ZenDNN User Guide

Publication Number: 57300 Revision: 5.0.1
Date: March 2025

Contents

Chapter 1: ZenDNN.....4

1.1 Scope.....4

1.2 Release Highlights.....4

1.3 High-level Overview.....6

1.4 Build from Source.....6

Chapter 2: PyTorch.....8

2.1 Release Highlights.....8

2.2 Supported OS.....8

2.3 Install ZenDNN Plug-in for PyTorch (zentorch).....8

2.3.1 Using the Release Binary.....9

2.3.2 Build from Source.....10

2.4 Usage.....10

2.4.1 Using torch.compile.....11

2.4.2 Examples.....11

2.4.3 Recommendations.....15

2.5 Limited Precision Support.....17

2.5.1 Weight Only Quantized Models.....17

2.6 zentorch Optimal Environment Settings.....19

2.7 Known Limitations.....19

Chapter 3: TensorFlow.....20

3.1 Release Highlights.....20

3.2 Supported OS.....20

3.3 Install ZenDNN Plug-in for TensorFlow (zentf).....20

3.3.1 Using the Release Binary.....21

3.3.2 Build from Source.....22

3.4 Examples.....22

3.4.1 BERT-based Model.....23

3.4.2 OPT.....23

3.4.3 ResNet.....24

3.5 Limited Precision Support........ 24

Chapter 4: Performance Tuning........ 25

4.1 Environment Variables........ 25

4.2 Performance Tuning Guidelines........ 30

4.3 System Used for Performance Tuning........ 31

4.4 Common Optimal Environment Variable Settings........ 31

4.5 Thread Affinity........ 32

4.6 Non-uniform Memory Access........ 32

4.7 Transparent Huge Pages........ 33

4.8 Memory Allocators........ 34

4.9 Optimal Environment Variable Settings for zentf........ 35

Chapter 5: Logging and Debugging........ 36

5.1 ZenDNN Library Logs........ 36

5.2 zentorch Logging and Debugging........ 37

5.3 Debugging........ 37

Chapter 6: Support........ 39

Appendix A: Additional Resources and Legal Notices........ 40

A.1 Revision History........ 40

A.2 Legal Notices........ 40

Appendix B: Notices........ 42

B.1 Trademarks........ 42

Chapter 1: ZenDNN

The latest ZenDNN 5.0.1 is here!

ZenDNN 5.0.1 is a minor release building upon the major ZenDNN 5.0 release. This upgrade continues the focus on optimizing inference performance with Recommender Systems and Large Language Models on AMD EPYC™ CPUs. ZenDNN includes AMD EPYC™ enhancements for bfloat16 performance, expanded support for cutting-edge models like Llama 3.1 and 3.2, Microsoft Phi, and more as well as support for INT4 quantized datatype. This includes the advanced Activation-Aware Weight Quantization (AWQ) algorithm.

Under the hood, ZenDNN's enhanced AMD-specific optimizations operate at every level. In addition to highly optimized operator microkernels, these include comprehensive graph optimizations including pattern identification, graph reordering, and fusions.

Notable improvements include optimized embedding bag kernels and enhanced zenMatMul matrix splitting strategies, both designed to maximize throughput and minimize latency.

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.

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 5.0.1

- **Compatibility with Deep-learning Frameworks**: Fully aligned with PyTorch 2.5 and TensorFlow 2.18, ensuring smooth upgrades and interoperability.
- **Efficient Model Execution**: Added support for INT8/INT4-quantized DLRM models in zentorch, unlocking faster inference with lower memory usage compared to BF16-precision. This release supports the MLPerf version of DLRMv2; support for generic models will be added in the next release.

Native Framework Support

- The ZenDNN library is based on oneDNN v2.6.3
- The ZenDNN library can be used in the following frameworks through a plug-in:
  - TensorFlow v2.16 and later

    > **Note:** The ZenDNN 5.0.1 plug-in for TensorFlow is optimized to give the best performance with TensorFlow v2.18.

  - PyTorch v2.2 to v2.5.

    > **Note:** The ZenDNN 5.0.1 plug-in for PyTorch is optimized to give the best performance with PyTorch v2.5.0.

- In ZenDNN 5.0, the ZenDNN library is directly integrated with ONNX Runtime v1.19.2. As of ZenDNN 5.0.1, support for ONNXRT has been temporarily paused.

