ZenDNN User Guide

Publication Number: 57300

Date: May 2024

Revision Number: 4.2

AMD

Table of Contents

AMD

Legal Notices

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

Trademarks

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Dolby is a trademark of Dolby Laboratories.

ENERGY STAR is a registered trademark of the U.S. Environmental Protection Agency.

HDMI is a trademark of HDMI Licensing, LLC.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Microsoft, Windows, Windows Vista, and DirectX are registered trademarks of Microsoft Corporation.

MMX is a trademark of Intel Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos.

PCIe is a registered trademark of PCI-Special Interest Group (PCI-SIG).

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Dolby Laboratories, Inc.

Manufactured under license from Dolby Laboratories.

Rovi Corporation

This device is protected by U.S. patents and other intellectual property rights. The use of Rovi Corporation's copy protection technology in the device must be authorized by Rovi Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Rovi Corporation.

Reverse engineering or disassembly is prohibited.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG-2 STANDARD IS EXPRESSLY PROHIBITED WITHOUT A LICENSE UNDER APPLICABLE PATENTS IN THE MPEG-2 PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111.

AMD

Revision History

A summary of the revisions made to this document.

Table 1: Revision History

Version Number	Date	Description
1	28-May-2024	Ported document to the new template, rearranged sections, and updated content for the 4.2 release.
2	30-May-2024	Fixed formatting issues

1 ZenDNN

AMD is unveiling a game-changing upgrade to ZenDNN with version 4.2, introducing a cutting-edge plug-in mechanism and an enhanced architecture under the hood. This isn't just about extensions; ZenDNN's aggressive AMD-specific optimizations operate at every level. It delves into comprehensive graph optimizations including pattern identification, graph reordering, and seeking opportunities for graph fusions to boost diverse AI workloads including vision, natural language processing (NLP), and recommender systems.

At the operator level, ZenDNN offers enhancements with microkernels, mempool optimizations, and efficient multi-threading on the large number of AMD EPYC™ cores. Microkernel optimizations further exploit all possible low-level math libraries, including AOCL BLIS.

The result? Enhanced performance with respect to the vanilla frameworks. Beyond its powerful optimizations, the ZenDNN plug-ins offer broad compatibility, seamlessly integrating with popular frameworks like TensorFlow and PyTorch. The ZenDNN plug-ins are in "Early Access" mode in the 4.2 release. For ONNX Runtime, ZenDNN is directly integrated.

Scope

The ZenDNN library and plug-ins have been developed to enable Deep Learning inference on AMD EPYC™ CPUs. The ZenDNN 4.2 release introduces plug-ins to TensorFlow and PyTorch, and 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.

Release Highlights

The highlights of this release are as follows:

- The ZenDNN library is based on oneDNN v2.6.3, and is not currently set up to be used independently.
- The ZenDNN library can be used in the following frameworks through a plug-in:
  - TensorFlow v2.16 and later
  - PyTorch v2.0 and later
- The ZenDNN library is integrated with ONNX Runtime v1.17.0.

In this document, we refer to the ZenDNN plug-in for TensorFlow as zentf, and the ZenDNN plug-in for PyTorch as zentorch.

- Wheel Files
  - zentorch wheel files (*.whl) have been generated using:
    - Python v3.8-v3.11
    - PyTorch v2.1.2
  - zentf wheel files (*.whl) have been generated using:
    - Python v3.9-v3.12
    - TensorFlow v2.16
  - ZenDNN-ONNX Runtime stack wheel files (*.whl) have been generated using:
    - Python v3.8-v3.11
    - ONNX Runtime v1.17.0
- Environment Variables for Tuning Performance

  The following environment variables have been added to tune performance:
  - Memory Pooling (Persistent Memory Caching)
    - `ZENDNN_ENABLE_MEMPOOL` for all TensorFlow models
    - Added MEMPOOL support for BF16 models in TensorFlow models
  - Convolution Operation
    - `ZENDNN_CONV_ALGO` for all TensorFlow models
  - Added new options to `ALGO` paths
    - Matrix Multiplication Operation:
      - `ZENDNN_MATMUL_ALGO` for TensorFlow, PyTorch, and ONNX Runtime models
      - Added new options, ALGO paths, and an experimental version of auto-tuner for TensorFlow
- Embedding Bag and Embedding Operators
  - Support for Embedding operator
  - AVX512 support for Embedding and Embedding Bag kernel
  - Two new parallelization strategies for Embedding and Embedding bag operators, namely, Table threading and Hierarchical threading
- Matrix Multiplication (MatMul) Operators
  - MatMul post-ops computation with BLIS kernels
  - Weight caching for FP32 JIT and BLIS kernels
  - BLIS BF16 kernel support

