Application Acceleration Using a Heterogeneous MPSoC Architecture with MPU and FPGA Processors

Student: Mikel Solabarrieta Román
Director: Ignacio Angulo Martínez
Degree: Industrial Electronics and Automation Engineering
University of Deusto - 2020

Index

Introduction

Key areas

Edge Computing

· Process (and store) data close to its origin

· Used in IoT

· General rule: closer to the data means lower processing power

· Embedded devices

↓

· Limited capabilities

System-on-Chip

· Integrated circuits

· CPU, memory, I/O ports...

· Usually without primary storage

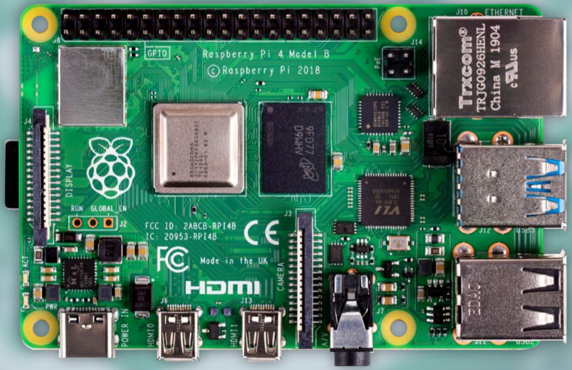
· Usually used for lightweight edge computing

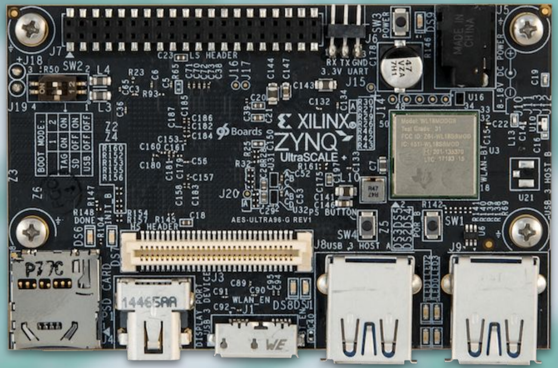
Can embedded devices run computationally heavy tasks in real-time?