> **Note:** 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 v3.9-v3.12
    - PyTorch v2.5.0
  - zentf wheel files (*.whl) have been generated using:
    - Python v3.9-v3.12
    - TensorFlow v2.18

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

Highlights of Previous Major Release ZenDNN 5.0

- Support for the Zen5 family of AMD EPYC™ processors, codenamed Turin
- Compatibility with AOCL BLIS 5.0
- AMD EPYC™ specific enhancements to matmul operators and related fusions, specifically for BF16 precision
- An auto-tuning algorithm BF16:0 specifically targeting generative LLM models. Support for weight only quantization (WOQ) with INT4 weights and BF16 activations for LLMs; ZenDNN 5.0 natively supports models optimized and exported using the AMD Quantizer Quark.
- AMD EPYC™ specific enhancements for WOQ matmul operators and related fusions
- Performance enhancements targeted at generative LLM models using the function

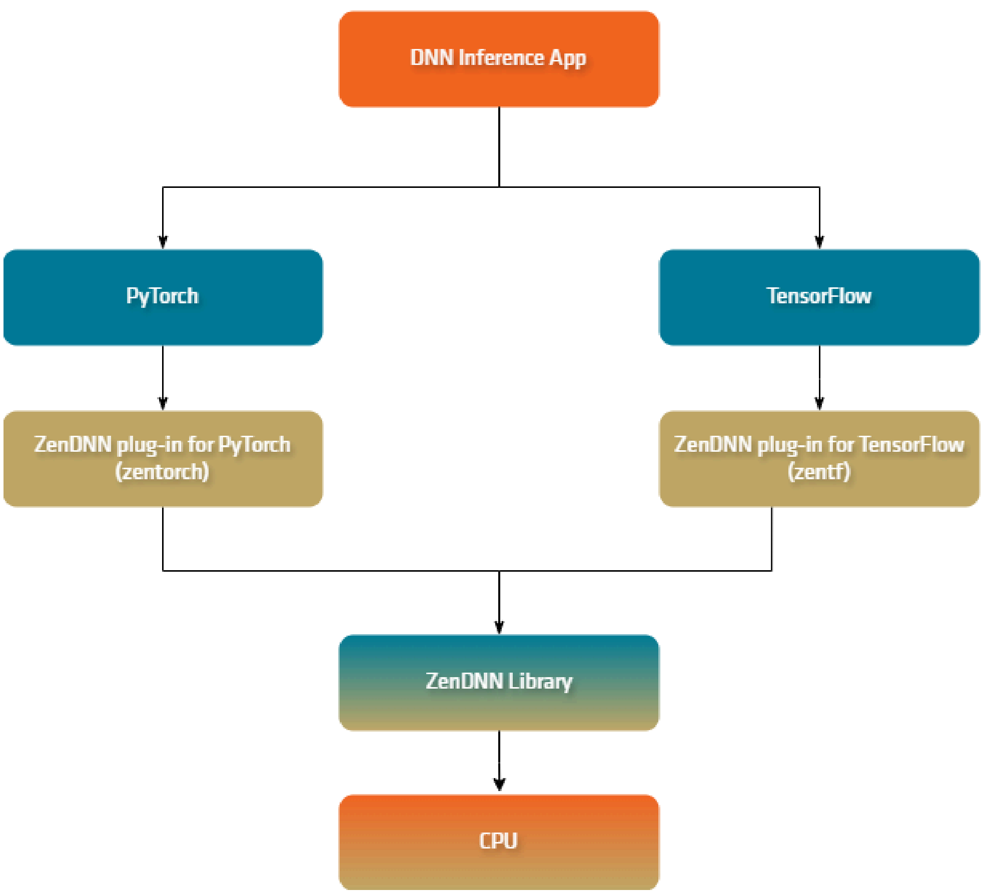
`zentorch.llm.optimize( )` available in the ZenDNN PyTorch plug-in; this function contains additive AMD EPYC™ specific optimizations on top of the x86 optimizations available in `ipex.llm.optimize()`

- An optimized Scalar Dot Product Attention (SDPA) operator in the PyTorch plug-in, including KV cache performance optimizations tailored to AMD EPYC™ cache architectures
- Support for BF16 precision for Recommender System models in the PyTorch plug-in
- Graph optimization and pattern matching improvements in the PyTorch plug-in

1.3 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) and the ZenDNN plug-in for TensorFlow (zentf). The ZenDNN library uses the AOCL-BLIS library internally, as well as other third-party libraries such as FBGEMM.

