ZenDNN User Guide

Publication # **57300**	Revision # **4.1**
Issue Date **September 2023**	

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
September 2023	4.1	• Merged all the ZenDNN user guides into one. • Updated supported TensorFlow, ONNX Runtime, and PyTorch versions.
January 2023	4.0	Updated supported TensorFlow, ONNX Runtime, and PyTorch versions.
June 2022	3.3	• Updated supported TensorFlow and PyTorch versions. • Removed Chapter 5 Prerequisites and Chapter 6 AOCC and AOCL (AMD-BLIS) Library Installation.
December 2021	3.2	Updated supported TensorFlow, ONNX Runtime, and PyTorch versions.
August 2021	3.1	Updated supported TensorFlow versions.
April 2021	3.0	Initial version.

Chapter 1 ZenDNN

1.1 Introduction

ZenDNN (Zen Deep Neural Network) Library accelerates deep learning inference applications on AMD CPUs. This library, which includes APIs for basic neural network building blocks optimized for AMD CPUs, targets deep learning application and framework developers with the goal of improving inference performance on AMD CPUs across a variety of workloads, including computer vision, natural language processing (NLP), and recommender systems. ZenDNN leverages oneDNN/ DNNL v2.6.3's basic infrastructure and APIs. ZenDNN optimizes several APIs and adds new APIs, which are currently integrated into TensorFlow, ONNX Runtime, and PyTorch. ZenDNN depends on:

- BLAS-like Library Instantiation Software (AOCL-BLIS) library for its BLAS (Basic Linear Algebra Subprograms) API needs
- AMD Math Library (LibM) for Core Math needs

AOCL-BLIS and AOCL-LibM are required dependencies for ZenDNN.

1.2 High-level Overview

The following is a high-level block diagram for the ZenDNN library, which uses the AOCL-BLIS library internally:

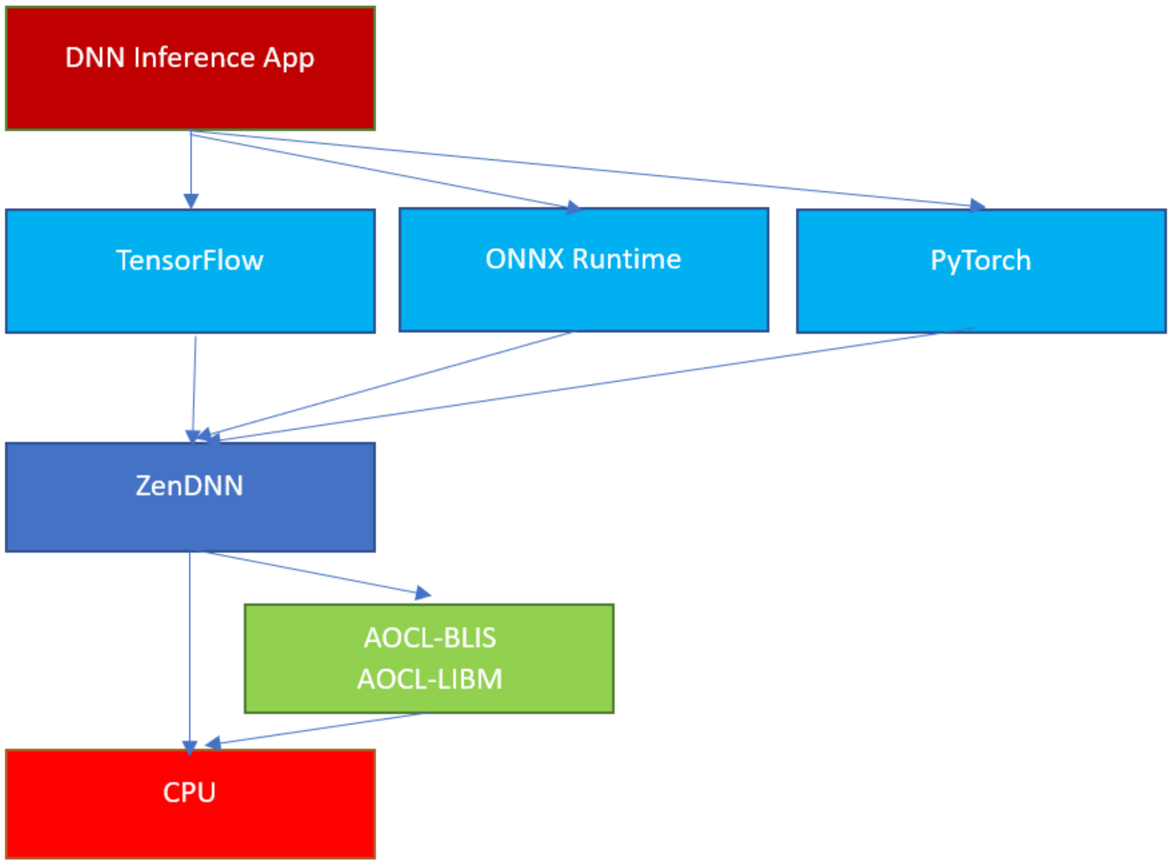

Figure 1. ZenDNN Library

In the current release, ZenDNN is integrated with TensorFlow, PyTorch, and ONNX Runtime.

1.3 Scope

The scope of ZenDNN is to support AMD EPYC^{TM} CPUs on the Linux$^{®}$ and Windows$^{®}$ (Beta support for ONNX Runtime) platforms. ZenDNN v4.1 offers optimized primitives, such as Convolution, MatMul, Elementwise, and Pool (Max and Average) that improve the performance of many convolutional neural networks, recurrent neural networks, transformer-based models, and recommender system models. For the primitives not supported by ZenDNN, execution will fall back to the native path of the framework.

1.4 Release Highlights

Following are the highlights of this release:

- ZenDNN library is integrated with:
  - TensorFlow v2.12
  - PyTorch v1.13
  - ONNX Runtime v1.15.1 on Linux and Windows (Beta)
- Python v3.8-v3.11 have been used to generate the following wheel files (*.whl):
  - TensorFlow v2.12
  - ONNX Runtime v1.15.1 on Linux and Windows (Beta)
- Python v3.7-v3.10 have been used to generate the PyTorch v1.13 wheel files (*.whl).
- Added the following environment variables for tuning performance:
  - Memory Pooling (Persistent Memory Caching):
    - ZENDNN_ENABLE_MEMPOOL for all the TensorFlow models
    - Added MEMPOOL support for INT8 and BF16 models
  - Convolution Operation:
    - ZENDNN_CONV_ALGO for all the TensorFlow models
    - Added new ALGO paths
  - Matrix Multiplication Operation:
    - ZENDNN_GEMM_ALGO for TensorFlow, PyTorch, and ONNX Runtime models
    - Added new ALGO paths and experimental version of auto-tuner

      Note: *Auto-tuner is available only for TensorFlow.*
- NHWC (default format) and Blocked Format (NCHWc8) continue to be supported.

ZenDNN library is intended to be used in conjunction with the frameworks mentioned above and cannot be used independently. It is inherited from oneDNN v2.6.3.

The latest information on the ZenDNN release and installers is available on AMD Developer Central (*https://www.amd.com/en/developer/zendnn.html*).

1.5 Supported OS and Compilers

This release of ZenDNN supports the following Operating Systems (OS) and compilers:

1.5.1 OS

- Ubuntu® 22.04 LTS and later
- Red Hat® Enterprise Linux® (RHEL) 9.1 and later
- CentOS Stream 8.4
- SUSE Linux Enterprise Server (SLES) 15 SP5
- Anolis OS 8.8 for PyTorch v1.13 wheel files
- Windows® 10 and 11

1.5.2 Compilers

- GCC 9.3 and later
- Microsoft Visual Studio 2019 with Clang compiler

1.6 Dependencies

1.6.1 Build

ZenDNN has the following build dependencies:

- For TensorFlow, ONNX Runtime, and PyTorch, Ccache package is required to build from source.
- For TensorFlow, Bazel package is required to build from source.

1.6.2 Runtime

ZenDNN has the following runtime dependencies:

- GNU C library (*glibc.so*)
- GNU Standard C++ library (*libstdc++.so*)
- Dynamic linking library (*libdl.so*)
- POSIX Thread library (*libpthread.so*)
- C Math Library (*libm.so*)
- OpenMP (*libomp.so and libomp.dll*)

- Python v3.8-3.11 for:
  - TensorFlow v2.12
  - ONNX Runtime v1.15.1 on Linux and Windows (Beta)
- Python v3.7-3.10 for PyTorch v1.13

Since ZenDNN is configured to use OpenMP, a C++ compiler with OpenMP 2.0 or later is required for runtime execution.

1.7 Logs

Logging is disabled in the ZenDNN library by default. It can be enabled using the environment variable **ZENDNN_LOG_OPTS** before running any tests. Logging behavior can be specified by setting the environment variable **ZENDNN_LOG_OPTS** to a comma-delimited list of **ACTOR:DBGLVL** pairs.

The different ACTORS are as follows:

Table 1. Log Actors

Actor	Description
ALGO	Logs all the executed algorithms.
CORE	Logs all the core ZenDNN library operations.
API	Logs all the ZenDNN API calls.
TEST	Logs all the calls used in API, functionality, and regression tests.
PROF	Logs the performance of operations in millisecond.
FWK	Logs all the framework (Tensorflow, ONNX Runtime, and PyTorch) specific calls.

For example:

- To turn on info logging, use **ZENDNN_LOG_OPTS=ALL:2**
- To turn off all logging, use **ZENDNN_LOG_OPTS=ALL:-1**
- To only log errors, use **ZENDNN_LOG_OPTS=ALL:0**
- To only log info for ALGO, use **ZENDNN_LOG_OPTS=ALL:-1,ALGO:2**
- To only log info for CORE, use **ZENDNN_LOG_OPTS=ALL:-1,CORE:2**
- To only log info for FWK, use **ZENDNN_LOG_OPTS=ALL:-1,FWK:2**
- To only log info for API, use **ZENDNN_LOG_OPTS=ALL:-1,API:2**
- To only log info for PROF (profile), use **ZENDNN_LOG_OPTS=ALL:-1,PROF:2**

Note: *For PyTorch NLP models, use **ZENDNN_LOG_OPTS=ALL:-1,PROF:4**.*

Enable Profiling Logs

To enable the profiling logs **zendnn_primitive_create** and **zendnn_primitive_execute**, you can use:

ZENDNN_PRIMITIVE_LOG_ENABLE=1

The Different Debug Levels (DBGLVL) are as follows:

```
enum LogLevel

{

    LOG_LEVEL_DISABLED = -1,

    LOG_LEVEL_ERROR = 0,

    LOG_LEVEL_WARNING = 1,

    LOG_LEVEL_INFO = 2,

    LOG_LEVEL_VERBOSE0 = 3,

    LOG_LEVEL_VERBOSE1 = 4,

    LOG_LEVEL_VERBOSE2 = 5

};
```

CORE, API, and PROF are mandatory logs when ZenDNN library is called. ALGO, TEST, and FW are optional logs and may not appear in all the cases.