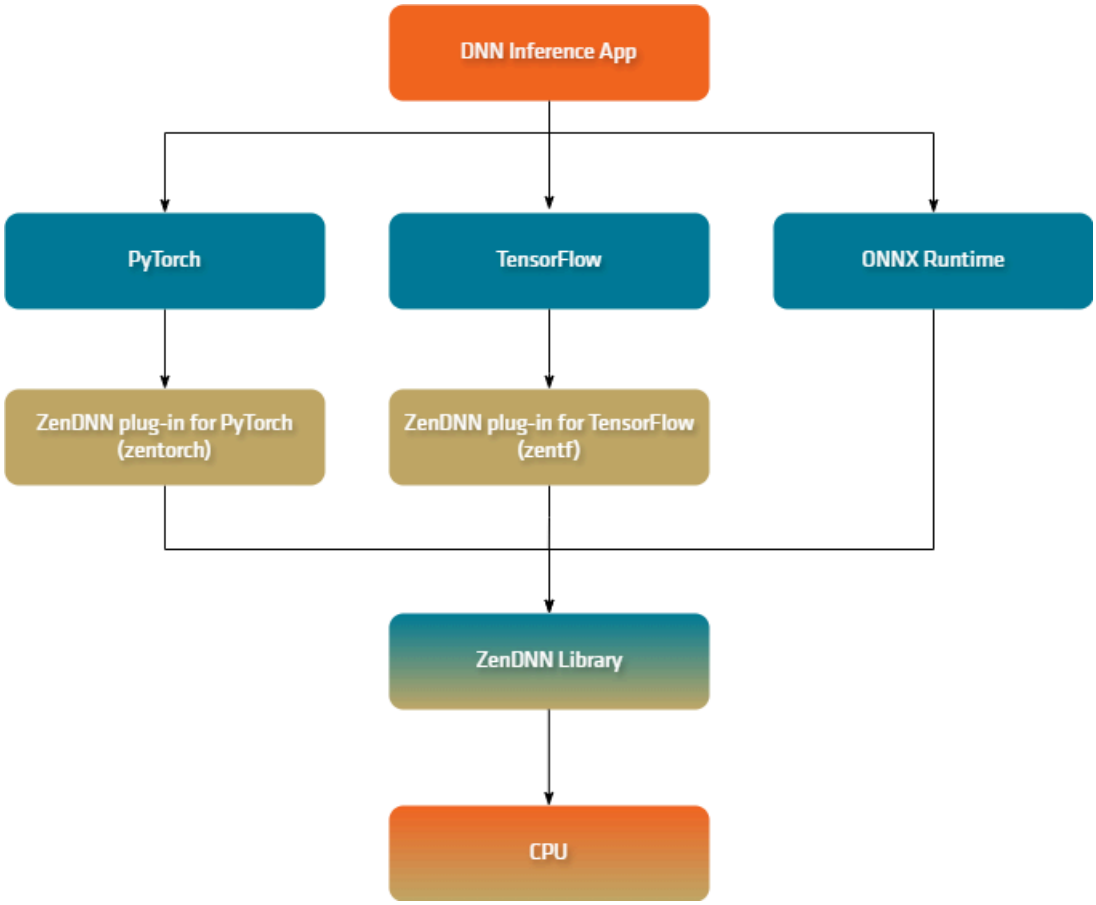
AOCL-LibM has been removed. The AOCL-BLIS Library is now used to compute mathematical functions such as `tanh`, `erf`, `gelu`, and so on.

For the latest information on the ZenDNN release and installers, visit AMD Developer Central.

High-level Overview

This high-level block diagram of the ZenDNN inference stack depicts how the ZenDNN library interfaces with the ZenDNN plug-in for PyTorch (zentorch), ZenDNN plug-in for TensorFlow (zentf), and the ONNX Runtime direct integration. The ZenDNN library uses the AOCL-BLIS library internally.

Figure 1.1: ZenDNN Software Stack



Supported OS

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

- Ubuntu® 22.04 LTS and later
- Red Hat® Enterprise Linux® (RHEL) 9.2 and later
- CentOS Stream 8.4

- SUSE Linux Enterprise Server (SLES) 15 SP5
- Anolis OS 8.8 for zentorch v4.2 wheel files

Build from Source

This GitHub page provides instructions to install the ZenDNN software stack using the **Build from Source** option.