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 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.

2.1 Release Highlights

zentorch is compatible with base versions of PyTorch v2.2 or later. This release provides zentorch for PyTorch v2.5.0.

zentorch 5.0.1 Release highlights

- Added support for Pytorch 2.5.0
- Added support for INT8/INT4-quantized DLRM models in zentorch, unlocking faster inference with lower memory usage compared to BF16-precision. This release supports the MLPerf version of DLRMv2; support for generic models will be added in the next release.
- Support for weight caching when running LLM/NLP models with Auto Mixed Precision (AMP) between FP32 and BF16

Previous Major Release Highlights

zentorch 5.0

- Datatypes FP32, BF16, INT8 and INT4 (WOQ)
- Introduction of a new `zentorch.llm.optimize()` method for Hugging Face Generative LLM models
- New `zentorch.load_woq_model()` method to support loading of Weight Only Quantized models generated through the AMD Quark tool. This method only supports models quantized and exported with per-channel quantization using the AWQ algorithm.
- Improved graph optimizations, enhanced SDPA (Scalar Dot Product Attention) operator and more.
- Automatic Mixed Precision (AMP) between FP32 and BF16 providing a performance improvement with minimal changes in accuracy

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-5.0.1`.
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `zentorch-5.0.1` exists, use the following command to remove it.
  ```
  conda remove --name zentorch-5.0.1 --all
  ```

Important: zentorch is compatible with Python v3.9-3.12. 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 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-5.0.1 python=3.10 -y
   conda activate zentorch-5.0.1
   ```

Install zentorch

To install the zentorch release binary:

1. Install PyTorch v2.5.0.
   ```
   pip install torch==2.5.0 --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==5.0.1
      ```
      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_v5.0.1_Python_v3.10.zip
cd ZENTORCH_v5.0.1_Python_v3.10/
```

Note: zentorch is compatible with Python v3.9-3.12. We have used 3.10 here only as an example.

3) Install the binary.

```
pip install zentorch-5.0.1-cp310-cp310-manylinux_2_28_x86_64.whl
```

4) To use the recommended environment settings, execute:

```
source scripts/benchmarking_optimal_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-5.0.1-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 r5.0.1 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. Additionally, for Hugging Face large language models, we provide `zentorch.llm.optimize()`, a specialized method that delivers further performance enhancements. 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. 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
```

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 Hugging Face Language Models

Here is an example of using the new `zentorch.llm.optimize()` method in BFloat16.

For this example, you will need a Hugging Face token and configure accordingly in the code snippet below.

```
import torch
import zentorch
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load Tokenizer and Model
model_id = "meta-llama/Llama-3.1-8B"
model = AutoModelForCausalLM.from_pretrained(
            model_id,
            torchscript=True,
            return_dict=False,
            torch_dtype=torch.bfloat16,
        )
tokenizer = AutoTokenizer.from_pretrained(model_id, trust_remote_code=True)
model = model.eval()


# Prepare Inputs
generate_kwargs = dict(
            do_sample=False,
            temperature=0.0,
            num_beams=4,
            max_new_tokens=10,
            min_new_tokens=2,
        )
prompt = "Hi, How are you today?"

# Inference
############### Code modification ###############
model = zentorch.llm.optimize(model, dtype=torch.bfloat16)
#################################################

with torch.inference_mode(), torch.no_grad(), torch.amp.autocast('cpu', enabled=True):
    ############### Code modification ###############
    model.forward = torch.compile(model.forward, backend="zentorch")
    #################################################
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids
    output = model.generate(input_ids, **generate_kwargs)
    gen_text = tokenizer.batch_decode(output, skip_special_tokens=True)
```

