TECHNOLOGY GUIDE

Intel Corporation

Intel® Deep Learning Boost - Boost Network Security AI Inference Performance in Google Cloud Platform (GCP)

Using Intel® oneAPI Deep Neural Network (oneDNN) and Intel® Neural Compressor with the 3rd Generation Intel® Xeon® Scalable Processor

Authors

David Lu

Heqing Zhu

Feng Tian

Tong Zhang

Shuangpeng Zhou

John DiGiglio

AG Ramesh

1. Introduction

Applying AI in network and security is a novel approach to address the ever-increasing cyber threats. AI inference is the new way to prevent advanced cyberattacks. One of the challenges is the latency when applying the AI technology. TensorFlow is a widely used deep-learning (DL) framework. Intel has been collaborating with Google to optimize its performance on Intel® Xeon® processor-based platforms using Intel® oneAPI Deep Neural Network Library (oneDNN), an open-source, cross-platform performance library for DL applications. TensorFlow optimizations are enabled via oneDNN to accelerate key performance-intensive operations such as convolution, matrix multiplication, activation, inner product, batch normalization, and other primitives. TensorFlow can be configured to run optimally on a specific target (for example, CPU with AVX instructions or GPU with Tensor Cores) resulting in performance acceleration greater than 3x. oneDNN optimizations were first introduced in official TensorFlow 2.5.

Users can enable those CPU optimizations by setting the environment variable TF_ENABLE_ONEDNN_OPTS=1 when using the official x86-64 TensorFlow release 2.5 or later. This will help developers seamlessly benefit from the Intel oneAPI Deep Neural Network Library (oneDNN) optimization. Additional TensorFlow-based applications, including TensorFlow Extended, TensorFlow Hub, and TensorFlow Serving also include the oneDNN optimizations. However, oneDNN is disabled by default. The user needs to enable it manually before running the workload to take advantage of oneDNN capabilities.

Intel® Neural Compressor helps developers convert a model's weights from floating point (32-bits) to integers (8-bits). Although some loss of accuracy may result, it significantly decreases model size in memory, while also enhancing CPU and hardware accelerator latency.

This technology guide illustrates how to use Intel's oneDNN and Intel Neural Compressor to boost deep learning inference performance. The guide also shows gen-2-gen performance comparison of Google Cloud Platform (GCP) instances among the 1st Generation Intel® Xeon® Scalable processor, 2nd Generation Intel® Xeon® Scalable processor, and 3rd Generation Intel® Xeon® Scalable processor.

Customers can use this solution and associated collateral as a reference to replicate other workloads as well. This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	April 2022	Initial release.

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
AI	Artificial Intelligence
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
DL	Deep Learning
GCP	Google Cloud Platform
GPU	Graphics Processing Unit
LSTM	Long Short-term Memory
oneDNN	Intel® oneAPI Deep Neural Network Library (oneDNN)
RNN	Recurrent Neural Network
VNNI	Vector Neural Network Instructions

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
Malware Detection by Eating a Whole EXE	https://arxiv.org/pdf/1710.09435.pdf
Intel® Deep Learning Boost (Intel® DL Boost)	https://www.intel.com/content/www/us/en/artificial-intelligence/deep-learning-boost.html
Intel® oneAPI Deep Neural Network Library	https://github.com/oneapi-src/oneDNN
Intel® Neural Compressor	https://github.com/intel/neural-compressor
MalConv model	https://github.com/elastic/ember/tree/master/malconv

2. Technology Overview

2.1 Intel® Deep Learning Boost (Intel® DL Boost) Technologies

- Intel® Advanced Vector Extensions 512 (Intel® AVX-512): A 512-bit instruction set that can accelerate performance for demanding workloads and usages like AI inferencing.
- Intel® Deep Learning Boost (Intel® DL Boost): A group of acceleration features introduced in the 2nd Generation Intel Xeon Scalable processors that aim to provide significant performance[1] increases to inference applications built with leading DL frameworks such as PyTorch, TensorFlow, MXNet, and ONNX (Open Neural Network Exchange). The foundation of Intel Deep Learning Boost is Vector Neural Network Instructions (VNNI), a specialized instruction set that uses a single instruction for DL computations that formerly required three separate instructions.