2 PyTorch

The ZenDNN plug-in for PyTorch (zentorch) enables inference optimizations for deep learning workloads on AMD EPYC™ CPUs. It uses the ZenDNN library, which contains deep learning operators tailored for high performance on AMD EPYC™ CPUs. The zentorch extension to PyTorch has been developed to leverage the `torch.compile` graph compilation flow, and all optimizations can be enabled by a call to `torch.compile` with zentorch as the backend. Multiple passes of graph level optimizations run on the `torch.fx` graph and provide further performance acceleration.

Release Highlights

zentorch is compatible with base versions of PyTorch v2.0 or later. This release provides zentorch for PyTorch v2.1.2. This baseline release of the plug-in supports:

- Datatypes FP32 and BF16
- Auto mixed precision (AMP) between FP32 and BF16 providing a performance improvement with minimal changes in accuracy

Install ZenDNN Plug-in for PyTorch (zentorch)

Use either the **Binary Release** or **Build from Source** option to install zentorch.

Using the Release Binary

To install zentorch, you must create and activate a Conda environment, and then install the zentorch Release Binary.

Install the Release Binary

Create and Setup Conda Environment

Before you begin:

- The recommended naming convention is: `zentorch-v4.2.0-zendnn-v4.2-rel-env`
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `zentorch-v4.2.0-zendnn-v4.2-rel-env` exists, use the command `conda remove --name zentorch-v4.2.0-zendnn-v4.2-rel-env --all` to remove it.

Important zentorch is compatible with Python v3.8-3.11. Make sure you create a Conda environment only with Python versions supported by zentorch.

Conda Environment Setup

To setup a Conda environment:

1. Refer to the Anaconda documentation available here to install Anaconda on your system. Testing has been performed with *Anaconda3-2020.11-Linux- x86_64*.
2. Create and activate a Conda environment that houses all the zentorch specific installations.
```
conda create -n zentorch-v4.2.0-zendnn-v4.2-rel-env python=3.8 -y
conda activate zentorch-v4.2.0-zendnn-v4.2-rel-env
```

Install zentorch

For more information on the supported OS for the Python wheel files, refer to the "Supported OS" section.

The release binaries for zentorch are compiled with manylinux2014 and they provide compatibility with some older Linux distributions.

The release Python binaries have been tested with the recent Linux distributions such as:

- Ubuntu 22.04 and later
- RHEL 9.2 and later

To install the zentorch release binary:

1. Activate plug-in environment.
```
conda activate zentorch-v4.2.0-zendnn-v4.2-rel-env
```
2. Install PyTorch v2.1.2.
```
conda install pytorch==2.1.2 cpuonly -c pytorch
```
3. Choose one of the following two methods to install zentorch.
   a. Using the pip utility:
```
pip install zentorch==4.2.0
```
   b. Using the release package:
      1) Download the package from the AMD developer portal.
      2) Run the following commands to unzip the package and install the binary:
```
unzip ZENTORCH_v4.2.0_Python_v3.8.zip
cd ZENTORCH_v4.2.0_Python_v3.8/
pip install zentorch-4.2.0-cp38-cp38-manylinux2014_x86_64.whl
```

> While importing zentorch, if you get an undefined symbol error such as: *ImportError: <anaconda_install_path>/envs/<your-env>/lib/python3.x/site-packages/ zentorch/ _C.cpython-3x-x86_64-linux-gnu.so : undefined symbol: <some string>*, it could be due to version differences with PyTorch. If you are using a PyTorch version other than v2.1.2, use the **Build from Source** option to install.

Build from Source

To build the zentorch pip package from source:

1. Clone the repository and check out the r4.2 branch.

```
git clone https://github.com/amd/ZenDNN-pytorch-plugin.git
cd ZenDNN-pytorch-plugin/
```

2. Follow instructions provided here to configure, build, and install zentorch.
3. After the build is successful, the wheel file will be generated in the folder: *./dist/zentorch-*.whl*.

Usage

The custom zentorch backend can be called through `torch.compile`.

Using torch.compile

This section specifies the flow and commands for using zentorch in the `torch.compile` flow.

```
import torch
import zentorch
from torchvision import models
model = models.__dict__ ['resnet50'](pretrained=True).eval()
compiled_model = torch.compile(model, backend='zentorch', dynamic = False)
with torch.no_grad():
    output = compiled_model(input)
```

Recommendations

It is recommended you use `torch.no_grad()` for optimal inference performance with zentorch.