Facial recognition

Object detection

Embedded Devices

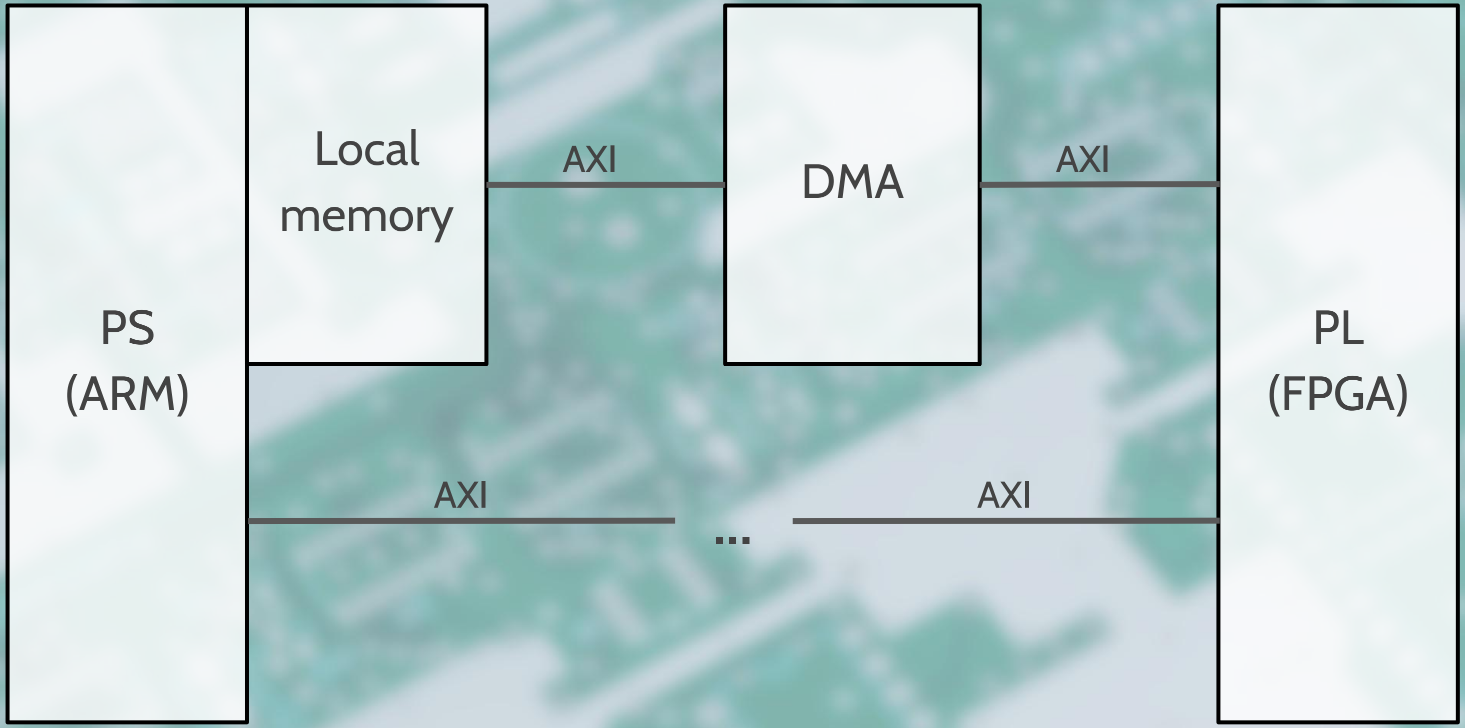




- Homogeneous MPSoC (multicore)
- Low-power, portable (relevant for IoT)
- Very extended

- Heterogeneous MPSoC: PS+PL
- PS is dominant
- PL allows certain tasks to run **fast**

Zynq



PYNQ

Overview

PYNQ exposes Zynq functionalities through a Python API.

Accessibility in mind.

Overlays (PL design) management.

Memory management.

Asyncio compatible.

Built over Linux.

Jupyter notebooks.

How?

<u>Directly</u> - less abstraction

```
from pynq import Overlay, Xlink
overlay = Overlay(...); overlay.download()
mem_array = Xlink().cma_array(...)
overlay.write(address, data)
```

<u>Indirectly</u> - more abstraction

```
import bnn
classifier = bnn.LfcClassifier(...)
result = classifier.classify_mnist(...)
# or even
result = await classifier.classify_mnist(...)
```

lib	mmio.py
notebooks	overlay.py
overlays	pl.py
pl_server	pmbus.py
tests	ps.py
__init__.py	registers.py
bitstream.py	tinynumpy.py
buffer.py	uio.py
devicetree.py	utils.py
ert.py	xclbin.py
gpio.py	xlnk.py
interrupt.py	xrt.py

Facial recognition

Initial situation (1/2)

Face recognition programs exist for PYNQ but they don't take full advantage of PL.
Accelerated image transformation, but not classification.

- github.com/larvelJ/PYNQ_facialRec

    OpenCV (mostly non-accelerated) for classification

```
def classify_face(face_frame, faces): # Func
ngerprint
    facenet = cv2.dnn.readNetFromCaffe('bvlc
    face_crop = face_frame[faces[0][1]:faces
age has to be a certain size for the Nueral
    faceblob = cv2.dnn.blobFromImage(face_cr
    facenet.setInput(faceblob)
    facenet_fingerprint = facenet.forward()
    return facenet_fingerprint
```

- github.com/julianbartolone/doorbellcam

    External library (non-accelerated) for classification

```
# Find all the faces and face encodings in the current frame of vid
face_locations = face_recognition.face_locations(rgb_small_frame)
face_encodings = face_recognition.face_encodings(rgb_small_frame, f

face_names = []
for face_encoding in face_encodings:
    # See if the face is a match for the known face(s)
    matches = face_recognition.compare_faces(known_face_encodings,
    name = "Unknown"

    # If a match was found in known_face_encodings, just use the fi
    if True in matches:
```

Initial situation (2/2)

