# AN02013: FaceId And Keyword Spotting Application Note

Release: 0.1.0
Publication Date: 2024/09/05
Document Number: XM-015123-AN

XMOS

# Table of Contents

# 1 Introduction

This Application Note provides a detailed guide for developing integrated vision and audio solutions using the XCORE.AI chip. Along with this document, the source code for such an application is included.

The guide offers general information about both vision and audio systems in an embedded environment, explains the key components of the source code, and provides instructions on how to build, run, or modify the code to create a custom solution.

It is designed to help users develop vision or audio solutions using the XCORE.AI Vision Development Kit. The guide can be used to create a plug-and-play application, serve as a reference for developing other models, or be implemented on other XCORE.AI boards.

# 2 FaceID and Keyword Spotting Overview

On one hand, face recognition is a task that involves identifying or verifying a person from a digital image or video frame. There are different ways to achieve that. A common approach is to have a two-stage model, where the first stage is responsible for detecting the face in the image (where is the face), and the second stage is responsible for recognizing the given face (whose face is this). This Application Note uses a model for face detection and another model for face recognition. The face detection model is a Single Shot MultiBox Detector (SSD) model, and the face recognition model is a MobileNetV2 model. This last model is able to generate a 128-dimensional array that represents the face features. This array is then compared with a database of known face features to identify the person. The application will use the Cosine Distance to compare the arrays and identify the person.

On the other hand, Keyword Spotting refers to being able to identify and respond to certain voice commands. That means capturing samples from a microphone, processing those to extract features, and passing those to a Neural Network to predict which keyword was pronounced, within a known set of keywords.

## 2.1 Thread Diagram

The application is divided into several threads, each responsible for a specific task. The following diagram illustrates the thread structure of the application:

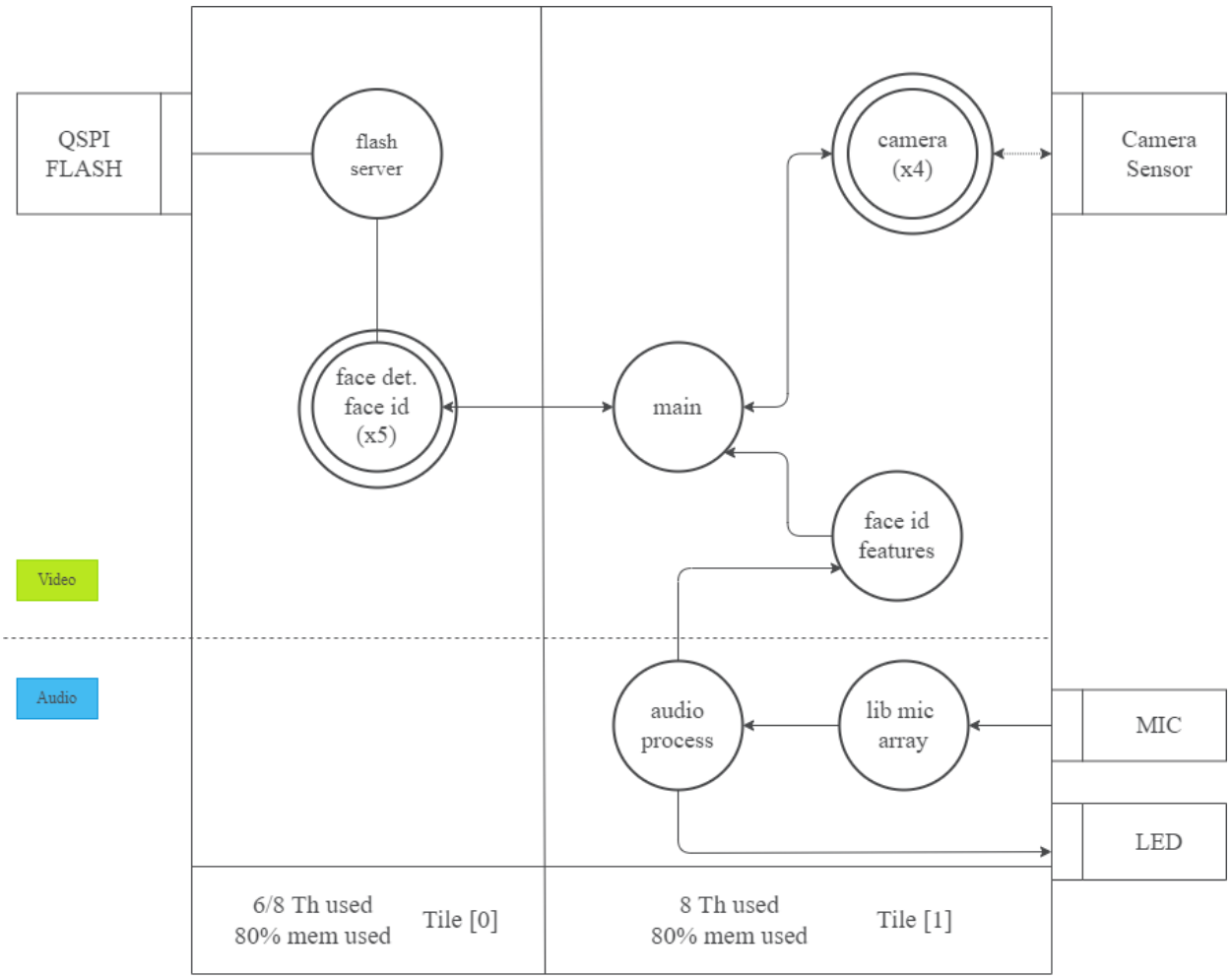


Fig. 2.1: Thread diagram

Tile[0] primarily focuses on the model, while Tile[1] covers high-level API, camera, and communication. Tasks are detailed in the following sections.

## 2.2 Folder Structure

The following is the folder structure of the application:

```
repository
    |
    |--- src
       |--- mapfile.xc
       |--- user_app.c
       |--- user_audio.c
       |--- user_mic.c
       |--- user_camera.xc
       |--- model
       |--- inference
       |--- audio
       |--- utils
```

- **src**: Contains the source code for the application.
- **mapfile.xc**: Entry point of the application.

- **user_app.c**: High-level thread that manages the application.
- **user_camera.xc**: Handles capturing images from the camera.
- **user_audio.c**: Handles the high level audio pipeline.
- **user_mic.c**: Handles capturing audio from the built-in microphone on the vision board.
- **inference**: Manages the vision inference process on the `XCORE.AI` chip, including pre- and post-processing of data.
- **model**: Stores models (non optimized) for face detection and face recognition. It also includes code for exporting unoptimized models to the `XCORE.AI` chip.
- **audio**: Provides audio functions to do the audio preprocessing and to store the audio model.
- **utils**: Provides utility functions to support the application or inference, such as bounding box manipulation.

## 2.3 Integration

As mentioned earlier, each thread or group of threads is doing a specific task. Thread communication is done through channels. The definition of those channels and the initial configuration for of all threads are done in the `mapfile.xc` file. It corresponds to the entry point of the application. It is responsible for setting up the threads and channels. The following is the structure of the `mapfile.xc` file:

```
int main(void)
{
  // Channel declarations
  chan c_xflash[N_NETWORKS];    // flash server <> inference
  chan c_recognition;           // inference <> recognition post process
  chan c_inference;             // inference <> user app
  chan c_mic;                   // inferencer_rec <> mic task
  chan c_key;                   // audio <> user_audio

  // Initialize parallel tasks
  par{
    // flash server has to be tile 0
    on tile[FLASH_SERVER_TILE]: {
      flash_t headers[N_NETWORKS];
      flash_server(c_xflash, headers, N_NETWORKS, qspi, flash_spec, NFLASH_SPECS);
    }
    // tile 0 : control and inference
    on tile[0]: {unsafe {inferencer(c_inference, c_xflash[0], c_xflash[1], c_recognition);}}
    // tile 1 : user app and camera
    on tile[1]: inferencer_rec(c_recognition, c_key);
    on tile[1]: user_mic(c_mic);
    on tile[1]: user_audio(c_mic, c_key);
    on tile[1]: user_camera();
    on tile[1]: user_app(c_inference);
  }
  return 0;
}
```

Camera is tied to tile 1 due to physical constraints. Therefore, the camera thread runs on tile 1. The model, on the other hand, runs on tile 0 as the model arena size and post-processing are large enough to consume the whole tile. Since the model runs on tile 0, flash communication also operates on this tile. The remaining tasks, such as the high-level thread and vision and audio processing, are assigned to tile 1 due to its greater availability of resources.

## 2.4 The Features Database

The face recognition model is based on an Arcface Model (Arcface_Paper) with a MobileNetV2 backbone. Trained on a vast dataset of 10 million faces, this model can generate a 128-dimensional vector representing facial features. These features are optimized to minimize intraclass differences while maximizing interclass disparities, crucial for a robust face recognition model.

The subsequent steps involve collecting faces that we want to identify and running them through the model to generate the base features. For instance, if there are 10 individuals to detect, 10 images are collected and passed through the model, resulting in 10 arrays of 128 features. A Python script facilitates this process. For further details, refer to the `generate_features.py` script and the `generate_features.rst` in the `python` folder.

The mentioned script generates arrays in a standardized structure, intended for use by the `XCORE.AI` chip to compare features and identify individuals. The structure is as follows:

```
// Define the person_t struct
typedef struct {
    std::string name;
    std::vector<float> embeddings;
} person_t;
```

The `person_t` struct contains the name of the person and the 128-dimensional feature vector. Therefore, the database is an array of `person_t` structs. The `XCORE.AI` chip compares the features of the detected face with the features in the database to identify the person. It utilizes logic to threshold the cosine distance, and it also incorporates averaging of previous detections to enhance accuracy.