Chapter 2 TensorFlow

2.1 Installing ZenDNN with TensorFlow

Note: *Refer to the section "ZenDNN" before starting the installation.*

In this release, ZenDNN library is supported for TensorFlow v2.12. This is a baseline release for TensorFlow v2.12 with:

- FP32 support
- *AMD UIF* INT8 model support
- Limited support for BF16 on AMD UIF ResNet50 and VGG16

2.1.1 Binary Release Setup

This section describes the procedure to setup the ZenDNN binary release for TensorFlow v2.12.

2.1.1.1 Conda

Complete the following steps to setup Conda:

1. Refer to Anaconda documentation (*https://docs.anaconda.com/anaconda/install/linux/*) to install Anaconda on your system. The testing has been done with Anaconda3-2020.11-Linux-x86_64.
2. Create and activate a Conda environment which will house all the TensorFlow-ZenDNN specific installations:

   ```
   conda create -n tf-v2.12-zendnn-v4.1-rel-env python=3.8 -y

   conda activate tf-v2.12-zendnn-v4.1-rel-env
   ```

   Ensure that you install the TensorFlow-ZenDNN package corresponding to the Python version with which you created the Conda environment.

   If there is any conda environment named *tf-v2.12-zendnn-v4.1-rel-env*, delete it (using command `conda remove --name tf-v2.12-zendnn-v4.1-rel-env --all`) before running *scripts/TF_ZenDNN_setup_release.sh*.

   Note: *TensorFlow-ZenDNN is compatible with Python v3.8-3.11 but 3.8 has been used as an example.*

3. It is recommended to use the naming convention:

   ```
   tf-v2.12-zendnn-v4.1-rel-env
   ```

4. Install all the necessary dependencies:

```
pip install --upgrade pyparsing

pip install --upgrade appdirs

pip install --upgrade --no-deps --force-reinstall --no-cache-dir numpy absl-py

pip install -U pip six wheel importlib-metadata setuptools mock future

pip install -U keras_applications --no-deps

pip install -U keras_preprocessing --no-deps
```

2.1.1.2 TensorFlow v2.12

Complete the following steps to install the ZenDNN binary release:

1. Copy the zipped release package to the local system being used. The name of the release package will be similar to *TF_v2.12_ZenDNN_v4.1_Python_v3.8.zip*.
2. Execute the following commands:
   a. `unzip TF_v2.12_ZenDNN_v4.1_Python_v3.8.zip`
   b. `cd TF_v2.12_ZenDNN_v4.1_Python_v3.8/`
   c. `source scripts/TF_ZenDNN_setup_release.sh`
      This installs the TensorFlow wheel package provided in the zip file.

      Note: *Ensure that it is sourced only from the folder TF_v2.12_ZenDNN_v4.1_Python_v3.8/.*
      You must run the command conda activate tf-v2.12-zendnn-v4.1-rel-env whenever you open a new terminal.

The release binaries for TensorFlow v2.12 are compiled with manylinux2014 and they provide compatibility with some older Linux distributions. The support for Docker releases has been discontinued.

For more information on the supported OS and compilers for the Python wheel file, refer to the section "Supported OS and Compilers".

The C++ interface will work on operating systems (with glibc version 2.31 or later):

- Ubuntu 22.04 and later
- RHEL 9.1 and later

2.1.2 Build from Source

To build ZenDNN with TensorFlow pip package from source, download TensorFlow-ZenDNN source code from:

https://github.com/amd/ZenDNN-tensorflow

The repository defaults to the master development branch that does not have ZenDNN support. To build, you must check out the release branch **r2.12_zendnn_rel**.

For more information on the building procedure, refer to *BUILD_SOURCE.md*.

2.2 Directory Structure

The release folder consists of a TensorFlow wheel (.whl) and the following directory:

- *scripts* contains scripts to set up the environment and run benchmarks

2.3 High-level Overview

For more information, refer to the section "High-level Overview".

2.4 TensorFlow CNN Benchmarks

The benchmark scripts provide performance benchmarking at the TensorFlow level, printing latency and throughput results for AlexNet, GoogLeNet, InceptionV3, InceptionV4, ResNet50, ResNet152, VGG16, and VGG19 models.

Complete the following steps:

1. Download the TensorFlow CNN benchmarks repository from GitHub:

   https://github.com/tensorflow/benchmarks.git

   ```
   cd $ZENDNN_PARENT_FOLDER

   git clone https://github.com/tensorflow/benchmarks.git $ZENDNN_PARENT_FOLDER/bench-
   marks
   ```

2. Export the environment variable BENCHMARKS_GIT_ROOT with the path to the benchmarks repository:

   ```
   export BENCHMARKS_GIT_ROOT=$ZENDNN_PARENT_FOLDER/benchmarks
   ```

For latency, execute the following commands:

1. `cd TF_v2.12_ZenDNN_v4.1_Python_v3.8/`
2. `source scripts/zendnn_TF_env_setup.sh`
3. `source scripts/tf_cnn_benchmarks_latency.sh`

For throughput, execute the following commands:

1. `cd TF_v2.12_ZenDNN_v4.1_Python_v3.8/`
2. `source scripts/zendnn_TF_env_setup.sh`
3. `source scripts/tf_cnn_benchmarks_throughput.sh`

To run the individual models rather than the entire suite, execute the following commands:

```
cd $BENCHMARKS_GIT_ROOT/scripts/tf_cnn_benchmarks/

numactl --cpunodebind=<NPS> --interleave=<NPS> python tf_cnn_benchmarks.py --
device=cpu --model=<model_name>  --data_format=NHWC --batch_size=$BATCH_SIZE --
num_batches=100 --num_inter_threads=1 --num_intra_threads=96 --nodistortions --for-
ward_only=True
```

Replace `<NPS>` with the following based on your number of NUMA nodes. Execute the command `lscpu` to identify the number of NUMA nodes for your machine:

- If you have 1 NUMA node, replace `<NPS>` with 0
- If you have 2 NUMA nodes, replace `<NPS>` with 0-1
- If you have 4 NUMA nodes, replace `<NPS>` with 0-3

Replace `<model_name>` with one of the following options:

- For AlexNet, replace `<model_name>` with `alexnet`
- For GoogLeNet, replace `<model_name>` with `googlenet`
- For InceptionV3, replace `<model_name>` with `inception3`
- For InceptionV4, replace `<model_name>` with `inception4`
- For ResNet50, replace `<model_name>` with `resnet50`
- For ResNet152, replace `<model_name>` with `resnet152`
- For VGG16, replace `<model_name>` with `vgg16`
- For VGG19, replace `<model_name>` with `vgg19`

While executing the commands, make a note of the following:

- For optimal settings, refer to the section “"Tuning Guidelines"”. Current setting refers to 96C, 2P, SMT=ON configuration.
- If a warning similar to the following appears during benchmark runs, configure your GOMP_CPU_AFFINITY setting to match the number of CPU cores supported by your machine:

  ```
  OMP: Warning #181: OMP_PROC_BIND: ignored because GOMP_CPU_AFFINITY is defined

  OMP: Warning #123: Ignoring invalid OS proc ID 48

  OMP: Warning #123: Ignoring invalid OS proc ID 49

  .

  .

  .

  OMP: Warning #123: Ignoring invalid OS proc ID 63
  ```

  For example, if your CPU has 24 cores, your GOMP_CPU_AFFINITY should be set as "export GOMP_CPU_AFFINITY=0-23".

Note: *TensorFlow Eager mode execution is supported through ZenDNN operations using only the following commands:*
export ZENDNN_CONV_ALGO=1
export ZENDNN_ENABLE_MEMPOOL=0

2.5 TensorFlow v2.12

In this release of ZenDNN:

- ZenDNN library is supported for TensorFlow v2.12.
- AMD Unified Inference Frontend (UIF) optimized models are supported. For the model details, refer to the AMD UIF documentation.
- TensorFlow v2.12 wheel file is compiled with GCC v9.3.1.
- TensorFlow v2.12 is expected to deliver similar or better performance as compared to TensorFlow v2.10.

2.6 Environment Variables

ZenDNN uses the following environment variables to setup paths and control logs, tune performance:

Table 2. Tensorflow-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
Generic (Setup paths and control logs)	
ZENDNN_LOG_OPTS	ALL:0
ZENDNN_PARENT_FOLDER	Path to unzipped release folder
TF_ZEN_PRIMITIVE_REUSE_DISABLE	FALSE
ZENDNN_ENABLE_MEMPOOL	The default value is set to 1, you can provide the value 0 to disable it. 1 is for Graph-based MEMPOOL and 2 is for Node-based MEMPOOL.
ZENDNN_PRIMITIVE_CACHE_CAPACITY	The default value is set to 1024, you can modify it as required[a].
ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE	0
OMP_DYNAMIC	FALSE
ZENDNN_INFERENCE_ONLY	Default value is set to 1. ZenDNN does not currently support training. You can set it to 0 when you want to enable vanilla training and inference.
Optimized (Tune performance)	
OMP_NUM_THREADS	The default value is set to 96. You can set it as per the number of cores in the user system[a].

Table 2. Tensorflow-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
OMP_WAIT_POLICY	ACTIVE
OMP_PROC_BIND	false
GOMP_CPU_AFFINITY	Set it as per the number of cores in the system being used. For example, use 0-95 for 96-core server.
ZENDNN_TENSOR_POOL_LIMIT	The default value is set to 1024. You can modify it to 512 for CNNs for optimal performance. For more information, refer to the section "Recommended Settings".
ZENDNN_INT8_SUPPORT	The default value is set to 0. You can modify it to 1 to enable the INT8 data type support. This works only with ZENDNN_CONV_ALGO=4. ***Note: This environment variable is not required for AMD UIF models.***
ZENDNN_TF_CONV_ADD_FUSION_SAFE	The default value is set to 0. You can modify it to 1 to enable Conv, Add fusion. Currently it is safe to enable this switch for resnet50v1_5, resnet101, and inception_resnet_v2 models only.
ZENDNN_GEMM_ALGO	The default value is 3. You can modify it to one of the following: • 0 = Auto • 1 = AOCL-BLIS path • 2 = Partial AOCL-BLIS • 3 = ZenDNN JIT path • 4 = ZenDNN partial JIT path ***Note: Auto is an experimental feature and should be used with application warmup iteration >=15.***
ZENDNN_CONV_ALGO	The default value is set to 4. Decides the convolution algorithm to be used in execution and the possible values are: • 1 = im2row followed by GEMM • 2 = WinoGrad (fallback to im2row GEMM for unsupported input sizes) • 3 = Direct convolution with blocked inputs and filters • 4 = Direct convolution with blocked filters
ZENDNN_LOG_OPTS=FWK:4	Dump graph after ZenDNN rewrites pass for all the TensorFlow models.
TF_ENABLE_ZENDNN_OPTS	The default value is set to 1 and ZenDNN code path will be used. You can modify it to 0 to use native TensorFlow code path.

a. You must define these environment variables explicitly.