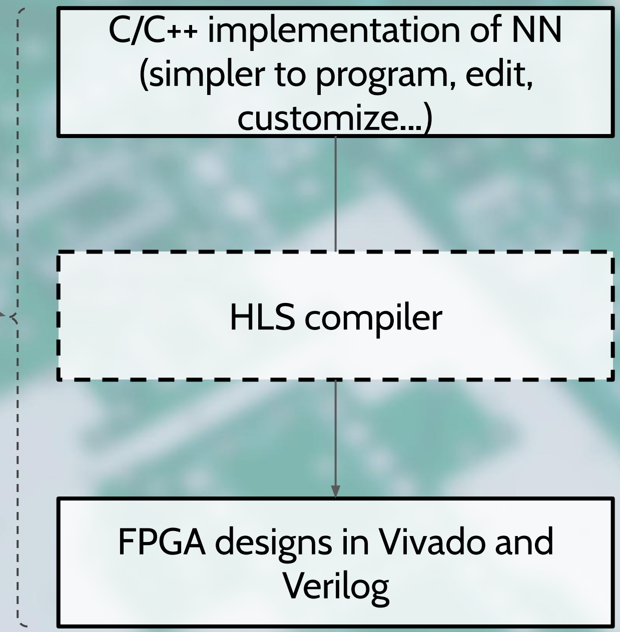
There are similar accelerated classifiers: BNN-PYNQ.

MNIST, GTSRB...

Based on FINN, exposed to user as a high level API.

Therefore:

Repurpose BNN-PYNQ for this project.



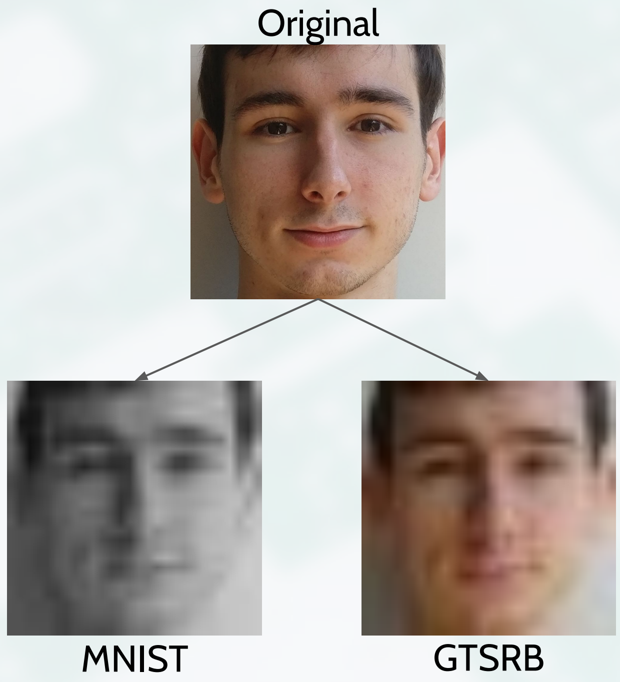
How? (1/2)

Tools are provided to train LFC and CNV topologies.

Adapt a face dataset to the format used by each topology.

MNIST for LFC: 28x28 grayscale, specific header, dataset structure.

GTSRB for CNV: 32x32 RGB, location of faces required, specific dataset structure.



How? (2/2)

Prepare dataset
↓
Train, validate and test model on PC
↓
Generate weight bitstream on PC
↓
Import bitstream on PYNQ
↓
Call classifier

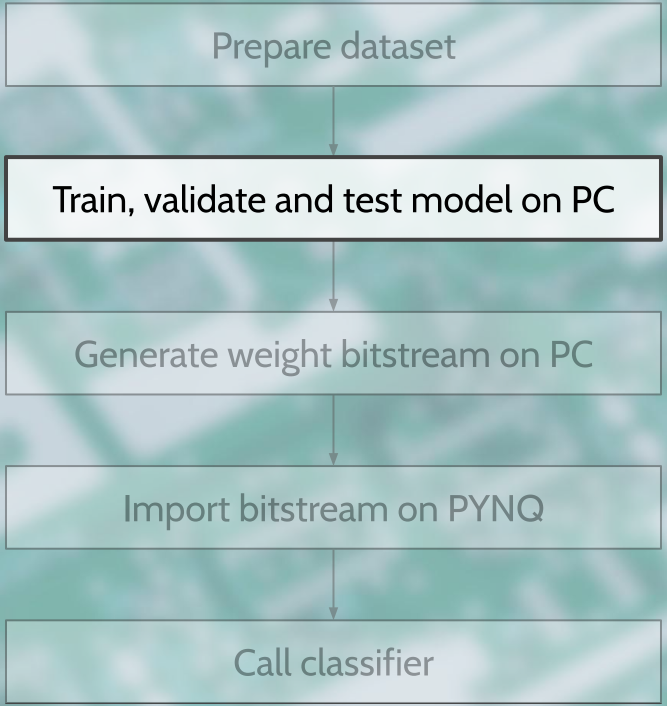
```
Epoch 997 of 1000 took 19.433754921s
  LR:                            3.09893715972e-07
  training loss:                 0.00693385727983
  validation loss:               0.0849743406848
  validation error rate:         13.3333333333%
  best epoch:                    997
  best validation error rate:    13.3333333333%
  test loss:                     0.0115805853067
  test error rate:               6.66666666667%
```