```
print(gen_text)
```

Sample output

```
Hi! I'm here and ready to help. How are you?
```

ZenDNN supports INT4 weight-only quantization with BFloat16 activations (W4A16). Here is an example of how to load a pre-quantized INT4 model from Hugging Face. This model has been quantized using AMD Quark. For detailed instructions on using the tool, please refer to the AMD Quark User Guide.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, AutoConfig
import zentorch

# Load Tokenizer and Model
model_id = "meta-llama/Llama-3.1-8B"
config = AutoConfig.from_pretrained(
            model_id,
            torchscript=True,
            return_dict=False,
            torch_dtype=torch.bfloat16,
        )
model = AutoModelForCausalLM.from_config(config, trust_remote_code=True, torch_dtype=torch.bfloat16)

# Load WOQ model
############### Code modification ###############
safetensor_path = "<Path to Quantized Model"
model = zentorch.load_quantized_model(model, safetensor_path)
model = model.eval()
#################################################

tokenizer = AutoTokenizer.from_pretrained(model_id, trust_remote_code=True, padding_side="left",use_fast=False)

# Prepare Inputs
generate_kwargs = dict(
            do_sample=False,
            temperature=0.0,
            num_beams=4,
            max_new_tokens=10,
            min_new_tokens=2,
        )

prompt = "Hi, How are you today?"

# Inference
############### Code modification ###############
model = zentorch.llm.optimize(model, dtype=torch.bfloat16)
#################################################

with torch.inference_mode(), torch.no_grad(), torch.amp.autocast('cpu', enabled=True):
    ############### Code modification ###############
    model.forward = torch.compile(model.forward, backend="zentorch")
    #################################################

    input_ids = tokenizer(prompt, return_tensors="pt").input_ids
    output = model.generate(input_ids, **generate_kwargs)
    gen_text = tokenizer.batch_decode(output, skip_special_tokens=True)

print(gen_text)
```

Sample Output

```
Hello! I'm here and ready to help. How are you?
```

2.4.2.3 Recommendation Systems with DLRM

Below is the main code snippet showing how you can accelerate DLRM with zentorch.

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/main/examples/dlrm_model.py
```

```
# 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 = ['int_0', 'int_1', 'int_2', 'int_3', 'int_4', 'int_5', 'int_6', 'int_7', 'int_8', 'int_9', 'int_10',
 'int_11', 'int_12']
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.inference_mode(), torch.no_grad(), torch.amp.autocast('cpu', enabled=True):
    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
```

2.4.2.4 ResNet

```
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")
#################################################

with torch.inference_mode(), torch.no_grad(), torch.amp.autocast('cpu', enabled=True):
    logits = model(**inputs).logits

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

Sample Output

```
plain airliner
```

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

For Hugging Face Generative LLM models, usage of `zentorch.llm.optimize` is recommended. All optimizations included in this API are specifically targeted for Generative Large Language Models from Hugging Face. If a model is not a valid Generative Large Language Model from Hugging Face, the following warning will be displayed and `zentorch.llm.optimize` will act as a dummy with no optimizations applied to the model that is passed to the method:

```
“Cannot detect the model transformers family by model.config.architectures. Please pass a valid Hugging Face LLM model to the
 zentorch.llm.optimize API.”
```

This check confirms the presence of the "config" and "architectures" attributes of the model to get the model ID. Considering the check, two scenarios the `zentorch.llm.optimize` can still act as a dummy function:

1. Hugging Face has a plethora of models, of which Generative LLMs are a subset of. So, even if the model has the attributes of `config` and `architectures`, the model ID might not yet be present in the supported models list from zentorch. In this case `zentorch.llm.optimize` will act as a dummy function.

   A model can be a valid generative LLM from Hugging Face but may miss the `config` and `architectures` attributes. In this case also, the `zentorch.llm.optimize` API will act as a dummy function.

2. If the model passed is valid, all the supported optimizations will be applied, and performant execution is ensured. To check the supported models, run the following command:

   ```
   python -c 'import zentorch; print("\n".join([f"{i+1:3}. {item}" for i, item in
    enumerate(zentorch.llm.SUPPORTED_MODELS)]))'
   ```

If a model ID other than the listed above are passed, `zentorch.llm.optimize` will not apply the above specific optimizations to the model and the following warning will be displayed:

```
“Complete set of optimizations are currently unavailable for this model.”
```

Control will pass to the “zentorch” custom backend in `torch.compile` for applying optimizations.

Note: To leverage the best performance of `zentorch_llm_optimize`, install IPEX corresponding to the PyTorch version that is installed in the environment.

