a set of sentences, where each sentence represents 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 related equally 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.

## Format

The ibm-data2vec component can only be invoked using the `ZADE Data2Vec` instance function:

```
⋮
ZADE zade = new ZADE();
zade.Data2Vec(args);
⋮
```

## Parameters

**int *num_threads***
The number of threads to use for parallelization.

**String *input_file***
The name of the input file.

**String *output_file***
The name of the output file.

**String *format***
The format for the Db2 storage layout.

**String *vocab_file***
An optional parameter that is the name of the vocab file.

If the **`vocab_file_fmt`** parameter is not specified, format 1 of the vocab file will be generated.

**int vocab_file_fmt**
An optional parameter that is the format number of the vocab file. The supported formats are:

**1**
The first vocab file format.

**2**
Adds DB2_GENERATED_COLUMNNAME columns.

If omitted, format 1 of the vocab file will be generated.

**Note:** The **`vocab_file`** and **`vocab_file_fmt`** parameters will be deprecated in a future release.

## Output files

**`binary`**
A file, stored in *output_dir*, that contains the trained model, written using the Db2 zload format.

**`log`**
A file, stored in the current working directory, that contains the execution log messages from the function.

# 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

The following two examples demonstrate how to invoke the base10Cluster function on an input CSV file. In these examples, an input file, `/path/libzade/test/csvs/V_SMINT.csv`, is processed, and the output file containing the cluster minimum values, `V_SMINT_output_minimums`, is stored in the `/path/libzade/test/csv` directory.

1. Using the ZADE `main` function:

```
java com.ibm.zos.zdnn.zade.jni.ZADE Base10Cluster 1 /path/libzade/test/csvs/V_SMINT.csv /
path/libzade/test/csv
```

2. Using the ZADE `Base10Cluster` instance function of the Java ZADE class. The following code fragment shows a file, `test_b10c.java`.

```
import com.ibm.zos.zdnn.zade.jni.ZADE;

public class test_b10c{

     public static void main(String[] args){

         ZADE zade = new ZADE();
         zade.Base10Cluster(args);

     }

}
```

To run the code:

```
java test_b10c 1 /path/libzade/test/csvs/V_SMINT.csv /path/libzade/test/csv
```

## 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 file with `Churn` prefix using the binaryDB2 binary storage format (`Churn-320-10-5-normal-db2zos_zload.bin`). Finally, a list of unique tokens in the trained model (vocabulary) is stored in the file, `vocab`.

1. Using the ZADE `main` function:

```
java com.ibm.zos.zdnn.zade.jni.ZADE Data2Vec 8 Textified_Churn_Data.txt Churn binaryDB2 vocab
```

2. Using the ZADE `Data2Vec` instance function of the Java ZADE class. The following code fragment shows a file, `test_d2v.java`.

```
import com.ibm.zos.zdnn.zade.jni.ZADE;

public class test_d2v{

     public static void main(String[] args){
```

 

```
        ZADE zade = new ZADE();
        zade.Data2Vec(args);

    }

}
```

To run the code:

```
java test_d2v 8 Textified_Churn_Data.txt Churn binaryDB2 vocab
```

# Chapter 10. 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 105.

## 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 105, 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 105.

## 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 105.

## 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 105.

## 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 105.

## Unable to find function to invoke

This issue occurs when zade_main is directly invoked. Ensure that you are using the proper JNI invocation, as described in Chapter 8, "IBM Z Artificial Intelligence Data Embedding Library API reference," on page 109.

# Accessibility

Accessible publications for this product are offered through IBM Documentation (www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact the z/OS team web page (www.ibm.com/systems/campaignmail/z/zos/contact_z) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

# 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, JES3, 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)
- 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).

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

## H



## I



## K



## L



## M



## N



## O



## P



## Q

Product Number: 5650-ZOS

SC31-5702-50