Note: *There are a few other environment variables that are initialized by the setup script, however these are not applicable for the binary release setup.*

When `source scripts/zendnn_TF_env_setup.sh` is invoked, the script initializes all the environment variables except the one(s) which must be set manually. The environment variable **ZENDNN_PARENT_FOLDER** is initialized relative to the unzipped release folder. To ensure that the paths are initialized correctly, it is important that the script is invoked from the unzipped release folder.

2.7 Tuning Guidelines

The hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. The details for the environment variables used on a 4th Gen AMD EPYC^{TM} server to achieve the optimal performance numbers are as follows:

2.7.1 System

A system with the following specifications has been used:

Table 3. System Specification

Model name	4th Gen AMD EPYC^{TM} 9654 1P 96-Core Processor
DPU MHz	Up to 3.7 GHz
No. of Cores	96
1P/2P	1
SMT: Thread(s) per Core	2
Mem-Dims	12x64 GB

OS Used: Ubuntu 22.04 LTS

2.7.2 Environment Variables

The following environment variables have been used:

ZENDNN_LOG_OPTS=ALL:0

TF_ENABLE_ONEDNN_OPTS=0

TF_ENABLE_ZENDNN_OPTS=1

OMP_NUM_THREADS=96

OMP_WAIT_POLICY=ACTIVE

OMP_PROC_BIND=FALSE

OMP_DYNAMIC=FALSE

ZENDNN_ENABLE_MEMPOOL=1

ZENDNN_GEMM_ALGO=3

Note: *For NLP and Recommender models, better performance is observed with ZENDNN_GEMM_ALGO=4. However, these details should be verified empirically.*

ZENDNN_TENSOR_POOL_LIMIT=1024

ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE=0

ZENDNN_CONV_ALGO=4

ZENDNN_PARENT_FOLDER=/home/<user_id>/my_work

BENCHMARKS_GIT_ROOT=/home/<user_id>/my_work/benchmarks

ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024

ZENDNN_INT8_SUPPORT=0

ZENDNN_INFERENCE_ONLY=1

ZENDNN_TF_CONV_ADD_FUSION_SAFE=0

Other than the ZENDNN environment variables, there are a few other parameters that influence the memory policy across the nodes, thread binding to the available physical cores. Considerable performance improvements may be achieved by setting these parameters carefully. Following sections describe the behavior and possible values for these parameters.

2.7.2.1 Recommended Settings

A few recommended settings for AMD UIF v1.1 models are as follows:

- UIF v1.1 FP32 tf_efficientnet-edgetpu and INT8 tf_RefineDet-Medical_EDD model ZENDNN_CONV_ALGO=3 leads to accuracy drop.

  For optimal results, use ZENDNN_CONV_ALGO=4.
- UIF v1.1 FP32 tf_mobilebert_SQuADv1.1 ZENDNN_ENABLE_MEMPOOL=1 leads to accuracy drop.

  For optimal results, use ZENDNN_ENABLE_MEMPOOL =2.
- UIF v1.1 FP32 tf_RefineDet-Medical_EDD and densenet169 models lead to segmentation fault with higher value of ZENDNN_TENSOR_POOL_LIMIT. This should be verified empirically.

  For optimal results, use ZENDNN_TENSOR_POOL_LIMIT=32.

2.7.3 Thread Wait Policy

OMP_WAIT_POLICY provides options to the OpenMP runtime library based on the expected behavior of the waiting threads. It can take the abstract values **PASSIVE** and **ACTIVE**. The default value is **ACTIVE**. When **OMP_WAIT_POLICY** is set to **PASSIVE**, the waiting threads will be passive and will not consume the processor cycles. Whereas, setting it to **ACTIVE** will consume processor cycles.

Note: *For ZenDNN stack, setting **OMP_WAIT_POLICY** to **ACTIVE** may give better performance.*

2.7.4 Thread Affinity

To improve ZenDNN performance, the behavior of OpenMP threads can be guarded precisely with thread affinity settings. A thread affinity defined at startup cannot be modified or changed during runtime of the application. Following are the ways through which you can bind the requested OpenMP threads to the physical CPUs:

- **GOMP_CPU_AFFINITY** environment variable binds threads to the physical CPUs, for example:

  export GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"

  This command will bind the:

  - Initial thread to CPU 0
  - Second thread to CPU 3
  - Third and fourth threads to CPU 1 and CPU 2 respectively
  - Fifth thread to CPU 4
  - Sixth through tenth threads to CPUs 6, 8, 10, 12, and 14 respectively

  Then, it will start the assigning back from the beginning of the list.

  export GOMP_CPU_AFFINITY="0" binds all the threads to CPU 0.

- KMP affinity belongs to LLVM OpenMP runtime library and is used by setting appropriate values for the environment variable **KMP_AFFINITY**. It has the following syntax:

  KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]

  Example:

  export KMP_AFFINITY='verbose,respect,granularity=fine,compact,1,0 binds threads as close as possible to the master thread but on a different core. Once each core is assigned with one OpenMP thread, the remaining OpenMP threads are assigned in the same order as before, but on different thread contexts.

  Among the two, **KMP_AFFINITY** takes highest precedence followed by **GOMP_CPU_AFFINITY**. If none of them is set, the host system will defer the assignment of threads to CPUs. Given the same thread binding (see example below), it is expected that both the affinity settings would give the same performance.

  Example:

  Following affinity settings should give the same thread bindings:

  - **export GOMP_CPU_AFFINITY=0-95**
  - **export KMP_AFFINITY='verbose,respect,granularity=fine,compact,1,0'**

2.7.5 Non-uniform Memory Access

2.7.5.1 numactl

numactl provides options to run processes with specific scheduling and memory placement policy. It can restrict the memory binding and process scheduling to specific CPUs or NUMA nodes:

- `--cpunodebind=nodes`: Restricts the process to a specific group of nodes.
- `--physcpubind=cpus`: Restricts the process to a specific set of physical CPUs.
- `--membind=nodes`: Allocates the memory from the nodes listed. The allocation fails if there is not enough memory on the listed nodes.
- `--interleave=nodes`: Memory will be allocated in a round robin manner across the specified nodes. When the memory cannot be allocated on the current target node, it will fall back to the other nodes.

Example:

If <tensorflow_script> is the application that needs to run on the server, then it can be triggered using `numactl` settings as follows:

```
numactl --cpunodebind=0-3 -membind=0-3 python <tensorflow_script>
```

The `interleave` option of numactl works only when the number nodes allocated for a particular application is more than one. `cpunodebind` and `physcpubind` behave the same way for ZenDNN stack, whereas `interleave` memory allocation performs better than `membind`.

2.7.5.2 Concurrent Execution

As every application, AI workload requires special considerations during performance tuning to get the best out of the non-uniform memory access (NUMA) enabled machine. Improvement in performance can be achieved by carefully analyzing memory access time, memory bandwidth, and congestion on the shared bus. These factors depend on how far away the allocated memory and the process that requested the memory are in the NUMA system. In NUMA machines, the local memory access is faster as compared to the remote memory access. Consider the following workload:

```
numactl --cpunodebind=0-3 --membind=0-3 python <tensorflow_script>
```

Performance can be optimized by partitioning the workload into multiple data shards and then running concurrently on more than one NUMA node. Following example shows the concurrent execution across 4 NUMA nodes:

```
numactl --cpunodebind=0 --membind=0 python <tensorflow_script> & numactl --cpuno-
debind=1 --membind=1 python <tensorflow_script> & numactl --cpunodebind=2 --membind=2
python <tensorflow_script> & numactl --cpunodebind=3 --membind=3 python <tensor-
flow_script>
```

The number of concurrent executions can be increased beyond 4 nodes. The following formula can be used to decide the number of concurrent executions to be triggered at a time:

```
Number Concurrent Executions = Number of Cores Per Socket / Numbers of Cores sharing L3
cache
```

This can also be extended to even cores. However, these details should be verified by the user empirically.

2.7.6 Transparent Huge Pages

Transparent Huge Pages (THPs) are a Linux kernel feature for memory management to improve performance of the application by efficiently using the processor's memory-mapping hardware. THP should reduce the overhead of the Translation Lookaside Buffer. You must login as root to enable or disable THP settings. It operates mainly in two modes:

- always: You can run the following command to set THP to 'always':

  ```
  echo always > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, the system kernel tries to assign huge pages to the processes running on the system.
- madvise: You can run the following command to set THP to 'madvise':

  ```
  echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, kernel only assigns huge pages to the individual process memory areas.

You can use the following command to disable THP:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

It is recommended to use the following THP setting for better performance:

- CNN models - 'never' (batch size =1), 'always' (batch size >1)
- NLP and Recommender models - 'madvise'

Note: *These details should be verified empirically.*

2.7.7 Batch Size

Batch Size is a sensitive factor for the throughput performance of any model. The following formula could be used to calculate the optimal Batch Size:

```
 Batch Size = number_of_physical_cores * batch_factor
```

Batch factor may vary from 8-32. The value 32 may provide optimal performance. However, you should verify this empirically.

2.7.8 Memory Allocators

Based on the model, if there is a requirement for a lot of dynamic memory allocations, a memory allocator can be selected from the available allocators which would generate the most optimal performance out of the model. These memory allocators override the system provided dynamic memory allocation routines and use a custom implementation. They also provide the flexibility to override the dynamic memory management specific tunable parameters (for example, logical page size, per thread, or per-CPU cache sizes) and environment variables. The default configuration of these allocators would work well in practice. However, you should verify empirically by trying out what setting works best for a particular model after analyzing the dynamic memory requirements for that model.

TCMalloc and jemalloc are the most commonly used allocators.

2.7.8.1 TCMalloc

TCMalloc is a memory allocator which is fast, performs uncontended allocation and deallocation for most objects. Objects are cached depending on the mode, either per-thread or per-logical CPU. Most allocations do not need to take locks. So, there is low contention and good scaling for multi-threaded applications. It has flexible use of memory and hence, freed memory can be reused for different object sizes or returned to the operating system. Also, it provides a variety of user-accessible controls that can be tuned based on the memory requirements of the workload.

2.7.8.2 jemalloc

jemalloc is a memory allocator that emphasizes fragmentation avoidance and scalable concurrency support. It has a powerful multi-core/multi-thread allocation capability. The more cores the CPU has, the more program threads, the faster jemalloc allocates. jemalloc classifies memory allocation granularity better, leading to less lock contention. It provides various tunable runtime options, such as enabling background threads for unused memory purging, allowing jemalloc to utilize transparent huge pages for its internal metadata, and so on.