[1] For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

Intel® Deep Learning Boost Technologies		
Microarchitecture	AVX-512_VNNI	AVX-512_BF16
Client		
3rd Generation Intel® Xeon® Scalable processor (Ice Lake Client)	✓	✗
Server		
2nd Generation Intel® Xeon® Scalable processor (Cascade Lake)	✓	✗
3rd Generation Intel® Xeon® Scalable processor (Cooper Lake)	✓	✓
3rd Generation Intel® Xeon® Scalable processor (Ice Lake Server)	✓	✗

Figure 1. Intel® Deep Learning Boost Technologies

2.2 TensorFlow

The binary distribution of TensorFlow with Intel oneAPI Deep Neural Network Library primitives, a popular performance library for deep-learning applications. TensorFlow is a widely used machine-learning framework in the deep-learning arena, demanding efficient use of computational resources. To take full advantage of Intel® architecture and extract maximum performance, the TensorFlow framework has been optimized using oneDNN primitives[2].

2.3 Intel® oneAPI Deep Neural Network Library

The Intel oneAPI Deep Neural Network Library (oneDNN) helps developers improve productivity and enhance the performance of their deep-learning frameworks. Use the same API to develop for CPUs, GPUs, or both. Then implement the rest of the application using Data Parallel C++. This library is included in both the Intel® oneAPI Base Toolkit and Intel® oneAPI DL Framework Developer Toolkit (DLFD Kit). It supports key data type formats, including 16- and 32-bit floating points, bfloat16, and 8-bit integers. oneDNN implements rich operators, including convolution, matrix multiplication, pooling, batch normalization, activation functions, recurrent neural network (RNN) cells, and long short-term memory (LSTM) cells. It accelerates inference performance with automatic detection of Intel Deep Learning Boost technology.

2.4 Intel® Neural Compressor

Intel® Neural Compressor (formerly known as Intel® Low Precision Optimization Tool) is an open-source Python library running on Intel® CPUs and GPUs, which delivers unified interfaces across multiple deep-learning frameworks for popular network compression technologies, such as quantization, pruning, and knowledge distillation. This tool supports automatic accuracy-driven tuning strategies to help users quickly find out the best quantized model. It also implements different weight pruning algorithms to generate pruned models with predefined sparsity goals and supports knowledge distillation to distill the knowledge from the teacher model to the student model.

It's a Python library designed to help you quickly deploy low-precision inference solutions on popular deep-learning frameworks such as TensorFlow, PyTorch, MXNet, and ONNX runtime. The tool automatically optimizes low-precision recipes for deep-learning models to achieve optimal product objectives, such as inference performance and memory usage, with expected accuracy criteria.

3. Using Intel® Deep Learning Boost to Enhance Performance

3.1 Prepare the benchmark environment

In this technical solution, we show how to improve the performance of an open-source deep-learning model named MalConv (https://github.com/elastic/ember/tree/master/malconv), which performs malware detection on raw bytes of entire executable files. A public GitHub repository ember already provides the pre-trained Keras H5 format file. You can download the model file with the following command.

```
# wget https://raw.githubusercontent.com/elastic/ember/master/malconv/malconv.h5
```

[2] For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

Before running the malware detection model, it needs a stand-alone Python virtual environment:

```
# python3.7 -m venv tf-dnn
# source tf-dnn/bin/activate
# pip install --upgrade pip
```

All package dependencies are listed in requirements.txt as show below:

```
# cat requirements.txt
tensorflow==2.7.0
tensorflow-estimator==2.7.0
keras==2.7.0
neural-compressor
progress
```