CNN

For torchvision CNN models, set `dynamic=False` when calling for `torch.compile` as follows:

```
model = torch.compile(model, backend='zentorch', dynamic=False)
with torch.no_grad():
    output = model(input)
```

NLP & RecSys

Optimize Hugging Face NLP models as follows.

```
model = torch.compile(model, backend='zentorch')
with torch.no_grad():
    output = model(input)
```

LLM

Optimize Hugging Face LLM models as follows.

- If output is generated through a call to the direct model, optimize it as follows:
  ```
  model = torch.compile(model, backend='zentorch')
  with torch.no_grad():
      output = model(input)
  ```
- If output is generated through a call to `model.forward`, optimize it as follows:
  ```
  model.forward = torch.compile(model.forward, backend='zentorch')
  with torch.no_grad():
      output = model.forward(input)
  ```

- If output is generated through a call to `model.generate`, optimize it as follows:
  - Optimize the `model.forward` with `torch.compile` instead of `model.generate`
  - However, generate the output through a call to `model.generate`

```
model.forward = torch.compile(model.forward, backend='zentorch')
with torch.no_grad():
    output = model.generate(input)
```

If the same model is optimized with `torch.compile` for multiple backends within a single script, it is recommended you use `torch._dynamo.reset()` before calling the `torch.compile` on that model.

Limited Precision Support

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

zentorch provides support for BF16 models through casting and AMP.

3 TensorFlow

TensorFlow provides a PluggableDevice mechanism that enables modular, plug-and-play integration of device-specific code.

AMD adopted PluggableDevice when developing the zentf for inference on AMD EPYC™ CPUs. zentf adds custom kernel implementations and operations specific to AMD EPYC™ CPUs to TensorFlow via its kernel and op registration C APIs.

zentf is a supplemental package to be installed alongside standard TensorFlow packages with TensorFlow version 2.16.1. From a TensorFlow developer's perspective, the zentf approach simplifies the process of leveraging ZenDNN optimizations.

This section provides instructions to setup zentf v4.2.

Release Highlights

This release of AMD's CPU solution for TensorFlow v2.16 provides a binary built with the PluggableDevice approach.

This zentf release:

- Supports TensorFlow v2.16.
- Integrates with ZenDNN v4.2 as the core inference library and compiled with GCC v10.2.
- Provides graph optimizations and layout pass directly integrated with the plug-in binary.
- Supports BF16 execution through auto-mixed precision.
- Supports fusions such as: the Conv2D + BatchNorm/BiasAdd + LeakyRelu fusion, BatchMatMul + Mul + Addv2 fusions, Dense keras layer forward fusion and FusedBatchNormEx optimization.
- Supports output buffer caching with mempool 3 path.
- Supports binary ops like Add, Sub, Mul, and SquaredDifference.
- Provides experimental support of C++ APIs.

The open source release of zentf is available here.

Install ZenDNN Plug-in for TensorFlow (zentf)

Use either the **Binary Release** or **Build from Source** option to install zentf.

Using the Release Binary

zentf can be set up with either Python or C++ interfaces.

Python Interface

Choose from one of three options to access the zentf binary release.

1. AMD developer portal (as a package). This release package consists of a zentf wheel file with a `.whl` extension and a `Scripts/` folder consisting of the environment setup script.
2. PyPI Repo as a wheel (.whl) file.
3. Community supported TensorFlow builds (as a .whl file)

C++ Interface

You can find the zentf C++ Interface package on the AMD developer portal.

Install the Release Binary

This section provides information required to install zentf v4.2 for a Python interface.

However, if you are interested in installing zentf v4.2 on a C++ interface, click here and refer to the instructions provided in *README.md*.

Compatibility

For more information on the supported OS for the Python wheel files, refer to the "Supported OS" section.

The release binaries for zentf are compiled with manylinux2014 and they provide compatibility with some older Linux distributions.

The release Python binaries and C++ binary (with glibc version 2.27 or later) have been tested with the recent Linux distributions such as:

- Ubuntu 22.04 and later
- RHEL 9.2 and later

Create and Setup Conda Environment

Before you begin:

- The recommended naming convention is: `tf-v2.16-zendnn-v4.2-rel-env`
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `tf-v2.16-zendnn-v4.2-rel-env` exists, use the command `conda remove --name tf-v2.16-zendnn-v4.2-rel-env --all` to remove it.

Important zentf is compatible with Python v3.9-3.12. Make sure you create a Conda environment only with Python versions supported by zentf.

To setup the Conda environment:

1. Refer to the Anaconda documentation available here to install Anaconda on your system. Testing has

been performed with *Anaconda3-2020.11-Linux- x86_64*.