2.7.8.3 Usage

You can install the TCMalloc/jemalloc dynamic library and use **LD_PRELOAD** environment variable as follows:

```
Before using TCMalloc:
export LD_PRELOAD=/path/to/TCMallocLib/

Before using jemalloc:
export LD_PRELOAD=/path/to/jemallocLib/

Or

Benchmarking command using TCMalloc:
LD_PRELOAD=/path/to/TCMallocLib/ < python benchmarking command>

Benchmarking command using jemalloc:
LD_PRELOAD=/path/to/jemallocLib/ < python benchmarking command>
```

To verify if TCMalloc/jemalloc memory allocator is in use, you can grep for tcmalloc/jemalloc in the output of `lsof` command:

```
lsof -p <pid_of_benchmarking_command> | grep <tcmalloc/jemalloc>
```

2.8 Convolution Algorithm Logic

Convolution kernels take Input and Filter/Weights as arguments and return Output. The table below describes the expected Layout for each of the convolution algorithms currently supported by TensorFlow-ZenDNN.

Table 4. Convolution Algorithm Logic

zenConvAlgoType	ZENDNN_CONV_ALGO	Input Layout	Filter Layout	Output Layout
GEMM	1	NHWC	HWIO	NHWC
WINOGRAD	2	NHWC	HWIO	NHWC
DIRECT1	3	nChw8c	Ohwi8o	nChw8c
DIRECT2	4	NHWC	Ohwi8o/ Ohwi16o	NHWC

Note: *In the context of Filter Layouts, HWIO is equivalent to HWCN but with I instead of C representing input channels and O instead of N representing output channels.*

2.9 Limited Precision Support

Quantization is an active area of research and a popular compression technique to accelerate neural network performance.

A few of these quantized neural networks models and TensorFlow protobuf (pb) files are publicly available. On AMD 4th Gen EPYC^{TM} platforms, ZenDNN offers options to enable INT8 quantization with AMD's UIF INT8 models. These models can be leveraged using AMD UIF benchmarking scripts.

ZenDNN provides limited support for BF16 on AMD UIF ResNet50 and VGG16.

To optimize performance, use the following environment variables:

```
export ZENDNN_ENABLE_MEMPOOL=1/2

export ZENDNN_TENSOR_POOL_LIMIT=1024
```

Chapter 3 ONNX Runtime

3.1 Installing ZenDNN with ONNX Runtime

Note: *Refer to the section "ZenDNN" before starting the installation.*

In this release, ZenDNN library is supported for ONNX Runtime v1.15.1. This is a baseline release for ONNX Runtime v1.15.1 with:

- FP32 support
- *AMD UIF* INT8 model support
- Limited support for BF16 on a few CNN models

3.1.1 Binary Release Setup

3.1.1.1 Conda

Complete the following steps to set up Conda:

1. Refer to Anaconda documentation (*https://docs.anaconda.com/anaconda/install/linux/*) to install Anaconda on your system. The testing has been done with Anaconda3-2020.11-Linux-x86_64.
2. Create and activate a Conda environment which will house all the ONNX Runtime-ZenDNN specific installations:

   ```
   conda create -n onnxrt-v1.15.1-zendnn-4.1-rel-env python=3.8 -y

   conda activate onnxrt-v1.15.1-zendnn-4.1-rel-env
   ```

   Ensure that you install the ONNX Runtime-ZenDNN package corresponding to the Python version with which you created the Conda environment.

   If there is any conda environment named *onnxrt-v1.15.1-zendnn-v4.1-rel-env*, delete it (using the command `conda remove --name onnxrt-v1.15.1-zendnn-v4.1-rel-env --all`) before running *scripts/ONNXRT_ZenDNN_setup_release.sh*.

   Note: *ONNX Runtime-ZenDNN is compatible with Python v3.8-3.11 but 3.8 has been used as an example.*

3. It is recommended to use the naming convention:

   ```
   onnxrt-v1.15.1-zendnn-v4.1-rel-env
   ```

4. Install all the necessary dependencies:

   ```
   pip install -U cmake numpy pytest psutil torch==2.0.1 coloredlogs

   pip install -U transformers sympy --ignore-installed ruamel.yaml

   pip install onnx==1.14.0
   ```

3.1.1.2 ONNX Runtime v1.15.1

Complete the following steps to install the ZenDNN binary release:

1. Copy the zipped release package to the local system being used. The name of the release packages will be similar to *ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8/*.
2. Execute the following commands:
   a. `unzip ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8.zip`
   b. `cd ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8/`
   c. `source scripts/ONNXRT_ZenDNN_setup_release.sh`
   d. `pip install protobuf==3.20.2`

   Note: *Ensure that it is sourced only from the unzipped release folder.*

For more information on the supported OS and compilers for the Python wheel file, refer to the section "Supported OS and Compilers".

C++ Interface will work on the following operating systems (with glibc version 2.17 or later):

- Ubuntu 22.04 and later
- RHEL 9.1 and later

3.1.2 Build from Source

To build ZenDNN with ONNX Runtime pip package from source, download the ONNX Runtime-ZenDNN source code from:

https://github.com/amd/ZenDNN-onnxruntime

The repository defaults to the master development branch which does not have ZenDNN support. To build, you must check out the release branch **rel-1.15.1_zendnn_rel**.

For more information on the building procedure, refer to *BUILD_SOURCE.md*.

3.2 Directory Structure

The release folder consists of a ONNX Runtime wheel (.whl) and the following directory:

- *scripts/* contains scripts to set up the environment

3.3 High-level Overview

For more information, refer to the section "High-level Overview".

3.4 ONNX Runtime Benchmarks

To understand latency and throughput metrics with ZenDNN execution paths, standard models such as BERT can be run on ONNX Runtime with ZenDNN backend.

Use the following setup scripts and commands to download and run BERT models and derive performance metrics:

1. `cd ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8/`
2. `source scripts/zendnn_ONNXRT_env_setup.sh`
3. Activate your conda environment.
4. The following command runs BERT benchmarking with batch-size=24, sequence length=16, and threads=96 on a NPS=4 machine setting:

```
numactl --cpunodebind=0-3 --interleave=0-3 python -m onnxruntime.transformers.bench-
mark -m bert-large-uncased --model_class AutoModel -p fp32 -i 3 -t 10 -b 4 -s 16 -n 96
-v --provider zendnn
```

While executing the commands, make a note of the following:

- For optimal settings, refer to the section "Tuning Guidelines". Current setting refers to 96C, 2P, SMT=ON configuration.

3.5 ONNX Runtime v1.15.1

In this release of ZenDNN:

- ZenDNN library is supported for ONNX Runtime v1.15.1.
- AMD Unified Inference Frontend (UIF) optimized models are supported. For the model details, refer to the AMD UIF documentation.
- ONNX Runtime v1.15.1 wheel file is compiled with GCC v9.3.1.

3.6 Environment Variables

ZenDNN uses the following environment variables:

Table 5. ONNX Runtime-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
Generic (Setup paths and control logs)	
ZENDNN_LOG_OPTS	ALL:0
ZENDNN_PARENT_FOLDER	Path to unzipped release folder
ZENDNN_PRIMITIVE_CACHE_CAPACITY	The default value is set to 1024, you can modify it as required[a].

Table 5. ONNX Runtime-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
OMP_DYNAMIC	FALSE
Optimized (Tune performance)	
OMP_NUM_THREADS	The default value is set to 96. You can set it as per the number of cores in the user system[a].
OMP_WAIT_POLICY	ACTIVE
OMP_PROC_BIND	FALSE
GOMP_CPU_AFFINITY	Set it as per the number of cores in the system being used. For example, use 0-95 for 96-core servers.
ZENDNN_CONV_ADD_FUSION_ENABLE	The flag is to enable convolution and add operator fusion. It is disabled (set to 0) by default. You can modify it to 1 to enable the fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_RESNET_STRIDES_OPT1_ENABLE	The flag is to enable strides trick optimization for ResNet blocks. It is disabled (set to 0) by default. You can modify it to 1 to enable the optimization. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_BN_RELU_FUSION_ENABLE	This flag is disabled by default. You can use export command in Linux to set it to 1 and enable it. It is used to optimize executions on limited CNN models.
ZENDNN_CONV_CLIP_FUSION_ENABLE	This flag is disabled by default. You can use export command in Linux to set it to 1 and enable it. It is used to optimize executions on all the variants of MobileNet models.
ZENDNN_CONV_RELU_FUSION_ENABLE	The flag is to enable convolution and relu operator fusion. It is enabled (set to 1) by default. You can modify it to 0 to disable the fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_CONV_ELU_FUSION_ENABLE	The flag is to enable convolution and elu operator fusion. It is disabled (set to 0) by default. You can modify it to 1 to enable the fusion. It is used to optimize executions on limited CNN models.

Table 5. ONNX Runtime-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
ORT_ZENDNN_ENABLE_INPLACE_CONCAT	This flag is used to perform in place concatenation of intermediate tensors arrays. It is disabled (set to 0) by default. You can modify it to 1 to enable the optimization. This optimization is useful for the variants of Inception and GoogleNet models.
ZENDNN_GEMM_ALGO	The default value is 3. You can modify it to one of the following: • 1 = AOCL-BLIS path • 2 = Partial AOCL-BLIS • 3 = ZenDNN JIT path • 4 = ZenDNN partial JIT path
ONNXRT_ZENDNN_CPU_ALLOC	This flag is to enable the usage of the CPU memory allocator in ZenDNN Execution Provider. By default, it is disabled.
ZENDNN_CONV_SWISH_FUSION_ENABLE	This flag is to enable the fusion of the sigmoid operator with the preceding conv operator. By default, it is disabled.
ZENDNN_QUANTIZE_CONV_ADD_FUSION_E NABLE	This flag is to enable convolution and add operator fusion in quantized models with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_QUANTIZE_CONV_RELU_FUSION_E NABLE	The flag is to enable convolution and relu operator fusion in quantized model with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_QCONV_CLIP_FUSION_ENABLE	This flag is to enable convolution and clip operator fusion in the quantize model with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of MobileNet models.
ZENDNN_ONNXRT_ENABLE_BF16_SUPPORT	This flag is disabled (set to 0) by default, you can set it to 1 to enable the execution of compute expensive operation in BF16. It optimizes the execution only for the CNN models.
***Note:** There are a few other environment variables that are initialized by the setup script, however these are not applicable for the binary release setup.*	

a. These environment variables work only for Blocked Format.

When `source scripts/zendnn_ONNXRT_env_setup.sh` is invoked, the script initializes all the environment variables except the one(s) which must be set manually. The environment variable **ZENDNN_PARENT_FOLDER** is initialized relative to the path defined by the unzipped release folder. To ensure that the paths are initialized correctly, it is important that the script is invoked from the unzipped release folder.

