ZenDNN User Guide

Publication Number: 57300 Revision: 5.2.1
Date: April 2026

AMD

Contents

Chapter 1: ZenDNN

ZenDNN continues its focus on optimizing inference performance for Recommender Systems and Large Language Models on AMD EPYC™ CPUs. This latest upgrade brings a host of enhancements designed to push the boundaries of efficiency and speed. We've introduced significant improvements for bfloat16 performance, expanded support for cutting-edge models like Llama 3.1 and 3.2, and added capabilities like Weight only quantization with INT4 datatype support, dynamic quantization and optimized quantized DLRM models.

Under the hood, ZenDNN's enhanced AMD-specific optimizations operate at every level. In addition to highly optimized operator microkernels, these optimizations include comprehensive graph optimizations such as pattern identification, graph reordering, and fusions. Notable improvements include optimized embedding bag kernels and an enhanced zenMatMul matrix splitting strategy, both designed to maximize throughput and minimize latency. The result is a significant performance boost over vanilla frameworks. Beyond its powerful optimizations, the ZenDNN plug-ins offer broad compatibility, seamlessly integrating with popular frameworks like TensorFlow and PyTorch.

This release also adds support for PyTorch 2.11.0 and 2.10.0 and TensorFlow 2.21.0 through 2.16.0, along with a stable vLLM + zentorch plugin supporting the vLLM V1 engine that delivers better performance on various models compared to vLLM-Native.

Note: The vLLM + zentorch plugin was available in the 5.1 release as an experimental feature, with support limited to the vLLM V0 engine.

We've also enabled Java® Integration by contributing and upstreaming a new Pluggable Device feature to the TensorFlow-Java repository, strengthening its core capabilities.

1.1 Scope

The ZenDNN library and plug-ins have been developed to enable Deep Learning inference on AMD EPYC™ CPUs. The library offers optimized primitives, such as EmbeddingBag operators, Matrix multiplications and related fusions, Elementwise operations, Attention operators and Pool (Max and Average) that improve the performance of many transformer-based models, recommender system models, convolutional neural networks, and recurrent neural networks. For the primitives not supported by ZenDNN, execution will fall back to the native path of the framework.

1.2 Release Highlights

ZenDNN 5.2.1

- **Compatibility with Deep-learning Frameworks**: Fully aligned with PyTorch 2.11.0 and TensorFlow 2.21.0, ensuring smooth upgrades and interoperability.
- **Java Interface**: Java Interface to the TensorFlow plugin (zentf) compatible with TensorFlow Java

v1.2.0-SNAPSHOT.
  - TensorFlow-Java v1.2.0-SNAPSHOT supports TensorFlow v2.20.0.
  - For TensorFlow-Java 1.2.0-SNAPSHOT, you must build from source.
- `zentorch.llm.optimize` has been deprecated from the 5.2 release.
- zentorch provides functional support for INT4 weight-only quantized models using TorchAO's (v0.16.0) `IntxWeightOnlyConfig`, supporting both per-channel and per-group quantization granularity. These models can be executed seamlessly via `torch.compile` (backend='zentorch') or through the vLLM zentorch plugin.
- Zentorch provides functional support for INT8 dynamic quantization using TorchAO's (v0.16.0) `Int8DynamicActivationInt8WeightConfig` with symmetric activation mapping, where activations are dynamically quantized to INT8 at runtime alongside INT8 quantized weights. These models can be executed seamlessly through the vLLM zentorch plugin.

Native Framework Support

- The ZenDNN library can be used in the following frameworks through a plugin:
  - TensorFlow v2.21.0 through v2.16.0

    Note: The ZenDNN 5.2.1 plugin for TensorFlow is optimized to give the best performance with TensorFlow v2.21.0.
  - PyTorch v2.11.0 to v2.10.0.

    Note: The ZenDNN 5.2.1 plugin for PyTorch is optimized to give the best performance with PyTorch v2.11.0.
  - vLLM v0.15.0 to v0.18.0

    Note: vLLM support is added to ZenDNN PyTorch plugin (zentorch). The vLLM-ZenTorch plugin integrates with vLLM's V1 engine, enabling plug-and-play acceleration of large language model inference on AMD EPYC™ CPUs.
- In ZenDNN 5.0, the ZenDNN library is directly integrated with ONNX Runtime v1.19.2. As of ZenDNN 5.1 and 5.2, support for ONNXRT has been temporarily paused.

Note: In this document, we refer to the ZenDNN plugin for TensorFlow as `zentf`, and the ZenDNN plugin for PyTorch and vLLM as `zentorch`.

- Wheel Files
  - zentorch wheel files (*.whl) have been generated using:
    - Python v3.10-v3.13
    - PyTorch v2.11.0

- zentf wheel files (*.whl) have been generated using:
  - Python v3.10-v3.13
  - TensorFlow v2.21.0

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

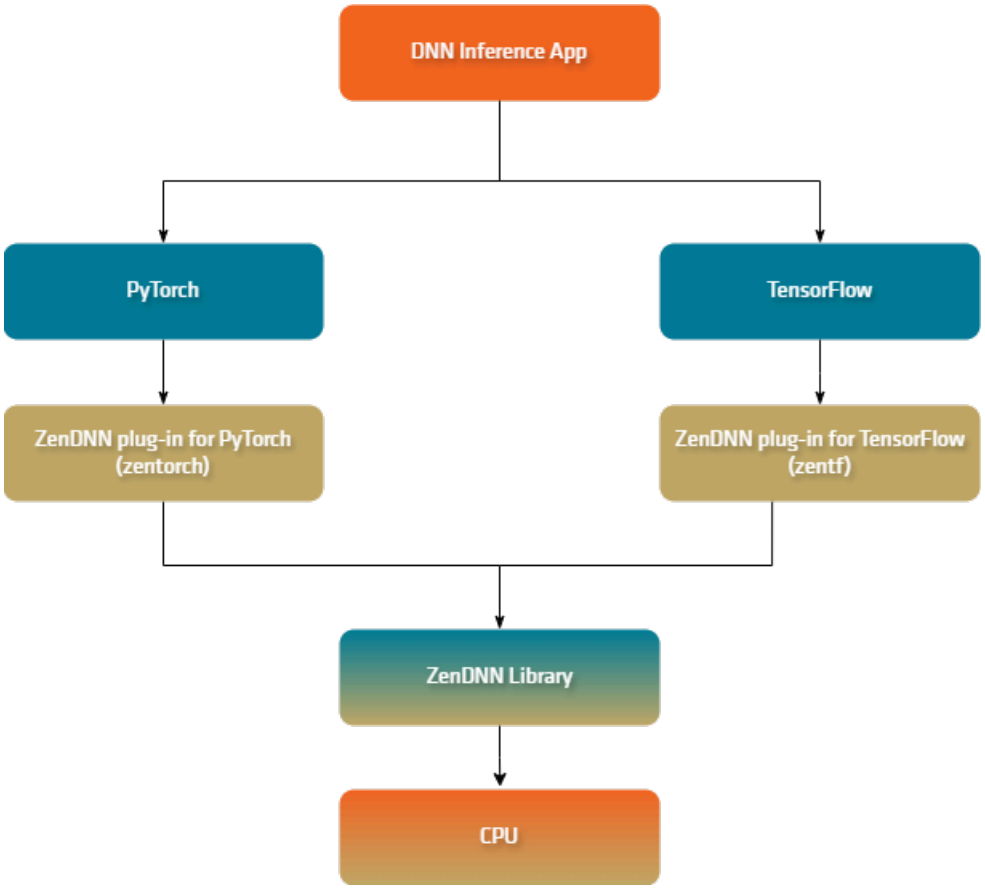
Highlights of Major Release ZenDNN 5.2

- Framework Compatibility: PyTorch 2.10.0, TensorFlow 2.20.0, vLLM v0.12.0–v0.15.1
- Java Interface: zentf compatible with TensorFlow Java v1.2.0-SNAPSHOT (build from source required)
- Deprecation: `zentorch.llm.optimize` is deprecated in 5.2
- INT4 Quantization: Functional support for INT4 weight-only quantized models via TorchAO v0.16.0 (`IntxWeightOnlyConfig`), with per-channel and per-group granularity; works with `torch.compile(backend='zentorch')` or the vLLM zentorch plugin
- vLLM Integration: zentorch plugin integrates with vLLM's V1 engine for LLM inference acceleration on AMD EPYC™ CPUs
- ONNX Runtime: Support paused since ZenDNN 5.1 (last supported in 5.0 with ONNXRT v1.19.2)
- Wheel Files: zentorch built for Python 3.10–3.13 + PyTorch 2.10.0; zentf built for Python 3.9–3.13 + TensorFlow 2.20.0

1.3 High-level Overview

This high-level block diagram of the ZenDNN inference stack depicts how the ZenDNN library interfaces with the ZenDNN plugin for PyTorch (zentorch) and the ZenDNN plugin for TensorFlow (zentf). The ZenDNN library uses the AOCL-DLP library internally, as well as other third-party libraries such as FBGEMM and LIBXSMM.