2. Create and activate a Conda environment that houses all the zentf specific installations:

```
conda create -n tf-v2.16-zendnn-v4.2-rel-env python=3.10 -y
conda activate tf-v2.16-zendnn-v4.2-rel-env
```

Install zentf

To install the zentf binary release:

1. Install TensorFlow v2.16.
```
pip install tensorflow-cpu~=2.16
```
2. Use one of three methods to install zentf:
   a. **Using the release package from AMD developer portal**
      1) Download the package from AMD developer portal.
      2) Run the following commands to unzip the package:
```
unzip ZENTF_v4.2.0_Python_v3.10.zip
cd ZENTF_v4.2.0_Python_v3.10/
```
      > zentf is compatible with Python v3.9-3.12. We have used 3.10 here only as an example.

      3) Install the binary.
```
pip install zentf-4.2.0-cp310-cp310-manylinux2014_x86_64.whl
```
      4) To use the recommended environment settings, execute:
```
source scripts/zentf_env_setup.sh
```
   b. **Using the PyPI repo**
      1) Run the command:
```
pip install zentf==4.2.0
```
   c. **Using the wheel file from Community supported TensorFlow builds**
      1) Use this link to download the package for build type: "*Linux AMD ZenDNN Plug-in CPU Stable: TF 2.x*".
```
pip install zentf-4.2.0-cp310-cp310-manylinux2014_x86_64.whl
```

Setup zentf

Set the following environment variables to enable zentf for inference:

- `TF_ENABLE_ZENDNN_OPTS=1`
- `TF_ENABLE_ONEDNN_OPTS=0`

> **Important** By default, TensorFlow is shipped with oneDNN enabled. To disable ZenDNN optimizations and revert to the default TensorFlow setting, set `TF_ENABLE_ZENDNN_OPTS=0` and `TF_ENABLE_ONEDNN_OPTS=1`.

Build from Source

To install zentf using the **Build from Source** option:

1. Clone the repository and check out the r4.2 branch.
```
$ git clone https://github.com/amd/ZenDNN-tensorflow-plugin.git
$ cd ZenDNN-tensorflow-plugin/
```
2. Follow the steps to build and install from source given here to configure, build, and install zentf.

Limited Precision Support

zentf supports BF16 execution through Auto-mixed precision (AMP) optimization. To enable BF16 support, use the environment variable: `export TF_ZENDNN_PLUGIN_BF16=1`.

4 ONNX Runtime

The ZenDNN library is directly integrated with ONNX Runtime in the ZenDNN 4.2 release.

Release Highlights

This release supports:

- BF16 for CNN models
- ZenDNN library for ONNX Runtime v1.17.0
- Compilation of ONNX Runtime v1.17.0 wheel file with GCC v10.2

Installing ZenDNN with ONNX Runtime

In this release, ZenDNN library is supported for ONNX Runtime v1.17.0. This is a baseline release for ONNX Runtime v1.17.0 with support for:

- FP32
- BF16

The release folder consists of an ONNX Runtime wheel (.whl) and the `scripts/` directory, which contains scripts to set up the environment.

Using the Release Binary

To install ZenDNN with ONNX Runtime, you must create and activate a Conda environment and then install the ZenDNN Binary Release.

Install the Release Binary

Create and Setup Conda Environment

Before you begin:

- The recommended naming convention is: `onnxrt-v1.17.0-zendnn-v4.2-rel-env`
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `onnxrt-v1.17.0-zendnn-v4.2-rel-env` exists, use the command `conda remove --name onnxrt-v1.17.0-zendnn-v4.2-rel-env --all` before running `scripts/ONNXRT_ZenDNN_setup_release.sh`.
- **Important** ONNX Runtime-ZenDNN is compatible with Python v3.8-3.11. Make sure you create a Conda environment only with Python versions supported by ONNX Runtime-ZenDNN.

To set up the Conda environment:

1. Refer to the Anaconda documentation available here to install Anaconda on your system. Testing has been performed with *Anaconda3-2020.11-Linux- x86_64*.
2. Create and activate a Conda environment that houses all the ONNX Runtime-ZenDNN specific installations:
```
conda create -n onnxrt-v1.17.0-zendnn-4.2-rel-env python=3.8 -y
conda activate onnxrt-v1.17.0-zendnn-4.2-rel-env
```
3. Install all the necessary dependencies:
```
pip install -U cmake numpy pytest psutil torch==2.1.2 coloredlogs
pip install -U transformers==4.39.3 sympy --ignore-installed ruamel.yaml
pip install onnx==1.15.0
```

Install ONNX Runtime v1.17.0

For more information on the supported OS for the Python wheel files, refer to the "Supported OS" section.