CNV

```
Epoch 992 of 1000 took 1.93348097801 seconds
LR:                            3.25927687085e-07
training loss:                 0.0916081467252
validation loss:               0.0945482701374
validation error rate:         12.5%
best epoch:                    992
best validation error rate:    12.5%
test loss:                     0.0483024631657
test error rate:               8.33333333333%
```

LFC

How? (2/2)




```
Epoch 997 of 1000 took 19.433754921s
  LR:                           3.09893715972e-07
  training loss:                0.00693385727983
  validation loss:              0.0849743406848
  validation error rate:        13.3333333333%
  best epoch:                   997
  best validation error rate:   13.3333333333%
  test loss:                    0.0115805853067
  test error rate:              6.66666666667%
```

CNV

```
Epoch 992 of 1000 took 1.93348097801 seconds
LR:                          3.25927687085e-07
training loss:               0.0916081467252
validation loss:             0.0945482701374
validation error rate:       12.5%
best epoch:                  992
best validation error rate: 12.5%
test loss:                   0.0483024631657
test error rate:             8.33333333333%
```

LFC

Usage

High level Python API

```
import bnn
from PIL import Image

img = Image(...)
img = format_image(img)

classifier = bnn.LfcClassifier(...) # or CnvClassifer
result = classifier.classify(img)
```

Pre-process image

Download overlay of topology, load weight bitstream, wrap methods

Allocate memory, assign channels to MM, trigger classifier, wait for results

Object detection

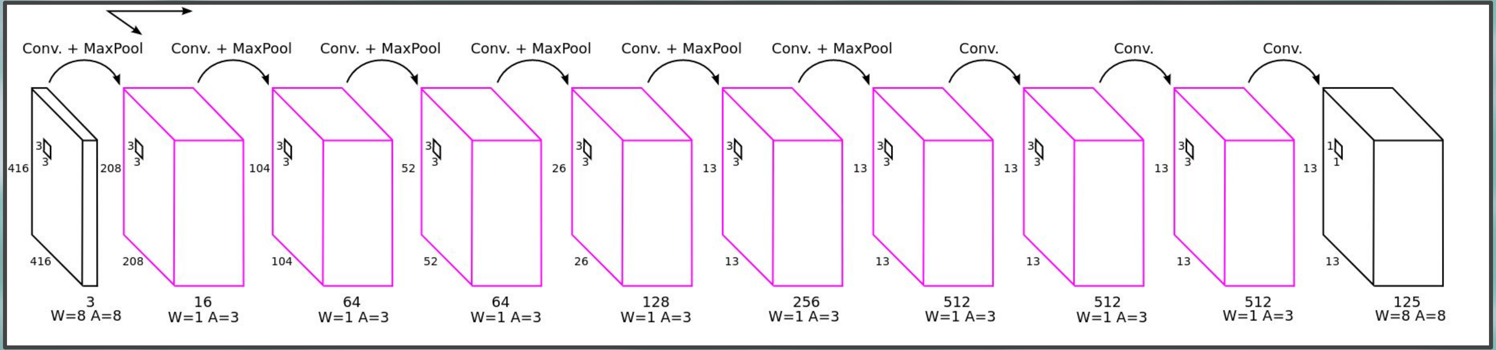
Initial situation (1/2)

Recent implementation of YOLO on PYNQ: QNN-MO-PYNQ.

Modified version of YOLOv2: less demanding but less accurate.

Partial implementation, but the most complete one available at the moment.

First and last layers have precision weights: can't be quantized.