Then, install all the dependencies with following command:

```
# pip install -r requirements.txt
```

Create inference test file. You can use below sample Python file to evaluate the performance of a TensorFlow model on specified datasets.

```
# cat test.py
import os
import time
import argparse
import numpy as np
import tensorflow as tf
from progress.bar import Bar

def read_file(filepath, expect_size=1048576):
    if filepath[-4:] == '.npy':
        data = np.load(filepath, allow_pickle=True)
    else:
        data = np.fromfile(filepath, np.ubyte)
    if data.size < expect_size:
        data = np.pad(data, (0, expect_size - data.size), 'constant', constant_values=(0, 0))
    else:
        data = data[:expect_size]
    return np.array([data])

class H5Model:
    def __init__(self, h5_path):
        self.model = tf.keras.models.load_model(h5_path)

    def predict(self, input_data):
        start = time.time()
        result = self.model.predict(input_data)
        finish = time.time()
        return result[0], 1000 * (finish - start)

class FrozenModel:
    def __init__(self, pb_filepath, config=None):
        graph = tf.Graph()
        with graph.as_default():
            graph_def = tf.compat.v1.GraphDef()
            with open(pb_filepath, "rb") as f:
                self.model path = pb filepath
                graph_def.ParseFromString(f.read())
                _ = tf.import_graph_def(graph_def, name='')
                self.session = tf.compat.v1.Session(config=config, graph=graph)
                self.input_t1 = graph.get_tensor_by_name("input_1:0")
                self.output_data = graph.get_tensor_by_name("Identity:0")

    def predict(self, input_data):
        start = time.time()
        result = self.session.run(self.output_data, feed_dict={self.input_t1: input_data})
        finish = time.time()
        return result[0][0], 1000 * (finish - start)

class TestOnDataset:
    def __init__(self, model, input_path):
        self.model = model
        self.avg_infer_time = None
        self.standard_deviation = None
        self.all_files = []
        mal_path = os.path.join(input_path, 'MALICIOUS')
        mal_files = [(1, os.path.join(mal_path, fp)) for fp in os.listdir(mal_path)]
```

```
        self.all_files.extend(mal_files)
        clean_path = os.path.join(input_path, 'KNOWN')
        clean_files = [(0, os.path.join(clean_path, fp)) for fp in os.listdir(clean_path)]
        self.all_files.extend(clean_files)

    def run(self):
        input_tensor = read_file(self.all_files[0][1])
        for _ in range(30):
            self.model.predict(input_tensor)
        all_infer_time = []
        bar = Bar('Progress... ', max=len(self.all_files))
        for label, filepath in self.all_files:
            int8_data = read_file(filepath)
            _, infer_time = self.model.predict(int8_data)
            all_infer_time.append(infer_time)
            bar.next()
        bar.finish()
        self.avg_infer_time = np.mean(all_infer_time)
        self.standard_deviation = np.std(all_infer_time)
        print(f'average inference time: {self.avg_infer_time} ms')
        print(f'standard deviation: {self.standard_deviation} ms')
        print(f'filecount: {len(self.all_files)}')

def load_model(model_path):
    if model_path[-2:] == 'h5':
        return H5Model(model_path)
    if os.path.isdir(model_path):
        return SavedModel(model_path)
    return FrozenModel(model_path)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '-m', '--model_path', type=str, dest='model_path', help='model path', required=True)
    parser.add_argument(
        '-i', '--input_path', type=str, dest='input_path', help='input dataset path', required=True)
    args = parser.parse_args()
    model = load_model(args.model_path)
    TestOnDataset(model, args.input_path).run()

if __name__ == '__main__':
    main()
```

3.2 Evaluate performance using Keras H5 format

The mean inference time is an important performance index for AI inference. First, we need to get mean inference time as the baseline. The open-source project provides a trained MalConv model, however, it is only provided in Keras H5 format. Run the following command to get mean inference time without enabling oneDNN.