The release binaries for ONNX Runtime are compiled with manylinux2014 and they provide compatibility with some older Linux distributions.

The release Python binaries and C++ binary (with glibc version 2.17 or later) have been tested with the recent Linux distributions such as:

- Ubuntu 22.04 and later
- RHEL 9.2 and later

To install the ZenDNN binary release:

1. Copy the zipped release package to the local system on which installation is being performed. The name of the release packages will be similar to `ONNXRT_v1.17.0_ZenDNN_v4.2_Python_v3.8/`.

   ONNX Runtime-ZenDNN is compatible with Python v3.8-3.11. We have used v3.8 here only as an example.

2. Execute the following commands.
   a. `unzip ONNXRT_v1.17.0_ZenDNN_v4.2_Python_v3.8.zip`
   b. `cd ONNXRT_v1.17.0_ZenDNN_v4.2_Python_v3.8/`
   c. `source scripts/ONNXRT_ZenDNN_setup_release.sh`
   d. `pip install protobuf==3.20.2`

Ensure that the script `ONNXRT_ZenDNN_setup_release.sh` is sourced only from the unzipped release folder.

Build from Source

To configure and build ZenDNN with ONNX Runtime pip package from source:

1. Download the ONNX Runtime-ZenDNN source code from here.
2. The repository defaults to the master development branch which does not have ZenDNN support. To build, you must check out the release branch `rel-1.17.0_zendnn_rel`.
3. For build instructions, refer to the *BUILD_SOURCE.md* file given here.

