rocAL Documentation

Release 2.5.0

Advanced Micro Devices, Inc.

Mar 27, 2026

INSTALLATION

The ROCm Augmentation Library (rocAL) is a Python library that provides a way to customize audio, video, and image pipelines for different datasets and models, improving the throughput and performance of deep learning applications. rocAL is optimized for loading and pre-processing data for deep learning applications, with support for multiple data formats and augmentations.

The rocAL public repository is located at https://github.com/ROCm/rocAL.

Installation

- *rocAL prerequisites*
- *Installing rocAL with the package installer*
- *Building and installing rocAL from source*

Conceptual

- *rocAL Operators*

How to

- *Create and run the rocAL pipeline*
- *Run PyTorch training with rocAL*
- *Run TensorFlow training with rocAL*
- *Run JAX training with rocAL*

Examples

- *rocAL framework integration examples*

Reference

- *rocAL pipelines*
- *rocAL RNNT dataloading*
- *rocAL Python API overview*
- *rocAL Python reference*

The C++ API for contributors:

- *rocAL C++ API overview*
- *rocAL C++ reference*

To contribute to the documentation refer to Contributing to ROCm Docs.

You can find licensing information on the Licensing page.

CHAPTER

ONE

ROCAL PREREQUISITES

rocAL requires ROCm running on GPUs based on the CDNA architecture installed with the AMDGPU installer.

rocAL has been tested on the following Linux environments:

- Ubuntu 22.04 and 24.04
- RHEL 8 and 9
- SLES 15 SP7

See Supported operating systems for the complete list of ROCm supported Linux environments.

Building rocAL from source requires CMake Version 3.10 or later, AMD Clang++ Version 18.0.0 or later, and the following compiler support:

- C++17
- OpenMP
- Threads

Most prerequisites are installed with the *package installer*.

When building rocAL from source, the `rocAL-setup.py` setup script can be used to install prerequisites:

```
rocAL-setup.py [-h] [--directory DIRECTORY; default ~/] \
                   [--rocm_path ROCM_PATH; default /opt/rocm] \
                   [--backend HIP|OCL; default HIP] \
                   [--ffmpeg ON|OFF; default OFF] \
                   [--reinstall ON|OFF; default OFF]
```

The following prerequisites are required and are installed with both the package installer and the setup script:

- MIVisionX with AMD OpenVX and the VX_RPP and AMD Media extensions
- HIP
- The half-precision floating-point library version 1.12.0 or later
- Google Protobuf version 3.12.4 or later
- LMBD Library
- TurboJPEG
- PyBind11 version 2.11.1
- RapidJSON
- OpenCV
- Python3, Python3 pip, and Python3 wheel

libstdc++-12-dev is required on Ubuntu 22.04 only and must be installed manually.

FFMPEG is not required, but is installed by the package installer. It can also be installed with the setup script using the `--ffmpeg` option.

rocDecode and rocJPEG are installed by the package installer and the setup script, but aren't required by rocAL. When installed, rocJPEG is used as the hardware image decoder and rocDecode is used as the hardware video decoder.

Note

TurboJPEG must be installed manually on SLES.
To use FFMPeg on SLES and RHEL, the `FFMPeg-dev` package must be installed manually.

CHAPTER

TWO

INSTALLING ROCAL WITH THE PACKAGE INSTALLER

Three rocAL packages are available:

`rocal`: The rocAL runtime package. This is the basic rocAL package that only provides dynamic libraries. It must always be installed.
`rocal-dev`: The rocAL development package. This package installs a full suite of libraries, header files, and samples. This package needs to be installed to use samples.
`rocal-test`: A test package that provides a CTest to verify the installation.

All the required prerequisites are installed when the package installation method is used.

Note

TurboJPEG must be installed manually on SLES.
To use FFMPeg on SLES and RedHat, the `FFMPeg-dev` package must be installed manually.

2.1 Basic installation

Use the following commands to install only the rocAL runtime package:

Ubuntu

```
sudo apt install rocal
```

RHEL

```
sudo yum install rocal
```

SLES

```
sudo zypper install rocal
```

2.2 Complete installation

Use the following commands to install `rocal`, `rocal-dev`, and `rocal-test`:

Ubuntu

```
sudo apt-get install rocal rocal-dev rocal-test
```

RHEL

```
sudo yum install rocal rocal-devel rocal-test
```

SLES

```
sudo zypper install rocal rocal-devel rocal-test
```

The rocAL test package will install a CTest module. Use the following steps to test the installation:

```
mkdir rocAL-test
cd rocAL-test
cmake /opt/rocm/share/rocal/test/
ctest -VV
```

CHAPTER

THREE

BUILDING AND INSTALLING ROCAL FROM SOURCE CODE

Before building and installing rocAL, ensure ROCm has been installed with the AMDGPU installer and the `rocm` usecase.