The PyTorch version for performant execution of supported LLMs should be greater than or equal to 2.3.0. The recommended version for optimal performance is PyTorch 2.5.0.

Case #1: If output is generated through a call to direct model, optimize it as shown here:

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

Case #2. If output is generated through a call to `model.forward`, optimize it as shown here:

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

Case #3: If output is generated through a call to `model.generate`, optimize it as shown here:

- Optimize the `model.forward` with `torch.compile` instead of `model.generate`
- However, proceed to generate the output through a call to `model.generate`

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

Note: For PyTorch versions lower than 2.3.0, 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.

2.5 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 supports Weight Only Quantization with INT4 weights and BF16 activations as described in Weight Only Quantized Models.

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

2.5.1 Weight Only Quantized Models

Hugging Face models are quantized using the AMD Quark tool. After downloading the zip file, install Quark and follow these steps:

1. Navigate to the `examples/torch/language_modeling/llm_ptq/` directory.
2. Install the necessary dependencies:

```
pip install -r requirements.txt
pip install -r ../llm_eval/requirements.txt
```

3. Run the following command to quantize the model:
   - For per-channel quantization:

```
OMP_NUM_THREADS=<physical-cores-num> numactl --physcpubind=<physical-cores-list> python quantize_quark.py
--model_dir <hugging_face_model_id> --device cpu --data_type bfloat16 --model_export hf_format
--custom_mode awq --quant_algo awq --quant_scheme w_int4_per_group_sym --group_size -1
--num_calib_data 128 --dataset pileval_for_awq_benchmark --seq_len 128 --output_dir <output_dir>
--pack_method order
```

   - For per-group quantization:

```
OMP_NUM_THREADS=<physical-cores-num> numactl --physcpubind=<physical-cores-list> python quantize_quark.py
--model_dir <hugging_face_model_id> --device cpu --data_type bfloat16 --model_export hf_format --custom_mode awq
--quant_algo awq --quant_scheme w_int4_per_group_sym --group_size <group_size> --num_calib_data 128
--dataset pileval_for_awq_benchmark --seq_len 128 --output_dir <output_dir> --pack_method order
```

Note: The channel/out_features dimension (property of your model) must be divisible by the specified group_size. To find out the values for channel and out_features in your model, refer to the model definition. We recommend using a group_size of 128, as this configuration has been validated by zentorch across a broad set of mainstream models.

For example:

The **Llama-3.2** model contains multiple linear layers subject to quantization, with out_features values of [2048, 512, 512, 2047, 8192, 8192, 2048, 128256].

Similarly, the **Llama-2** model has linear layers that can be quantized with out_features values of [4096, 4096, 4096, 4096, 11008, 11008, 4096, 32000].

The **ChatGLM** model includes linear layers with out_features values of [4068, 4096, 27392, 4096, 65024].

For effective quantization, the chosen group_size must be a factor of each channel/out_features value within the model.

```
OMP_NUM_THREADS=<physical-cores-num> numactl --physcpubind=<physical-cores-list> python quantize_quark.py
--model_dir <hugging_face_model_id> --device cpu --data_type bfloat16 --model_export quark_safetensors
--quant_algo awq --quant_scheme w_int4_per_group_sym --group_size -1 --num_calib_data 128
--dataset pileval_for_awq_benchmark --seq_len 128 --output_dir <output_dir> --pack_method order
```

Note: zentorch v5.0.1 is compatible with Quark v0.8. Make sure you download the right version.

Table 2.1: Constraints for zentorch WOQ with the AWQ algorithm

Constraint	Remarks
`--device cpu`	zentorch only supports CPU device.
`--data_type bfloat16`	Currently, zentorch only supports the BFloat16 model data type.
`--group_size -1`	`group-size -1` refers to per-channel quantization; for per-group quantization, the channel/out_features dimension should be divisible by group_size value.
`--quant_algo awq`	Currently, the zentorch release supports only the AWQ quantization algorithm.
`--quant_scheme w_int4_per_group_sym`	Currently, the zentorch release supports only the w_int4_per_group_sym quantization scheme.
`--packing_method order`	Currently, the zentorch release supports only the packing_method order.

