AN02010: XMOS Face Recognition Application Note

Release: 0.1.0
Publication Date: 2024/09/05
Document Number: XM-015121-AN

Table of Contents

1 Introduction

This application note provides a comprehensive guide to developing a face recognition solution for the XCORE.AI chip. It covers the entire process, from camera acquisition to output interface, specifically UART communication. The current note and provided code are designed to assist users in designing personalized vision solutions.

2 Face Recognition System Overview

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 feature faces to identify the person. The application will use the Cosine Distance to compare the arrays and identify the person.

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:

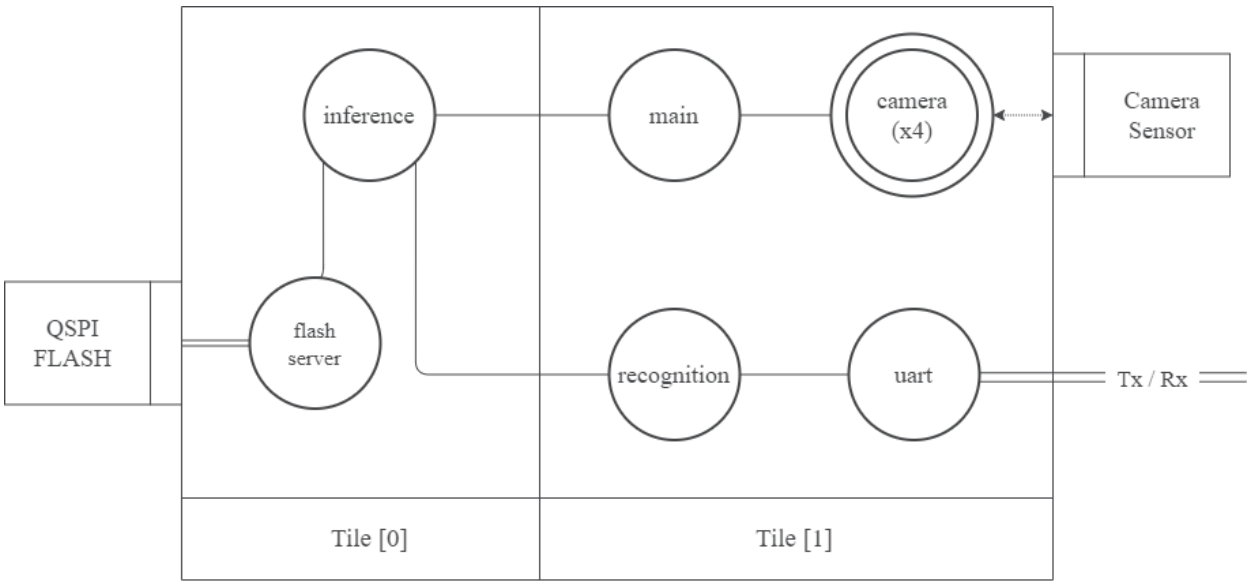

Fig. 2.1: Thread diagram

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_main.c
       |--- user_camera.xc
       |--- user_gpio.c
       |--- model
       |--- inference
       |--- uart
       |--- utils
```

- **src**: Contains the source code for the application.
- **mapfile.xc**: Entry point of the application.
- **user_main.c**: High-level thread that manages the application.
- **user_camera.xc**: Handles capturing images from the camera.
- **user_gpio.c**: Manages the GPIO communication between the `XCORE.AI` chip and the host.
- **inference**: Manages the 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.
- **uart**: Programs to manage the UART communication between the `XCORE.AI` chip and the host.
- **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_uart;                  // inferencer_rec <> user uart
  chan c_gpio;                  // inferencer_rec <> gpio task

  // 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
```

(continues on next page)

(continued from previous page)

```
    on tile[0]: {unsafe {inferencer(c_inference, c_xflash[0], c_xflash[1], c_recognition);}}
    // tile 1 : user app and camera
    on tile[1]: user_camera();
    on tile[1]: user_app(c_inference);
    on tile[1]: inferencer_rec(c_recognition, c_uart, c_gpio);
    on tile[1]: user_uart(c_uart);
    on tile[1]: user_gpio(c_gpio);
  }
  return 0;
}
```

GPIO and Camera are tied to tile 1 due to physical constraints. Therefore, the camera thread runs on tile 1. On the other hand, the model arena size and post-processing are large enough to fill a tile, leading to their placement on tile 0. Since the model runs on tile 0, flash communication also operates on this tile. The remaining tasks, such as the high-level thread, UART communication, and post-process thread, are assigned to tile 1 due to its greater availability of resources.

2.4 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 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 pressing the hardware button on the vision board. When this button is pressed, the `user_gpio` thread detects the event. It then communicates with the `face_recognition_thread` to include the new person in the database for future detections. Once the person is successfully added, the `face_recognition_thread` notifies the gpio_thread. As a visual confirmation, the `gpio_thread` makes an LED blink and turn green for 2 seconds. If a person is added using this method, they will be named *person_*, followed by the current size of the database.

The following is the main logic of the `user_gpio.c` program:

```
    SELECT_RES(
        CASE_THEN(button_port, on_button_change))
    {
```

(continues on next page)

(continued from previous page)

```
        on_button_change: {
            p_button_state = button_state;
            button_state = port_in(button_port);

            // rising edge
            rising_edge = p_button_state == BTN_NO_PRESSED && button_state == BTN_PRESSED;
            if (rising_edge) {
                // if a button is pressed, inform the inference thread
                chan_out_byte(c_gpio, 1);
                // wait for response
                if (chan_in_byte(c_gpio)){
                    gpio_led_activity();
                }
            }

            // here reset the condition before leaving
            port_set_trigger_in_not_equal(button_port, button_state);
            SELECT_CONTINUE_NO_RESET;
```

When we detect a rising edge in the button, we send the information, and once we receive the response, we perform the blink sequence.