Initial situation (2/2)

Darknet: NN library written in C for SW processing.

Dependency lacking functions to extract results, can only display them.

Extracting results is vital for integration.

Therefore:

Modify Darknet and update the dependency.

How? (1/2)

Update latest Darknet version to include the methods of QNN-MO-PYNQ and any other method required by the project.

Example:

C function

```c
detection *get_network_boxes(network *net, int w, int h, floa
{
    detection *dets = make_network_boxes(net, thresh, num);
    fill_network_boxes(net, w, h, thresh, hier, map, relative,
    return dets;
}
```

Python binding

```python
get_network_boxes = lib.get_network_boxes
get_network_boxes.argtypes = [c_void_p, c_int, c_int, c_floa
get_network_boxes.restype = POINTER(DETECTION)
```

How? (2/2)

1. Modify Darknet
2. Compile new version in PYNQ device
3. Change QNN dependency
4. Download overlay and import bitstream
5. Configure network
6. Run classifier
7. Process results

Usage

High level Python API

```
import qnn
from PIL import Image

img = Image(...)
img = format_image(img)

net = darknet.lib.parse_network(...)
classifier = TinierYolo()
classifier.init_accelerator()

first_layer(net, ...)                 # sw
classifier.middle_layers(net, ...) # hw
last_layer(net, ...)                  # sw

results = post_process(net, ...)
```

Pre-process image

Create and configure SW network and HW classifier

First layer: software
Middle layers: accelerated
Final layer: software

Allocate memory, assign channels to MM, trigger classifier, wait for results

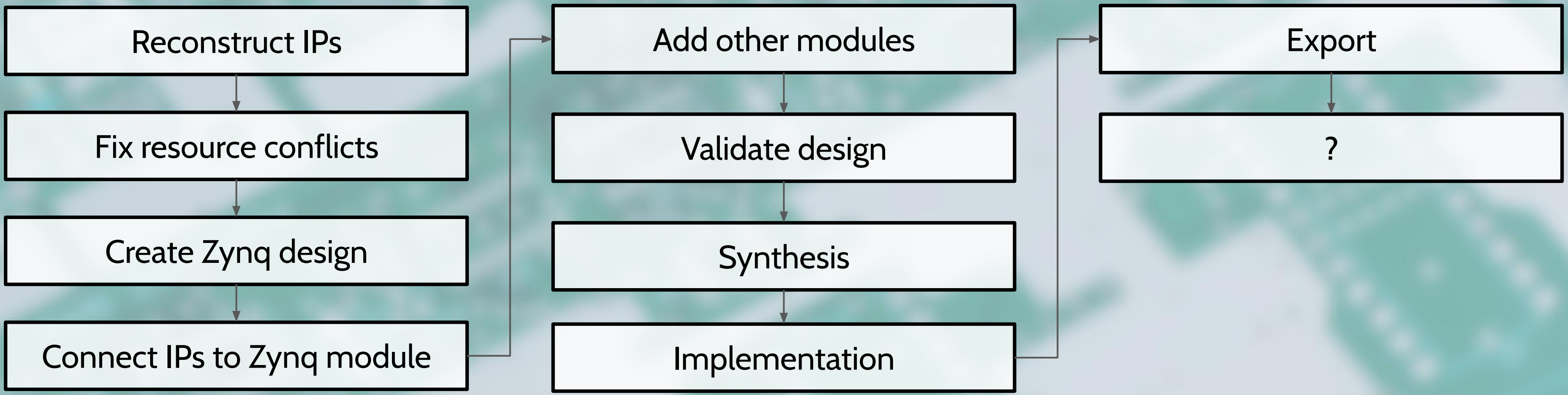
Get detection boxes, sort and filter results

Integration

How?

Expose the functionalities of both projects in a single Overlay.