As an alternative to the previous method, it is also possible to add a new person to the database at runtime by saying the activation voice command `right`. If this happens the audio thread will send a signal to the vision postprocess thread to add the current face to the database.

```
            if (argmax != p_argmax && output_prob > OUTPUT_TH) {
                p_argmax = argmax;
                print_results_fast(argmax, output_prob);
                if (argmax == LABEL_ACTIVATION){
                    printstrln(">>>>>>>>>> Write");
                    chan_out_byte(c_key, LABEL_ACTIVATION); // to : face_rec_post_process_thread
                    led_blinkl_time_ms(1, 1000, 20);
                }
            }
```

If the voice command is different from the previous voice command and the score is higher than a given threshold (default to 0.7) the result is printed to the console. If the voice commands match the chosen command for adding someone to the database, then the information is sent through a channel to `face_det_post_process`.

## 2.5 Image Capture

Image capture involves acquiring images from the camera and sending them to the `XCORE.AI` chip for processing. The camera thread is responsible for this task. It captures images from the camera and sends them to the inference thread for processing. The camera thread is implemented in the *user_camera.xc* file. The following is the structure of the *user_camera.xc* file:

```
void user_camera()
{
  streaming chan c_pkt;
```

(continues on next page)

(continued from previous page)

```
  streaming chan c_ctrl;
  chan c_isp;
  chan c_control;
  delay_milliseconds(200);
  camera_mipi_init(
    p_mipi_clk,
    p_mipi_rxa,
    p_mipi_rxv,
    p_mipi_rxd,
    clk_mipi);

  par{
    MipiPacketRx(p_mipi_rxd, p_mipi_rxa, c_pkt, c_ctrl);
    mipi_packet_handler(c_pkt, c_ctrl, c_isp);
    isp_thread(c_isp, c_control);
    sensor_control(c_control);
    }
}
```

The camera is connected to the MIPI interface, the first thread is responsible for receiving packets from the camera. The second thread handles the packets and forwards them to the ISP. A third thread is dedicated to image processing, and a fourth thread manages sensor control. Users are not required to modify this code.

The high-level api `camera_capture` is used to capture images from the camera. This function is called in the `user_app.c` file. The capture involves by default a downsample, debayer, AWB and AE processes. The capture has to wait for the last image when the function is called, so it might take some time to return if the call is made at the start of frame.

```
  // init data
  const size_t img_data_size = H * W * CH * sizeof(int8_t);
  int8_t img_data[H][W][CH] = {{{0}}};
```

```
    // grab a frame
    //printstr("Requesting image...\n");
    assert(camera_capture_image(img_data) == 0);
    // printstr("Image captured\n");
```

## 2.6 Face Detection

Once we have the features database and we capture an image, the next step is to locate where the face is located in the image, if any. For this task, we use a Single Shot MultiBox Detector (SSD) model based on a YunNet model (Yunet_Paper). The output of the Yunet model, like other SSD models, typically consists of bounding boxes for detected objects, such as faces, and confidence scores for each detected object.

Each bounding box is represented by a set of coordinates that specify the location of the detected face in the input image. The confidence score indicates the model's certainty that the detected object is indeed a face.

The detection model and the recognition model share the same arena size in RAM as they are executed sequentially. This means that the detection model is executed first, followed by the recognition model. The `inferencer.cpp` file is responsible for running the inference on the `XCORE.AI` chip. Depending on what the main thread asks, it will run either the detection model or the recognition model. The following is the structure of the `inferencer.cpp` file:

```
      // ------------------ Detection ------------------
      // Init the model
```

(continues on next page)

(continued from previous page)

```
    model1_init((void *)flash1);

    // sizes
    const size_t input_size = model1_input_size(0);
    // const size_t output_size = model1_output_size(0);

    // Receive image size
    size_t chan_in_size = chan_in_word(c_inference);
    assert(chan_in_size == input_size);

    // Receive the image
    int8_t *inputs1 = (int8_t *)model1_input_ptr(0);
    chan_in_buf_byte(c_inference, (uint8_t *)inputs1, input_size);

    // Run the model
    model1_invoke_decorator();

    // Retrive output
    int8_t *conf_int8 = model1_output(0)->data.int8;
    int8_t *loc_int8 = model1_output(1)->data.int8;
    int8_t *iou_int8 = model1_output(2)->data.int8;

    // Retrieve quant params
    quant_params_t params_det[3];
    for (uint8_t i = 0; i < 3; i++)
    {
      params_det[i].scale = model1_output(i)->params.scale;
      params_det[i].zero_point = model1_output(i)->params.zero_point;
      params_det[i].out_size = (size_t)model1_output_size(i);
    }

    // Pass to the post process
    bbox_t result = face_det_post_process(
        conf_int8,
        loc_int8,
        iou_int8,
        params_det);

    // return the result
    model_chan_out_bbox(&result, c_inference);
```

The pipeline consists of setting the input, running the model and applying a post-process to the output.

