Qualcomm Cloud AI 100 ARCFACE using SCRFD + RESNET 100

This notebook provides instructions for compiling and executing deep learning networks on the Qualcomm® Cloud AI 100 (AIC) Accelerator Card.

Prerequisites for these exercises:

- Qualcomm Cloud AI 100 Platform SDK and Apps SDK. These are available at http://createpoint.qti.qualcomm.com (http://createpoint.qti.qualcomm.com).
- Conda Python 3.6 environment

This notebook should be run from a conda environment:

```
conda create -n arc python=3.6
```

Activate the conda environment and install required Python packages

```
conda activate arc

sudo apt-get install protobuf-compiler libprotoc-dev

pip install onnx

pip install mxnet

pip install numpy

pip install matplotlib

pip install opencv-python

pip install easydict

pip install scikit-image

pip install torch
```

Start the Jupyter notebook:

```
jupyter notebook --no-browser --port 8080 --allow-root
```

Qualcomm Cloud AI 100 Apps and Platform SDKs provide the necessary tools, drivers and firmware to quantize, compile, optimize and run deep learning models on Qualcomm Cloud AI 100 Accelerators. The SDKs are supported in CentOS 7, Ubuntu 18.04 and Ubuntu 20.04 Linux distributions with Kernel 5.4.1 or later.

Here are the SDK tools used in this notebook.

qaic-util Query AIC100 card status
qaic-exec Quantize and compile networks for AIC100. Supports ONNX, TensorFlow, PyTorch, Caffe2 and Caffe formats

Cloud AI 100 Card Setup

Check Card Status

First, let's check the status of AIC100 cards on the system with qaic-util. Here are some key items to check:

- Number of devices - Shows the number of AIC100 cards detected on the system
- Dram Free - Shows the amount of free RAM on the card
- Nsp Free - Shows the number of available AI compute cores

In [ ]:

```
!/opt/qti-aic/tools/qaic-util -q
```

Workflow

The workflow for optimizing models for the Cloud AI 100 Accelerator Card include:

- **Download model** Use pretrained model scrfd_10g_bnkps_480_480.onnx for face detection and arcfaceresnet100-8.onnx for arcface
- **Prepare inputs** Download or generate input data such as image or text. Can optionally use randomly generated input data.
- **Quantize (optional)** Quantize network layers to INT8 to improve performance.
- **Compile** Compile model file into network binaries for AIC100.
- **Run network** Run network binaries on AIC100 using PYTHON API
- **Check Key Performance Indicators (KPI)** Use SDK tools to check performance, latency, and power.

Prepare Model

Retrain models using custom datasets

Cleanup

In [ ]:

```
!rm -rf ./compiler_output_fp16
!rm -rf ./compiler_output_fp16_1
!rm -rf crop*
!rm -rf arcfaceresnet100-8.onnx
!rm -rf QAicArcFace.py
```

Download model zoo

In [ ]:

```
!wget -O arcfaceresnet100-8.onnx
https://github.com/onnx/models/blob/main/vision/body_analysis/arcface/model/arcfaceresnet1
00-8.onnx?raw=true
```

In [ ]:

```
!ls
```

Quantization (Optional)

Next we'll generate a quantization profile using the compiler's profile guided quanitzation feature. ...

Compile the Network

In [ ]:

```

#print("Compile SCRFD model")

# FP16
!/opt/qti-aic/exec/qaic-exec \
-m=./scrfd_10g_bnkps_480_480.onnx \
-aic-hw \
-mos=1 \
-ols=1 \
-aic-num-cores=1 \
-convert-to-fp16 \
-aic-minimize-host-traffic \
-aic-binary-dir=./compiler_output_fp16 \
-compile-only \
-aic-num-of-instances=14 \
-batchsize=1 \
-host-preproc

# print("")
print("Compile RESNET100 model zoo")
!/opt/qti-aic/exec/qaic-exec \
-m=./arcfaceresnet100-8.onnx \
-aic-hw \
-mos=1 \
-ols=1 \
-aic-num-cores=1 \
-convert-to-fp16 \
-aic-minimize-host-traffic \
-aic-binary-dir=./compiler_output_fp16_1 \
-compile-only \
-aic-num-of-instances=14 \
-batchsize=1 \
-host-preproc
```

In [ ]:

```
from numpy import dot, sqrt
def cosine_similarity(x, y):
    return dot(x, y) / (sqrt(dot(x, x)) * sqrt(dot(y, y)))
```