The rocAL source code is available from https://github.com/ROCm/rocAL. Use the rocAL version that corresponds to the installed version of ROCm.

rocAL supports the HIP backend.

rocAL is installed in the ROCm installation directory by default. If rocAL for both HIP and OpenCL backends will be installed on the system, each version must be installed in its own custom directory and not in the default directory.

You can choose to use the `rocAL-setup.py` setup script to install most *prerequisites*

Note

TurboJPEG must be installed manually on SLES.
To use FFmpeg on SLES and RedHat, the `FFmpeg-dev` package must be installed manually.

To build and install rocAL, create the `build` directory under the `rocAL` root directory. Change directory to `build`:

```
mkdir build
cd build
```

Use `cmake` to generate a makefile:

```
cmake ../
```

Use the `-DCMAKE_INSTALL_PREFIX` directive to set the installation directory. For example:

```
cmake -DCMAKE_INSTALL_PREFIX=/opt/rocAL/
```

Run make:

```
make
```

Run `cmake` again to generate Python bindings for `rocal_pybind` then install:

```
sudo cmake --build . --target PyPackageInstall
sudo make install
```

After the installation, the rocAL files will be installed under `/opt/rocm/` unless `-DCMAKE_INSTALL_PREFIX` was specified. If `-DCMAKE_INSTALL_PREFIX` was specified, the rocAL files will be installed under the specified directory.

To make and run the tests, use `make test`.

CHAPTER

FOUR

ROCAL OPERATORS

rocAL operators offer the flexibility to run on CPU or GPU for building hybrid pipelines. They also support classification and object detection on the workload. Some of the useful operators supported by rocAL are listed below:

- **Augmentations:** These are used to enhance the data set by adding effects to the original images. To use the augmentations, import the instance of `amd.rocal.fn` into the Python script. These augmentation APIs further call the RPP kernels underneath (HIP/HOST) depending on the backend used to build RPP and rocAL.
- **Readers:** These are used to read and understand the different types of datasets and their metadata. Some examples of readers are list of files with folders, LMDB, TFRecord, and JSON file for metadata. To use the readers, import the instance of `amd.rocal.readers` into the Python script.
- **Decoders:** These are used to support different input formats of images and videos. Decoders extract data from the datasets that are in compressed formats such as JPEG, MP4, etc. To use the decoders, import the instance of `amd.rocal.decoders` into the Python script.

4.1 Table 1. Augmentations Available through rocAL

Color Augmentations	Effects Augmentations	Geometry Augmentations
Blend	Fog	Crop
Blur	Jitter	Crop Mirror Normalization
Brightness	Pixelization	Crop Resize
Color Temperature	Raindrops	Fisheye Lens
Color Twist	Snowflakes	Flip (Horizontal, Vertical, and Both)
Contrast	Salt and Pepper Noise	Lens Correction
Exposure		Random Crop
Gamma		Resize
Hue		Resize Crop Mirror
Saturation		Rotation
Vignette		Warp Affine

4.2 Table 2. Readers Available through rocAL

Readers	Description
File Reader	Reads images from a list of files in a folder(s)
Video Reader	Reads videos from a list of files in a folder(s)
Caffe LMDB Reader	Reads (key, value) pairs from Caffe LMDB
Caffe2 LMDB Reader	Reads (key, value) pairs from Caffe2 LMDB
COCO Reader - file source and keypoints	Reads images and JSON annotations from COCO dataset
TFRecord Reader	Reads from a TFRecord dataset
MXNet Reader	Reads from a RecordIO dataset
Web Dataset Reader	Reads from a web dataset
CIFAR-10 Dataset Reader	Reads from a binary CIFAR-10 dataset

4.3 Table 3. Decoders Available through rocAL

Decoders	Description
Image	Decodes JPEG images
Image_raw	Decodes images in raw format
Image_random_crop	Decodes and randomly crops JPEG images
Image_slice	Decodes and slices JPEG images

To see examples demonstrating the usage of decoders and readers, see rocAL Python Examples.