3.7 Tuning Guidelines

The hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. The details for the environment variables used on a 4th Gen AMD EPYC™ server to get the best performance numbers are as follows:

3.7.1 System

A system with the following specifications has been used:

Table 6. System Specification

Model name	4th Gen AMD EPYC™ 9654P 96-Core Processor
CPU MHz	Up to 3.7 GHz
No of Cores	96
1P/2P	1
SMT: Thread(s) per Core	2
Mem-Dims	12x64 GB

3.7.2 Environment Variables

The following environment variables have been used:

ZENDNN_LOG_OPTS=ALL:0

OMP_NUM_THREADS=96

OMP_WAIT_POLICY=ACTIVE

OMP_PROC_BIND=FALSE

OMP_DYNAMIC=FALSE

ZENDNN_GEMM_ALGO=3

Note: *For NLP models, a better performance is observed with ZENDNN_GEMM_ALGO=4. However, these details should be verified empirically.*

ZENDNN_PARENT_FOLDER=/home/<user_id>/my_work

ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024

ZENDNN_ONNXRT_VERSION=1.15.1

ZENDNN_ONNX_VERSION=1.14.0

ZENDNN_CONV_ADD_FUSION_ENABLE=0

ZENDNN_RESNET_STRIDES_OPT1_ENABLE=0

GOMP_CPU_AFFINITY=0-95

ZENDNN_CONV_CLIP_FUSION_ENABLE=0

ZENDNN_BN_RELU_FUSION_ENABLE=0

ZENDNN_CONV_ELU_FUSION_ENABLE=0

ORT_ZENDNN_ENABLE_INPLACE_CONCAT=0

ONNXRT_ZENDNN_CPU_ALLOC=0

ZENDNN_CONV_SWISH_FUSION_ENABLE=0

ZENDNN_QUANTIZE_CONV_RELU_FUSION_ENABLE=0

ZENDNN_QUANTIZE_CONV_ADD_FUSION_ENABLE=0

As mentioned in the section "Environment Variables", the script *scripts/zendnn_ONNXRT_env_setup.sh*, initializes all the environment variables except the one(s) which you must set manually. The environment variables **OMP_NUM_THREADS**, **OMP_WAIT_POLICY**, **OMP_PROC_BIND**, and **GOMP_CPU_AFFINITY** can be used to tune performance. For optimal performance, the **Batch Size** must be a multiple of the total number of cores (used by the threads). On a 4th Gen AMD EPYC server (configuration: AMD EPYC 9654P 96-Core, 2P, and **SMT=ON**) with the above environment variable values, **OMP_NUM_THREADS=96** and **GOMP_CPU_AFFINITY=0-95** yield the best throughput numbers for a single socket.

Batch Size is a sensitive factor for the throughput performance of any model. The following formula could be used to calculate the optimal **Batch Size**:

Batch Size = number_of_physical_cores * batch_factor

batch_factor may vary from 8-32. Usually, the value 32 gives the optimal performance.

3.7.3 Thread Wait Policy

OMP_WAIT_POLICY environment variable provides options to the OpenMP runtime library based on the expected behavior of the waiting threads. It can take the abstract values **PASSIVE** and **ACTIVE**. The default value is **ACTIVE**. When **OMP_WAIT_POLICY** is set to **PASSIVE**, the waiting threads will be passive and will not consume the processor cycles. Whereas, setting it to **ACTIVE** will consume processor cycles.

Note: *For ZenDNN stack, setting **OMP_WAIT_POLICY** to **ACTIVE** may give better performance.*

3.7.4 Thread Affinity

To improve ZenDNN performance, the behavior of OpenMP threads can be guarded precisely with thread affinity settings. A thread affinity defined at start up cannot be modified or changed during runtime of the application. Following are the ways through which you can bind the requested OpenMP threads to the physical CPUs:

- **GOMP_CPU_AFFINITY** environment variable binds threads to the physical CPUs, for example:

  export GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"

  This command will bind the:

  – Initial thread to CPU 0
  – Second thread to CPU 3
  – Third and fourth threads to CPU 1 and CPU 2 respectively
  – Fifth thread to CPU 4
  – Sixth through tenth threads to CPUs 6, 8, 10, 12, and 14 respectively

  Then, it will start the assigning back from the beginning of the list.

  export GOMP_CPU_AFFINITY="0" binds all the threads to CPU 0.

- KMP affinity belongs to LLVM OpenMP runtime library and is used by setting appropriate values for the environment variable **KMP_AFFINITY**. It has the following syntax:

  KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]

  Example:

  export KMP_AFFINITY='verbose,respect,granularity=fine,compact,1,0 binds threads as close as possible to the master thread but on a different core. Once each core is assigned with one OpenMP thread, the remaining OpenMP threads are assigned in the same order as before, but on different thread context.

  Among the two, **KMP_AFFINITY** takes highest precedence followed by **GOMP_CPU_AFFINITY**. If none of the two is set, the host system will defer the assignment of threads to CPUs. Given the same thread binding (see example below), it is expected that both the affinity settings would give the same performance.

  Example:

  Following affinity settings should give the same thread bindings:

  – **export GOMP_CPU_AFFINITY=0-95**
  – **export KMP_AFFINITY='verbose,respect,granularity=fine,compact,1,0'**

3.7.5 Non-uniform Memory Access

3.7.5.1 numactl

numactl provides options to run processes with specific scheduling and memory placement policy. It can restrict the memory binding and process scheduling to specific CPUs or NUMA nodes:

- `--cpunodebind=nodes`: Restricts the process to specific group of nodes.

- `--physcpubind=cpus`: Restricts the process to specific set of physical CPUs.
- `--membind=nodes`: Allocates the memory from the nodes listed. The allocation fails if there is not enough memory on the listed nodes.
- `--interleave=nodes`: Memory will be allocated in a round robin manner across the specified nodes. When the memory cannot be allocated on the current target node, it will fall back to the other nodes.

Example:

If <onnxruntime_script> is the application that needs to run on the server, then it can be triggered using `numactl` settings as follows:

```
numactl --cpunodebind=0-3 -membind=0-3 python <onnxruntime_script>
```

The `interleave` option of numactl works only when the number nodes allocated for a particular application is more than one. `cpunodebind` and `physcpubind` behave the same way for ZenDNN stack, whereas `interleave` memory allocation performs better than `membind`.

3.7.5.2 Concurrent Execution

As every application, AI workload requires special considerations during performance tuning to get the best out of the non-uniform memory access (NUMA) enabled machine. Improvement in performance can be achieved by carefully analyzing memory access time, memory bandwidth, and congestion on the shared bus. These factors depend on how far away the allocated memory and the process that requested the memory are in the NUMA system. In NUMA machines, the local memory access is faster as compared to the remote memory access. Consider the following workload:

```
numactl --cpunodebind=0-3 -membind=0-3 python <onnxruntime_script>
```

Performance can be optimized by partitioning the workload into multiple data shards and then running concurrently on more than one NUMA node. Following example shows the concurrent execution across 4 NUMA nodes:

```
numactl --cpunodebind=0 --membind=0 python <onnxruntime_script> & numactl --cpuno-
debind=1 --membind=1 python <onnxruntime_script> & numactl --cpunodebind=2 --membind=2
python <onnxruntime_script> & numactl --cpunodebind=3 --membind=3 python <onnxrun-
time_script>
```

The number of concurrent executions can be increased beyond 4 nodes. The following formula can be used to decide the number of concurrent executions to be triggered at a time:

```
Number Concurrent Executions = Number of Cores Per Socket / Numbers of Cores sharing L3
cache
```

This can also be extended to even cores. However, these details should be verified empirically.

3.7.6 Transparent Huge Pages

Transparent Huge Pages (THPs) are a Linux kernel feature for memory management to improve performance of the application by efficiently using processor's memory-mapping hardware. THP should reduce the overhead of the Translation Lookaside Buffer. User must login as root to enable or disable THP settings. It operates mainly in two modes:

- always: You can run the following command to set THP to 'always':

  ```
  echo always > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, the system kernel tries to assign huge pages to the processes running on the system.
- madvise: You can run the following command to set THP to 'madvise':

  ```
  echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, kernel only assigns huge pages to the individual processes memory areas.

You can use the following command to disable THP:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

It is recommended to use the following THP setting for better performance:

- CNN and NLP models - 'madvise'

Note: *For AMD UIF CNN models, a better performance is observed with 'always'. However, these details should be verified empirically.*

3.7.7 Batch Size

Batch Size is a sensitive factor for the throughput performance of any model. The following formula could be used to calculate the optimal Batch Size:

```
 Batch Size = number_of_physical_cores * batch_factor
```

Batch factor may vary from 8-32. The value 32 may provide optimal performance. However, user should verify this empirically.

3.7.8 Memory Allocators

Based on the model, if there is a requirement for a lot of dynamic memory allocations, a memory allocator can be selected from the available allocators which would generate the most optimal performance out of the model. These memory allocators override the system provided dynamic memory allocation routines and use a custom implementation. They also provide the flexibility to override the dynamic memory management specific tunable parameters (for example, logical page size, per thread, or per-cpu cache sizes) and environment variables. The default configuration of these allocators would work well in practice. However, you should verify empirically by trying out what setting works best for a particular model after analyzing the dynamic memory requirements for that model.

Most commonly used allocators are TCMalloc and jemalloc.

3.7.8.1 TCMalloc

TCMalloc is a memory allocator which is fast, performs uncontended allocation and deallocation for most objects. Objects are cached depending on the mode, either per-thread or per-logical CPU. Most allocations do not need to take locks. So, there is low contention and good scaling for multi-threaded applications. It has flexible use of memory and hence, freed memory can be reused for different object sizes or returned to the operating system. Also, it provides a variety of user-accessible controls that can be tuned based on the memory requirements of the workload.

3.7.8.2 jemalloc

jemalloc is a memory allocator that emphasizes fragmentation avoidance and scalable concurrency support. It has a powerful multi-core/multi-thread allocation capability. The more cores the CPU has, the more program threads, the faster jemalloc allocates. jemalloc classifies memory allocation granularity better, leading to less lock contention. It provides various tunable runtime options, such as enabling background threads for unused memory purging, allowing jemalloc to utilize transparent huge pages for its internal metadata, and so on.

3.7.8.3 Usage

You can install the TCMalloc/jemalloc dynamic library and use **LD_PRELOAD** environment variable as follows:

```
Before using TCMalloc:
export LD_PRELOAD=/path/to/TCMallocLib/

Before using jemalloc:
export LD_PRELOAD=/path/to/jemallocLib/

Or

Benchmarking command using TCMalloc:
LD_PRELOAD=/path/to/TCMallocLib/ < python benchmarking command>

Benchmarking command using jemalloc:
LD_PRELOAD=/path/to/jemallocLib/ < python benchmarking command>
```

To verify if TCMalloc/jemalloc memory allocator is in use, you can grep for tcmalloc/jemalloc in the output of `lsof` command:

```
lsof -p <pid_of_benchmarking_commad> | grep <tcmalloc/jemalloc>
```

3.7.9 Optimal Setting

Optimal performance of several ZenDNN workloads is observed when interleaving is enabled in conjunction with the NPS4 mode.