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;
  streaming chan c_ctrl;
  chan c_isp;
  chan c_control;

  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, while 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_main.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 captured 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
    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();

    // Retrieve 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);
```

(continues on next page)

(continued from previous page)

```
    }

    // 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,
    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);

    // 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;
  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;
```

(continues on next page)

(continued from previous page)

```
    // 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)
        {
            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.

The result is stored in the `result_t` struct, which is then sent to the main thread to be displayed on the UART interface.

```
// Define the rec_result_t struct
```

(continues on next page)

(continued from previous page)

```
typedef struct {
    std::string name;
    int index;
    float similarity;
} rec_result_t;

// Database definition
extern std::vector<person_t> database;
```

2.8 UART Communication

UART stands for Universal Asynchronous Receiver Transmitter, which is a protocol that allows serial communication between devices. It is commonly used for transmitting and receiving data between microcontrollers, sensors, and other peripheral devices.

The UART task consists of two main parts:

1. the device thread, that waits to receive data and sends it through UART. The data sent is the person identified by the system.
2. the host app, that receives and prints in the console the detected person, which could enable for example another board to start a process of opening a gate or personalizing information for that user.

Both programs are located in the `uart` folder, the device program is called `user_uart.cpp` with its headers, and the host program is called `host.py`.

Note: If the user needs to update any UART initialization values, such as baud rate and stop bit, the same values must be used on the device and host side.

Here below the device side:

```
static void user_uart_cpp(chanend_t c_uart) {
        uart_tx_t uart;
        port_t p_uart_tx = USER_UART_PORT;
        hwtimer_t tmr = hwtimer_alloc();

        // Initialize the UART Tx
        uart_tx_blocking_init(&uart, p_uart_tx, USER_BAUD_RATE, 8, UART_PARITY_NONE, 1, tmr);

        // Send hello message
        send_string_uart(&uart, "Hello from user_uart\n");

        // Receive from app and send to uart
        while (1) {
                // convert index in name
                uint8_t person_index = chan_in_byte(c_uart);
                std::string name = database[person_index].name;
                send_string_uart(&uart, name);
        }
        xassert(0 && "This should never be reached");
```

And here the host side:

```
import serial
from time import sleep

print("Initializing serial communication...")
```

(continues on next page)

(continued from previous page)

```
try:
    ser = serial.Serial("/dev/ttyS0", timeout=1)  # Open port with baud rate and timeout
    ser.baudrate = 115200  # Set Baud rate
    ser.bytesize = 8  # Number of data bits = 8
    ser.parity = serial.PARITY_NONE  # No parity
    ser.stopbits = 1  # Number of Stop bits = 1
    print("Serial communication initialized.")

    while True:
        received_data = ser.read_all()  # read all available serial data
        if received_data:
            data = received_data.decode('utf-8')
            print("RX:", data)  # print received data as UTF-8 decoded string
        sleep(0.1)

except serial.SerialException as e:
    print("Serial communication error:", e)

finally:
    if ser.is_open:
        ser.close()  # Close serial port
        print("Serial port closed.")
```

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 an02010
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_face_identification_uart.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.

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
```

The device provides information when an image is requested or captured, including the face location and a 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.

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.

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:

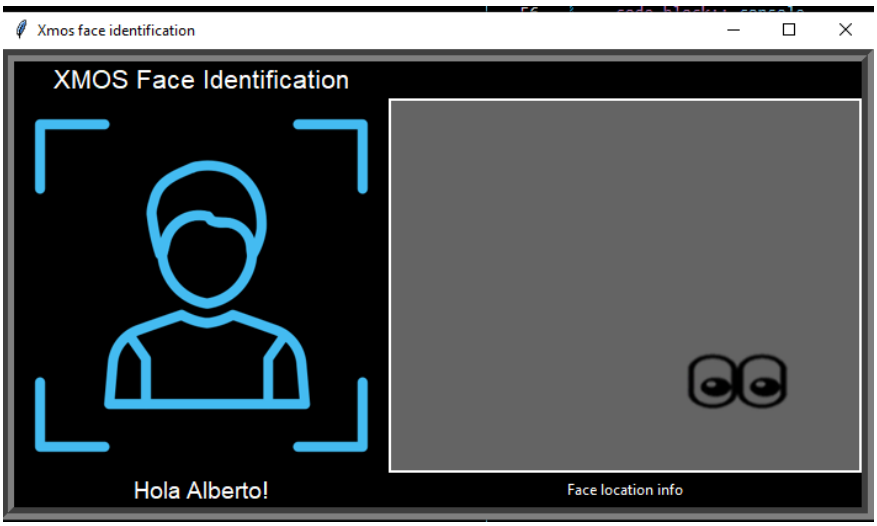

Fig. 3.2: Person Identified

4 Resource Usage

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

- Tile[0]: 2/8 threads used.
- Tile[1]: 7/8 threads used.

Memory Usage: 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	465412	89
Tile [1]	524288	149384	29

The application uses 465412 bytes of memory on tile 0 and 149384 bytes on tile 1. That means that the model is almost taking all the memory of tile[0], and the application is taking a small part of the memory of tile[1].

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 (125ms vs 96ms) 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.