CHAPTER

FIVE

CREATING AND RUNNING A ROCAL PIPELINE

rocAL pipelines are used to load, decode, and augment audio, video, and image files that will be used in training and inference.

The pipeline can either be instantiated using the `Pipeline()` constructor or by using the `@pipeline_def` decorator. This document demonstrates how to use the decorator. For information about using the constructor, see *the rocAL pipeline reference*.

To create and use a pipeline in your rocAL application with the `@pipeline_def` decorator, you'll need to import `@pipeline_def` from `amd.rocal.pipeline`:

```
from amd.rocal.pipeline import pipeline_def
```

There are two ways to run a pipeline. The first is using the `pipeline.run()` function. This function will run the pipeline exactly once on one batch of files. The second way is to use an iterator that runs all batches of files through the pipeline.

Audio, video, and image iterators are available, with some iterators designed to work with specific framework integrations. For example, if you're using *PyTorch for training*, you would import the `ROCALClassificationIterator` in `amd.rocal.plugin.pytorch`:

```
from amd.rocal.plugin.pytorch import ROCALClassificationIterator
```

Generic iterators are imported from `amd.rocal.plugin.generic`:

```
from amd.rocal.plugin.generic import ROCALClassificationIterator
```

A pipeline is created by decorating a graph definition function with `@pipeline_def`.

Graph definition functions are user-defined functions that import audio, video, or image files, decode them, and augment them. The `@pipeline_def` decorator turns a graph definition function into a pipeline factory. The output of the graph definition function becomes the output of the pipeline.

For example, in `decoder.py` the graph definition function, `image_decoder_pipeline`, reads in an image file, decodes it, and resizes it. It then returns the resized image:

```
@pipeline_def(seed=seed)
def image_decoder_pipeline(device="cpu", path=image_dir):
  jpegs, labels = fn.readers.file(file_root=path)
  images = fn.decoders.image(jpegs, file_root=path, device=device, output_type=types.RGB,
↪ shard_id=0, num_shards=1, random_shuffle=False)
  return fn.resize(images, device=device, resize_width=300, resize_height=300)
```

The pipeline object requires additional parameters such as batch size, number of threads, and device ID. These are passed to the decorated function.

For example, in `decoder.py`:

```
pipe = image_decoder_pipeline(batch_size=bs, num_threads=1, device_id=gpu_id, rocal_
↪cpu=rocal_cpu, tensor_layout=types.NHWC, reverse_channels=True, mean = [0, 0, 0],␣
↪std=[255,255,255], device=rocal_device, path=img_folder)
```

See the pipeline API reference for the complete list of parameters.

Once the pipeline is created, `pipeline.build()` is called to build the pipeline. The pipeline can only be run after it's been built.

Use an iterator to run the pipeline over every batch of files. In `decoder.py`, the pipeline is run using `ROCALClassificationIterator`.

```
def show_pipeline_output(pipe, device):
  pipe.build()
  data_loader = ROCALClassificationIterator(pipe, device=device)
  images = next(iter(data_loader))
  show_images(images[0][0])

[...]

def main():
  [...]
  pipe = image_decoder_pipeline(batch_size=bs, num_threads=1, device_id=gpu_id, rocal_
↪cpu=rocal_cpu, tensor_layout=types.NHWC,reverse_channels=True, mean = [0, 0, 0],␣
↪std=[255,255,255], device=rocal_device, path=img_folder)
  show_pipeline_output(pipe, device=rocal_device)
```

The iterator will run the pipeline until all batches of files have been processed.

CHAPTER

SIX

USING ROCAL WITH PYTORCH FOR TRAINING

The PyTorch plugin for rocAL includes three iterators that can be used to process different types of pipelines.

- `ROCALAudioIterator` is used for audio processing pipelines.
- `ROCALGenericIterator` is used for general data processing pipelines.
- `ROCALClassificationIterator` is used for classification and training pipelines.

The iterators run the training and validation *pipelines*, prefetching and loading the next batch of files while the previous batch is being processed.

Pipelines are created by either instantiating them with `Pipeline()` or decorating a graph function with `@pipeline_def`.