Figure 1.1: ZenDNN Software Stack



1.4 Build from Source

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

Chapter 2: PyTorch

The ZenDNN plugin 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.

2.1 Release Highlights

zentorch is compatible with base versions of PyTorch v2.11.0 and 2.10.0. Testing with PyTorch v2.11.0 has been performed, and it is recommended you use zentorch 5.2.1 with PyTorch v2.11.0.

zentorch 5.2.1 Release highlights

- Added support for PyTorch 2.11.0 in addition to 2.10.0.
- Added vLLM zentorch plugin support: The new zentorch plugin for vLLM delivers a significant performance uplift on a variety of models compared to native vLLM.
- Added functional support for INT8 dynamic quantization using TorchAO's (v0.16.0) `Int8DynamicActivationInt8WeightConfig` with symmetric activation mapping.
- Added support for 4-bit WOQ using TorchAO's (v0.16.0) using `Int4WeightOnlyOpaqueTensorConfig` and `IntxWeightOnlyConfig`.

Previous Major Release (zentorch 5.2.0) Highlights

- Added supprt for PyTorch 2.10.0.
- Added vLLM zentorch plugin support: The new zentorch pligin for vLLM delivers a significant performance uplift on a variety of models compared to native vLLM.

2.2 Supported OS

Refer to the support matrix for the list of supported operating systems.

2.3 Install ZenDNN Plug-in for PyTorch (zentorch)

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

2.3.1 Using the Release Binary

To install zentorch, you may choose from one of two options to access the zentorch binary release.

1. PyPI Repo as a wheel (.whl) file.
2. AMD developer portal (as a package). This release package consists of a zentorch wheel file with

a .whl extension and a `scripts/` folder to set up optimal environment settings. Refer to section zentorch Optimal Environment Settings for more details on the usage of the script.

2.3.1.1 Install the Release Binary

Create and Setup Conda Environment

Before you begin:

- Choose a unique name for your new Conda environment. Example: `zentorch-5.2.1`.
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `zentorch-5.2.1` exists, use the following command to remove it.
  ```
  conda remove --name zentorch-5.2.1 --all
  ```

Important: zentorch is compatible with Python v3.10-3.13. Make sure you create a Conda environment only with Python versions supported by zentorch.

Conda Environment Setup

To setup the Conda environment:

1. Refer to the Miniforge documentation available here to install Miniforge on your system. Testing has been performed with *Miniforge3-24.11.3-2-Linux-x86_64.sh*.
2. Create and activate a Conda environment that houses all the zentorch specific installations.
   ```
   conda create -n zentorch-5.2.1 python=3.10 -y
   conda activate zentorch-5.2.1
   ```

Install zentorch

To install the zentorch release binary:

1. Install PyTorch v2.11.0.
   ```
   pip install torch==2.11.0 --index-url https://download.pytorch.org/whl/cpu
   ```
2. Use one of the following two methods to install zentorch.
   a. Using the PyPI repo. Run the following command:
      ```
      pip install zentorch==5.2.1
      ```
      For optimal environment settings, refer to Performance Tuning, or use the script shipped in the release package from the AMD developer portal.
   b. Using the release package from the AMD developer portal:
      1) Download the package from the AMD developer portal.
      2) Run the following commands to unzip the package and install the binary:
         ```
         unzip ZENTORCH_v5.2.1_Python_v3.10.zip
         cd ZENTORCH_v5.2.1_Python_v3.10/
         ```

         Note: zentorch is compatible with Python v3.10-3.13. We have used 3.10 here only as an example.

3) Install the binary.
```
pip install zentorch-5.2.1-cp310-cp310-manylinux_2_28_x86_64.whl
```

4) To use the recommended environment settings, execute:
```
source scripts/zentorch_env_setup.sh
```

Note: While importing zentorch, if you get the error: *ImportError: /lib64/libstdc++.so.6: version `GLIBCXX_.a.b.cc' not found (required by <path-to-conda>/envs/<env-name>/lib/python<py-version>/site-packages/zentorch-5.2.1-pyx.y-linux-x86_64.egg/zentorch/_C.cpython-xy-x86_64-linux-gnu.so)*, export `LD_PRELOAD` as:
```
export LD_PRELOAD=<path-to-conda>/envs/<env-name>/lib/libstdc++.so.6:$LD_PRELOAD
```

2.3.2 Build from Source

To build the zentorch pip package from source:

1. Clone the repository and check out the r5.2.1 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: *<path to zentorch repo>/dist/zentorch-*.whl*.

2.4 Usage

The custom zentorch backend can be called through `torch.compile`. See the Examples section for a few code examples.

Note: For optimal performance when using `torch.compile` with zentorch as a backend in PyTorch, it is recommended to set a warm-up count of five. This entails running the inference section of the code five times—such as within a loop—before executing the actual run used for measuring inference performance.

Note: The warm-up process allows the compiled model to be pre-loaded into memory, reducing the likelihood of costly cache misses and improving overall efficiency.

2.4.1 Using torch.compile

In most cases, you can simply set `backend='zentorch'` as an argument in `torch.compile()` to enable optimizations.

For Hugging Face large language models, up to r5.1 we used to provide `zentorch.llm.optimize()`, a specialized method that delivers further performance enhancements.

However, beginning with r5.2, `Zentorch.llm.optimize` has been removed. LLMs are run with zentorch as a plugin in vLLM. Refer to the vLLM-zentorch Plugin section for details.

For additional guidance on usage scenarios, refer to the Recommendations section.

```
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)
```

2.4.2 Examples

Here are examples of running inference for various models in PyTorch. The examples described in this section are also available at this page.

Note that additional packages may be required in your environment. If the zentorch plugin is already installed, you can add the remaining packages by running the following command:

```
pip install datasets scikit-learn pillow transformers==4.57.6
```

2.4.2.1 BERT-based Models

```
import torch
import zentorch
from transformers import BertTokenizer, BertModel
from datasets import load_dataset

# Load the dataset
dataset = load_dataset("imdb", split="test")

print(dataset[0]['text'])

# Load the tokenizer and the model
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased', trust_remote_code=True)

# Load the model
model_id = "google-bert/bert-large-uncased"
model = BertModel.from_pretrained(
            model_id,
            torch_dtype=torch.bfloat16,
            trust_remote_code=True,
        )
model = model.eval()

############### Code modification ###############
model.forward = torch.compile(model.forward, backend="zentorch")
################################################

# Inference
with torch.inference_mode(), torch.no_grad():
    # Prepare inputs by tokenizing the examples
    inputs = tokenizer(dataset['text'][:3], return_tensors="pt", padding=True, truncation=True)

    # Generate outputs
```

```
    outputs = model(**inputs)

# Get last hidden states
last_hidden_states = outputs.last_hidden_state

# Print the shape of the last hidden states
print("Last hidden states shape:", last_hidden_states.shape)
```

Sample Output

```
Last hidden states shape: torch.Size([3, 339, 1024])
```

2.4.2.2 Using Warm-up Iterations for Optimal Performance

For optimal performance when using `torch.compile` with `zentorch` as a backend, it is recommended to set a warm-up count of five. This entails running the inference section of the model five times before measuring or relying on inference latency. The initial runs trigger graph compilation, operator fusion, and internal ZenDNN cache population (example: weight reordering). Subsequent runs after warm-up reflect the true optimized performance.

Here is a complete example using a ResNet-50 model.

```
import torch
import zentorch
from torchvision.models import resnet50

# Step 1: Load a pretrained model and set it to eval mode
model = resnet50(pretrained=True).eval()

# Step 2: Compile the model with zentorch backend
compiled_model = torch.compile(model, backend="zentorch")

# Step 3: Create a sample input
sample_input = torch.randn(1, 3, 224, 224)

# Step 4: Warm-up phase — run inference 5 times to trigger compilation and
#          populate internal caches for optimal steady-state performance
WARMUP_COUNT = 5
with torch.no_grad():
    for i in range(WARMUP_COUNT):
        output = compiled_model(sample_input)
        print(f"Warm-up iteration {i + 1}/{WARMUP_COUNT} complete")

# Step 5: Timed inference — measure performance after warm-up
import time
with torch.no_grad():
    start = time.time()
    output = compiled_model(sample_input)
    end = time.time()
    print(f"Inference latency after warm-up: {(end - start) * 1000:.2f} ms")
```

Sample Output

```
Inference latency after warm-up: 4.11 ms
```

Note:

- The first call to the compiled model triggers `torch.compile` graph capture, tracing, and zentorch

backend optimizations (operator fusion, ZenDNN op replacement). This makes the first iteration significantly slower.