By default, ONNX Runtime uses GNU OpenMP (libgomp) for parallel computation. For ZenDNN backend, you can preload LLVM OpenMP's libomp for a better performance compared to libgomp as follows:

1. Download:
   ```
   cd $ZENDNN_PARENT_FOLDER
   ```
2. Unzip:
   ```
   tar -xf openmp-10.0.1.src.tar.xz
   cd openmp-10.0.1.src
   ```
3. Web get:
   ```
   wget https://github.com/llvm/llvm-project/releases/download/llvmorg-10.0.1/openmp-
   10.0.1.src.tar.xz
   ```
4. Configure cmake command:
   ```
   cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
   ```
5. Build command:
   ```
   make
   ```
6. Clean command:
   ```
   make clean
   ```
7. Build artifacts location:
   ```
   $ZENDNN_PARENT_FOLDER/openmp-10.0.1.src/runtime/src/libomp.so
   ```
8. Do LD_PRELOAD to the *.so* file:
   ```
   export LD_PRELOAD=$ZENDNN_PARENT_FOLDER/openmp-10.0.1.src/runtime/src/libomp.so:$LD_PRE-
   LOAD
   ```

This is a one-time activity and for all the benchmarking, you can just point to the already built lib.

A sample command line to run a Python code with 96C in NPS4 mode is as follows:

```
export GOMP_CPU_AFFINITY=0-95 && export OMP_NUM_THREADS=96 && numactl --cpunodebind=0-3 --
interleave=0-3 python -m onnxruntime.transformers.benchmark -m bert-large-uncased --mod-
el_class AutoModel -p fp32 -i 3 -t 10 -b 24 -s 16 -n 96 -v
```

3.8 Limited Precision Support

Quantization is an active area of research and a popular compression technique to accelerate neural network performance.

A few of these quantized neural networks models are publicly available. On AMD 4th Gen EPYC™ platforms, ZenDNN offers options to enable INT8 quantization with AMD's UIF INT8 models. These models can be leveraged using AMD UIF benchmarking scripts.

ZenDNN provides limited support for BF16 with the following CNN models:

- AlexNet
- GoogleNet
- ResNet variants
- SqueezeNet1.1
- VGG11

Chapter 4 PyTorch

4.1 Installing ZenDNN with PyTorch

Note: *Refer to the section "ZenDNN" before starting the installation.*

In this release, we are providing ZenDNN library support for PyTorch v1.13. This is a baseline release for PyTorch v1.13 with:

- FP32 support
- *AMD UIF* INT8 model support
- Limited support for BF16 on AMD UIF ResNet50

4.1.1 Binary Release Setup

4.1.1.1 Conda

Complete the following steps to setup Conda:

1. Refer to Anaconda documentation (*https://docs.anaconda.com/anaconda/install/linux/*) to install Anaconda on your system. The testing has been done with Anaconda3-2020.11-Linux-x86_64.
2. Create and activate a Conda environment which will house all the PyTorch-ZenDNN specific installations:

   ```
   conda create -n pt-v1.13-zendnn-v4.1-rel-env python=3.8 -y

   conda activate pt-v1.13-zendnn-v4.1-rel-env
   ```

   Ensure that you install the PyTorch-ZenDNN package corresponding to the Python version with which you created the Conda environment.

   If there is any conda environment named *pt-v1.13-zendnn-v4.1-rel-env*, delete it (using command `conda remove --name pt-v1.13-zendnn-v4.1-rel-env --all`) before running *scripts/PT_ZenDNN_setup_release.sh*.

   Note: *PyTorch-ZenDNN is compatible with Python v3.7-3.10 but 3.8 has been used as an example.*

3. It is recommended to use the naming convention:

   ```
   pt-v1.13-zendnn-v4.1-rel-env
   ```

4. Install all the necessary dependencies:

   ```
   pip install --upgrade typing-extensions

   pip install --upgrade numpy==1.23.2
   ```

Note: *For binary packages built with Python v3.7, it is recommended to use numpy v1.21.6 (numpy==1.21.6).*

4.1.1.2 PyTorch v1.13

Complete the following steps to install the ZenDNN binary release:

1. Copy the zipped release package to the local system being used. The name of the release package will be similar to *PT_v1.13_ZenDNN_v4.1_Python_v3.8.zip*.
2. Execute the following commands:
   a. `unzip PT_v1.13_ZenDNN_v4.1_Python_v3.8.zip`
   b. `cd PT_v1.13_ZenDNN_v4.1_Python_v3.8/`
   c. `source scripts/PT_ZenDNN_setup_release.sh`
      This installs the PyTorch wheel package provided in the zip file.

      Note: *Ensure that it is sourced only from the folder PT_v1.13_ZenDNN_v4.1_Python_v3.8/.*
   d. To run the benchmarks with different CNN models at the PyTorch level, refer the section "PyTorch CNN Benchmarks".

The release binaries for PyTorch v1.13 are now compiled with manylinux2014 and they provide compatibility with some older Linux distributions.

For more information on the supported OS and compilers for the Python wheel file, refer to the section "Supported OS and Compilers".

C++ Interface will work on the following operating systems (with glibc version 2.31 or later):

- Ubuntu 22.04 and later
- RHEL 9.1 and later

4.1.2 Build from Source

To build ZenDNN with PyTorch pip package from source, download PyTorch-ZenDNN source code from:

https://github.com/amd/ZenDNN-pytorch

The repository defaults to the master development branch that does not have ZenDNN support. To build, you must check out the branch release/1.13_zendnn_rel.

For more information on building procedure, refer to *BUILD_SOURCE.md*.

4.2 Directory Structure

The release folder consists of a PyTorch wheel (.whl) and the following directory:

- *scripts/* contains scripts to install the wheel file and run benchmarks

4.3 High-level Overview

For more information, refer to the section "High-level Overview".

4.4 PyTorch CNN Benchmarks

The benchmark scripts provide performance benchmarking at the PyTorch level. It prints the latency and throughput results for the following torchvision supported models:

ResNet50, ResNet152, GoogLeNet, and VGG11

To install JEMalloc, complete the following steps:

1. Follow the steps on jemalloc installation (*https://github.com/jemalloc/jemalloc/blob/dev/INSTALL.md*).
2. Export following environment variables:

   ```
   export LD_PRELOAD=<Installation path>lib/libjemalloc.so

   export MALLOC_CONF="oversize_threshold:1,background_thread:true,meta-
   data_thp:auto,dirty_decay_ms:-1,muzzy_decay_ms:-1"
   ```

To install torchvision, execute the following command:

```
pip install torchvision==0.14.1+cpu --extra-index-url https://download.pytorch.org/whl/cpu
```

For latency, execute the following commands:

1. `cd PT_v1.13_ZenDNN_v4.1_Python_v3.8/`
2. `source scripts/zendnn_PT_env_setup.sh`
3. `conda activate pt-v1.13-zendnn-v4.1-rel-env`
4. `bash scripts/pt_cnn_benchmarks_latency.sh`

For throughput, execute the following commands:

1. `cd PT_v1.13_ZenDNN_v4.1_Python_v3.8/`
2. `source scripts/zendnn_PT_env_setup.sh`
3. `conda activate pt-v1.13-zendnn-v4.1-rel-env`
4. `bash scripts/pt_cnn_benchmarks_throughput.sh`

To run individual models rather than the entire suite, execute the following commands:

```
cd $ZENDNN_PARENT_FOLDER/scripts/

numactl --cpunodebind=<NPS> --interleave=<NPS> python pt_cnn_benchmarks.py --arch
<model_name> --batch_size $BATCH_SIZE --iterations $NUM_OF_BATCHES --warmups $WARMUP_-
SIZE
```

Replace `<NPS>` with the following based on your number of NUMA nodes. Execute the command `lscpu` to identify the number of NUMA nodes for your machine:

- If you have 1 NUMA node, replace `<NPS>` with 0
- If you have 2 NUMA nodes, replace `<NPS>` with 0-1
- If you have 4 NUMA nodes, replace `<NPS>` with 0-3

Replace `<model_name>` with one of the following options:

- For ResNet50, replace `<model_name>` with `resnet50`
- For ResNet152, replace `<model_name>` with `resnet152`
- For GoogLeNet, replace `<model_name>` with `googlenet`
- For VGG11, replace `<model_name>` with `vgg11`

While executing the commands, make a note of the following:

- For optimal settings, refer to the Tuning Guidelines section. Current setting refers to 96C, 2P, SMT=ON configuration.
- If a warning similar to the following appears during benchmark runs, configure your GOMP_CPU_AFFINITY setting to match the number of CPU cores supported by your machine:

  ```
  OMP: Warning #181: OMP_PROC_BIND: ignored because GOMP_CPU_AFFINITY is defined

  OMP: Warning #123: Ignoring invalid OS proc ID 48

  OMP: Warning #123: Ignoring invalid OS proc ID 49

  .

  .

  .

  OMP: Warning #123: Ignoring invalid OS proc ID 63
  ```

  For example, if your CPU has 24 cores, your GOMP_CPU_AFFINITY should be set as "export GOMP_CPU_AFFINITY=0-23".

- If a warnings similar to the following appear during the benchmark runs, they may be ignored:

  ```
  "../site-packages/torchvision/models/googlenet.py:212: UserWarning: Scripted Goo-
  gleNet always returns GoogleNetOutputs Tuplewarnings.warn("Scripted GoogleNet always
  returns GoogleNetOutputs Tuple")"
  ```

  ```
  "UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated
  since 0.13 and will be removed in 0.15" and "UserWarning: The parameter 'pretrained' is
  deprecated since 0.13 and will be removed in 0.15, please use 'weights' instead."
  ```

4.5 PyTorch v1.13

In this release of ZenDNN:

- ZenDNN library is supported for PyTorch v1.13.
- AMD Unified Inference Frontend (UIF) optimized models are supported. For the model details, refer to the UIF documentation.
- PyTorch v1.13 wheel file is compiled with GCC v9.3.1.
- PyTorch v1.13 is expected to deliver similar or better performance compared to PyTorch v1.12.

4.6 Environment Variables

ZenDNN uses the following environment variables to setup paths and control logs, tune performance:

Table 7. PyTorch-ZenDNN Environment Variables

Environment Variable	Default Value/User Defined Value
Generic (Setup paths and control logs)	
ZENDNN_LOG_OPTS	ALL:0
ZENDNN_PARENT_FOLDER	Path to unzipped release folder
ZENDNN_PRIMITIVE_CACHE_CAPACITY	The default value is set to 1024, you can modify it as required[a].
OMP_DYNAMIC	FALSE
Optimized (Tune performance)	
OMP_NUM_THREADS	The default value is set to 96. You can set it as per the number of cores in the user system[a].
OMP_WAIT_POLICY	ACTIVE
GOMP_CPU_AFFINITY	Set it as per the number of cores in the system being used. For example, use 0-95 for 96-core servers.
ZENDNN_GEMM_ALGO	The default value is 3. You can modify it to any of the following: • 1 = AOCL-BLIS path • 2 = Partial AOCL-BLIS • 3 = ZenDNN JIT path
ZENDNN_PT_CONV_ADD_FUSION_SAFE	The default value is set to 0. You can set it to 1 while running UIF INT8 models.

a. You must set these environment variables explicitly.