The training and validation pipelines in `imagenet_training.py` are both instantiated with `Pipeline()`:

```
from amd.rocal.pipeline import Pipeline

[...]

def train_pipeline(data_path, batch_size, local_rank, world_size, num_thread, crop,␣
↪rocal_cpu, fp16):
  pipe = Pipeline(batch_size=batch_size, num_threads=num_thread, device_id=local_rank,␣
↪seed=local_rank+10, rocal_cpu=rocal_cpu, tensor_dtype=types.FLOAT16 if fp16 else types.
↪FLOAT, tensor_layout=types.NCHW,
                  prefetch_queue_depth=6, mean=[0.485 * 255, 0.456 * 255, 0.406 * 255],␣
↪std=[0.229 * 255, 0.224 * 255, 0.225 * 255], output_memory_type=types.HOST_MEMORY if␣
↪rocal_cpu else types.DEVICE_MEMORY)
  with pipe:
      jpegs, labels = fn.readers.file(file_root=data_path)
      rocal_device = 'cpu' if rocal_cpu else 'gpu'
      decode = fn.decoders.image_slice(jpegs, output_type=types.RGB,
                                       file_root=data_path, shard_id=local_rank, num_
↪shards=world_size, random_shuffle=True)
      res = fn.resize(decode, resize_width=224, resize_height=224, output_layout=types.
↪NHWC,
                      output_dtype=types.UINT8, interpolation_type=types.TRIANGULAR_
↪INTERPOLATION)
      flip_coin = fn.random.coin_flip(probability=0.5)
      cmnp = fn.crop_mirror_normalize(res,
                                      output_layout=types.NCHW,
                                      output_dtype=types.FLOAT,
                                      crop=(crop, crop),
```

(continues on next page)

(continued from previous page)

```
                                    mirror=flip_coin,
                                    mean=[0.485 * 255, 0.456 *
                                          255, 0.406 * 255],
                                    std=[0.229 * 255, 0.224 * 255, 0.225 * 255])
        pipe.set_outputs(cmnp)
    print('rocal "{0}" variant'.format(rocal_device))
    return pipe
```

Data is read from the dataset using `readers.file`, which is appropriate for PyTorch:

```
import amd.rocal.fn as fn

[...]

with pipe:
        jpegs, labels = fn.readers.file(file_root=data_path, shard_id=local_rank, num_
↪shards=world_size, random_shuffle=True)
```

The appropriate iterator is then used to load data and run the pipeline. In `imagenet_training.py`, the `ROCALClassificationIterator` is used.

```
from amd.rocal.plugin.pytorch import ROCALClassificationIterator

[...]

def get_rocal_train_loader(data_path, batch_size, local_rank, world_size, num_thread,␣
↪crop, rocal_cpu, fp16=False):
    traindir = os.path.join(data_path, 'train')
    pipe_train = train_pipeline(
        traindir, batch_size, local_rank, world_size, num_thread, crop, rocal_cpu, fp16)
    pipe_train.build()
    train_loader = ROCALClassificationIterator(
        pipe_train, device="cpu" if rocal_cpu else "cuda", device_id=local_rank)
    return Prefetcher(train_loader, rocal_cpu, batch_size)
```

Two examples of PyTorch training using rocAL are available in the rocAL GitHub repository. Jupyter Notebooks are also available.

A Docker container is available for PyTorch training with rocAL.

CHAPTER

SEVEN

USING ROCAL WITH TENSORFLOW FOR TRAINING

The TensorFlow plugin for rocAL includes `ROCALIterator`. `ROCALIterator` runs the training and validation *pipelines*. It prefetches and loads the next batch of files while the previous batch is being processed.

Pipelines are created by either instantiating them with `Pipeline()` or decorating a graph function with `@pipeline_def`:

```
from amd.rocal.pipeline import Pipeline
import amd.rocal.fn as fn

[...]

trainPipe = Pipeline(batch_size=train_batch_size, num_threads=8, rocal_cpu=rocal_cpu,␣
↪device_id=device_id, prefetch_queue_depth=6,
                        tensor_layout=types.NHWC, mean=[0, 0, 0], std=[255, 255, 255],␣
↪tensor_dtype=types.FLOAT)
with trainPipe:
  inputs = fn.readers.tfrecord(path=train_records_dir, reader_type=TFRecordReaderType,␣
↪user_feature_key_map=featureKeyMap,
                               features={
                               'image/encoded': tf.io.FixedLenFeature((), tf.string, "
↪"),
                               'image/class/label': tf.io.FixedLenFeature([1], tf.int64,
↪ -1),
                               'image/filename': tf.io.FixedLenFeature((), tf.string, "
↪")
                               }
                               )
  jpegs = inputs["image/encoded"]
  labels = inputs["image/class/label"]
  images = fn.decoders.image(jpegs, user_feature_key_map=featureKeyMap, output_
↪type=types.RGB, path=train_records_dir)
  resized = fn.resize(images, resize_width=image_size[0], resize_height=image_size[1])
  flip_coin = fn.random.coin_flip(probability=0.5)
  cmn_images = fn.crop_mirror_normalize(resized, crop=(image_size[1], image_size[0]),
                                        mean=[127.5, 127.5, 127.5],
                                        std=[127.5, 127.5, 127.5],
                                        mirror=flip_coin,
                                        output_dtype=types.FLOAT,
                                        output_layout=types.NHWC)
  trainPipe.set_outputs(cmn_images)
```