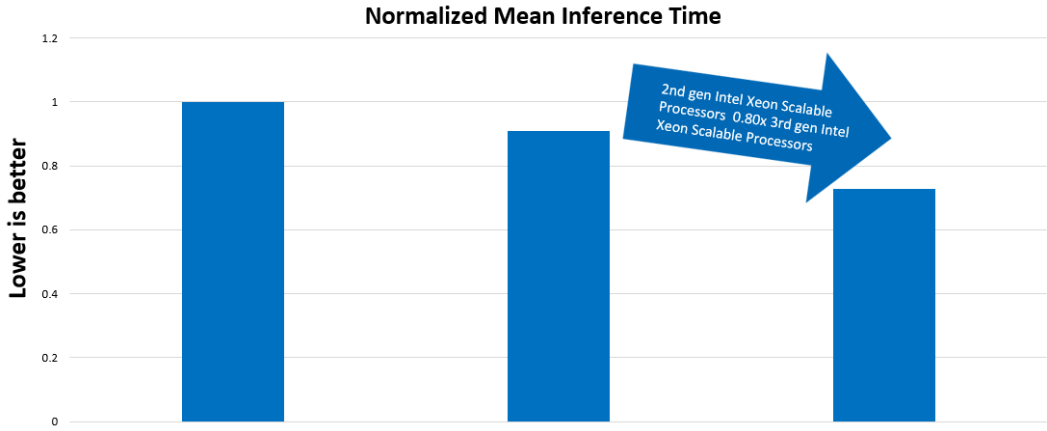
```
# numactl --physcpubind=0 python test.py -m malconv.h5 -i ./datasets
```

We executed the MalConv model with Keras H5 format under the three generations of Intel Xeon Scalable processor GCP instances. The mean inference time on single physical core can be found in the following table.

Table 3. Mean inference time of MalConv H5 model without oneDNN under different Intel gen CPUs

Mean Inference Time (ms) One Core	H5 format without oneDNN
n1-std-8-SKX (1st gen Intel Xeon Scalable @2.00GHz)	121.56
n2-std-8-CLX (2nd gen Intel Xeon Scalable @2.80GHz)	112.22
n2-std-8-ICX (3rd gen Intel Xeon Scalable @2.60GHz)	90.34
Performance Boost n2-std-8-ICX vs n1-std-8-SKX	1.34 X

The following graph compares the normalized mean inference time of MalConv H5 model without oneDNN on the three GCP instances.



	GCP: n1-std-8-SKX (1st gen Intel Xeon Scalable Processors)	GCP: n2-std-8-CLX (2nd gen Intel Xeon Scalable Processors)	GCP: n2-std-8-ICX (3rd gen Intel Xeon Scalable Processors)
TF 2.7 H5 model without oneDNN	1	0.92	0.74

Figure 2. Comparison of normalized mean inference time of MalConv H5 model without oneDNN under different Intel gen CPUs

From the results from running the H5 model shown above, we have the following conclusion[3]:

- Overall, the mean inference time under n2-std-8-ICX (3rd generation Intel Xeon Scalable at 2.60GHz) is much faster than n1-std-8-SKX (1st generation Intel Xeon Scalable @2.00GHz) and n2-std-8-CLX (2nd generation Intel Xeon Scalable @2.80GHz). The results show that the performance of the n2-std-8-ICX instance is 1.34 X faster than the n1-std-8-SKX instance.

3.3 Evaluate performance using FP32 frozen graph model

In this part, we convert H5 model to FP32 frozen graph model to get further performance gains. To get an FP32 frozen graph model, we need to first convert Keras H5 format to SavedModel, which is a standard format starting TensorFlow 2.X. Here is the script to convert Keras H5 format to SavedModel.

```
# cat h5_to_savedmodel.py
import tensorFlow as tf
model = tf.keras.models.load_model("malconv.h5")
model.save("malconv_saved_model")
```

After we get a SavedModel, we can use the following steps to convert SavedModel to FP32 frozen graph model.