- Subsequent warm-up iterations allow ZenDNN internal caches (such as weight repacking and matmul strategies) to reach a steady state.
- After five warm-up iterations, the model runs at peak throughput and the measured latency accurately reflects production performance.
- This warm-up pattern applies to all model types (CNN, NLP, LLM) when using the zentorch backend.

2.4.2.3 Hugging Face Language Models

The `zentorch.llm.optimize` API has been deprecated.

You can run generative models using `torch.compile` (model, backend="zentorch"), but for optimal performance we recommend using vLLM. See vLLM-zentorch Plugin for more details.

zentorch provides support for Weight-Only Quantization (WOQ) models with both per-channel and per-group quantization granularity, enabling efficient 4-bit quantized inference, along with Dynamic quantization with INT8 activations with per-token granularity and INT8 quantized weights for large language models on AMD EPYC™ CPUs with significant memory savings and minimal impact on model accuracy.

Quantizing Models

Use the following steps to quantize Hugging Face models with different TorchAO configurations. While the first step is different for different configurations, steps 2 through 5 are common for all configurations.

Step 1

Weight-only Quantization using `IntxWeightOnlyConfig` with Per-channel granularity.

```
import torch
from transformers import TorchAoConfig, AutoModelForCausalLM, AutoTokenizer
from torchao.quantization.quant_api import IntxWeightOnlyConfig
from torchao.quantization.quant_primitives import MappingType

# Step 1: Create quantization config with IntxWeightOnlyConfig per-channel granularity
quantization_config = TorchAoConfig(
    IntxWeightOnlyConfig(
        weight_dtype=torch.int4,
        mapping_type=MappingType.SYMMETRIC,
        scale_dtype=torch.bfloat16,
    )
)
```

Weight-only Quantization using `Int4WeightOnlyOpaqueTensorConfig`

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, TorchAoConfig
from torchao.prototype.int4_opaque_tensor import Int4WeightOnlyOpaqueTensorConfig
```

```
quantization_config = TorchAoConfig(
    Int4WeightOnlyOpaqueTensorConfig(group_size=128)
)
```

Dynamic quantization with `Int8DynamicActivationInt8WeightConfig`

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, TorchAoConfig
from torchao.quantization import Int8DynamicActivationInt8WeightConfig
from torchao.quantization.quant_primitives import MappingType

quantization_config = TorchAoConfig(
    Int8DynamicActivationInt8WeightConfig(
        version=2,
        act_mapping_type=MappingType.SYMMETRIC,
    )
)
```

Steps 2 through 5 - Common for all the afore mentioned configurations

```
model_name = "meta-llama/Llama-3.2-1B-Instruct"
output_dir = "./quantized_model"

# Step 2: Load and quantize the model
quantized_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.bfloat16,
    device_map="cpu",
    quantization_config=quantization_config,
    trust_remote_code=True,
)

# Step 3: Save the quantized model
quantized_model.save_pretrained(
    output_dir,
    safe_serialization=False,
)

# Step 4: Save the tokenizer
tokenizer = AutoTokenizer.from_pretrained(
    model_name,
    trust_remote_code=True,
)
tokenizer.save_pretrained(output_dir)

# Step 5: Test the quantized model
input_text = "what are we having for dinner?"
```

Sample Output

```
You mentioned we were planning to go to a restaurant
```

Note:

- Ensure you have the required dependencies installed: `pip install transformers==4.57.6 torchao==0.16.0`
- The `MappingType.SYMMETRIC` option enables symmetric quantization which is recommended for optimal performance with zentorch.

- The `scale_dtype=torch.bfloat16` option ensures compatibility with AMD EPYC™ CPU optimizations.
- Use `safe_serialization=False` when saving for compatibility with zentorch.
- WOQ quantized models are only supported with freezing enabled

```
export TORCHINDUCTOR_FREEZING=1
export VLLM_USE_AOT_COMPILE=0
export TORCHINDUCTOR_AUTOGRAD_CACHE=0
```

- Sample output of the quantized model example described earlier is expected to be different for each run.

Running Quantized Models

Quantized models are supported in vLLM via the zentorch backend. This release introduces functional support for Weight-Only Quantization (WOQ). See vLLM-zentorch Plugin for detailed instructions on running models with vLLM.

Note: When running quantized models, use the path to your quantized model directory (example: `./quantized_model`) as the model path instead of the original Hugging Face model name.

2.4.2.4 Recommendation Systems with DLRM

The main code snippet showing how you can accelerate DLRM with zentorch is provided in this section.

To try the code snippet, you will need the DLRM model which is hosted on Github. You can download it using:

```
wget https://raw.githubusercontent.com/amd/ZenDNN-pytorch-plugin/refs/heads/r5.2.1/examples/dlrm_model.py
```

Note: We recommend running inference with freezing enabled. You can set this environment variable to enable it: `export TORCHINDUCTOR_FREEZING=1`

```
# Sourced from https://github.com/facebookresearch/dlrm
from dlrm_model import DLRMMLPerf
import torch
import numpy as np
import zentorch
import random

from sklearn.metrics import roc_auc_score

# Initialize the model
np.random.seed(123)
random.seed(123)
torch.manual_seed(123)

DEFAULT_INT_NAMES = [f'int_{i}' for i in range(13)]
model = DLRMMLPerf(
    embedding_dim=128,
    num_embeddings_pool=[
        40000000, 39060, 17295, 7424, 20265, 3, 7122, 1543, 63, 40000000,
```

```
        3067956, 405282, 10, 2209, 11938, 155, 4, 976, 14, 40000000,
        40000000, 40000000, 590152, 12973, 108, 36],
    dense_in_features=len(DEFAULT_INT_NAMES),
    dense_arch_layer_sizes=[512, 256, 128],
    over_arch_layer_sizes=[1024, 1024, 512, 256, 1],
    dcn_num_layers=3,
    dcn_low_rank_dim=512,
    use_int8=False,
    use_bf16=True
).bfloat16()

# Prepare Inputs
multi_hot = [3, 2, 1, 2, 6, 1, 1, 1, 1, 7, 3, 8, 1, 6, 9, 5, 1, 1, 1, 12, 100, 27, 10, 3, 1, 1]
batchsize = 32768
densex = torch.randn((batchsize, 13), dtype=torch.float).to(torch.bfloat16)
index = [torch.ones((batchsize * h), dtype=torch.long) for h in multi_hot]
offset = [torch.arange(0, (batchsize + 1) * h, h, dtype=torch.long) for h in multi_hot]

# Inference
################ Code modification ###############
model = torch.compile(model, backend="zentorch")
##################################################

with torch.no_grad():
    out = model(densex, index, offset)

# Simulating labels
true_labels = torch.randint(0, 2, (32768,))
# Convert to float32 for compatibility with sklearn
predicted_probabilities = out.to(torch.float32).cpu().detach().numpy().reshape(-1)
true_labels = true_labels.cpu().detach().numpy()

# Calculate AUC
auc_score = roc_auc_score(true_labels, predicted_probabilities)
print(f"AUC Score: {auc_score}")
```

Sample Output

```
AUC Score: 0.5
```

To run the Quantized DLRMv2 model, refer to this page. Also, follow steps described in the README for Multi-instance performance.