Data is read from the dataset using `readers.tfrecord`, which reads from TFRecord datasets. `ROCALIterator` is then used to load data and run the pipeline. For example, in `train.py`:

```
from amd.rocal.plugin.tf import ROCALIterator

[...]

trainIterator = ROCALIterator(trainPipe, device=device)
valIterator = ROCALIterator(valPipe, device=device)
```

An example of TensorFlow training using rocAL is available in the rocAL GitHub repository. Jupyter Notebooks are also available.

A Docker container is available for PyTorch training with rocAL.

CHAPTER

EIGHT

USING ROCAL WITH JAX FOR TRAINING

The Jax plugin for rocAL can process the entire dataset at once or it can divide the dataset into shards that are distributed over a mesh of GPUs.

```
from jax.sharding import PositionalSharding
from jax.experimental import mesh_utils

mesh = mesh_utils.create_device_mesh((jax.device_count(), 1))
sharding = PositionalSharding(mesh)
```

Note

The `jax_classification_reader.py` sample and the Jax Jupyter notebook both use the `PositionalSharding()` helper function to automatically divide the dataset into shards. In later versions of Jax, this function has been deprecated. The `NamedSharding()` function is used instead.

When the dataset is divided over multiple GPUs, one *pipeline* is assigned to each GPU. The pipelines process the data shard assigned to the GPU.

The pipelines are then processed by the iterators. Two rocAL Jax iterators are available:

- `ROCALJaxIterator` is used for general data processing pipelines.
- `ROCALPeekableIterator` is a peekable version of `ROCALJaxIterator` that lets you peek at the next item without consuming it.

Both Jax iterators can take a single pipeline if the dataset is being processed all at once, or an array of pipelines and the sharding value if the dataset has been divided over a mesh of GPUs.

Pipelines are created by either instantiating them with `Pipeline()` or decorating a graph function with `@pipeline_def`.

In the Jax Jupyter notebook, training is done over a mesh of GPUs.

Each training pipeline is instantiated, populated with graph elements, and built, before being added to the array of pipelines:

```
from amd.rocal.pipeline import Pipeline
from amd.rocal.plugin.jax import ROCALJaxIterator

[...]

 train_pipelines = []
  for id in range(device_count):
```

(continues on next page)

(continued from previous page)

```
  train_pipeline = Pipeline(batch_size=batch_size, num_threads=8, device_id=id,␣
↪seed=id+42, rocal_cpu=False, tensor_dtype = types.FLOAT, tensor_layout=types.NCHW,␣
↪prefetch_queue_depth = 3, mean=[0.5 * 255,0.5 * 255,0.5 * 255], std = [0.5 * 255,0.5 *␣
↪255,0.5 * 255], output_memory_type = types.DEVICE_MEMORY)

  with train_pipeline:
      cifar10_reader_output = fn.readers.cifar10(file_root=f'{data_path}/cifar-10-
↪batches-bin', shard_id=id, num_shards=device_count, filename_prefix='data_batch_',␣
↪random_shuffle=True, last_batch_policy=types.LAST_BATCH_DROP)
      cmnp = fn.crop_mirror_normalize(cifar10_reader_output,
                                      output_layout = types.NCHW,
                                      output_dtype = types.FLOAT,
                                      crop=(32, 32),
                                      mirror=0,
                                      mean=[0.5 * 255,0.5 * 255,0.5 * 255],
                                      std=[0.5 * 255,0.5 * 255,0.5 * 255])
      train_pipeline.set_outputs(cmnp)

  train_pipeline.build()
  train_pipelines.append(train_pipeline)

training_iterator = ROCALJaxIterator(train_pipelines, sharding)
```

The pipelines are then passed to the iterator:

```
imageIteratorPipeline = ROCALJaxIterator(pipelines, sharding=sharding)
```

Jax isn't tied to a specific data loader and can use any dataset reader.