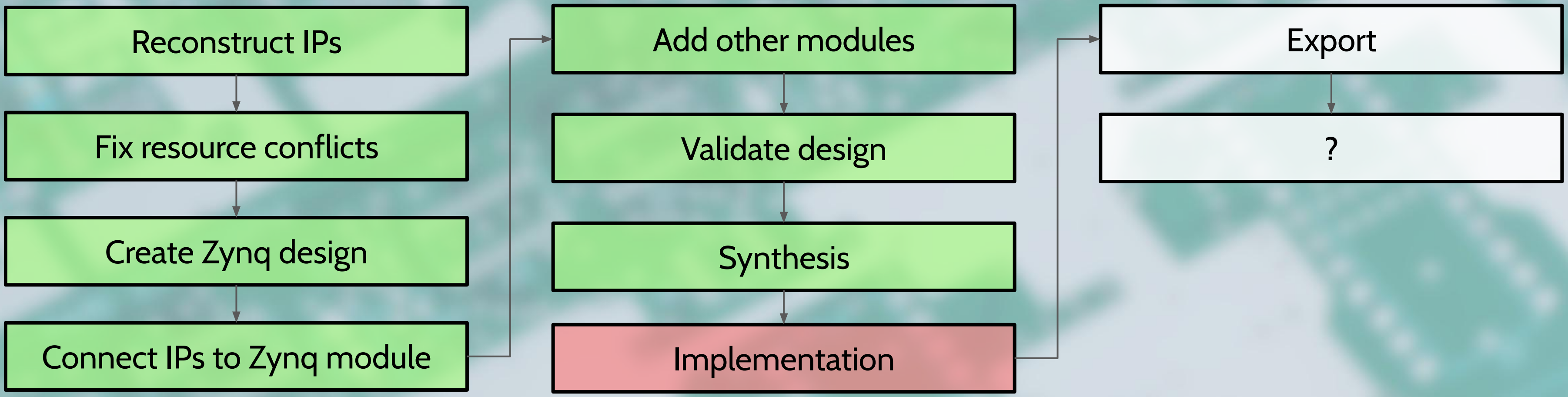
For that: reconstruct the IP of each overlay and integrate them in a single design.



Close, but not there yet.

Resulting design didn't fit the board PL.

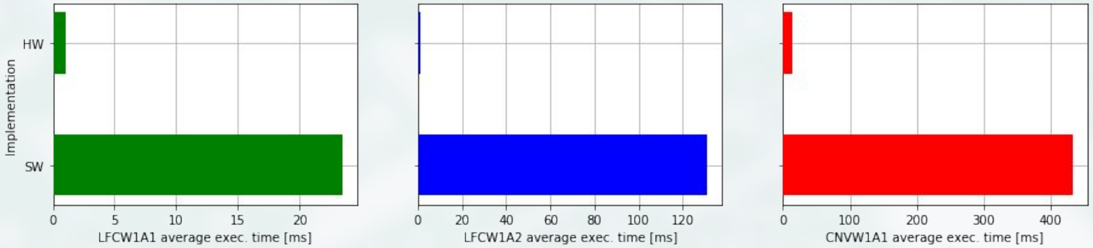
Unable to reduce the size enough to fit.



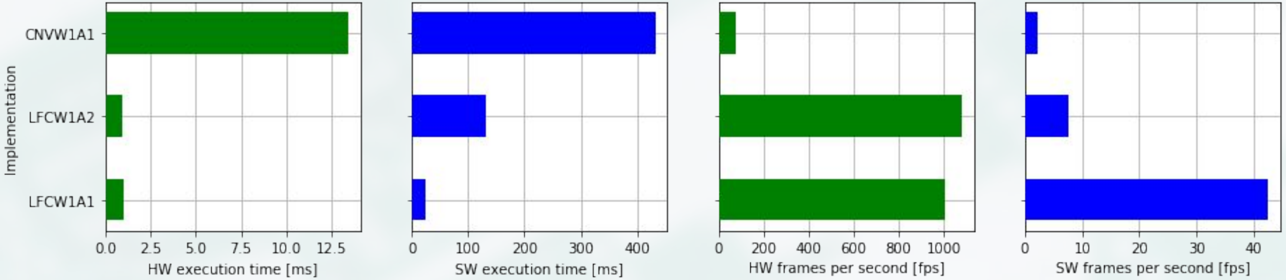
Results and Conclusions

Results (1/3)