2.4.2.5 ResNet

Note: To enable freezing for improved performance, set this environment variable before running the script: `export TORCHINDUCTOR_FREEZING=1`.

```
import torch
import zentorch
from transformers import AutoImageProcessor, ResNetForImageClassification
from PIL import Image

# Load the ResNet Model
processor = AutoImageProcessor.from_pretrained("microsoft/resnet-50")
model = ResNetForImageClassification.from_pretrained(
    "microsoft/resnet-50", torch_dtype=torch.bfloat16
)
```

```
# Prepare Inputs
image = Image.open("airplane.jpg")  # Pick an image of your choice
inputs = processor(image, return_tensors="pt")
# converting input to BF16
inputs = {k: v.to(torch.bfloat16) for k, v in inputs.items()}

# Inference
############### Code modification ###############
model.forward = torch.compile(model.forward, backend="zentorch", dynamic=False)
#################################################

with torch.no_grad():
    logits = model(**inputs).logits

predicted_label = logits.argmax(-1).item()
print(model.config.id2label[predicted_label])
```

Sample Output

```
plane, carpenter's plane, woodworking plane
```

2.4.3 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)
```

Hugging Face Generative LLM Models

The `zentorch.llm.optimize` API has been deprecated. You can run generative models using `torch.compile` (model, backend="zentorch"), but for optimal performance we recommend using vLLM. See vLLM-zentorch Plugin for more details.

2.5 vLLM-zentorch Plugin

The zentorch vLLM plugin integrates zentorch with vLLM's V1 engine to deliver optimized large language model inference on AMD EPYC™ CPUs. By leveraging ZenDNN's highly optimized kernels, this plugin accelerates both attention and non-attention operations in vLLM, providing significant throughput improvements for popular LLMs.

The plugin uses vLLM's platform and general plugin entry points to:

- Inject zentorch optimization passes into torch.compile
- Disable replacement with Intel oneDNN kernels to enable replacement with zentorch kernels
- Enable CPU-only profiling

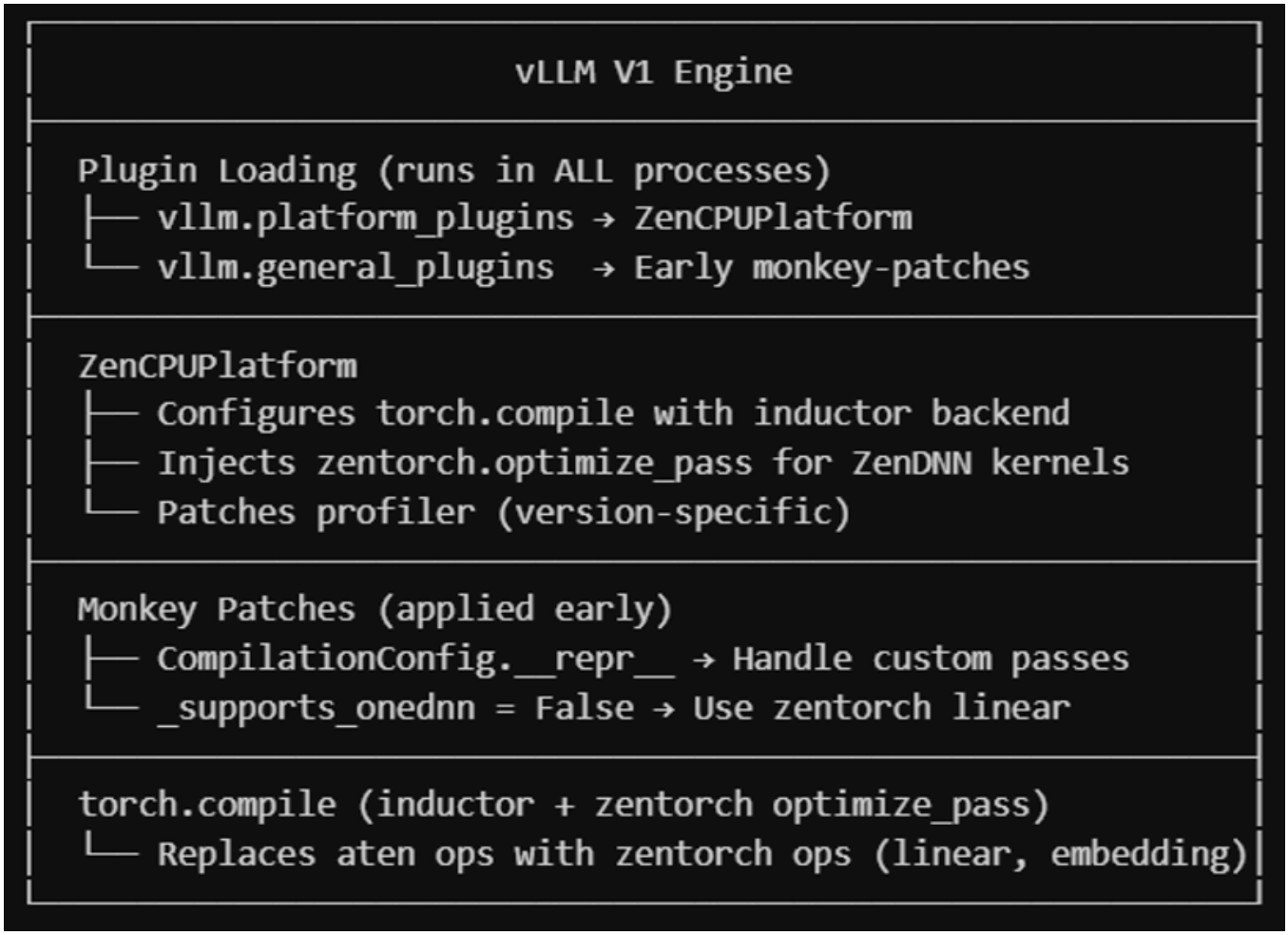
Key Features

- **Plug-and-Play Acceleration**: No code modifications required—just install zentorch alongside vLLM for automatic acceleration.
- **Seamless vLLM Integration**: vLLM detects zentorch and transparently uses ZenDNN-optimized GEMM and Embedding kernels for supported CPUs.
- **Optimized for Modern x86 CPUs**: Delivers best-in-class performance on AMD EPYC™ processors, while supporting a broad range of x86 CPUs with the necessary instruction set.
- **Powered by ZenDNN**: Leverages AMD's ZenDNN library for state-of-the-art, CPU-optimized neural network operations.

2.5.1 Architecture

When both vLLM and the zentorch packages are installed, vLLM automatically detects the zentorch platform and uses zentorch optimizations via `torch.compile`.

Figure 2.1: vLLM V1 Engine



The plugin leverages AMD EPYC™ specific intrinsics and optimizations to accelerate computations on AMD EPYC™ CPUs. However, it may also function on other x86 CPUs that meet the required ISA. We use zentorch to compile the LLM with `torch.compile`, replacing the native ops with zentorch's optimized ops.

Key Components

The system consists of two main components that work together to enable CPU optimization.

- The `ZenCPUPlatform` extends vLLM's `CpuPlatform` class and configures the system for CPU. It establishes the compilation configuration using `CompilationLevel.DYNAMO_ONCE` or `CompilationMode.DYNAMO_TRACE_ONCE` with the `inductor` backend, and integrates `zentorch` operators through the `zentorch._compile_backend.optimize_pass` injection.
- The Plugin Entry Points (defined in `init.py`) handle the initialization process by registering with vLLM through the `vllm.platform_plugins` and `vllm.general_plugins` mechanisms. These entry points apply necessary patches before model initialization occurs and ensure compatibility by validating the vLLM version requirements.

2.5.2 Installation

Follow these instructions to install.

1. Create a new Python environment.

   Note: To get started with Conda, refer to the Miniforge Installation Guide.

   Using Conda

   ```
   conda create -n vllm-env python=3.10
   conda activate vllm-env
   ```

2. Build vLLM from Source.

   a. Follow instructions provided in the vLLM Installation Guide for detailed instructions.

   Important: Pre-built vLLM CPU binaries are available from 0.13.0. You must build vLLM from source to enable CPU support for the previously supported versions. The supported versions are: 0.15.0, 0.15.1, 0.16.0, 0.17.0, 0.17.1, 0.18.0. Check out the appropriate release tag before building.

3. Install zentorch.

   Table 2.1: Compatibility Matrix: vLLM-PyTorch version-zentorch install method

vLLM version	PyTorch version (auto-installed by vLLM)	zentorch install method
0.15.0 - 0.18.0	2.10.0	PyPI or source

   If installing from PyPI (vLLM 0.15.0+):

   ```
   pip install zentorch
   ```

   From source (all supported versions): Refer to the zentorch Installation Guide for detailed instructions.

2.5.3 Usage

No code changes are required. Once installed, simply run your vLLM inference workload as usual. The plugin will be automatically detected and used for inference on supported x86 CPUs that meet the required ISA features. While optimized for AMD EPYC™ CPUs, it may also function on other compatible x86 processors.

Note: Upon importing vLLM, you should see the following message in the logs:

```
INFO [__init__.py] Platform plugin zentorch is activated
```

Environment Configuration

The plugin is recommended to be run with `ZENDNNL_MATMUL_ALGO=1` (the default).

Environment Variables

```
export VLLM_CPU_KVCACHE_SPACE=120         # GB for KV cache
export VLLM_CPU_OMP_THREADS_BIND=0-127    # CPU cores to use
export TORCHINDUCTOR_FREEZING=1
export VLLM_USE_AOT_COMPILE=0
export TORCHINDUCTOR_AUTOGRAD_CACHE=0
```

Performance Libraries

Install and preload tcmalloc and llvm-openmp for best performance:

```
# tcmalloc
#The following command is for Ubuntu
sudo apt-get install libtcmalloc-minimal4
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc_minimal.so.4:$LD_PRELOAD