As Hugging Face currently does not support the AWQ format for CPU, an additional codeblock has to be added to your inference script for loading the WOQ models.

```
config = AutoConfig.from_pretrained(model_id, trust_remote_code=True, torch_dtype=torch.bfloat16)
model = AutoModelForCausalLM.from_config(config, trust_remote_code=True, torch_dtype=torch.bfloat16)
model = zentorch.load_quantized_model(model, safetensor_path)
```

Here, the safetensor_path refers to the "<output_dir>" path of the quantized model. After the loading steps, the model can be executed in a similar fashion as the cases # 1-3 listed in Recommendations (Hugging Face Generative LLM Models).

Note: From zentorch 5.0.1, the `load_woq_model()` API is deprecated and will be removed in future releases. Use `load_quantized_model()` API instead.

2.6 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 5.0.1 is installed and source the *benchmarking_optimal_env_setup.sh* file.

```
source scripts/benchmarking_optimal_env_setup.sh --help
source scripts/benchmarking_optimal_env_setup.sh --framework <zentorch|ipex> --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 192 cores, you would source the *benchmarking_optimal_env_setup.sh* as follows:

```
source scripts/benchmarking_optimal_env_setup.sh --framework zentorch --model llm --threads 192 --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.

2.7 Known Limitations

ChatGLM and Falcon-7B are currently compatible with ZenDNN 5.0 and may not work as expected with ZenDNN 5.0.1.

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

This section provides instructions to setup zentf v5.0.1.

3.1 Release Highlights

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

This zentf release:

- Supports TensorFlow v2.18.
- Integrates with ZenDNN v5.0.1 as the core inference library and is compiled with GCC v12.2.

ZenTF 5.0 Release highlights

- Merged BF16 and FP32 compute flows and added broadcasting support for BatchMatMul kernel.
- INT8 support for the ResNet50 model.
- Softmax kernel supports up to 5D.
- Deprecated blocked format support for convolution ops and restriction of rewrite for the fused ops based on the post ops.
- Provides experimental support of C++ APIs.

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 or C++ 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.

3.3.1.1 Install the Release Binary

This section provides information required to install zentf v5.0.1 for a Python interface.

However, if you are interested in installing zentf v5.0.1 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-5.0.1`
- Make sure that you delete any older Conda environment with the same name. For example: If a Conda environment named `zentf-5.0.1` exists, use the following command to remove it.
  ```
  conda remove --name zentf-5.0.1 --all
  ```

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 zentf-5.0.1 python=3.10 -y
   conda activate zentf-5.0.1
   ```

Install zentf

To install the zentf binary release:

1. Install TensorFlow v2.18.
   ```
   pip install tensorflow-cpu==2.18
   ```

2. Use one of the following two methods to install zentf:

   a. Using the PyPi repo. Run the command:
   ```
   pip install zentf==5.0.1
   ```
   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_v5.0.1_Python_v3.10.zip
      cd ZENTF_v5.0.1_Python_v3.10
      ```
      Note: 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-5.0.1-cp310-cp310-manylinux_2_28_x86_64.whl
      ```
      4) To use the recommended environment settings, execute:
      ```
      source scripts/zentf_env_setup.sh
      ```
      5) Install requirements:
      ```
      pip install transformers==4.48.3
      ```

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 r5.0.1 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 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