SW and HW comparison

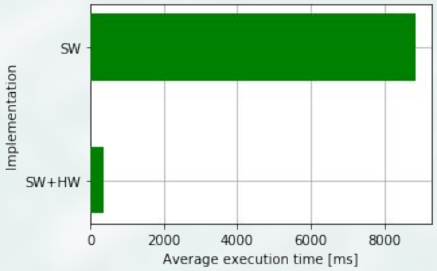


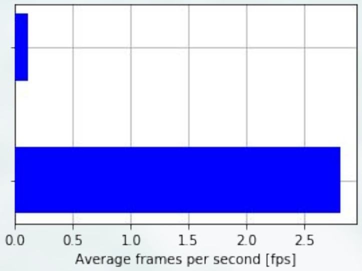
Topology comparison



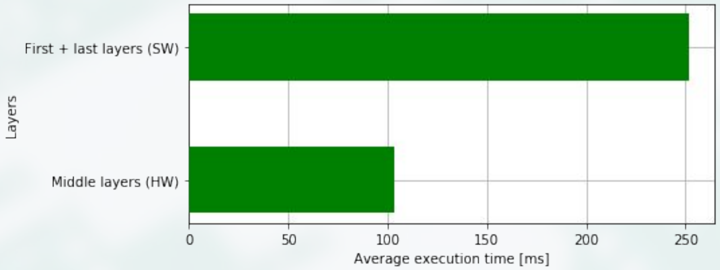
Results (2/3)