The postprocess stage, implemented in the `postprocess_det.cpp` file, filters the bounding boxes generated by the model and returns the most accurate one. This unique box indicates the location of the face in the image along with its corresponding score.

The outputs of the model or the inputs to the post-process stage are the following:

- conf_int8 : Confidence score of the detected faces (normalized).
- loc_int8 : Bounding boxes of the detected face (normalized).
- iou_int8 : Intersection over Union of the detected faces (normalized).

The post-process stage will first dequantize each array, average the IOU and the confidence score, and then filter the bounding boxes applying Non-Maximum Suppression (NMS). The output of the post-process stage is a single bounding box that indicates where the face is located in the image and its score.

```
bbox_t face_det_post_process(
    int8_t *conf_int8,
    int8_t *loc_int8,
    int8_t *iou_int8,
```

(continues on next page)

(continued from previous page)

```
    const quant_params_t (&params)[3]
)
{
```

```
    // Get the scores
    get_scores(
        conf,
        iou,
        params[2].out_size);

    // Transform bboxes
    transform_priors_in_bboxes(
        loc,
        priors,
        scores,
        bboxes);

    // Do NMS
    nms(bboxes, nms_th, nms_iou_th);

    // Do some checks to filter bboxes
    filter_bboxes(bboxes);

    // Second box
    if (bboxes[0].score != -1){ // if the first box is valid
      second_box_process(bboxes[1].score);
    }

    // return the first bbox
    return bboxes[0];
```

## 2.7 Face Recognition

Face recognition is the process of identifying or verifying a person from a digital image or video frame. As mentioned earlier, the face recognition model is based on an Arcface Model (Arcface_Paper) with a MobileNetV2 backbone. This model generates a 128-dimensional vector representing facial features. These features are compared against the mentioned database to identify the person.

After a face is detected by the first model and post-processed, we may have a candidate (bounding box) face to be identified. The next step is to crop the image to extract only the face region. This is accomplished by a wrapper function that takes the bounding box as input and uses the `isp_crop` function from the `lib_camera` library.

The cropping process extracts the specified region of interest from the image, resulting in a new image containing only the face. Subsequently, this cropped face is resized to fit the input dimensions of the second model, which are 64x64x3 pixels. Again, this resizing is facilitated by a wrapper function, using the `isp_resize` function from the `lib_camera` library.

Below are both function wrappers:

```
void img_crop_int8(
  int8_t * img,
  const unsigned in_width,
  const unsigned in_height,
  bbox_t * bbox)
{
  unsigned xu1 = (unsigned)bbox[0].x1;
  unsigned yu1 = (unsigned)bbox[0].y1;
```

(continues on next page)

(continued from previous page)

```
  unsigned xu2 = (unsigned)bbox[0].x2;
  unsigned yu2 = (unsigned)bbox[0].y2;
  isp_crop_int8(img, in_width, in_height, xu1, yu1, xu2, yu2);
}

void img_bilinear_resize_int8(
  bbox_t * bbox,
  int8_t * img,
  const uint16_t out_width,
  const uint16_t out_height,
  int8_t * out_img)
{
  unsigned in_width = (unsigned)bbox[0].x2 - (unsigned)bbox[0].x1;
  unsigned in_height = (unsigned)bbox[0].y2 - (unsigned)bbox[0].y1;
  isp_resize_int8(img, in_width, in_height, out_img, out_width, out_height);
}
```

Then similarly to the first model, the pipeline consists of setting the input, running the model and applying a post-process to the output.

In this case, the stage that changes is the post-process, which takes the output and compares against the database. This operation is performed in the `postprocess_rec.cpp` file. For convenience this task is living in its own thread, to offload resources from tile 0.

The following is the main function of the `postprocess_rec.cpp` file:

```
void compare_against_database(
    std::vector<person_t>& database,
    std::vector<float> &output_float,
    rec_result_t &recognition_result)
{
    const float SIM_TH = 0.35;
    float max_similarity = -1.0;
    int max_index = 0;

    // initialize index to -1
    recognition_result.index = -1;

    static std::vector<float> means(database.size(), 0.0f);
    // char buff[32];
    //snprintf(buff, sizeof(buff), "database size: %d\n", database.size()); printstr(buff);
    //snprintf(buff, sizeof(buff), "output size: %d\n", output_float.size()); printstr(buff);
    for (unsigned person = 0; person < database.size(); person++)
    {

        // check size
        xassert(database[person].embeddings.size() == output_float.size() && \
        ERROR_MSG_SIZE_MISMATCH);

        // compute similarity
        float similarity = compute_cosine_similarity(
            database[person].embeddings.data(),
            output_float.data(),
            output_float.size());

        // filter
        means[person] = (means[person] + similarity) / 2.0f;
        similarity = means[person];

        // update max
        if (similarity > max_similarity)
```