Step 1: Create the following sample Python file.

```
# cat saved_model_to_frozen.py
import sys
import tensorFlow as tf
from tensorFlow.python.saved_model import signature_constants
from tensorFlow.python.training import saver
from tensorFlow.python.framework import convert_to_constants
from tensorFlow.core.protobuf import config_pb2
from tensorFlow.python.grappler import tf_optimizer
from tensorFlow.core.protobuf import meta_graph_pb2
from tensorFlow.python.platform import gfile
from tensorFlow.python.eager import context
assert context.executing_eagerly()
if len(sys.argv) != 3:
    print('Usage:')
    print(f'\tpython3 {sys.argv[0]} model_path output_pbfile')
    sys.exit(1)

model = tf.keras.models.load_model(sys.argv[1])
model.summary()

func = model.signatures[signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY]
```

[3] For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

```
frozen_func = convert_to_constants.convert_variables_to_constants_v2(func)

grappler_meta_graph_def = saver.export_meta_graph(
    graph_def=frozen_func.graph.as_graph_def(), graph=frozen_func.graph)

fetch_collection = meta_graph_pb2.CollectionDef()
for array in frozen_func.inputs + frozen_func.outputs:
    fetch_collection.node_list.value.append(array.name)
grappler_meta_graph_def.collection_def["train_op"].CopyFrom(fetch_collection)

grappler_session_config = config_pb2.ConfigProto()
rewrite_options = grappler_session_config.graph_options.rewrite_options
rewrite_options.min_graph_nodes = -1
opt = tf_optimizer.OptimizeGraph(grappler_session_config, grappler_meta_graph_def, graph_id=b"tf_graph")

f = gfile.GFile(sys.argv[2], 'wb')
f.write(opt.SerializeToString())
```

Step 2: Run the following command to convert SavedModel to FP32 frozen graph model.

```
# python saved_model_to_frozen.py malconv_saved_model/ malconv_fp32.pb
```

Now we can run the following command to get mean inference time using FP32 frozen graph model with enabling oneDNN. We need to add TF_ENABLE_ONEDNN_OPTS=1 to enable oneDNN for TensorFlow 2.5 or later, since oneDNN is disabled by default. Run the following command to get mean inference time while oneDNN is enabled.

```
# TF_ENABLE_ONEDNN_OPTS=1 numactl --physcpubind=0 python test.py -m malconv_fp32.pb -i ./datasets
```

We tested MalConv FP32 frozen model under three generations of Intel Xeon Scalable processor GCP instances. The mean inference time on a single physical core can be found in the following table.

Table 4. Mean inference time under MalConv H5 model w/o oneDNN and FP32 model w/oneDNN under different Intel gen CPUs

Mean Inference Time (ms) One Core	H5 format without oneDNN	FP32 frozen model with oneDNN
n1-std-8-SKX (1st gen Intel Xeon Scalable @2.00GHz)	121.56	45.54
n2-std-8-CLX (2nd gen Intel Xeon Scalable @2.80GHz)	112.22	41.40
n2-std-8-ICX (3rd gen Intel Xeon Scalable @2.60GHz)	90.34	33.08
Performance Boost n2-std-8-ICX vs n1-std-8-SKX	**1.34 X**	**1.38 X**

Following graph compares the normalized mean inference time of MalConv FP32 frozen model with oneDNN on those three GCP instances.