Note: *There are a few other environment variables that are initialized by the setup script, however these are not applicable for the binary release setup.*

When source *scripts/zendnn_PT_env_setup.sh* is invoked, the script initializes all the environment variables except the one(s) which must be set manually. The environment variable **ZENDNN_PARENT_FOLDER** is initialized relative to the unzipped release folder. To ensure that the paths are initialized correctly, it is important that the script is invoked from the unzipped release folder.

4.7 Tuning Guidelines

The hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. The details for the environment variables used on a 4th Gen AMD EPYC™ server to get the best performance numbers are as follows:

4.7.1 System

A system with the following specifications has been used:

Table 8. System Specification

Model name	4th Gen AMD EPYC™ 9654P 96-Core Processor
CPU MHz	Up to 3.7 GHz
No of Cores	96
1P/2P	1
SMT: Thread(s) per Core	2
Mem-Dims	12x64 GB

4.7.2 Environment Variables

The following environment variables have been used:

ZENDNN_LOG_OPTS=ALL:0

OMP_NUM_THREADS=96

OMP_WAIT_POLICY=ACTIVE

OMP_DYNAMIC=FALSE

ZENDNN_GEMM_ALGO=3

Note: *For latency case of NLP models, a better performance is observed with ZENDNN_GEMM_ALGO=1. However, these details should be verified empirically.*

ZENDNN_PARENT_FOLDER=/home/<user_id>/my_work

ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024

GOMP_CPU_AFFINITY=0-95

As mentioned in the section "Environment Variables", the script *scripts/zendnn_PT_env_setup.sh*, initializes all the environment variables except the one(s) which you must set manually. The environment variables **OMP_NUM_THREADS**, **OMP_WAIT_POLICY**, **OMP_PROC_BIND**, and **GOMP_CPU_AFFINITY** can be used to tune performance. For optimal performance, the **Batch Size** must be a multiple of the total number of cores (used by the threads). On a 4th Gen AMD EPYC™ server (configuration: AMD EPYC™ 9654P 96-Core, 2P, and **SMT=ON**) with the above environment variable values, **OMP_NUM_THREADS=96** and **GOMP_CPU_AFFINITY=0-95** yield the best throughput numbers.

4.7.3 Thread Wait Policy

OMP_WAIT_POLICY environment variable provides options to the OpenMP runtime library based on the expected behavior of the waiting threads. It can take the abstract values **PASSIVE** and **ACTIVE**. The default value is **ACTIVE**. When **OMP_WAIT_POLICY** is set to **PASSIVE**, the waiting threads will be passive and will not consume the processor cycles. Whereas, setting it to **ACTIVE** will consume processor cycles.

Note: *For ZenDNN stack, setting **OMP_WAIT_POLICY** to **ACTIVE** may give better performance.*

4.7.4 Thread Affinity

To improve ZenDNN performance, the behavior of OpenMP threads can be guarded precisely with thread affinity settings. A thread affinity defined at start up cannot be modified or changed during runtime of the application. Following are the ways through which you can bind the requested OpenMP threads to the physical CPUs:

GOMP_CPU_AFFINITY environment variable binds threads to the physical CPUs, for example:

export GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"

This command will bind the:

- Initial thread to CPU 0
- Second thread to CPU 3
- Third and fourth threads to CPU 1 and CPU 2 respectively
- Fifth thread to CPU 4
- Sixth through tenth threads to CPUs 6, 8, 10, 12, and 14 respectively

Then, it will start the assigning back from the beginning of the list.

export GOMP_CPU_AFFINITY="0" binds all the threads to CPU 0.

Example:

Following affinity settings should give the same thread bindings:

export GOMP_CPU_AFFINITY=0-95

4.7.5 Non-uniform Memory Access

4.7.5.1 numactl

numactl provides options to run processes with specific scheduling and memory placement policy. It can restrict the memory binding and process scheduling to specific CPUs or NUMA nodes:

- `--cpunodebind=nodes`: Restricts the process to specific group of nodes.
- `--physcpubind=cpus`: Restricts the process to specific set of physical CPUs.
- `--membind=nodes`: Allocates the memory from the nodes listed. The allocation fails if there is not enough memory on the listed nodes.
- `--interleave=nodes`: Memory will be allocated in a round robin manner across the specified nodes. When the memory cannot be allocated on the current target node, it will fall back to the other nodes.

Example:

If <pytorch_script> is the application that needs to run on the server, then it can be triggered using `numactl` settings as follows:

```
numactl --cpunodebind=0-3 -interleave=0-3 python <pytorch_script>
```

The `interleave` option of numactl works only when the number nodes allocated for a particular application is more than one. `cpunodebind` and `physcpubind` behave the same way for ZenDNN stack, whereas `interleave` memory allocation performs better than `membind`.

The number of concurrent executions can be increased beyond 4 nodes. The following formula can be used to decide the number of concurrent executions to be triggered at a time:

```
Number Concurrent Executions = Number of Cores Per Socket / Numbers of Cores sharing L3
cache
```

This can also be extended to even cores. However, these details should be verified by the user empirically.

4.7.6 Transparent Huge Pages

Transparent Huge Pages (THPs) are a Linux kernel feature for memory management to improve performance of the application by efficiently using processor's memory-mapping hardware. THP should reduce the overhead of the Translation Lookaside Buffer. User must login as root to enable or disable THP settings. It operates mainly in two modes:

- always: You can run the following command to set THP to ‘always’:

  ```
  echo always > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, the system kernel tries to assign huge pages to the processes running on the system.

- madvise: You can run the following command to set THP to ‘madvise’:

  ```
  echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  In this mode, kernel only assigns huge pages to the individual processes memory areas.

You can use the following command to disable THP:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

It is recommended to use the following THP setting for better performance:

- CNN models - ‘always’
- NLP models - ‘madvise’

4.7.7 Memory Allocators

Based on the model, if there is a requirement for a lot of dynamic memory allocations, a memory allocator can be selected from the available allocators which would generate the most optimal performance out of the model. These memory allocators override the system provided dynamic memory allocation routines and use a custom implementation. They also provide the flexibility to override the dynamic memory management specific tunable parameters (for example, logical page size, per thread, or per-cpu cache sizes) and environment variables. The default configuration of these allocators would work well in practice. However, you should verify empirically by trying out what setting works best for a particular model after analyzing the dynamic memory requirements for that model.

Most commonly used allocator is jemalloc.

4.7.7.1 jemalloc

jemalloc is a memory allocator that emphasizes fragmentation avoidance and scalable concurrency support. It has a powerful multi-core/multi-thread allocation capability. The more cores the CPU has, the more program threads, the faster jemalloc allocates. jemalloc classifies memory allocation granularity better, leading to less lock contention. It provides various tunable runtime options, such as enabling background threads for unused memory purging, allowing jemalloc to utilize transparent huge pages for its internal metadata, and so on.

4.7.7.2 Usage

You can install the jemalloc dynamic library and use **LD_PRELOAD** environment variable as follows:

```
Before using jemalloc:
export LD_PRELOAD=/path/to/jemallocLib/

Benchmarking command using jemalloc:
LD_PRELOAD=/path/to/jemallocLib/ <python benchmarking command>
```

To verify if jemalloc memory allocator is in use, you can grep for jemalloc in the output of `lsof` command:

```
lsof -p <pid_of_benchmarking_commad> | grep <jemalloc>
```

4.8 Limited Precision Support

Quantization is an active area of research and a popular compression technique to accelerate neural network performance.

A few of these quantized neural networks models are publicly available. On AMD 4th Gen EPYC™ platforms, ZenDNN offers options to enable INT8 quantization with AMD's UIF INT8 models. These models can be leveraged using AMD UIF benchmarking scripts.

ZenDNN provides limited support for BF16 on AMD UIF ResNet50.

Chapter 5 ONNX Runtime Windows (Beta)

5.1 Installing ZenDNN with ONNX Runtime

Note: *Refer to the section "ZenDNN" before starting the installation.*

In this release, ZenDNN library is supported for ONNX Runtime v1.15.1. This is a baseline release for ONNX Runtime v1.15.1 with:

- FP32 support
- *AMD UIF* INT8 model support
- Limited support for BF16 on a few CNN models
- WinML application support

5.1.1 Binary Release Setup

5.1.1.1 Conda

Complete the following steps to setup Conda:

1. Refer to Anaconda documentation (*https://docs.anaconda.com/anaconda/install/windows/*) to install Anaconda on your system. The testing has been done with Anaconda v4.8.3.
2. Install visual C++ Redistributable for Windows. It installs Microsoft C and C++ (MSVC) runtime libraries. These libraries are required by many applications built by using Microsoft C and C++ tools.
3. Create and activate a Conda environment which will house all the ONNX Runtime-ZenDNN specific installations:

   ```
   conda create -n onnxrt-v1.15.1-zendnn-v4.1-rel-env python=3.8 -y

   conda activate onnxrt-v1.15.1-zendnn-v4.1-rel-env
   ```

   Ensure that you install the ONNX Runtime-ZenDNN package corresponding to the Python version with which you created the Conda environment.

   If there is any conda environment named *onnxrt-v1.15.1-zendnn-v4.1-rel-env* already present, delete the conda environment *onnxrt-v1.15.1-zendnn-v4.1-rel-env* (using command `conda remove --name onnxrt-v1.15.1-zendnn-v4.1-rel-env --all`).

   Note: *ONNX Runtime-ZenDNN is compatible with Python v3.8-3.11 but 3.8 has been used as an example.*

4. It is recommended to use the naming convention:

   ```
   onnxrt-v1.15.1-zendnn-v4.1-rel-env
   ```

5. Install all the necessary dependencies:

```
pip install -U cmake numpy pytest psutil torch coloredlogs

pip install -U transformers sympy --ignore-installed ruamel.yaml

pip install onnx==1.14.0
```

6. Download AOCL-BLIS from AMD Developer Central (*https://developer.amd.com/amd-aocl/*).
7. Add BLIS path to the environment variable "Path". For example, *C:\amd-blis\lib\ILP64*.
8. Download and install LLVM (Windows 64-bit) for *libomp.dll* (OpenMP: used for parallel programming) from GitHub (*https://github.com/llvm/llvm-project/releases/tag/llvmorg-14.0.6*).
9. Add *libomp.dll*, *libiomp5d.dll* path to the environment variable "Path". For example, *C:\Program Files\LLVM\lib*.