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_v1.17.0_ZenDNN_v4.2_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.benchmark -m bert-large-uncased --model_class AutoModel -p
fp32 -i 3 -t 10 -b 24 -s 16 -n 96 -v --provider zendnn
```

Current setting refers to a 96C, 1P, SMT=ON configuration. For information about optimal settings while you execute commands, refer to the Performance Tuning Guidelines section.

Limited Precision Support

ZenDNN ONNX Runtime provides support for BF16 with the following CNN models:

- MobileNet_v2
- Resnet50_v1
- Resnet101_v1
- Resnet152_v1
- VGG16
- VGG19

5 Performance Tuning

In this chapter, we discuss performance tuning of the ZenDNN software stack.

Environment Variables

In this section, we enumerate various environment variables to setup paths and control logs, and tune performance.

The settings given in the following table are used in the ZenDNN library and apply to zentorch, zentf, and ONNXRT direct integration.

Table 5.1: ZenDNN Environment Variables common to all frameworks

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. Set it based on the number of cores in the user system [a].
OMP_WAIT_POLICY	ACTIVE
GOMP_CPU_AFFINITY	Set it based on the number of cores in the system being used. For example, use 0-95 for 96-core servers.

Table 5.1: ZenDNN Environment Variables common to all frameworks

Environment Variable	Default Value/User Defined Value
ZENDNN_MATMUL_ALGO	The default value is FP32:4,BF16:3 For BF16: • 0 = Auto • 1 = AOCL_BLIS path (gcc 11.2 or later) • 2 = JIT path with weight caching • 3 = JIT path For FP32: • 0 = Auto • 1 = AOCL-BLIS path (CBLAS based) • 2 = AOCL-BLIS with ZenDNN parallelism • 3 = JIT path with weight caching • 4 = JIT path • 5 = AOCL_BLIS with weight caching Auto is an experimental feature and should be used with application warm-up iteration >=15.
[a] You must set these environment variables explicitly.	

Additional settings used to tune performance with the zentf to the TensorFlow framework

Table 5.2: zentf Environment Variables-Generic

Environment Variable	Default Value/User Defined Value
ZENDNN_LOG_OPTS	ALL:0
TF_ZEN_PRIMITIVE_REUSE_DISABLE	False
ZENDNN_ENABLE_MEMPOOL	The default value is set to 1. Set it to 0 if you want to disable it. Set it to: • 1 for Graph-based MEMPOOL • 2 for Node-based MEMPOOL • 3 for Output buffer caching
ZENDNN_PRIMITIVE_CACHE_CAPACITY	The default value is set to 1024; modify it as required.
ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE	0
TF_ENABLE_ZENDNN_OPTS	Default value is set to 0. Set it to 1 along with TF_ENABLE_ONEDNN_OPTS=0 to enable ZenDNN for inference. Set it to 0 when you want to enable vanilla training and inference.

Table 5.2: zentf Environment Variables-Generic

Environment Variable	Default Value/User Defined Value
TF_ENABLE_ONEDNN_OPTS	Default value is set to 1. By default, TensorFlow is shipped with oneDNN optimizations enabled. Hence, set it to 0 when you enable ZenDNN.
TF_ZENDNN_PLUGIN_BF16	The default value is 0. Set it to 1 to enable Auto mixed precision (AMP) for BF16.

Table 5.3: zentf Environment Variables-Optimization

Environment Variable	Default Value/User Defined Value
ZENDNN_TENSOR_POOL_LIMIT	The default value is set to 1024. For optimal performance, you can modify it to: • 512 for CNNs • 32 for densenet model
ZENDNN_CONV_ALGO	The default value is set to 1. It decides the convolution algorithm to be used in execution. 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  CONV ALGO 3 is not supported for this release. It will be deprecated in a future release.

Table 5.4: Environment variables used for tuning ONNX Runtime-ZenDNN

Environment Variable	Default Value/User Defined Value
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 the `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.

Table 5.4: Environment variables used for tuning ONNX Runtime-ZenDNN

Environment Variable	Default Value/User Defined Value
ZENDNN_CONV_RELU_FUSION_ENABLE	The flag is to enable convolution and relu operator fusion. It is enabled (set to 1) by default. Set 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. Set 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. Set it to 1 to enable the optimization. This optimization is useful for the variants of Inception and GoogleNet models.
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_ONNXRT_ENABLE_BF16_SUPPORT	This flag is disabled (set to 0) by default. Set it to 1 to enable the execution of compute expensive operation in BF16. It optimizes the execution for the CNN models only.
There are a few other environment variables that are initialized by the setup script.	

The script *scripts/ zendnn_ONNXRT_env_setup.sh*, initializes all the environment variables for ZenDNN-ONNXRT.

Performance Tuning Guidelines

Hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. Details of the environment variables used on a 4th Gen AMD EPYC™ server to get the best performance numbers are enumerated in the following sections.

System Used for Performance Tuning

Performance tuning settings are with respect to a system with the following specifications.

Table 5.5: System Specification

Specification	Value
Model Name	4th Gen AMD EPYC™ 9654 96-Core Processor

Table 5.5: System Specification

Specification	Value
CPU MHz	Up to 3.7 GHz
Core(s) per Socket	96
Socket(s) used	1
Thread(s) per Core	2
Mem-Dims	24x64 GB

Common Optimal Environment Variable Settings

The following environment variable settings are common to all three frameworks.

- `ZENDNN_LOG_OPTS=ALL:0`
- `OMP_NUM_THREADS=96`
- `OMP_WAIT_POLICY=ACTIVE`
- `OMP_DYNAMIC=FALSE`
- `ZENDNN_MATMUL_ALGO=FP32:4;BF16:3`
- `ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024`
- `GOMP_CPU_AFFINITY=0-95`

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).

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.

For ZenDNN stack, setting `OMP_WAIT_POLICY` to `ACTIVE` may give better performance.

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.

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. It will then start the assignment back from the beginning of the list.

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

Example

The affinity setting: `export GOMP_CPU_AFFINITY=0-95`, should give the same thread bindings.

Non-uniform Memory Access

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 `<model_run_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 <model_run_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, you must verify these details empirically.

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. It operates mainly in two modes:

- **always**: In this mode, the system kernel tries to assign huge pages to the processes running on the system. You can run the following command to set THP to `always`.
  ```
  echo always > /sys/kernel/mm/transparent_hugepage/enabled
  ```
- **madvise**: In this mode, the kernel only assigns huge pages to the individual processes memory areas. You can run the following command to set THP to `madvise`.
  ```
  echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
  ```

Disable THP

Log in as `root` to enable or disable THP settings. Use the following command to disable THP.

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

These are the recommended THP settings for better performance.

- For zentorch
  - CNN models: `always`
  - NLP models: `madvise`
- For zentf
  - CNN models: `never` (batch size =1), `always` (batch size >1)
  - NLP and Recommender models: `madvise`
- For ONNX Runtime
  - CNN and NLP models: `madvise` (batch size=1), `always` (batch size>1)

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.

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 use THPs for its internal metadata, and so on.

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_command> | grep <jemalloc>
  ```

Additional Environment Variable Settings for zentf

The following environment variable settings are specific to zentf, and should be used in addition to the environment variable settings.

- `ZENDNN_ENABLE_MEMPOOL=2` (for NLP and LLM models)
- `ZENDNN_ENABLE_MEMPOOL=3` (for CNN models)
- `TF_ENABLE_ZENDNN_OPTS=1`
- `TF_ENABLE_ONEDNN_OPTS=0`
- `ZENDNN_TENSOR_POOL_LIMIT=1024`
- `ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE=0`
- `ZENDNN_CONV_ALGO=4`

Additional Environment Variable Settings for ONNXRT Version of ZenDNN

The following is a list of additional Environment Variable settings for ONNXRT version of ZenDNN.