```
plain airliner
```

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`.

Note: For INT8, computations fall back to the native framework except for the ResNet50 model.

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)	
`ZENDNN_LOG_OPTS`	Enables ZenDNN logs. See Logging and Debugging for details on how to use logs.	ALL:0
`ZENDNN_PARENT_FOLDER`	Path to the folder where the unzipped ZenDNN folder is located.	Path to unzipped release folder.
`ZENDNN_PRIMITIVE_CACHE_CAPACITY`	Sets maximum capacity of LRU cache for primitives. You can modify it as required [a].	1024
`ZENDNN_WEIGHT_CACHE_CAPACITY`	Sets maximum capacity of LRU cache for blocked weights of MatMul algo. You can modify it as required [a].	1024
`ZENDNN_EB_THREAD_TYPE`	Sets Embedding Bag thread type. This is the recommended setting for RecSys models.	1
`OMP_DYNAMIC`	OMP variable to control dynamic adjustment of OMP threads. Refer to OpenMP documentation for details.	FALSE
	Optimized (Tune performance)	

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

Environment Variable	Description	Default Value/User Defined Value
`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
`OMP_WAIT_POLICY`	Sets the behavior of waiting threads. Refer to the OMP documentation for details.	ACTIVE
`GOMP_CPU_AFFINITY`	Binds threads to specific CPUs. This is a GNU OpenMP library flag and will work only with GNU OpenMP.	Set it based on the number of cores in the system being used. For example, use 0-127 for 128-core servers.

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

Environment Variable	Description	Default Value/User Defined Value

`ZENDNN_MATMUL_ALGO`	Specifies the MatMul algo to be used.
	For FP32/BF16/INT8:
	• AUTO (Auto-Tuner)
	• 0 = Static Decision Tree
	• 1 = AOCL_BLIS (Blocked with weight-caching)
	• 2 = BRGEMM (Blocked with weight-caching)
	• 3= AOCL_BLIS
	• 4 = BRGEMM
	Auto is an experimental feature and should be used with application warm-up iteration >=15.
	Note: Different workloads on different frameworks (PyTorch, TensorFlow) have specific `ZENDNN_MATMUL_ALGO` settings for optimized performance.
	• NLP-based models
	  • FP32 models
	    • ZENDNN_MATMUL_ALGO =FP32:2
	  • BF16 (AMP) models
	    • ZENDNN_MATMUL_ALGO =BF16:4
	• LLM-based models
	  • BF16 and WOQ (Per channel and Per group) models:
	    • ZENDNN_MATMUL_ALGO =BF16:0
	For RecSys models
	• FP32, INT8 and BF16 models
	  • ZENDNN_MATMUL_ALGO=FP32:2;INT8:2;BF16:2
	• BF16 (AMP) models
	  • ZENDNN_MATMUL_ALGO=BF16:4
		`ZENDNN_MATMUL_ALGO=FP32:4,BF16:4,INT8:4`

[a] You must set these environment variables explicitly.

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

Table 4.2: zentorch Environment Variables-Generic

Environment Variable	Description	Default Value/User Defined Value
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 4.2: zentorch Environment Variables-Generic (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.	granularity=fine,compact,1,0

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 4.3: zentf Environment Variables-Generic

Environment Variable	Description	Default Value/ User Defined Value
`TF_ZEN_PRIMITIVE_REUSE_DISABLE`		False
`ZENDNN_ENABLE_MEMPOOL`	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	1

Table 4.3: zentf Environment Variables-Generic (continued)

Environment Variable	Description	Default Value/ User Defined Value
ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE		0
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

Table 4.4: zentf Environment Variables-Optimization

Environment Variable	Description	Default Value/User Defined Value
ZENDNN_TENSOR_POOL_LIMIT	For optimal performance, you can modify it to: • 512 for CNNs • 32 for densenet model	1024
ZENDNN_CONV_ALGO	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 filters	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.

4.3 System Used for Performance Tuning

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

Table 4.5: 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	1
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.

- `ZENDNN_LOG_OPTS=ALL:0`
- `OMP_NUM_THREADS=128 # For a system with 128 cores per socket`
- `OMP_WAIT_POLICY=ACTIVE`
- `OMP_DYNAMIC=FALSE`
- `ZENDNN_MATMUL_ALGO=FP32:4,BF16:4`
- `ZENDNN_PRIMITIVE_CACHE_CAPACITY=1024`
- `GOMP_CPU_AFFINITY=0-127`

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

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.

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

4.5 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=0-127`, should give the same thread bindings.

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

4.6 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.

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: `never` (batch size =1), `always` (batch size >1)
  - 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 4.6: 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>`

Table 4.6: LD_PRELOAD environment variables in case of TCMalloc and jemalloc (continued)

Use this command	TCMalloc	jemalloc
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 zentf

The following environment variable settings are optimal settings for 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)
- `ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE=0`
- `ZENDNN_CONV_ALGO=3`
- `TF_NUM_INTEROP_THREADS=1` (for Hugging Face NLP and LLM Models)
- `TF_NUM_INTRAOP_THREADS=128` (for Hugging Face NLP and LLM Models)

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 `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 5.1: 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 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 5.2: 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.

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.

A.2 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.

Appendix B: Notices

B.1 Trademarks