# llvm-openmp
conda install -c conda-forge llvm-openmp=18.1.8=hf5423f3_1 -y
export LD_PRELOAD="$CONDA_PREFIX/lib/libiomp5.so:$LD_PRELOAD"
```

Example

```
from vllm import LLM, SamplingParams

llm = LLM(model="meta-llama/Llama-3.1-8B", dtype="bfloat16")
params = SamplingParams(temperature=0.8, top_p=0.95)
output = llm.generate(["Hello, world!"], sampling_params=params)
print(output)
```

Note: These hardware recommendations are specific to vLLM CPU workloads. ZenTorch can be used independently and may have different requirements or optimizations for other use cases.

Support and Feedback

For questions, feedback, or to contribute, visit the AMD ZenDNN PyTorch Plugin GitHub page.

2.5.4 Docker

Pre-built docker images with vLLM and zentorch are available for quick deployment.

Pulling the Image

```
# Pull the Ubuntu image
docker pull amdih/zendnn_zentorch:vllm_v0.18.0_zentorch_v5.2.1_ubuntu22.04_2026_r5.2.1

# Pull the RHEL image
docker pull amdih/zendnn_zentorch:vllm_v0.18.0_zentorch_v5.2.1_rhel9.5_r5.2.1
```

Running the Container

```
# Ubuntu Container
docker run -d --name vllm_zentorch-ubuntu \
  -v /path/to/models:/models \
  amdih/zendnn_zentorch:vllm_v0.18.0_zentorch_v5.2.1_ubuntu22.04_2026_r5.2.1 \
  tail -f /dev/null
```

```
# Enter the container
docker exec -it vllm_zentorch-ubuntu bash

cd workspace
```

Note: Mount volumes **(-v)** for model files and any datasets you need inside the container. Export `HF_TOKEN` of your account and `HF_HOME` to the model path Environment variables are pre-configured. Adjust `VLLM_CPU_OMP_THREADS_BIND` for your machine (example: 0-127 for Turin, 0-95 for Genoa).

2.6 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. For generative LLMs, zentorch provides support for Weight Only Quantization with 4-bit weights and BF16 activations, along with dynamically quantized INT8 activation and INT8 weights as described in Weight Only Quantized Models and Dynamically Quantized Models, respectively.

Note: For INT8, computations fall back to the native framework.

2.6.1 Weight Only Quantized Models

Quantizing Models

Hugging Face models can be quantized using `Int4WeightOnlyOpaqueTensorConfig` and `IntxWeightOnlyConfig` from TorchAO. zentorch supports Weight Only Quantization (WOQ) with 4-bit weights and BF16 activations, with both per-channel (limited to `IntxWeightOnlyConfig`) and per-group quantization granularity.

To quantize a model, use the TorchAoConfig integration in HuggingFace Transformers as shown below:

- For per-group quantization (recommended, group_size=128):

```
from transformers import TorchAoConfig, AutoModelForCausalLM, AutoTokenizer
from torchao.quantization.quant_api import IntxWeightOnlyConfig
from torchao.quantization.quant_primitives import MappingType
from torchao.quantization.granularity import PerGroup
import torch

quantization_config = TorchAoConfig(
    IntxWeightOnlyConfig(
        weight_dtype=torch.int4,
        mapping_type=MappingType.SYMMETRIC,
        scale_dtype=torch.bfloat16,
        granularity=PerGroup(128),
    )
)
quantized_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    dtype=torch.bfloat16,
    device_map="cpu",
```

```
    quantization_config=quantization_config,
    trust_remote_code=True
)
```

- For per-channel quantization, replace PerGroup(128) with PerChannel():

```
from torchao.quantization.granularity import PerChannel

quantization_config = TorchAoConfig(
    IntxWeightOnlyConfig(
        weight_dtype=torch.int4,
        mapping_type=MappingType.SYMMETRIC,
        scale_dtype=torch.bfloat16,
    )
)
```

- Using `Int4WeightOnlyOpaqueTensorConfig` for per-group weight-only quantization:

```
import torch
from transformers import TorchAoConfig, AutoModelForCausalLM, AutoTokenizer
from torchao.prototype.int4_opaque_tensor import Int4WeightOnlyOpaqueTensorConfig

quantization_config = TorchAoConfig(Int4WeightOnlyOpaqueTensorConfig(group_size=128))

quantized_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    dtype=torch.bfloat16,
    device_map="cpu",
    quantization_config=quantization_config,
    trust_remote_code=True
)
```

Attention: Currently, we do not support Mixture-of-Experts (MoE) models for any quantization scheme.

2.6.2 Dynamically Quantized Models

Quantizing models

Hugging Face models can be quantized using `Int8DynamicActivationInt8WeightConfig` from TorchAO. Zentorch supports dynamically quantized models with INT8 dynamically quantized activation with per-token granularity and INT8 quantized weights with per-channel granularity.

To quantize a model, use the TorchAoConfig integration in Hugging Face Transformers as shown here.

```
import torch

from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TorchAoConfig,
)
from torchao.quantization import Int8DynamicActivationInt8WeightConfig
from torchao.quantization.quant_primitives import MappingType

quantization_config = TorchAoConfig(
    Int8DynamicActivationInt8WeightConfig(
        version=2,
```

```
        act_mapping_type=MappingType.SYMMETRIC,
    )
)

quantized_model = AutoModelForCausalLM.from_pretrained(
    args.model_name,
    torch_dtype=torch.bfloat16,
    device_map="cpu",
    quantization_config=quantization_config,
    trust_remote_code=True,
)
```

Note:

- Ensure you have the required dependencies installed: `pip install transformers>=4.57.6 torchao==0.16.0`
- zentorch v5.2.1 is compatible with TorchAO. AMD Quark is no longer required for quantization.
- Use `MappingType.SYMMETRIC` for optimal performance with zentorch.
- Use `scale_dtype=torch.bfloat16` for compatibility with AMD EPYC™ CPU optimizations.
- Use `safe_serialization=False` when saving for compatibility with zentorch.
- For per-group quantization, we recommend a `group_size` of 128, as this configuration has been validated by zentorch across a broad set of mainstream models.

2.6.3 Running Quantized Models

Quantized models can be run using vLLM with the zentorch backend. No additional loading code (such as `zentorch.load_quantized_model`) is required — simply point vLLM to the quantized model directory. Refer to vLLM-zentorch Plugin for detailed instructions.

2.7 zentorch Optimal Environment Settings

The zentorch zip package which you can download from the AMD ZenDNN Developer Central page contains a convenient bash script to help you set optimal environment settings for best performance.

Before you run your workload, activate the conda environment where zentorch 5.2.1 is installed and source the *zentorch_env_setup.sh* file.

```
source scripts/zentorch_env_setup.sh --help
source scripts/zentorch_env_setup.sh --framework <zentorch> --model <llm|recsys|cnn|nlp> --threads <num_threads> --precision
 <amp|bf16|fp32|woq>
```

You can set the `num_threads` variable by checking the output of the following shell command:

```
lscpu | awk '/^Core\(s\) per socket:/ {print $4}'
```

For example, if you are running your LLM workload in BF16 format on an AMD 5th Gen EPYC™ Processor (codenamed Turin) with 128 cores, you would source the *zentorch_env_setup.sh* as follows:

```
source scripts/zentorch_env_setup.sh --framework zentorch --model llm --threads 128 --precision bf16
```

The script will make sure that necessary utilities like `llvm-openmp` as well as optimal tools for memory allocation (for example jemalloc) are installed and made available to zentorch.

Consult the Performance Tuning chapter for more details on the various environment variables.

2.8 Known Limitations

The following is a list of known limitations.

- The zentorch library requires the g++ compiler as a dependency. Ensure that the installed g++ version matches the system's gcc version.
- ALGO 3 has higher memory footprint during execution with AMP precision for a few models: Starcoder2-15b, Starcoder2-7b, and Qwen-QwQ-32B.
- We recommend using:
  - `ZENDNNL_MATMUL_ALGO=1` for CNN AMP & NLP AMP latency
  - `ZENDNNL_MATMUL_ALGO=5` for NLP AMP Throughput (mixed precision) models
- For best performance, we recommend running models with freezing enabled. This can be done by setting the environment variable: `export TORCHINDUCTOR_FREEZING=1`
- When running models with vLLM 0.18.0 and `export TORCHINDUCTOR_FREEZING=1`, set
  ```
  export VLLM_USE_AOT_COMPILE=0
  export TORCHINDUCTOR_AUTOGRAD_CACHE=0
  ```
- If you encounter an error while installing the sentencepiece package with Python 3.13, install it using:
  ```
  conda install -c conda-forge sentencepiece
  ```

Chapter 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 plugin 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 2.21.0. From a TensorFlow developer's perspective, the zentf approach simplifies the process of leveraging ZenDNN optimizations.

This section provides instructions to setup zentf v5.2.1.

3.1 Release Highlights

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

zentf v5.2.1 Release Highlights

- **TF 2.21 Integration**
  - zentf 5.2.1 is built for and validated against TensorFlow v2.21.0.
  - Upgraded from Bazel 7.4.1 to 7.7.0.
  - Python 3.10 - 3.13 -- Compatibility with TensorFlow (support for Python 3.9 discontinued).
  - As TensorFlow-Java has not been released for TensorFlow 2.21.0, zentf supports TensorFlow-Java main(75402bef) via source build only.
- **Backward Build Compatibility Support: TF 2.16 to 2.21**
  - zentf can now be built from source against TensorFlow versions 2.16.0 through 2.21.0 from a single unified codebase.
  - The `./configure` script auto-detects the installed TensorFlow version and applies the matching build configuration — no manual intervention required.
  - Per-version build configurations (`workspace`, `WORKSPACE`, `build_config.bzl`, `BUILD.tpl`) are maintained under `version_configs/` for TF 2.19, 2.20, and 2.21. Note that TF 2.16–2.18 share the TF 2.19 configuration.
  - Bazel version and third-party dependency settings (`protobuf`, `abseil`, `rules_cc`) are automatically adjusted to match the target TensorFlow version.
  - The standard build workflow (./configure > bazel build) remains unchanged; version adaptation is fully transparent.

zentf v5.2 Release Highlights

- **TF 2.20 Migration**
  - zentf 5.2.0 is built for and validated against TensorFlow v2.20.0.
  - Bazel 7.4.1: Upgraded from Bazel 5.3-6.5 range to a single supported version (7.4.1).
  - Python 3.9 - 3.13: Extended Python version support to include Python 3.13.
  - As TF JAVA is not released with 2.20 version, zentf is supported with main (75402bef) branch from TensoFlow-Java through source build only.
- **Migration from legacy ZenDNN library to ZenDNNL**
  - CMake-based ZenDNNL integration using `rules_foreign_cc`.
  - All operator kernels (MatMul, Conv2D, BatchMatMul, Softmax, Pooling) have been rewritten to use the ZenDNNL Low Overhead API (LOA), replacing the legacy ZenDNN primitives.
  - Old third-party dependencies on zen_dnn and amd_blis (BLIS) have been removed and replaced with ZenDNNL with integrated AOCL-DLP.
- **Removed Legacy Components**
  - Mempool optimization has been completely removed and equivalent performance has been achieved using jemalloc as the memory allocator instead.
  - INT8 support has been removed.
  - Removal of non-performant ops: ZenTranspose, ZenReshape, Binary ops.

3.2 Supported OS

Refer to the support matrix for the list of supported operating systems.

3.3 Install ZenDNN Plug-in for TensorFlow (zentf)

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

3.3.1 Using the Release Binary

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

Python Interface

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

1. PyPI Repo as a wheel (.whl) file.
2. 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.

C++ Interface

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

Note: zentf C++ package is tested with gcc 13.1.0.

Java Interface

Download the required zentf C++ package from the AMD Developer Portal.

For instructions on building the TensorFlow (TF) Java interface, refer to zentf Plugin with Java API of TensorFlow .

3.3.1.1 Install the Release Binary

This section provides information required to install zentf v5.2.1 for a Python interface.

However, if you are interested in installing zentf v5.2.1 on a C++ interface, click here for the README instructions.

Create and Setup Conda Environment

Before you begin:

- Choose a unique name for your Conda environment. Example: `zentf-5.2.1`
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `zentf-5.2.1` exists, use the following command to remove it.

```
conda remove --name zentf-5.2.1 --all
```

Important: zentf is compatible with Python v3.10-3.13. Make sure you create a Conda environment only with Python versions supported by zentf.

To setup the Conda environment:

1. Refer to the Miniforge documentation available here to install Miniforge on your system. Testing has been performed with *Miniforge3-24.11.3-2-Linux-x86_64.sh*.
2. Create and activate a Conda environment that houses all the zentf specific installations:

```
conda create -n zentf-5.2.1 python=3.10 -y
conda activate zentf-5.2.1
```

Install zentf

To install the zentf binary release:

1. Install TensorFlow v2.21.0.

```
pip install tensorflow==2.21.0
```

2. Use one of the following two methods to install zentf:
   a. Using the PyPi repo. Run the command:

```
pip install zentf==5.2.1
```

   For optimal environment settings, refer to Performance Tuning or use the script shipped in the

release package from the AMD developer portal.

b. Using the release package from the AMD developer portal.

1) Download the package from AMD developer portal.

2) Run the following commands to unzip the package:

```
unzip ZENTF_v5.2.1_Python_v3.10.zip
cd ZENTF_v5.2.1_Python_v3.10
```

> **Note:** zentf is compatible with Python v3.10-3.13. We have used 3.10 here only as an example.

3) Install the binary.

```
pip install  zentf-5.2.1-cp310-cp310-manylinux_2_28_x86_64.whl
```

4) To use the recommended environment settings, execute:

```
source scripts/zentf_env_setup.sh
```

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`.

3.3.2 Build from Source

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

1. Clone the repository and check out the r5.2.1 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.

3.4 Examples

Here are examples of running inference for various models in TensorFlow. Note that additional packages may be required in your environment. If the zentf plugin is already installed, you can add the remaining packages by running the following command:

```
pip install pillow transformers==4.48.3 tf-keras
```

3.4.1 BERT-based Model

```
import tensorflow as tf
from transformers import AutoTokenizer, TFBertModel, BertConfig
```

```
# Load the tokenizer, model and the model config
tokenizer = AutoTokenizer.from_pretrained("bert-large-uncased")
model_config = BertConfig.from_pretrained("bert-large-uncased")
model = TFBertModel.from_pretrained("bert-large-uncased",
                                    config=model_config,)

# Prepare inputs by tokenizing the examples
inputs = tokenizer("My puppy is very cute!", return_tensors="tf")

@tf.function
def generate():
    return model(**inputs)

# Generate outputs
outputs = generate()

# Get last hidden states
last_hidden_states = outputs.last_hidden_state

# Print the shape of the last hidden states
print("Last hidden states shape:", last_hidden_states.shape)
```

Sample Output

```
Last hidden states shape: (1, 8, 1024)
```

3.4.2 OPT

```
import tensorflow as tf
from transformers import AutoTokenizer, TFOPTForCausalLM

# Load the model and tokenizer
model = TFOPTForCausalLM.from_pretrained("facebook/opt-350m")
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")

# Run Inference
prompt = "Are you conscious? Can you talk?"

# Tokenize the input text
input_ids = tokenizer(prompt, return_tensors='tf').input_ids

@tf.function
def generate():
    return model.generate(input_ids, max_length=20)

# Run inference
outputs = generate()

# Decode the outputs
decoded_outputs = [tokenizer.decode(output, skip_special_tokens=True) for output in outputs]

for result in decoded_outputs:
    print(result)
```

Sample Output

```
I can talk, but I can't really think
```

3.4.3 ResNet

```
from PIL import Image

image = Image.open("airplane.jpg") # Choose an image of your choice and make sure it is in the same folder as this python
 script.

import tensorflow as tf
from transformers import AutoImageProcessor, TFResNetForImageClassification

# Load the model and image processor
model = TFResNetForImageClassification.from_pretrained("microsoft/resnet-50")
image_processor = AutoImageProcessor.from_pretrained("microsoft/resnet-50")


# Run Inference
inputs = image_processor(image, return_tensors="tf")

@tf.function
def predict():
    return model(**inputs)

logits = predict().logits

predicted_label = int(tf.math.argmax(logits, axis=-1))
print(model.config.id2label[predicted_label])
```

Sample Output

```
plane, carpenter's plane, woodworking plane
```

3.5 Limited Precision Support

zentf supports BF16 execution through Automatic Mixed Precision (AMP) optimization. To enable BF16 support, use the environment variable: `export TF_ZENDNN_PLUGIN_BF16=1`.

3.6 zentf Plugin with Java API of TensorFlow

This section provides the information you need to set up the zentf plugin for TensorFlow Java. This setup enables Java applications to exploit zentf in DNN inference using TensorFlow.

Prerequisites

Before building the project, ensure you have installed the following:

- Maven 3.6 or higher
- Java Development Kit (JDK) 11 or higher
- Environment variable `JAVA_HOME` set to your JDK installation path
- GLIBC v2.33 or higher

Note that you may need to set `JAVA_HOME` to the appropriate path to build this project with Maven. For example, set `JAVA_HOME` to the path: */usr/lib/jvm/java-<java-version>-openjdk-amd64*.

For example:

- For Ubuntu OS:
```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
```
- For RHEL OS:
```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk
export PATH=$JAVA_HOME/bin:$PATH
```

Note: We assume users have installed Java-v11. Check the version running on your machine by navigating to: */usr/lib/jvm* and set it accordingly.

Set up zentf Plugin for TensorFlow Java

Currently (as of v1.2.0-SNAPSHOT) TensorFlow Java with 2.20 version has not been released. Hence, you cannot use TensorFlow Java builds available publicly.

Complete the following steps to enable the zentf plugin for TensorFlow-Java:

1. Build TensorFlow-Java from source.
2. Set up zentf plugin.

Note: TensorFlow Java v1.2.0-SNAPSHOT supports TensorFlow v2.20.0.

Build TensorFlow Java from Source for the TF 2.20 Version

Run the script: `build_tf_java.sh` to get the source code of TensorFlow Java. Build and install TensorFlow Java from source.

We reproduce the following steps from `build_tf_java.sh` for easy reference.