(continues on next page)

(continued from previous page)

```
        {
            max_similarity = similarity;
            max_index = person;
        }

        #if PRINT_RESULT_COMPARISONS
        printf("Similarity with [%s]:\t%f\n",
                database[person].name.c_str(),
                similarity);
        #endif
    }

    if (max_similarity > SIM_TH){
        recognition_result.name = database[max_index].name;
        recognition_result.similarity = max_similarity;
        recognition_result.index = max_index;
        print_identified(recognition_result);
    }
    else{
        recognition_result.name = "unknown";
        recognition_result.similarity = 0.0f;
    }
}
```

This function takes the database and the output of the model, gets the cosine similarity and returns the name of the person that has the highest similarity. There is an average with the previous detection and a threshold to avoid false positives.

## 2.8 Keyword Spotting

Keyword spotting is a technique for speech applications, which enables users to activate devices by saying a keyword phrase. This allows users to interact with the device from a distance. There are different methods and approaches to this problem [kws_ref]. This application note uses a method that combines audio processing with a Neural Network to perform the keyword spotting.

The process is split into three stages:

1. **Audio Capture**: this involves taking audio samples from a microphone and sending them to the audio pipeline.
2. **Audio Preprocess**: in this stage, frames of samples are converted into features that the model can handle.
3. **Audio Network**: in this stage, these features are sent to the model which computes the distribution probability of a given set of keywords. Finally, a decision-making process is in place to initiate an action, when the scores align with our expectations for a certain keyword.

In the subsequent sections, we will detail each step of the audio pipeline.

**Audio Capture**

This stage involves setting up the microphone and capturing data from it. The `XCORE.AI Vision Development Kit` has two built-in microphones, although this application uses one microphone only. For more information on the microphones refer to the mic_datasheet. Those microphones are PDM microphones with good SNR (low self-noise) characteristics. PDM microphones are a kind of `digital microphone` that captures audio data as a stream of 1-bit samples at a very high sample rate. The high sample rate PDM stream is captured by the device, filtered and decimated to a 32-bit PCM audio stream.

The lib_mic_array library is used to perform this audio conversion operation. For doing so a wrapper file was made in order to set up the MIC configuration:

```
// ------------------- Port definitions --------------------
#define MIC_ARRAY_CONFIG_MCLK_FREQ            (24576000)
#define MIC_ARRAY_CONFIG_PDM_FREQ             (3072000) // will be 96 kHz after the first stage
#define MIC_ARRAY_CONFIG_MIC_COUNT            (1)
#define MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME    (320)
#define MIC_ARRAY_CONFIG_USE_DC_ELIMINATION   (1)
#define MIC_ARRAY_CONFIG_PORT_MCLK            XS1_PORT_1D // X0D11, J14 - Pin 15, '11'
#define MIC_ARRAY_CONFIG_PORT_PDM_CLK         PORT_MIC_CLK // X0D00, J14 - Pin 2, '00'
#define MIC_ARRAY_CONFIG_PORT_PDM_DATA        PORT_MIC_DATA // X0D14..X0D21 | J14 - Pin 3,5,12,14 and
↪Pin 6,7,10,11
#define MIC_ARRAY_CONFIG_CLOCK_BLOCK_A        XS1_CLKBLK_2
#define MIC_ARRAY_CONFIG_MCLK_DIVIDER  \
  ((MIC_ARRAY_CONFIG_MCLK_FREQ)/(MIC_ARRAY_CONFIG_PDM_FREQ))
// ----------------------------------------------------------
```

This set up the number of mics, the PDM frequency, and the samples per frame (in this case 320). The mic resource and the stage type are defined as follows:

```
pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_SDR(
                                MIC_ARRAY_CONFIG_PORT_MCLK,
                                MIC_ARRAY_CONFIG_PORT_PDM_CLK,
                                MIC_ARRAY_CONFIG_PORT_PDM_DATA,
                                MIC_ARRAY_CONFIG_CLOCK_BLOCK_A);

using TMicArray = mic_array::prefab::BasicMicArray<
                        MIC_ARRAY_CONFIG_MIC_COUNT,
                        MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME,
                        MIC_ARRAY_CONFIG_USE_DC_ELIMINATION>;

TMicArray mics;
```

Using the `prefab` example, the first stage converts samples to 96 kHz at 32-bit samples, and the second stage filters them to 16 kHz at 32-bit resolution.