- ZENDNN_LOG_OPTS=ALL:0
- OMP_NUM_THREADS=96
- OMP_WAIT_POLICY=ACTIVE
- OMP_PROC_BIND=FALSE
- OMP_DYNAMIC=FALSE
- ZENDNN_MATMUL_ALGO=FP32:3,BF16:3
- ZENDNN_PARENT_FOLDER=/home/<user_id>/my_work
- ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024
- ZENDNN_ONNXRT_VERSION=1.17.0
- ZENDNN_ONNX_VERSION=1.15.0
- 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_ONNXRT_ENABLE_BF16_SUPPORT=0

Optimal Memory Allocator Settings Specific to ONNXRT

Optimal performance of several ZenDNN workloads is observed when interleaving is enabled 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
wget https://github.com/llvm/llvm-project/releases/download/llvmorg-10.0.1/openmp-10.0.1.src.tar.xz
```

2. Unzip.
```
tar -xf openmp-10.0.1.src.tar.xz
cd openmp-10.0.1.src
```
3. Configure cmakecommand.
```
cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
```
4. Build command.
```
make
```
5. Clean command.
```
make clean
```
6. Build artifacts location.
```
$ZENDNN_PARENT_FOLDER/openmp-10.0.1.src/runtime/src/libomp.so
```
7. Do LD_PRELOAD to the .so file.
```
export LD_PRELOAD=$ZENDNN_PARENT_FOLDER/openmp-10.0.1.src/runtime/src/libomp.so:$LD_PRELOAD
```

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 --model_class AutoModel -p fp32 -i 3 -t 10 -b 24 -s 16 -n 96 -v
```

6 Logging and Debugging

In this chapter, logging mechanisms in both the ZenDNN library and the plug-ins are discussed.

ZenDNN Library Logs

Logging is disabled in the ZenDNN library by default. It can be enabled using the environment variable `ZENDNN_LOG_OPTS` before running any test. 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 6.1: 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.

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`

Enable Log Profiling

To enable the log profiling of `zendnn_primitive_create` and `zendnn_primitive_execute`, set `ZENDNN_PRIMITIVE_LOG_ENABLE=1`

The Different Debug Levels (`DBGLVL`) are as follows.

Table 6.2: Debug Levels

Debug Level	Value
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 invoked. ALGO, TEST, and FWK are optional logs and might not appear in all the cases.

zentorch Logging and Debugging

For zentorch, enable CPP specific logging by setting the environment variable `TORCH_CPP_LOG_LEVEL`. This has four levels: `INFO`, `WARNING`, `ERROR` and `FATAL` in decreasing order of verbosity.

Similarly, enable Python logging by setting the environment variable `ZENTORCH_PY_LOG_LEVEL`. This has five levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`, again in decreasing order of verbosity.

Here is an example of how to enable INFO level logs for cpp and DEBUG level for Python (most verbose):

```
export TORCH_CPP_LOG_LEVEL=INFO
export ZENTORCH_PY_LOG_LEVEL=DEBUG
```

`WARNING` is the default level of logs for both cpp and Python sources, but it can be overridden.

The log levels are the same as those provided by the Python logging module.

INFO: As all Operators implemented in zentorch are registered with torch using the `TORCH_LIBRARY()` and `TORCH_LIBRARY_IMPL()` macros in bindings, the PyTorch profiler can be used without any modification to measure the operator level performance.

Debugging

PyTorch offers a debugging toolbox that comprises a built-in stats and trace function. This functionality facilitates the display of the time spent by each compilation phase, output code, output graph visualization, and IR dump. `TORCH_COMPILE_DEBUG` invokes this debugging tool that allows for better problem-solving while troubleshooting the internal issues of TorchDynamo and TorchInductor.

This functionality works for the models optimized using zentorch, and hence it can be leveraged to debug these models too. To enable this functionality, either set the environment variable `TORCH_COMPILE_DEBUG=1` or specify the environment variable with the runnable file (for example, *test.py*) as input.

For example, if the file `test.py`contains a model optimized by `torch.compile` with `zentorch` as backend, use:

```
TORCH_COMPILE_DEBUG=1 python test.py
```

7 Support

Note that the ZenDNN 4.2 plug-ins and library are currently in “Early Access” mode. We welcome feedback, suggestions, and bug reports. Feel free to contact us at: zendnn.maintainers@amd.com