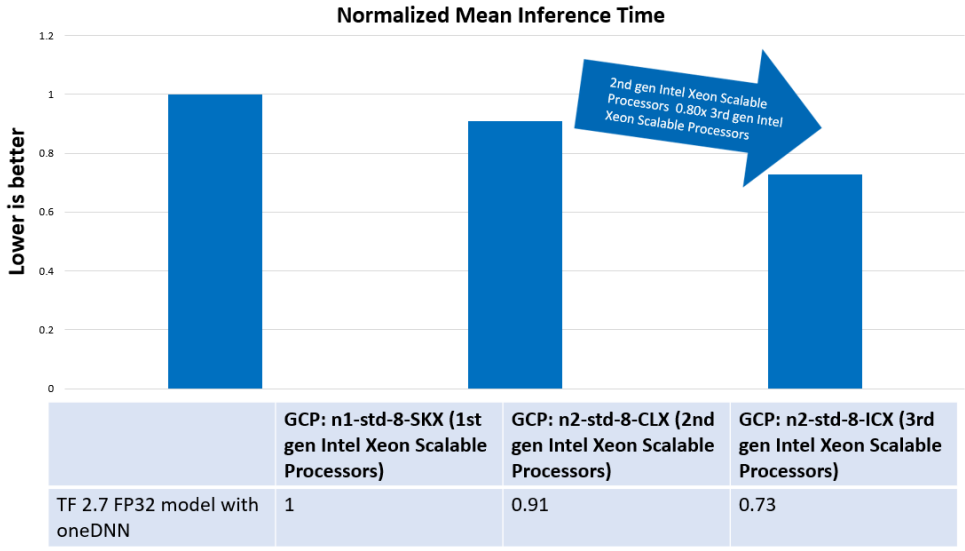


Figure 3. Comparison of normalized mean inference time of MalConv FP32 frozen model with oneDNN under different Intel gen CPUs

From the results of the FP32 frozen graph model shown above, we can conclude[4]:

- By using FP32 frozen model, the performance is improved under all those GCP instances especially after enabling oneDNN. However, n2-std-8-ICX (3rd generation Intel Xeon Scalable @2.60GHz) has the best performance with oneDNN enabled.
- FP32 Frozen model has better performance than H5 model.
- Overall, the mean inference time under n2-std-8 (3rd generation Intel Xeon Scalable processor at 2.60GHz) is much faster than n1-std-8-SKX (1st generation Intel Xeon Scalable @2.00GHz) and n2-std-8-CLX (2nd generation Intel Xeon Scalable @2.80GHz). The performance for the n2-std-8-ICX instance is 1.47x faster than the n1-std-8-SKX instance.

3.4 Evaluate performance using INT8 frozen graph model

We have already found that the mean inference is significantly decreased by converting Keras H5 format to F32 frozen graph model resulting from the use of oneDNN. We are now going to use the Intel® Neural Compressor to perform post-training quantization (PTQ for short) on MalConv FP32 frozen model, seeking for further performance improvements. Use the following command to install Intel Neural Compressor.

```
# pip install neural-compressor
```

For above Python script sample, besides FP32 frozen model of sample security AI workload, it also needs a YAML config file.

```
# cat malconv.yaml
version: 1.0
model:
  name: malconv
  framework: tensorflow
  inputs: input_1
  outputs: Identity

tuning:
  accuracy_criterion:
    relative:  0.01
  exit policy:
    timeout: 0
    max_trials: 1 random_seed: 9527
```

Below is the Python script sample that quantizes FP32 frozen model with Intel Neural Compressor.

```
# cat quantize.py
import os
import argparse
import numpy as np
from neural_compressor.experimental import Quantization, common

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '-m', '--input_model', type=str, dest='input_model', help='frozen fp32 model', required=True)
    parser.add_argument(
        '-c', '--input_config', type=str, dest='input_config', help='yaml config file', required=True)
    parser.add_argument(
        '-i', '--input_path', type=str, dest='input_path', help='input dataset', required=True)
    parser.add_argument(
        '-o', '--output_file', type=str, dest='output_file', help='output file', required=True)
    args = parser.parse_args()
    return args

def load_dataset(input_path):
    result = []
    mal_path = os.path.join(input_path, 'MALICIOUS')
    if os.path.exists(mal_path):
        mal_files = [(1, os.path.join(mal_path, fp)) for fp in os.listdir(mal_path)]
        result.extend(mal_files)
    clean_path = os.path.join(input_path, 'KNOWN')
    if os.path.exists(clean_path):
        clean files = [(0, os.path.join(clean path, fp)) for fp in os.listdir(clean path)]
        result.extend(clean_files)

    return result
def read_file(filepath: str, expect_size: int):
    if filepath[-4:] == '.npy':
```