Finally, a thread is in charge of creating this microphone instance and setting the channel to transmit the mentioned samples to the next thread. The PDM receiver is configured to not assert on dropped frames. Therefore, if packets are delayed, audio samples may be lost. This won't affect the current app since samples are received within time constraints, and any lost samples during keyword activation are not an issue.

```
void user_mic(chanend_t c_inference) {
    printf("Mic Init\n");
    device_pll_init();
    ma_wrapper_init();
    ma_wrapper_task(c_inference);
```

### Audio Preprocess

The 320 audio samples are sent by the `user_mic` thread and received by the `user_audio` thread. The audio is divided into overlapping frames of 512 samples each, which correspond to 32 ms at 16 kHz. Our network requires a 20 ms advance, translating to 320 samples at 16 kHz, with the remaining 192 samples coming from the old frame. Overlapping frames ensure that transient events, such as the beginning or the end of a keyword, are not missed. Buffers are managed to move the data from old to new, and provide space for new samples.

The next step is applying the Fast Fourier Transform (FFT) to convert the time-domain signal into the frequency domain. The result is a set of complex numbers representing the amplitude and phase of different frequency components in the frame.

Then, the frequency components from the FFT are mapped onto the mel scale using a set of triangular filters. This is known as the mel scale, which better represents human perception of sound. The Logarithm of Mel Spectrogram is applied, to compress the dynamic range of the mel-scaled frequencies.

After Log Mel, Discrete Cosine Transform (DCT) is computed to obtain the Mel-Frequency Cepstral Coefficients (MFCC). DCT transforms the log-mel features into a set of coefficients, where the first few coefficients capture the most significant information. MFCCs are compact and decorrelated features, making them well-suited for machine learning models.

```
                // receive 320 audio data
                chan_in_buf_word(c_mic, (uint32_t*)&tmp_buff[0], N_SAMPLES); // receive in buff_audio

                // Audio Process
                compute_mel_frame(mel_spec, tmp_buff); // Note: this moves audio buffer as well
                compute_cepstrum(features_int32_single, mel_spec);
                scale_cepstrum(features_int32_single, features_int8_single);
```

**Audio Network**

The MFCCs obtained from the previous stage are chopped to keep the first 10 features. Then 49 of them are stacked together. By providing a sequence of frames, the model can learn temporal dynamics and dependencies within the audio signal. At each new MFCC array, the last array is popped and the new is stacked to the features. This stacked feature set serves as the input to the neural network model, allowing it to capture both short-term and long-term patterns in the audio. The data is scaled to int8. Like this, we have finally our int8_t 49x10 input to the model. The features are then passed to the model and the inference is performed calling `model_invoke`.

```
                // Model Invoke
                model_invoke();

                // Get the output
                int8_t* outputs = (int8_t*)model_output_ptr(0);

                // Dequantize output
                dequantize_output(outputs, output_float, output_size, scale, zp);

                // Find the argmax
                unsigned argmax = argmax_float(output_float, output_size);
```

After that, the output is retrieved. The output corresponds to the softmax distribution of the set of keywords. The current set of keywords of this model are the following.

```
#define N_LABELS 12

const char *kws_labels[N_LABELS] = {
"Down",
"Go",
"Left",
"No",
"Off",
"On",
"Right",
"Stop",
"Up",
"Yes",
"Silence",
"Unknown",
};
```



# 3 Build and Run the Application

This section describes how to setup and run the current application.

This example can be run both only using `xrun` or with a `Python GUI` interface.

Please refer to the Application Note `AN02017` regarding Software and Hardware requirements, as well as how to get started and connect the `XCORE.AI Vision Development Kit` to the host computer.

Once, the XMOS tools environment is set, set up the Python environment by running the following commands from the top-level directory:

```
# Create a Python venv
python -m venv .venv
# Activate venv (Windows)
call .venv/Scripts/activate.bat
# Activate venv (Mac OS, Linux)
source .venv/bin/activate
# Install Python Requirements
cd an02013
pip install -r requirements.txt
```

In the same terminal, run the following commands to build the application and flash the models:

```
# build
cmake -G "Unix Makefiles" -B build
xmake -C build
# flash model weights
xflash --target-file XCORE-VISION-EXPLORER.xn --data src/model/xcore_flash_binary.out
```

The application can be run either using just the console:

```
# run (xrun option)
xrun --xscope bin/app_faceid_keyword.xe
```

or using the Python GUI:

```
# run (python GUI option)
python python/faceident.py
```

**Note:** In order to achieve optimal performance, the camera must be in a fixed position.

After running the application, the device will do the following:

1. Capture an image and process it.
2. Detect faces on the frame.
3. If a face is detected, see if the person is in the database.
4. If a second face is detected, the GUI is locked (over-the-shoulder protection).
5. In parallel, it will listen to different commands and interact with the vision system according to the selected command.

The recognised voice commands are the following:

1. `On` : starts the face ID demo.
2. `Off`: stops the vision systems.
3. `Right`: adds a new face to the database.