```
git clone https://github.com/tensorflow/java.git tf_java.git
cd $current_dir/tf_java
git checkout 75402befedce0e1cf847b6f93d654b708a7db1db
mvn clean install
```

```
cd <Path to zentf plugin parent folder>/ZenDNN_TensorFlow_Plugin/scripts/java
bash build_tf_java.sh
```

Set up the zentf Plugin

1. Download the zentf C++ plugin package v5.2.1 built with TensorFlow v2.20.0 from the AMD Developer Forum.
2. Set the environment variable `LD_LIBRARY_PATH` to the zentf plugin libraries path.
```
unzip ZENTF_v5.2.1_C++_API.zip
export LD_LIBRARY_PATH=<Path to zentf C++ parent folder>/ZENTF_v5.2.1_C++_API/lib-tensorflow-plugins
```
3. Source the script to set the ZenDNN specific environment variables as shown here:
```
cd <Path to zentf C++ parent folder>/ZENTF_v5.2.1_C++_API
source zentf_env_setup.sh
```

Example

To try a setup on an example inference application, refer to the README file available at the *./examples/java* folder.

Here is an example of how to run the WideDeepLarge model.

1. Download the model.

```
wget https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_8/wide_deep_fp32_pretrained_model.pb
cd <Path to zentf plugin parent folder>/ZenDNN_TensorFlow_Plugin/examples/java
mvn clean package
```

2. Run the model.

```
java -cp target/tensorflow-benchmark-0.1-jar-with-dependencies.jar org.tensorflow.benchmark.RunWideDeeplarge <path to
 wide deep large .pb model> <batch size>
```

3. On successful execution, the output would be as shown here:

```
Batch Size = <batch size>
Output:
0
End of execution.
```

Chapter 4: Performance Tuning

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

4.1 Environment Variables

The environment variables to setup paths and control logs, and tune performance are enumerated here.

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

Table 4.1: ZenDNN Environment Variables common to all frameworks

Environment Variable	Description	Default Value/User Defined Value
	Generic (Setup paths and control logs)	
`ZENDNNL_<module>_LOG_LEVEL`	Enables ZenDNN logs. See Logging and Debugging for details on how to use logs.	0
`ZENDNNL_LRU_CACHE_CAPACITY`	Sets maximum capacity of LRU cache for blocked weights of MatMul algo. You can modify it as required [a].	1024
`ZENDNNL_EMBAG_THREAD_ALGO`	Sets Embedding Bag thread type. This is the recommended setting for RecSys models.	1
`ZENDNNL_EMBAG_ALGO`	Sets the embedding bag backend kernel. For FP32/BF16/INT4: • 1 = Native • 2 = FBGEMM	2
`OMP_DYNAMIC`	OMP variable to control dynamic adjustment of OMP threads. Refer to OpenMP documentation for details.	FALSE
	Optimized (Tune performance)	
`OMP_NUM_THREADS`	Sets the number of OMP threads. Generally, this is equal to the number of cores present. Set it based on the number of cores in the user system [a].	128

Table 4.1: ZenDNN Environment Variables common to all frameworks (continued)

Environment Variable	Description	Default Value/User Defined Value
`OMP_WAIT_POLICY`	Sets the behavior of waiting threads. Refer to the OMP documentation for details.	ACTIVE
`KMP_BLOCKTIME`	Sets the amount of time, in milliseconds, that a thread should wait before sleeping when a parallel region ends. Setting it to 1 minimizes idle time and can improve responsiveness for short tasks by quickly putting threads to sleep after work is complete. **Note:** Do not set this for Recommender System models.	1
`KMP_TPAUSE`	Controls the behavior of threads when they are waiting for work, aiming to reduce CPU usage. Setting it to 0 indicates threads should not enter an active wait state, optimizing CPU efficiency.	0
`KMP_FORKJOIN_BARRIER_PATTERN`	Specifies the synchronization pattern for fork/join barriers. dist,dist means a distributed barrier pattern is applied both when threads are forked and joined, potentially reducing synchronization contention.	dist,dist
`KMP_PLAIN_BARRIER_PATTERN`	Sets the synchronization pattern for plain barriers to dist,dist indicating a distributed pattern that helps manage thread synchronization efficiently during plain barriers.	dist,dist
`KMP_REDUCTION_BARRIER_PATTERN`	Controls the barrier pattern used in reduction operations (for example, sum or product of arrays across threads). Using dist,dist specifies a distributed pattern to enhance efficiency.	dist,dist

Table 4.1: ZenDNN Environment Variables common to all frameworks (continued)

Environment Variable	Description	Default Value/User Defined Value
`KMP_AFFINITY`	Determines how threads are bound to CPU cores. The setting granularity=fine,compact,1,0 specifies fine-grained affinity with threads compacted to as few cores as possible, minimizing memory access latency and maximizing cache utilization, respectively.	granularity=fine,compact,1,0
`ZENDNNL_MATMUL_ALGO`	Specifies the MatMul algo to be used. For FP32/BF16/INT8: • AUTO (Auto-Tuner) • 0 = Static Decision Tree • 1 = AOCL_DLP (Blocked with weight-caching) • 2 = oneDNN (Blocked with weight-caching) • 3 = LIBXSMM-blocked • 4 = AOCL_DLP • 5 = oneDNN • 6 = LIBXSMM Auto is an experimental feature and should be used with application warm up iteration >=8. **Note:** Different workloads on different frameworks (PyTorch, TensorFlow) have specific `ZENDNNL_MATMUL_ALGO` settings for optimized performance. See Optimal Environment Variable Settings for zentorch and Optimal Environment Variable Settings for zentf.	`ZENDNNL_MATMUL_ALGO=1`
[a] You must set these environment variables explicitly.		

LLVM OpenMP

LLVM OpenMP runtimes provides the necessary libraries and compiler directives for implementing parallelism in programs.

Developers can use LLVM OpenMP 18.1.18 to compile and run parallel programs written in Fortran and C/C++, taking advantage of shared memory parallelism and improving the performance and scalability of their applications.

The LLVM OpenMP implementation supports various features, including:

- Compiler directives for specifying parallel regions, tasks, and data dependencies
- Library routines for creating and managing teams, parallel loops, and synchronization
- Environment variables for controlling OpenMP behavior

Complete the following steps to install and leverage llvm openmp in your Conda environment:

1. `conda install -c conda-forge llvm-openmp=18.1.8=hf5423f3_1 --no-deps -y`
2. `export LD_PRELOAD="<path to conda>/pkgs/llvm-openmp-18.1.8-hf5423f3_1/lib/libiomp5.so: $LD_PRELOAD"`

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

Table 4.2: zentf Environment Variables-Generic

Environment Variable	Description	Default Value/ User Defined Value
`TF_ENABLE_ZENDNN_OPTS`	Set `TF_ENABLE_ONEDNN_OPTS=0` when you want to enable vanilla training and inference. Set it to 1 along with `TF_ENABLE_ONEDNN_OPTS=0` to enable ZenDNN for inference.	0
`TF_ENABLE_ONEDNN_OPTS`	By default, TensorFlow is shipped with oneDNN optimizations enabled. Hence, set it to 0 when you enable ZenDNN.	1
`TF_ZENDNN_PLUGIN_BF16`	Set it to 1 to enable Automatic Mixed Precision (AMP) for BF16.	0
zentf Environment Variables-Optimization		
`USE_ZENDNN_MATMUL_DIRECT`	For optimal single core MatMul execution modify it to 1	1

4.2 Performance Tuning Guidelines

Hardware configuration, OS, Kernel, and BIOS settings play an important role in performance. Details of the environment variables used on a 5th Gen AMD EPYC™ server to get the best performance numbers are enumerated in the following sections.

Recommendation

For optimal performance with vLLM CPU inference, set the temperature parameter to 0.0 and use supported x86 CPUs (with best results on the latest AMD EPYC™ CPUs).

4.3 System Used for Performance Tuning

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

Table 4.3: System Specification

Specification	Value
Model Name	5th Gen AMD EPYC™ 9755 128-Core Processor
CPU MHz	Up to 4.1 GHz
Core(s) per Socket	128
Socket(s) used	2
Thread(s) per Core	2
Mem-Dims	24x64 GB

4.4 Common Optimal Environment Variable Settings

The following environment variable settings are common to both frameworks.

- `OMP_NUM_THREADS=128 # For a system with 128 cores per socket`
- `OMP_WAIT_POLICY=ACTIVE`
- `OMP_DYNAMIC=FALSE`
- `ZENDNNL_MATMUL_ALGO=1`

Note: `ZENDNNL_MATMUL_ALGO=1` is mandatory for vLLM.

The environment variables `OMP_NUM_THREADS`, `OMP_WAIT_POLICY`, and `OMP_PROC_BIND`, can be used to tune performance of both frameworks. These are OpenMP variables. Refer to the OpenMP documentation for details.

For achieving the best performance in zentf, use KMP variables, refer Table 4.1 for further details.

For achieving the best performance in zentf Recommender System models, use `GOMP_CPU_AFFINITY=0-127. # For a system with 128 cores per socket`.

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 `PASSIVE`. 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.

4.5 Thread Affinity

Important: The contents of this section do not apply to vLLM.

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. Use the following methods to 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=0-127`, binds the threads to CPUs 0-127.

Note: `GOMP_CPU_AFFINITY` will be ignored if you export the `KMP_AFFINITY` variable.

4.6 Non-uniform Memory Access

Important: The contents of this section do not apply to vLLM.

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.

4.7 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 and LLM models: `madvise`
- For zentf
  - CNN models: `always`
  - NLP and Recommender models: `madvise`

4.8 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.

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.

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 TCMalloc and jemalloc dynamic libraries and use the `LD_PRELOAD` environment variable as follows:

Table 4.4: LD_PRELOAD environment variables in case of TCMalloc and jemalloc

Use this command	TCMalloc	jemalloc
Before you begin	`export LD_PRELOAD=/path/to/TCMallocLib/`	`export LD_PRELOAD=/path/to/jemallocLib/`
For benchmarking	`LD_PRELOAD=/path/to/TCMallocLib/ < python benchmarking command>`	`LD_PRELOAD=/path/to/jemallocLib/ <python benchmarking command>`
To verify if TCMalloc or jemalloc memory allocator is in use	`lsof -p <pid_of_benchmarking_command> | grep tcmalloc`	`lsof -p <pid_of_benchmarking_command> | grep jemalloc`

4.9 Optimal Environment Variable Settings for zentorch

The following environment variable settings are optimal settings for zentorch, and should be used in addition to the environment variable settings.

- CNN-based models
  - FP32 models
    - ZENDNNL_MATMUL_ALGO=1
  - BF16 (AMP) models
    - ZENDNNL_MATMUL_ALGO=1
- NLP-based models
  - FP32 models and BF16 (AMP) models Latency
    - ZENDNNL_MATMUL_ALGO=1
  - FP32 models and BF16 (AMP) models Throughput
    - ZENDNNL_MATMUL_ALGO=5
- LLM-based models
  - BF16 and WOQ (Per channel and Per group) models
    - ZENDNNL_MATMUL_ALGO=1
- For RecSys models
  - FP32, BF16, INT8 and BF16 (AMP) models
    - ZENDNNL_MATMUL_ALGO=1

4.10 Optimal Environment Variable Settings for zentf

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

- Use LLVM OPENMP for all models except Recommendation System (Recsys) models
  - For Recsys models, use `GOMP_CPU_AFFINITY`
- Use KMP settings for all models except Recsys models
  - `export KMP_BLOCKTIME=1`
  - `export KMP_TPAUSE=0`
  - `export KMP_FORKJOIN_BARRIER_PATTERN=dist,dist`
  - `export KMP_PLAIN_BARRIER_PATTERN=dist,dist`
  - `export KMP_REDUCTION_BARRIER_PATTERN=dist,dist`
  - `export KMP_AFFINITY=granularity=fine,compact,1,0`
- Use Jemalloc as memory allocator
- `TF_NUM_INTEROP_THREADS=1` (for CNN, Hugging Face NLP and LLM Models)
- `TF_NUM_INTRAOP_THREADS=128` (for CNN, Hugging Face NLP and LLM Models on Turin machine)
- `OMP_PROC_BIND=FALSE`
- `USE_ZENDNN_MATMUL_DIRECT=1`
- `ZENDNNL_MATMUL_WEIGHT_CACHE=1`
- `ZENDNNL_MATMUL_ALGO`

NLP & LLM models

- For FP32 and direct BF16 models
  - Throughput:`export ZENDNNL_MATMUL_ALGO=2`
  - Latency: `export ZENDNNL_MATMUL_ALGO=1`
- For BF16 (AMP) models: `export ZENDNNL_MATMUL_ALGO=4`

DIEN (Recsys) models

- For FP32 and direct BF16 models: `export ZENDNNL_MATMUL_ALGO=1`
- For BF16 (AMP) models: `export ZENDNNL_MATMUL_ALGO=4`

CNN models

- For FP32 and direct BF16 models: `export ZENDNNL_MATMUL_ALGO=1`
- For BF16 (AMP) models: `export ZENDNNL_MATMUL_ALGO=4`

Chapter 5: Logging and Debugging

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

5.1 ZenDNN Library Logs

Logging is disabled in the ZenDNN library by default. It can be enabled using the environment variable `ZENDNNL_<log_module>_LOG_LEVEL` before running any test. `ZENDNNL_<log_module>_LOG_LEVEL` sets the default log level for a specific module. For example, the command `export ZENDNNL_API_LOG_LEVEL=4` sets the log level of a module (Example: API, PROFILE) to verbose (4).

The different ACTORS are as follows.

Table 5.1: Log Actors

Actor	Description
COMMON	General logging for common operations.
API	Logging related to API calls and interfaces.
TEST	Logs for testing and validation purposes.
PROFILE	Metrics and performance-related logs.
DEBUG	Detailed debugging information.

Example

- To only log info for COMMON, use `ZENDNNL_COMMON_LOG_LEVEL=4`
- To only log info for API, use `ZENDNNL_API_LOG_LEVEL=4`
- To only log info for TEST, use `ZENDNNL_TEST_LOG_LEVEL=4`
- To only log info for DEBUG, use `ZENDNNL_DEBUG_LOG_LEVEL=4`
- To only log info for PROFILE, use `ZENDNNL_PROFILE_LOG_LEVEL=4`

Enable Log Profiling

To enable profiling logs, set the log level for the PROFILE module to verbose (4):

```
export ZENDNNL_ENABLE_PROFILER=1
```

```
export ZENDNNL_PROFILE_LOG_LEVEL=4
```

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

Table 5.2: Debug Levels

Debug Level	Value
`LOG_LEVEL_DISABLED`	0

Table 5.2: Debug Levels (continued)

Debug Level	Value
`LOG_LEVEL_ERROR`	1
`LOG_LEVEL_WARNING`	2
`LOG_LEVEL_INFO`	3
`LOG_LEVEL_VERBOSE`	4

5.2 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.

Note: 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.

5.3 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
```

Chapter 6: Support

We welcome feedback, suggestions, and bug reports.

If you need technical support on ZenDNN, please file an issue ticket on the respective Github page:

- ZenDNN Library: https://github.com/amd/ZenDNN
- ZenDNN Plugin for PyTorch: https://github.com/amd/ZenDNN-pytorch-plugin
- ZenDNN Plugin for TensorFlow: https://github.com/amd/ZenDNN-tensorflow-plugin

Appendix A: Additional Resources and Legal Notices

A.1 Revision History

A summary of the revisions made to this document.

Table A.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
3	09-Jul-2024	• Added TCMalloc information to the Performance Tuning guidelines. • Updated the list of installation commands for ONNX Runtime release binary.
4	08-Nov-2024	Documented 5.0 release features and updated Readme for ZenDNN 5.0.
5	03-Mar-2025	Updates to 5.0.1 release.
6	09-Apr-2025	Updates to 5.0.2 release.
7	18-Aug-2025	Updates to 5.1 release.
8	13-Mar-2026	Updates to 5.2 release.
9	22-Apr-2026	Updates to 5.2.1 release.

A.2 Legal Notices

Appendix B: Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.1 Trademarks

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

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