5.1.1.2 ONNX Runtime v1.15.1

Complete the following steps to install the ZenDNN binary release:

1. Copy the zipped release package to the local system being used. The name of the release package will be similar to *ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8_Win.zip*.
2. Execute the following commands:
   a. `Extract ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8_Win.zip`
   b. `ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8_Win/`
   c. `call scripts/zendnn_ONNXRT_env_setup_win.bat`
      This script will set up the required environment to run ONNX Runtime in optimal mode.
   d. `python -m pip install <whlfile.whl>`
   e. `pip install protobuf==3.20.2`

   Notes:

   1. Ensure that it is sourced only from the unzipped release folder.

   2. If there is any conda environment named onnxrt-v1.15.1-zendnn-v4.1-rel-env already present, delete the conda environment onnxrt-v1.15.1-zendnn-v4.1-rel-env (using command conda remove --name onnxrt-v1.15.1-zendnn-v4.1-rel-env --all) before running scripts/zendnn_ONNXRT_env_setup_win.bat.

The Python release binaries are tested with the recent Windows releases, such as Windows 10 and 11.

C++ Interface will work on operating systems, such as Windows 10 and 11.

5.1.1.3 Building WinML Application

To build a WinML application, complete the following steps:

1. For building ONNXRT, execute the command `enable --use_winml --skip_winml_tests` along with other arguments for the *.bat* file.

2. For building WinML application, install *Microsoft.AI.MachineLearning* package for the project *vcxproj*.
3. Change the import statements from `Windows.AI.MachineLearning` to `Microsoft.AI.MachineLearning` in the *.cpp* files.
4. Build with debug mode.

5.1.2 Build from Source

To build ZenDNN with ONNX Runtime pip package from source, download the ONNX Runtime-ZenDNN source code from:

https://github.com/amd/ZenDNN-onnxruntime

The repository defaults to the master development branch which does not have ZenDNN support. To build, you must check out the release branch **rel-1.15.1_zendnn_rel**.

For more information on the building procedure, refer to *BUILD_SOURCE.md*.

5.2 Directory Structure

The release folder consists of a ONNX Runtime wheel (.whl) and the following directory:

- *scripts/* contains scripts to set up the environment

5.3 High-level Overview

For more information, refer to the section "High-level Overview".

5.4 ONNX Runtime Benchmarks

The benchmark scripts provide performance benchmarking at the ONNX Runtime level, printing latency and throughput results for BERT models.

To use the benchmarking scripts, execute the following commands:

1. `ONNXRT_v1.15.1_ZenDNN_v4.1_Python_v3.8_Win/`
2. `call scripts/zendnn_ONNXRT_env_setup_win.bat`
3. Activate your conda environment.
4. The following command runs BERT benchmarking with batch-size=4, sequence length=16, and threads=64:

```
set ORT_ZENDNN_SUBGRAPH=0 && set OMP_NUM_THREADS=64 && python -m onnxruntime.transform-
ers.benchmark -s 16 -b 4 -t 5 -n 64 -m bert-large-uncased --provider zendnn
```

5.5 Environment Variables

ZenDNN uses the following environment variables to setup paths and control logs, tune performance:

Table 9. ONNX Runtime-ZenDNN Windows Environment Variables

Environment Variable	Default Value/User Defined Value
Generic (Setup paths and control logs)	
ZENDNN_LOG_OPTS	ALL:0
ZENDNN_PARENT_FOLDER	Path to unzipped release folder
ZENDNN_PRIMITIVE_CACHE_CAPACITY	The default value is set to 1024, you can modify it as required[a].
OMP_DYNAMIC	FALSE
Optimized (Tune performance)	
OMP_NUM_THREADS	The default value is set to 64. You can set it as per the number of cores in the user system[a].
OMP_WAIT_POLICY	ACTIVE
OMP_PROC_BIND	FALSE
ZENDNN_CONV_ADD_FUSION_ENABLE	The flag is to enable convolution and add operator fusion. It is disabled (set to 0) by default. You can modify it to 1 to enable the fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_RESNET_STRIDES_OPT1_ENABLE	The flag is to enable strides trick optimization for ResNet blocks. It is disabled (set to 0) by default. You can modify it to 1 to enable the optimization. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_BN_RELU_FUSION_ENABLE	This flag is disabled by default. You can use set command (with value 1) in Windows to enable it. It is used to optimize executions on limited CNN models.
ZENDNN_CONV_CLIP_FUSION_ENABLE	This flag is disabled by default. You can use set command (with value 1) in Windows to enable it. It is used to optimize executions on all the variants of MobileNet models.
ZENDNN_CONV_RELU_FUSION_ENABLE	The flag is to enable convolution and relu operator fusion. It is enabled (set to 1) by default. You can modify it to 0 to disable the fusion. It is used to optimize executions on all the variants of ResNet models.

Table 9. ONNX Runtime-ZenDNN Windows Environment Variables

Environment Variable	Default Value/User Defined Value
ZENDNN_CONV_ELU_FUSION_ENABLE	The flag is to enable convolution and elu operator fusion. It is disabled (set to 0) by default. You can modify it to 1 to enable the fusion. It is used to optimize executions on limited CNN models.
ORT_ZENDNN_ENABLE_INPLACE_CONCAT	This flag is used to perform in place concatenation of intermediate tensors arrays. It is disabled (set to 0) by default. You can modify it to 1 to enable the optimization. This optimization is useful for the variants of Inception and GoogleNet models.
ZENDNN_GEMM_ALGO	The default value is 4. You can modify it to one of the following: • 1 = AOCL-BLIS path • 2 = Partial AOCL-BLIS • 3 = ZenDNN JIT path • 4 = ZenDNN partial JIT path
ONNXRT_ZENDNN_CPU_ALLOC	This flag is to enable the usage of the CPU memory allocator in ZenDNN Execution Provider. By default, it is disabled.
ZENDNN_CONV_SWISH_FUSION_ENABLE	This flag is to enable the fusion of the sigmoid operator with the preceding conv operator. By default, it is disabled.
ZENDNN_QUANTIZE_CONV_ADD_FUSION_E NABLE	This flag is to enable convolution and add operator fusion in quantized models with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_QUANTIZE_CONV_RELU_FUSION_E NABLE	The flag is to enable convolution and relu operator fusion in quantized model with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of ResNet models.
ZENDNN_QCONV_CLIP_FUSION_ENABLE	This flag is to enable convolution and clip operator fusion in the quantize model with QOperator format. It is disabled (set to 0) by default. You can modify it to 1 to enable fusion. It is used to optimize executions on all the variants of MobileNet models.
ZENDNN_ONNXRT_ENABLE_BF16_SUPPORT	This flag is disabled (set to 0) by default, you can set it to 1 to enable the execution of compute expensive operation in BF16. It optimizes the execution only for the CNN models.

a. These environment variables work only for Blocked Format.

There are a few other environment variables that are initialized by the setup script, however these are not applicable for the binary release setup.

When `call scripts/zendnn_ONNXRT_env_setup_win.bat` is invoked, the script initializes all the environment variables except the one(s) which must be set manually. The environment variable **ZENDNN_PARENT_FOLDER** is initialized relative to the path defined by the unzipped release folder. To ensure that the paths are initialized correctly, it is important that the script is invoked from the unzipped release folder.

5.5.1 Tuning Guidelines

The hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. The details for the environment variables used on a 4th Gen AMD Ryzen™ Threadripper™ to get the best performance numbers are as follows:

5.5.2 System

A system with the following specifications has been used:

Table 10. System Specification

Processor	AMD Ryzen™ Threadripper™ PRO 3995WX
RAM	512 GB
Socket	1
Physical Core	64
SMT: Thread(s) per Core	2
ONNXRT Version	1.15.1
ZenDNN Version	4.1

Note: *Optimal performance is observed on this system. ONNX Runtime-ZenDNN (Windows) library can be used on AMD 'Zen3' and AMD 'Zen4' processors including AMD Ryzen Notebook Series.*

5.5.3 Environment Variables

The following environment variables have been used:

ZENDNN_LOG_OPTS=ALL:0

OMP_NUM_THREADS=64

OMP_WAIT_POLICY=ACTIVE

OMP_PROC_BIND=FALSE

OMP_DYNAMIC=FALSE

ZENDNN_GEMM_ALGO=4

ZENDNN_PARENT_FOLDER=/home/<user_id>/my_work

BENCHMARKS_GIT_ROOT=/home/<user_id>/my_work/benchmarks

ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024

ZENDNN_ONNXRT_VERSION=1.15.1

ZENDNN_ONNX_VERSION=1.14.0

ZENDNN_CONV_ADD_FUSION_ENABLE=0

ZENDNN_RESNET_STRIDES_OPT1_ENABLE=0

ONNXRT_ZENDNN_CPU_ALLOC=0

ZENDNN_CONV_SWISH_FUSION_ENABLE=0

ZENDNN_QUANTIZE_CONV_ADD_FUSION_ENABLE=0

ZENDNN_QUANTIZE_CONV_RELU_FUSION_ENABLE=0

As mentioned in the section "Environment Variables", the script *scripts/zendnn_ONNXRT_env_setup_win.bat*, initializes all the environment variables except the one(s) which you must set manually. The environment variables **OMP_NUM_THREADS**, **OMP_WAIT_POLICY** and **OMP_PROC_BIND** can be used to tune performance. For optimal performance, the **Batch Size** must be a multiple of the total number of cores (used by the threads). On a 3rd Gen AMD Ryzen™ Threadripper™ workstation (configuration: AMD Ryzen™ Threadripper™ PRO 3995WX, 1P and **SMT=ON**) with the above environment variable values, **OMP_NUM_THREADS=64** yield the best throughput numbers for a single socket.

KMP_DUPLICATE_LIB_OK=TRUE is used to load multiple libomp instances.

Batch Size is a sensitive factor for the throughput performance of any model. The following formula could be used to calculate the optimal **Batch Size**:

Batch Size = number_of_physical_cores * batch_factor

batch_factor may vary from 8-32. Usually, the value 32 gives the optimal performance.

5.5.4 Optimal Setting

By default, ONNX Runtime uses Visual Studio OpenMP (libomp) for parallel computation. For ZenDNN backend, you can download LLVM OpenMP's libomp for better performance.

You can download:

- LLVM from GitHub (*https://github.com/llvm/llvm-project/releases/tag/llvmorg-14.0.6*)
- Visual Studio from Microsoft website (*https://learn.microsoft.com/en-us/visualstudio/releases/2019/history*).

5.6 Limited Precision Support

Quantization is an active area of research and a popular compression technique to accelerate neural network performance.

A few of these quantized neural networks models are publicly available. On AMD 4th Gen EPYC™ platforms, ZenDNN offers options to enable INT8 quantization with AMD's UIF INT8 models. These models can be leveraged using AMD UIF benchmarking scripts.

ZenDNN provides limited support for BF16 with the following CNN models:

- AlexNet
- GoogleNet
- ResNet variants
- SqueezeNet1.1
- VGG11