It is worth mentioning that the image never leaves the chip, all process is done locally, so we keep the privacy of the captured face. The device will only provide information about coordinates and names.

The terminal outputs the following information:

```
Image captured...
Box: [x1:10.832067,y1:60.337509,x2:54.573555,y2:109.986816], score: 1.360449}
Requesting image...
Identified: [alberto], similarity:0.542654
Label: Off, Prob: 78
```

The device provides information when an image is requested or captured, including the face location and a confidence score. The score, ranging from -1.5 to 1.5, reflects how confident the model is that the detected class is a face.

If the face is in the database, it prints the identified person with a similarity score. This similarity score goes from -1 to 1 and represents the cosine distance between the features in the database and the features of the current picture. While the image is being processed, audio is being processed in parallel to detect different commands, if a command is detected it will print the label and its corresponding probability.

The Python GUI is a frontend that displays this information in a more user-friendly manner. The GUI is split vertically into two sections. The right side is dedicated to face detection; if a face is in the frame, it will track the eyes. The left side represents the Face ID section; if a person is detected and identified, the frame turns blue, otherwise, it remains grey. As mentioned before, it is also possible to add a person who is not initially recognized by the system to the database using a voice command. To do this, the user can say the command `right`, and the person will be added to the database. The user will be assigned a random name from a list of animals, allowing them to know which name the device has assigned.

**Warning:** To exit the GUI, use the `ctrl+c` command in the terminal. Closing the GUI window will not stop the application.

An example of a detected person is presented below:

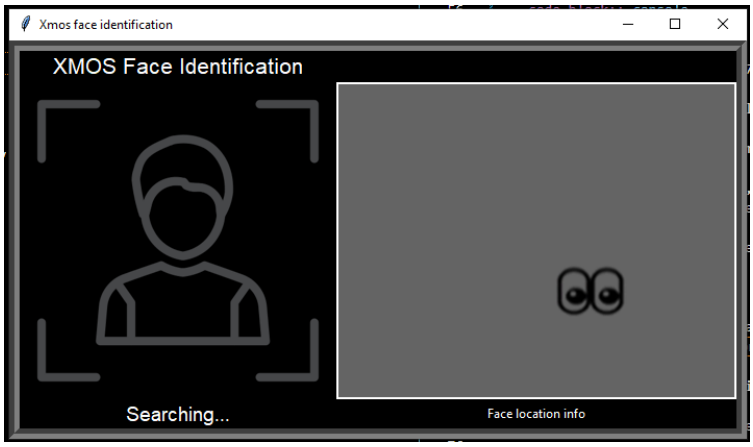


Fig. 3.1: Person Detected

An example of an identified person is presented below:

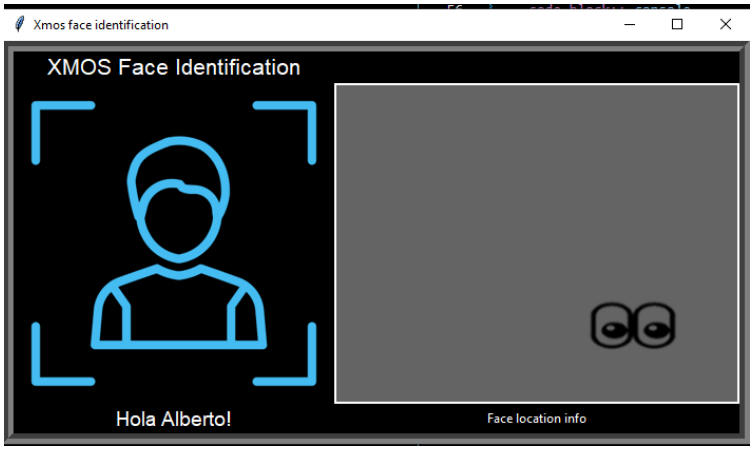


Fig. 3.2: Person Identified

An example of the GUI status after saying the command `Off` is presented below:


Fig. 3.3: Sleep Command

If during a detection, a second face is spotted in the same frame, the GUI will alert the user, as shown below:

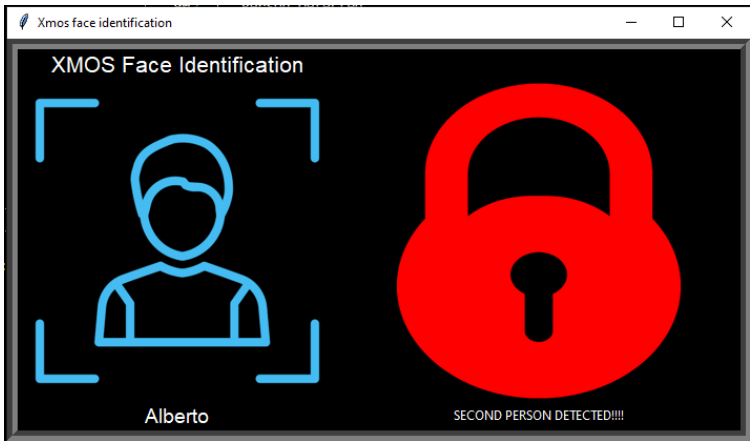


Fig. 3.4: Second Person Detected

In the following section, we will dive into the Software components of this application and its performance metrics.