The validation pipeline in the Jax Jupyter notebook processes the entire dataset without sharding:

```
val_pipeline = Pipeline(batch_size=batch_size, num_threads=8, device_id=0, seed=42,␣
↪rocal_cpu=False, tensor_dtype = types.FLOAT, tensor_layout=types.NCHW, prefetch_queue_
↪depth = 3, mean=[0.5 * 255,0.5 * 255,0.5 * 255], std = [0.5 * 255,0.5 * 255,0.5 * 255],
↪  output_memory_type = types.DEVICE_MEMORY)

with val_pipeline:
  val_cifar10_reader_output = fn.readers.cifar10(file_root=f'{data_path}/cifar-10-
↪batches-bin', shard_id=0, num_shards=1, filename_prefix='test_batch', last_batch_
↪policy=types.LAST_BATCH_DROP)
  val_cmnp = fn.crop_mirror_normalize(val_cifar10_reader_output,
                                      output_layout = types.NCHW,
                                      output_dtype = types.FLOAT,
                                      crop=(32, 32),
                                      mirror=0,
                                      mean=[0.5 * 255,0.5 * 255,0.5 * 255],
                                      std=[0.5 * 255,0.5 * 255,0.5 * 255])
  val_pipeline.set_outputs(val_cmnp)

val_pipeline.build()
validation_iterator = ROCALJaxIterator(val_pipeline)
```

Prebuilt Docker images with Jax pre-installed are available.

CHAPTER

NINE

ROCAL EXAMPLES

Use the links below to access rocAL examples:

- Image Processing
- Pytorch
- Tensorflow
- Jupyter Notebooks

CHAPTER

TEN

THE ROCAL PIPELINE

rocAL pipelines are used to load, decode, and augment audio, video, and image files that will be used in training and inference.

Audio, video, and image data is passed through the pipeline in batches.

Pipelines can be defined by decorating a graph definition function with `@pipeline_def` or by using the `Pipeline()` constructor.

The `@pipeline_def` decorator converts a graph definition function into a pipeline factory. The graph definition function needs to load a file, decode it, and *augment it*. The return value is the result of the augmentation.

For example, the `my_pipe()` function defines a graph that flips an image:

```
from amd.rocal.pipeline import pipeline_def

[...]

@pipeline_def
def my_pipe(flip_vertical, flip_horizontal):
    data, _ = fn.readers.file(file_root=images_dir)
    img = fn.decoders.image(data, device="mixed")
    flipped = fn.flip(img, horizontal=flip_horizontal, vertical=flip_vertical)
    return flipped, img
```

Parameters such as the batch size, device ID, and tensor layout need to be set before the pipeline can be built. For example:

```
pipe = my_pipe(True, False, batch_size=32, num_threads=1, device_id=0)
```

For more information on creating a pipeline with `@pipeline_def`, see *Creating and running a pipeline*.

When the pipeline is created with `Pipeline()`, the pipeline must be populated with readers and augmentations. For example, in `train.py` the training pipeline is populated with a graph that decodes, resizes, flips, and crop-mirror-normalizes images:

```
from amd.rocal.pipeline import Pipeline

[...]

def trainPipeline(data_path, batch_size, num_classes, one_hot, local_rank, world_size,␣
↪num_thread, crop, rocal_cpu, fp16):
    pipe = Pipeline(batch_size=batch_size, num_threads=num_thread, device_id=local_rank,␣
↪seed=local_rank+10,
```

(continues on next page)

(continued from previous page)

```
            rocal_cpu=rocal_cpu, tensor_dtype = types.FLOAT16 if fp16 else types.FLOAT,␣
↪tensor_layout=types.NCHW,
            prefetch_queue_depth = 7)
    with pipe:
        jpegs, labels = fn.readers.file(file_root=data_path, shard_id=local_rank, num_
↪shards=world_size, random_shuffle=True)
        rocal_device = 'cpu' if rocal_cpu else 'gpu'
        decode = fn.decoders.image_slice(jpegs, output_type=types.RGB, file_root=data_
↪path, shard_id=local_rank,
                num_shards=world_size, random_shuffle=True)
        res = fn.resize(decode, resize_x=224, resize_y=224)
        flip_coin = fn.random.coin_flip(probability=0.5)
        cmnp = fn.crop_mirror_normalize(res, device="gpu",
                                        output_dtype=types.FLOAT,
                                        output_layout=types.NCHW,
                                        crop=(crop, crop),
                                        mirror=flip_coin,
                                        image_type=types.RGB,
                                        mean=[0.485 * 255,0.456 * 255,0.406 * 255],
                                        std=[0.229 * 255,0.224 * 255,0.225 * 255])
        if(one_hot):
            _ = fn.one_hot(labels, num_classes)
        pipe.set_outputs(cmnp)
    return pipe
```