Input

In [ ]:

```python
from IPython.display import Image

Image('./image1.png')
```

Using AIC100 Python API

In [ ]:

```python
Image('./image2.png')
```

In [ ]:

```
%%writefile QAicArcFace.py

from __future__ import print_function
from __future__ import absolute_import
from __future__ import division
import os
import argparse
from argparse import ArgumentParser
import sys
import json
import numpy as np
from PIL import Image
import cv2
import torch
import time
import torchvision
import string
import sklearn
from sklearn import preprocessing
from typing import List

sys.path.append("/opt/qti-aic/dev/lib/x86_64/")
from qaicrt import Util, ExecObj, Context, Program, Queue, QBuffer, QStatus, QDevInfo,
Qpc, InferenceVector, BufferMapping, BufferIoTypeEnum, QIDList

def softmax(z):
    assert len(z.shape) == 2
    s = np.max(z, axis=1)
    s = s[:, np.newaxis] # necessary step to do broadcasting
    e_x = np.exp(z - s)
    div = np.sum(e_x, axis=1)
    div = div[:, np.newaxis] # dito
    return e_x / div

def distance2bbox(points, distance, max_shape=None):
    """Decode distance prediction to bounding box.

    Args:
        points (Tensor): Shape (n, 2), [x, y].
        distance (Tensor): Distance from the given point to 4
            boundaries (left, top, right, bottom).
        max_shape (tuple): Shape of the image.

    Returns:
        Tensor: Decoded bboxes.
    """
    x1 = points[:, 0] - distance[:, 0]
    y1 = points[:, 1] - distance[:, 1]
    x2 = points[:, 0] + distance[:, 2]
    y2 = points[:, 1] + distance[:, 3]
    if max_shape is not None:
        x1 = x1.clamp(min=0, max=max_shape[1])
        y1 = y1.clamp(min=0, max=max_shape[0])
        x2 = x2.clamp(min=0, max=max_shape[1])
        y2 = y2.clamp(min=0, max=max_shape[0])
    return np.stack([x1, y1, x2, y2], axis=-1)

def distance2kps(points, distance, max_shape=None):
    """Decode distance prediction to bounding box.
    Args:
        points (Tensor): Shape (n, 2), [x, y].
        distance (Tensor): Distance from the given point to 4
            boundaries (left, top, right, bottom).
        max_shape (tuple): Shape of the image.

    Returns:
        Tensor: Decoded bboxes.
    """
    preds = []
    for i in range(0, distance.shape[1], 2):
        px = points[:, i%2] + distance[:, i]
        py = points[:, i%2+1] + distance[:, i+1]
        if max_shape is not None:
            px = px.clamp(min=0, max=max_shape[1])
            py = py.clamp(min=0, max=max_shape[0])
        preds.append(px)
        preds.append(py)
    return np.stack(preds, axis=-1)

def nms(dets, nms_thresh=0.4):
    thresh = nms_thresh
    x1 = dets[:, 0]
    y1 = dets[:, 1]
    x2 = dets[:, 2]
    y2 = dets[:, 3]
    scores = dets[:, 4]

    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    order = scores.argsort()[::-1]

    keep = []
    while order.size > 0:
        i = order[0]
        keep.append(i)
        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])

        w = np.maximum(0.0, xx2 - xx1 + 1)
        h = np.maximum(0.0, yy2 - yy1 + 1)
        inter = w * h
        ovr = inter / (areas[i] + areas[order[1:]] - inter)

        inds = np.where(ovr <= thresh)[0]
        order = order[inds + 1]

    return keep

def dir_path(path):
    if os.path.isdir(path):
        return path
    else:
        raise argparse.ArgumentTypeError(
            'readable_dir:{} is not a valid path'.format(path))

def arc_preproc(img):
    #img = cv2.imread(img_path)
    resized = cv2.resize(img, (112, 112))
```

```python
    resized = cv2.cvtColor(resized, cv2.COLOR_BGR2RGB)
    aligned = np.transpose(resized, (2,0,1))
    blob = np.expand_dims(aligned, axis=0).astype(np.float32)
    return blob

def arc_postproc(input_path, pred):
    qBuffer = pred[1]
    output_buffer = bytearray(qBuffer)
    output_data = np.frombuffer(output_buffer, dtype=np.float32)
    out = output_data.reshape(1, -1)
    embedding = sklearn.preprocessing.normalize(out).flatten()
    return embedding

def scrfd_preproc(img_path):
    img = cv2.imread(img_path)
    input_size = (480, 480)
    input_mean = 127.5
    input_std = 128.0

    im_ratio = float(img.shape[0]) / img.shape[1]
    model_ratio = float(input_size[1]) / input_size[0]

    if im_ratio > model_ratio:
        new_height = input_size[1]
        new_width = int(new_height / im_ratio)