[4] For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

```
        data = np.load(filepath, allow_pickle=True)
    else:
        data = np.fromfile(filepath, np.ubyte)

    if data.size < expect_size:
        data = np.pad(data, (0, expect_size - data.size), 'constant', constant_values=(0, 0))
    else:
        data = data[:expect_size]

    return np.array([data])

class Dataset:
    def __init__(self, input_path):
        self.batch_size = 32
        self.dataset = load_dataset(input_path)

    def __iter__(self):
        for label, filepath in self.dataset:
            data = read_file(filepath, expect_size=1048576)
            yield data, label

    def __len__(self):
        return len(self.dataset)

if __name__ == '__main__':
    os.environ['TF_ENABLE_ONEDNN_OPTS'] = '1'
    args = parse_args()
    quantizer = Quantization(args.input_config)
    quantizer.model = common.Model(args.input_model)
    quantizer.calib_dataloader = Dataset(args.input_path)
    quantizer().save(args.output_file)
```

By running the quantization Python script on FP32 frozen model with this YAML config file, we can get a quantized INT8 model.

```
# TF_ENABLE_ONEDNN_OPTS=1 python3 quantize.py -m malconv_fp32.pb -c malconv.yaml -i ./datasets -o
malconv_int8.pb
```

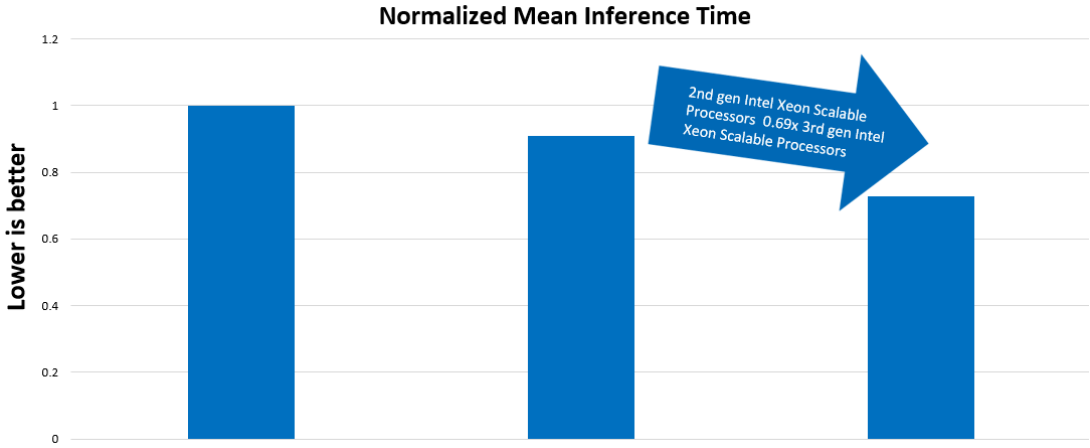
Similar to above, we can run the following command to get mean inference time.

```
# TF_ENABLE_ONEDNN_OPTS=1 numactl --physcpubind=0 python test.py -m malconv_int8.pb -i ./datasets
```

Table 5. Mean inference time of MalConv frozen INT8 model under three generations of Intel Xeon Scalable processor GCP instances.

Mean Inference Time (ms) One Core	H5 format without oneDNN	Frozen INT8 model
n1-std-8-SKX (1st gen Intel Xeon Scalable @2.00GHz)	121.56	46.49
n2-std-8-CLX (2nd gen Intel Xeon Scalable @2.80GHz)	112.22	28.15
n2-std-8 (3rd gen Intel Xeon Scalable @2.60GHz)	90.34	19.47
Performance Boost n2-std-8-ICX vs n1-std-8-SKX	**1.34 X**	**2.39 X**

The following graph compares the normalized mean inference time of MalConv INT8 frozen model with oneDNN on the three GCP instances.