In both cases, the pipeline must be built with `pipe.build()` before being run.

There are two ways to run a pipeline. The first is using the `pipeline.run()` function. This function will run the pipeline exactly once on a single batch of files. The second way is to use an iterator that prefetches and loads the next batch of files while the initial batch is being processed.

```
pipe.build()
data_loader = ROCALClassificationIterator(pipe, device=device)
images = next(iter(data_loader))
```

The output of the pipeline is the output of the graph definition function.

CHAPTER

ELEVEN

ROCAL RNNT DATALOADING IN PYTHON

rocAL supports the RNNT speech recognition model through audio readers and other functions that can be used with PyTorch.

All the functions used for RNNT dataloading are available in the `amd.rocal.fn` module. See *Using rocAL with the Python API* for more details about this module.

All the augmentations used in the RNNT dataloader pipeline are available as part of rocAL. These augmentations need to be plugged into the rocAL PyTorch dataloader to run the training. PyTorch samples can be found in the rocAL GitHub repository. .. Note:

```
The rocAL GitHub repository does not host the entire RNNT dataloader source.
```

Table 1: Supported augmentations

Function	Description	Details
`fn.resample`	Resamples an audio signal.	Resampling is achieved by applying a sinc filter with a Hann window. The extent is controlled by the function's `quality` argument.
`fn.nonsilen`	Detects leading and trailing silences.	Returns the beginning and length of the non-silent region. Compares the short-term power calculated for the window length of the signal with a silence cut-off threshold. The signal is considered to be silent when the short term power in decibels is less than the cut-off threshold in decibels.
`fn.slice`	Slices the input.	The slice is specified by an anchor and a shape for the slice.
`fn.preempha`	Applies a preemphasis filter to the input.	The filter used is `Output[t] = Input[t] - coeff * Input[t-1]` if `t > 1` and `Output[t] = Input[t] - coeff * Input_border` if `t == 0`
`fn.spectro`	Produces a spectrogram from a 1D audio signal.	
`fn.mel_fil`	Converts a spectrogram to a mel spectrogram.	Conversion is done by applying a bank of triangular filters where the frequency dimension is selected from the input layout.
`fn.to_decib`	Converts a magnitude to decibels.	The conversion is done using `out[i] = multiplier * log10( max(min_ratio, input[i]/reference) )` where `min_ratio = pow(10, cutoff_db/multiplier)`.
`fn.normali`	Normalizes an input.	Normalization is done by removing the mean and dividing by the standard deviation.

CHAPTER

TWELVE

THE ROCAL C++ API FOR CONTRIBUTORS

The rocAL C++ API is meant for developers who want to contribute to the rocAL project, or who want to create their own custom augmentations.

12.1 C++ Common APIs

The following sections list the commonly used C++ APIs.

12.1.1 rocalCreate

Use: To create the pipeline

Returns: The context for the pipeline

Arguments:

- RocalProcessMode: Defines whether rocal data loading should be on the CPU or GPU.

```
RocalProcessMode:: ROCAL_PROCESS_GPU
RocalProcessMode::ROCAL_PROCESS_CPU
```

- RocalTensorOutputType: Defines whether the output of rocal tensor is FP32 or FP16.

```
RocalTensorOutputType::ROCAL_FP32
RocalTensorOutputType::ROCAL_FP16
```

See rocalCreate example.

```
extern "C"  RocalContext  ROCAL_API_CALL rocalCreate(size_t batch_size, RocalProcessMode␣
↪affinity, int gpu_id = 0, size_t cpu_thread_count = 1, size_t prefetch_queue_depth = 3,
↪ RocalTensorOutputType output_tensor_data_type = RocalTensorOutputType::ROCAL_FP32);
```