    else:
        new_width = input_size[0]
        new_height = int(new_width * im_ratio)

    det_scale = float(new_height) / img.shape[0]

    resized_img = cv2.resize(img, (new_width, new_height))
    det_img = np.zeros((input_size[1], input_size[0], 3), dtype=np.uint8)
    det_img[:new_height, :new_width, :] = resized_img
    input_size = tuple(det_img.shape[0:2][::-1])

    blob = cv2.dnn.blobFromImage(
        det_img,
        1.0/input_std,
        input_size,
        (input_mean, input_mean, input_mean),
        swapRB=True) # NCHW
    return blob

def scrfd_postproc(input_path, pred):
    scores_list = []
    bboxes_list = []
    kpss_list = []
    input_height = 480
    input_width = 480
    input_size = (480, 480)
    feat_stride_fpn = [8, 16, 32]
    fmc = 3
    num_anchors = 2
    center_cache = {}
    threshold = 0.5
    nms_threshold = 0.4
    max_num = 0
    img0 = cv2.imread(input_path)

    im_ratio = float(img0.shape[0]) / img0.shape[1]
    model_ratio = float(input_size[1]) / input_size[0]
    if im_ratio>model_ratio:
        new_height = input_size[1]
        new_width = int(new_height / im_ratio)
    else:
        new_width = input_size[0]
        new_height = int(new_width * im_ratio)
    det_scale = float(new_height) / img0.shape[0]

    for idx, stride in enumerate(feat_stride_fpn):
        scores = np.frombuffer(bytearray(pred[idx+1]), dtype=np.float32).reshape(-1,1)
        bbox_preds = np.frombuffer(bytearray(pred[idx+1+fmc]),
dtype=np.float32).reshape(-1,4)
        bbox_preds = bbox_preds * stride
        kps_preds = np.frombuffer(bytearray(pred[idx+1+fmc*2]),
dtype=np.float32).reshape(-1, 10)
        kps_preds = kps_preds * stride

        height = input_height // stride
        width = input_width // stride
        K = height * width
        key = (height, width, stride)
        if key in center_cache:
            anchor_centers = center_cache[key]
        else:
            anchor_centers = np.stack(np.mgrid[:height, :width][::-1],
axis=-1).astype(np.float32)
            anchor_centers = (anchor_centers * stride).reshape( (-1, 2) )
            if num_anchors > 1:
                anchor_centers = np.stack([anchor_centers] * num_anchors, axis=1).reshape(
(-1,2) )

            if len(center_cache) < 100:
                center_cache[key] = anchor_centers

        pos_inds = np.where(scores>=threshold)[0]
        bboxes = distance2bbox(anchor_centers, bbox_preds)
        pos_scores = scores[pos_inds]
        pos_bboxes = bboxes[pos_inds]
        scores_list.append(pos_scores)
        bboxes_list.append(pos_bboxes)
        kpss = distance2kps(anchor_centers, kps_preds)
        kpss = kpss.reshape( (kpss.shape[0], -1, 2) )
        pos_kpss = kpss[pos_inds]
        kpss_list.append(pos_kpss)

    scores = np.vstack(scores_list)
    scores_ravel = scores.ravel()
    order = scores_ravel.argsort()[::-1]
    bboxes = np.vstack(bboxes_list) / det_scale
    kpss = np.vstack(kpss_list) / det_scale
    pre_det = np.hstack((bboxes, scores)).astype(np.float32, copy=False)
    pre_det = pre_det[order, :]
    keep = nms(pre_det, nms_thresh=nms_threshold)
    det = pre_det[keep, :]
    kpss = kpss[order,:,:]
    kpss = kpss[keep,:,:]
    if max_num > 0 and det.shape[0] > max_num:
        area = (det[:, 2] - det[:, 0]) * (det[:, 3] - det[:, 1])
        img_center = img0.shape[0] // 2, img0.shape[1] // 2
        offsets = np.vstack([
```

```python
            (det[:, 0] + det[:, 2]) / 2 - img_center[1],
            (det[:, 1] + det[:, 3]) / 2 - img_center[0]
        ])
        offset_dist_squared = np.sum(np.power(offsets, 2.0), 0)
        if metric=='max':
            values = area
        else:
            values = area - offset_dist_squared * 2.0  # some extra weight on the
centering
        bindex = np.argsort(
            values)[::-1]  # some extra weight on the centering
        bindex = bindex[0:max_num]
        det = det[bindex, :]
        if kpss is not None:
            kpss = kpss[bindex, :]
    for i in range(det.shape[0]):
        bbox = det[i]
        x1,y1,x2,y2,score = bbox.astype(np.int)
        crop = img0[y1:y2, x1:x2]
    return crop

def parse_args():
    parser = ArgumentParser(description="Face recognition example")

    parser.add_argument(
        '-d', "--device", dest="device", default=0, type=int, help="Device for QAIC