# 4 Resource Usage

The XCORE.AI chip offers 8 threads per tile. Below is the application thread usage:

- Tile[0]: 6/8 threads used.
- Tile[1]: 8/8 threads used.

The memory usage of the application is as follows:

Table 4.1: Memory Usage of Current Application Note

| Tile N | Available [bytes] | Used [bytes] | Used [%] |
|---|---|---|---|
| Tile [0] | 524288 | 450588 | 86 |
| Tile [1] | 524288 | 441912 | 84 |

The application uses 450588 bytes of memory on tile 0 and 441912 bytes on tile 1. It is using then 80% of the chip's memory.

# 5 Performance

Here are the performance metrics of the application running on the XCORE.AI Vision Development Kit:

| Function | TICKS | MS | FPS |
|---|---|---|---|
| Total capture | 10721405 | 107.21 | 9.33 |
| Model A - detect. | 7352519 | 73.53 | 13.60 |
| Model B - recog. | 7440434 | 74.40 | 13.44 |
| Models | 14792953 | 147.93 | 6.76 |
| Others | 1043829.00 | 10.44 | 95.80 |
| Total | 26558187.0 | 265.58 | 3.77 |

We can see that the total time to capture an image, run the detection model, and run the recognition model is 265.58 ms, which corresponds to 3.77 FPS of global performance. The detection model takes 73.53 ms, while the recognition model takes 74.40 ms. The total time for both models is 147.93 ms, which corresponds to 6.76 FPS. The remaining time is spent on other tasks, such as UART communication and post-processing.

Depending on the model used the performance of the inference may vary. Here are the performance metrics of the application running on the XCORE.AI Vision Development Kit using different models:

Table 5.1: MobileNetV2 Comparison

| Model | Back-bone | Input (H,W,Ch) | Al-pha | Classes | Params (M) | Time (ms) | Model (KB) | Arena (Bytes) |
|---|---|---|---|---|---|---|---|---|
| A | Mb-NetV2 | 120,160,3 | 0.5 | 1000 | 1.987 | 96.12 | 2175 | 193240 |
| B | Mb-NetV2 | 120,160,3 | 0.5 | 10 | 0.719 | 69.9 | 960 | 191424 |
| C | Mb-NetV2 | 200,200,3 | 0.5 | 10 | 0.719 | 125.14 | 937 | 350056 |
| D | Mb-NetV2 | 120,160,3 | 0.75 | 10 | 1.395 | 116.19 | 1679 | 279872 |
| E | Mb-NetV2 | 120,160,3 | 1 | 10 | 2.271 | 136.96 | 2611 | 280336 |

The execution time of a model is not only determined by the number of parameters it has. Factors such as the number of operations, compiler optimizations, and conversion tuning impact on performance. For example, even though model C has fewer parameters than model A (0.7M vs 1.98M), it takes longer to execute (125 ms vs 96 ms) due to the input size and hence the number of convolution operations.

Here below is a zoom on the performance of the profiling of model C:

Table 5.2: Detailed Profiling of MobilenetV2 (a=0.5, i=200x200x3)

| N | Operation | Cumulative Time (ms) |
|---|---|---|
| **15** | **OP_XC_strided_slice** | **1.02ms** |
| 14 | OP_XC_pad_3_to_4 | 3.12ms |
| 63 | OP_XC_pad | 4.17ms |
| 59 | OP_XC_ld_flash | 16.01ms |
| 151 | OP_XC_conv2d_v2 | 98.32ms |
| 16 | OP_XC_add | 2.05ms |
| 2 | OP_CONCATENATION | 0.39ms |
| 1 | OP_RESHAPE | 0.00ms |
| 1 | OP_SOFTMAX | 0.06ms |
| | Total time invoke() | 125.14ms |

**Note:** The model uses 5 threads for the optimised layers. Optimised layers are the ones that start with `OP_XC_`. The rest of the operations are executed single-threaded. The parameter to set the number of threads is in the `--xcore-thread-count=5` parameter.

The most time-consuming operation is the convolution operation, which takes 98.32 ms. Note that is also the one that repeats the most (151 times). The second most time-consuming operation is the load from flash, which takes 16.01 ms. The rest of the operations are not significant in terms of total time.

# 6 References

- XMOS XTC Tools User Guide: XTC tools.
- XMOS XTC Tools Installing Guide: XTC Installing Guide.

# 7 Support

For all support issues please visit XMOS Support.