SW and HW **classifier** comparison

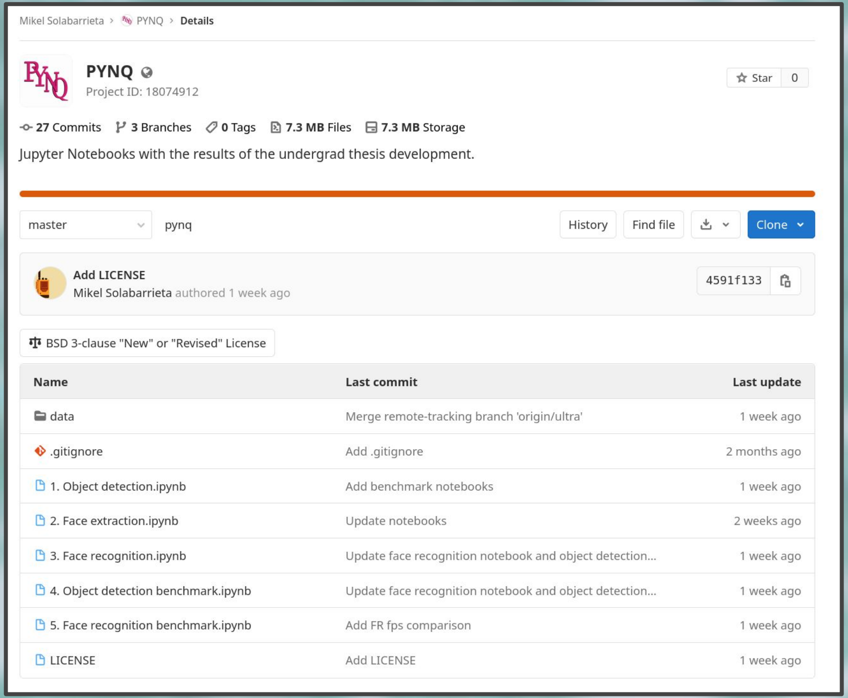


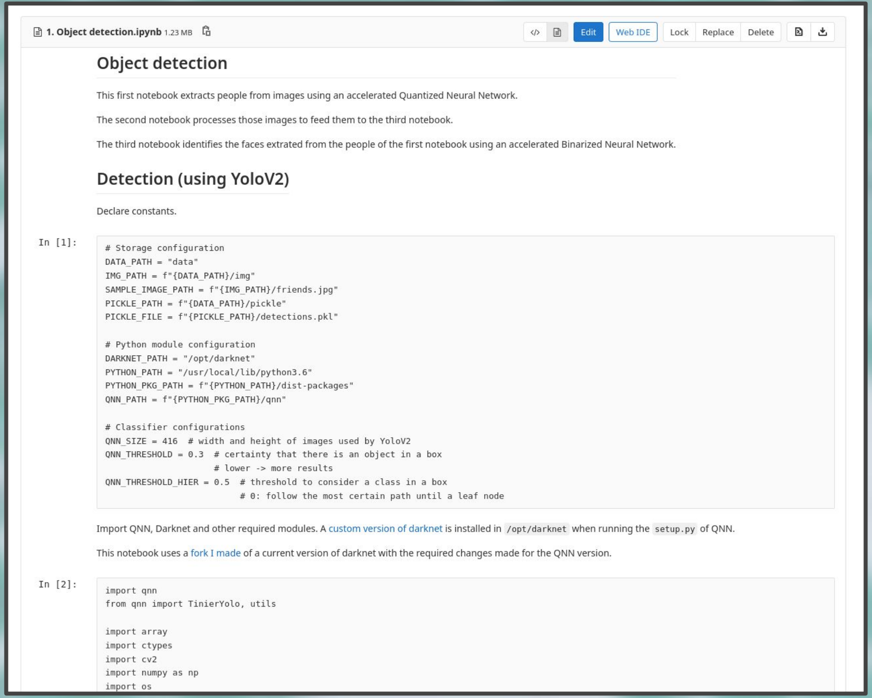


SW and HW **layer** comparison



Results (3/3)


Mikel Solabarrieta › PYNQ › Details
PYNQ
Project ID: 18074912
Star 0
27 Commits 3 Branches 0 Tags 7.3 MB Files 7.3 MB Storage
Jupyter Notebooks with the results of the undergrad thesis development.
master
pynq
History
Find file
Clone
Add LICENSE
Mikel Solabarrieta authored 1 week ago
4591f133
BSD 3-clause "New" or "Revised" License
Name
Last commit
Last update
data
Merge remote-tracking branch 'origin/ultra'
1 week ago
.gitignore
Add .gitignore
2 months ago
1. Object detection.ipynb
Add benchmark notebooks
1 week ago
2. Face extraction.ipynb
Update notebooks
2 weeks ago
3. Face recognition.ipynb
Update face recognition notebook and object detection...
1 week ago
4. Object detection benchmark.ipynb
Update face recognition notebook and object detection...
1 week ago
5. Face recognition benchmark.ipynb
Add FR fps comparison
1 week ago
LICENSE
Add LICENSE
1 week ago


1. Object detection.ipynb 1.23 MB
Edit
Web IDE
Lock
Replace
Delete
Object detection
This first notebook extracts people from images using an accelerated Quantized Neural Network.
The second notebook processes those images to feed them to the third notebook.
The third notebook identifies the faces extrated from the people of the first notebook using an accelerated Binarized Neural Network.
Detection (using YoloV2)
Declare constants.
In [1]:
# Storage configuration
DATA_PATH = "data"
IMG_PATH = f"{DATA_PATH}/img"
SAMPLE_IMAGE_PATH = f"{IMG_PATH}/friends.jpg"
PICKLE_PATH = f"{DATA_PATH}/pickle"
PICKLE_FILE = f"{PICKLE_PATH}/detections.pkl"
# Python module configuration
DARKNET_PATH = "/opt/darknet"
PYTHON_PATH = "/usr/local/lib/python3.6"
PYTHON_PKG_PATH = f"{PYTHON_PATH}/dist-packages"
QNN_PATH = f"{PYTHON_PKG_PATH}/qnn"
# Classifier configurations
QNN_SIZE = 416 # width and height of images used by YoloV2
QNN_THRESHOLD = 0.3 # certainty that there is an object in a box
# lower -> more results
QNN_THRESHOLD_HIER = 0.5 # threshold to consider a class in a box
# 0: follow the most certain path until a leaf node
Import QNN, Darknet and other required modules. A custom version of darknet is installed in /opt/darknet when running the setup.py of QNN.
This notebook uses a fork I made of a current version of darknet with the required changes made for the QNN version.
In [2]:
import qnn
from qnn import TinierYolo, utils
import array
import ctypes
import cv2
import numpy as np
import os

Conclusions

· Performance is radically improved in critical sections of applications.

· PYNQ enables embedded devices to run demanding tasks with little latency, making real-time execution possible.

· This confirmation will possibly impact edge-computing and IoT architecture design paradigms.

· There is a lot of research and optimization potential.

Future Work

Future work

1. Test integration implementation in another device with a more resourceful PL.
2. Compare PYNQ device performance with other accelerated devices, such as NVIDIA Jetsons.
3. Accelerate other less critical parts of the applications.
4. Test PYNQ on new applications.

Q&A