Inference"
    )
    parser.add_argument(
        "-t",
        "--model-path",
        dest="model_path",
        required=True,
        help="Path of the onnx/pytorch model. In case of AIC it can be "
        + "directory of QAIC compiled model binaries.",
    )
    parser.add_argument(
        "-t1",
        "--model-path1",
        dest="model_path1",
        required=True,
        help="Path of the onnx/pytorch model. In case of AIC it can be "
        + "directory of QAIC compiled model binaries.",
    )
    parser.add_argument(
        "-i",
        "--input",
        dest="input_path",
        required=True,
        help="Path of input",
    )

    parser.add_argument(
        "-i1",
        "--input1",
        dest="input_path1",
        required=True,
        help="Path of input1",
    )
    return parser.parse_args()
```

```python
class QAiCInf:
    def __init__(self, device_id, binary_path, preproc_func, postproc_func):
        self.device_id = device_id
        self.binary_path = binary_path
        self.preproc_func = preproc_func
        self.postproc_func = postproc_func

        if not os.path.isdir(self.binary_path):
            print("Binary directory not found.")
            exit()

        if not os.path.isfile(self.binary_path + "/programqpc.bin"):
            print("programqpc.bin not found at given binary path.")
            exit()

        self.dev_list = QIDList()
        self.dev_list.append(self.device_id)
        context = Context(self.dev_list)
        self.qpc = Qpc(self.binary_path)

        if self.qpc == None:
            print("Unable to open program container: " + self.binary_path)
            sys.exit(1)

        self.buf_mappings = self.qpc.getBufferMappings()
        self.program = Program(context, self.dev_list[0], self.qpc)
        self.execObj = ExecObj(context, self.program)
        self.queue = Queue(context, self.dev_list[0])

    def run(self, inp):
        img0 = self.preproc_func(inp)
        all_input_data = [img0]
        raw_buffers = []

        for inp_data in all_input_data:
             raw_buffers.append(inp_data.tobytes())
        #raw_buffers.append(im_preproc)
        i = 0
        for mapping in self.buf_mappings:
            if mapping.ioType == BufferIoTypeEnum.BUFFER_IO_TYPE_OUTPUT:
                raw_buffers.append(bytearray(mapping.size))

        qbuf_list = []
        for buffer in raw_buffers:
            qbuf_list.append(QBuffer(buffer))

        self.execObj.setData(qbuf_list) # ---> add
        self.execObj.run(qbuf_list)  # ---> add
        # Run asynchonous
        self.queue.enqueue(self.execObj)
        self.execObj.waitForCompletion()
        status, execObjData = self.execObj.getData()
        if self.postproc_func is not None:
            output = self.postproc_func(inp, execObjData)
            return output
        else :
            return execObjData

def main():
    args = parse_args()
    qaic_scrfd_infer = QAiCInf(device_id=args.device,
                                binary_path=args.model_path,
                                preproc_func=scrfd_preproc,
                                postproc_func=scrfd_postproc)
    #input_path = './trump.png'
    pred = qaic_scrfd_infer.run(args.input_path)
    cv2.imwrite("crop.jpg", pred)
    qaic_arc_infer = QAiCInf(device_id=args.device,
                              binary_path=args.model_path1,
                              preproc_func=arc_preproc,
                              postproc_func=arc_postproc)
    embedding = qaic_arc_infer.run(pred)
    embeddings = []
    embeddings.append(embedding)
    #input_path = './trump1.png'
    pred = qaic_scrfd_infer.run(args.input_path1)
    cv2.imwrite("crop1.jpg", pred)
    embedding = qaic_arc_infer.run(pred)
    embeddings.append(embedding)
    dist = np.sum(np.square(embeddings[0]-embeddings[1]))
    # Compute cosine similarity between embedddings
    sim = np.dot(embeddings[0], embeddings[1].T)
    # Print predictions
    print('Distance = %f' %(dist))
    print('Similarity = %f' %(sim))

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

In [ ]:

```
!python QAicArcFace.py -t ./compiler_output_fp16/ -t1 ./compiler_output_fp16_1 -i
image1.png -i1 image2.png
```

In [ ]:

```
Image('./crop.jpg')
```

In [ ]:

```
Image('./crop1.jpg')
```