12.1.2 rocalVerify

Use: To verify the graph for all the inputs and outputs

Returns: A status code indicating the success or failure

See rocalVerify example.

```
extern "C"  RocalStatus ROCAL_API_CALL rocalVerify(RocalContext context);
```

12.1.3 rocalRun

Use: To process and run the built and verified graph

Returns: A status code indicating the success or failure

See rocalRun example.

```
extern "C"  RocalStatus  ROCAL_API_CALL rocalRun(RocalContext context);
```

12.1.4 rocalRelease

Use: To free all the resources allocated during the graph creation process

Returns: A status code indicating the success or failure

See rocalRelease example.

```
extern "C"  RocalStatus  ROCAL_API_CALL rocalRelease(RocalContext rocal_context);
```

12.1.5 Image Augmentation Using C++ API

The example below shows how to create a pipeline, read JPEG images, perform certain augmentations on them, and show the output using OpenCV by utilizing C++ API.

Listing 1: Example Image Augmentation

```
 Auto handle = rocalCreate(inputBatchSize, processing_device?RocalProcessMode::ROCAL_
↪PROCESS_GPU:RocalProcessMode::ROCAL_PROCESS_CPU, 0,1);
 input1 = rocalJpegFileSource(handle, folderPath1,  color_format, shard_count, false,␣
↪shuffle, false,  ROCAL_USE_USER_GIVEN_SIZE, decode_width, decode_height, dec_type);

 image0 = rocalResize(handle, input1, resize_w, resize_h, true);

 RocalImage image1 = rocalRain(handle, image0, false);


     RocalImage image11 = rocalFishEye(handle, image1, false);


     rocalRotate(handle, image11, true, rand_angle);


     // Creating successive blur nodes to simulate a deep branch of augmentations
     RocalImage image2 = rocalCropResize(handle, image0, resize_w, resize_h, false, rand_
↪crop_area);;
     for(int i = 0 ; i < aug_depth; i++)
     {
         image2 = rocalBlur(handle, image2, (i == (aug_depth -1)) ? true:false );
     }
 // Calling the API to verify and build the augmentation graph
     if(rocalVerify(handle) != ROCAL_OK)
     {
         std::cout << "Could not verify the augmentation graph" << std::endl;
         return -1;
     }
```

(continues on next page)

(continued from previous page)

```
while (!rocalIsEmpty(handle))
    {
        if(rocalRun(handle) != 0)
            break;
}
```

To see a sample image augmentation application in C++, see Image Augmentation.

CHAPTER

THIRTEEN

ROCAL C++ API

Consult the following files for detailed information on the rocAL C++ API:

- Functions
- Data types
- Augmentations
- Data loaders
- Data transfer
- Info
- Metadata
- Parameters

CHAPTER

FOURTEEN

ROCAL PYTHON API OVERVIEW

The rocAL Python package has been created using Pybind11 which enables data transfer between the rocAL C++ API and Python API. The `rocal_pybind` package includes both PyTorch and TensorFlow framework support and support for multiple data readers such as `FileReader`, `COCOReader`, and `TFRecordReader`.

The rocAL data types are defined in amd.rocal.types.

amd.rocal.fn
: Contains the image augmentations linked to the rocAL C++ API.

amd.rocal.decoders
: Image, video, and audio decoders.

amd.rocal.readers
: Image, video, and audio readers.

amd.rocal.pipeline
: The pipeline class encapsulates the data needed to build and run a rocAL graph. This includes support for context and graph creation, functions to verify and run the graph, and data transfer functions.

amd.rocal.types
: enums exported from the C++ API to Python.

amd.rocal.plugin.pytorch
: PyTorch plugin that includes the `ROCALGenericIterator` for Pytorch. The `ROCALClassificationIterator` class implements an iterator for image classification that returns labelled images.

amd.rocal.plugin.tf
: TensorFlow plugin that includes the `readers.tfrecord` TensorFlow reader.

amd.rocal.plugin.jax
: JAX plugin that includes the `ROCALJaxIterator` for JAX. The `ROCALJaxIterator` implements an iterator for running a pipeline over multiple devices.

CHAPTER

FIFTEEN

ROCAL PYTHON API

Consult the following files for detailed information on the rocAL Python API:

- Readers
- Decoders
- Augmentations
- Generic iterators
- Pipeline
- Randomization
- PyTorch integration
- TensorFlow integration
- JAX integration

CHAPTER

SIXTEEN

LICENSE