	GCP: n1-std-8-SKX (1st gen Intel Xeon Scalable Processors)	GCP: n2-std-8-CLX (2nd gen Intel Xeon Scalable Processors)	GCP: n2-std-8-ICX (3rd gen Intel Xeon Scalable Processors)
TF 2.7 INT8 model	1	0.61	0.42

Figure 4. Comparison of normalized mean inference time of MalConv INT8 frozen model with oneDNN under different Intel gen CPUs

From the above results, we have the following conclusions[5]:

- By using both oneDNN and Intel Neural Compressor, the n2-std-8 (3rd gen Intel Xeon Scalable @2.60GHz) instance gets performance improvement from an initial 1.34x to 2.39x.
- The accuracy is minimally impacted after using Intel Neural Compressor to convert FP32 model to INT8 model.
- The mean inference time under n2-std-8 (3rd gen Intel Xeon Scalable @2.60GHz) using INT8 model is 19.47ms, which is 6.11x faster than the H5 model.

4. Summary

From the previous tests, we find that the mean inference time can be dramatically improved under TensorFlow by applying oneDNN and Intel Neural Compressor. With oneDNN and Neural Compressor, the 3rd generation Intel Xeon Scalable processor-based instance gets performance improvement from an initial 1.34x to 2.39x. We also showed that the performance tuning with oneDNN and using the Intel Neural Compressor is easy and straightforward. Intel DL Boost, which contributes to the performance improvement we observed, is a standard and universally available feature in 2nd and 3rd generation Intel Xeon Scalable processors without the need to attach auxiliary hardware accelerator.

[5] For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

Appendix A Platform Configuration

Name	n1-std-8 SKX	n2-std-8 CLX	n2-std-8 ICX
Time	Mon Jan 10 06:27:51 UTC 2022	Mon Jan 10 06:27:51 UTC 2022	Mon Jan 10 06:27:51 UTC 2022
Manufacturer	Google	Google	Google
Product Name	Virtual Machine	Virtual Machine	Virtual Machine
BIOS Version	Google V 1.0	Google V 1.0	Google V 1.0
OS	Ubuntu 20.04.3 LTS	Ubuntu 20.04.3 LTS	Ubuntu 20.04.3 LTS
Kernel	5.11.0-1026-gcp	5.11.0-1026-gcp	5.11.0-1026-gcp
Microcode	0xffffffff	0xffffffff	0xffffffff
IRQ Balance	Disabled	Disabled	Disabled
CPU Model	Intel® Xeon® CPU @ 2.00GHz	Intel® Xeon® CPU @ 2.80GHz	Intel® Xeon® CPU @ 2.60GHz
Base Frequency	2.0GHz	2.8GHz	2.6GHz
CPU(s)	8	8	8
Thread(s) per Core	2	2	2
Core(s) per Socket	4	4	4
Socket(s)	1	1	1
NUMA Node(s)	1	1	1
Turbo	Disabled	Disabled	Disabled
Installed	32 GB	32 GB	32 GB
Automatic NUMA Balancing	Disabled	Disabled	Disabled

Appendix B Software Configuration

Software	
Operating System	Ubuntu 20.04.3 LTS
Kernel	5.11.0-1026-gcp
Workload & version	elastic/ember/malconv
AI Framework	TensorFlow 2.7.0
Libraries	N/A
WL Specific Details	
	Mean Inference time under following 1, H5 model under TensorFlow 2, FP32 frozen model under TensorFlow with oneDNN enabled 3, INT8 frozen model under TensorFlow The WL performance will take advantage of oneDNN and Intel Neural Compressor, which convert FP32 model to INT8 model

0422/DN/WIPRO/PDF 723973-001US