AN02005: XMOS Logo Detector Application Note

Release: 0.1.0
Publication Date: 2024/09/14
Document Number: XM-015116-AN

Table of Contents

1 Introduction **1**

2 XMOS Logo Detector Overview **2**

2.1 Thread Diagram 3
2.2 Software Application Structure 3
2.3 Model Preparation 4
2.4 Top level Main 5
2.5 Main Program 5
2.6 User Inferencing Thread 6
2.7 Camera Thread 7
2.8 GPIO Thread 7

3 Building and Running the Application **9**

4 Resource Usage **10**

5 Performance **11**

6 References **12**

7 Support **13**

1 Introduction

This application note provides an example of how to connect a camera, a small ai-model and a GPIO on an XCORE.AI processor. The model is purposely small in order to show how a model can run efficiently using just RAM memory.

The model is a multi-stage CNN with a sigmoid output that can recognize the XMOS logo. The model is trained on a small dataset of images of the XMOS logo. The model is trained using the TensorFlow Lite framework and is converted to an XCORE optimised model using the xmos_ai_tools.

The application illuminates a green LED if the logo is detected. If the logo is not detected, the LED is red. The application also prints the probability of the logo being seen (as output by the sigmoid) if connected via `DEBUG` Micro USB port to a host.

2 XMOS Logo Detector Overview

The structure of the application follows a multi-threading approach where each thread is focused on doing a specific task. Threads communicate through channels. The threads are distributed across two tiles (each has its own 512 kByte of memory): the NN model executes on tile[0], and the main program and camera interface execute on tile[1]. Allocating a whole tile to the NN model enables us to run the biggest model possible. Hence, we have allocated a specific tile to the model and run all other components on the other tile.

One thread is dedicated to running the main program that interacts with the camera, GPIO and inferencer. This thread asks the camera thread for the latest image available and subsequently sends that image to the NN model. The NN-model thread receives the image, and runs an inference on the image, whereupon it sends the result back to the main program. The main thread then calls the GPIO thread to turn on the LED if the logo is detected. This partitioning of tasks allows the NN model to execute in parallel with the main program and camera library, so the main program can get the latest image whilst the model is running an inference.

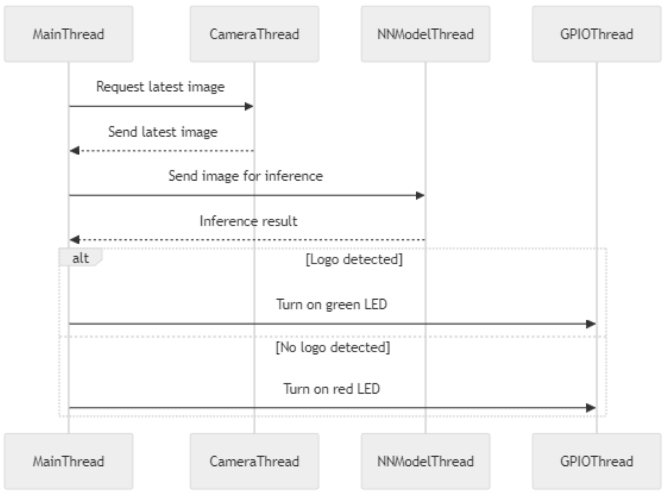

Fig. 2.1: High-Level Sequence Diagram

In the next section, the structure of the application and how the model is imported into the application will be discussed.

2.1 Thread Diagram

The following shows the thread diagram of the application.

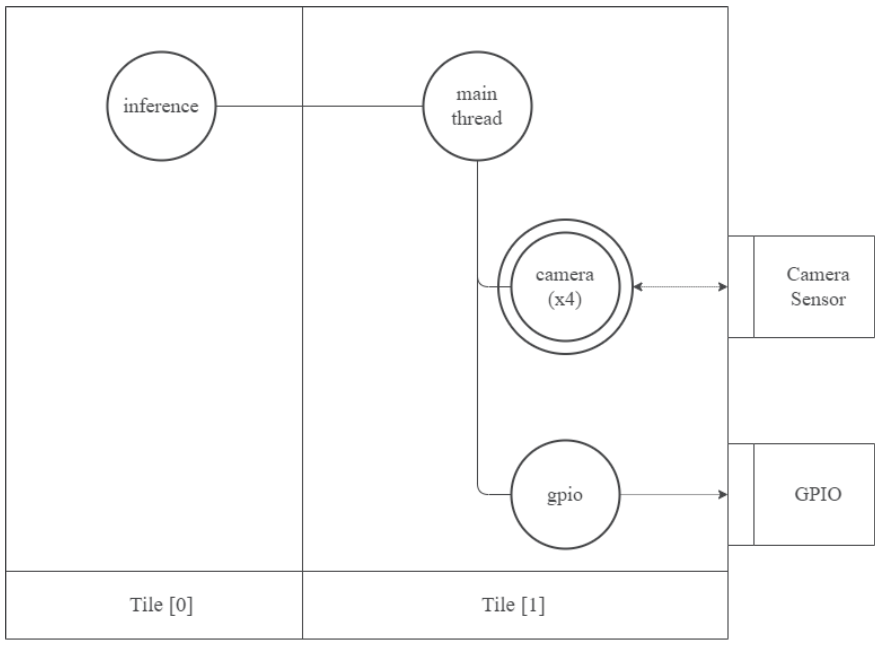

Fig. 2.2: Thread diagram

This application uses two tiles. Inferencing takes place on tile[0]. Tile[1] includes the main, camera and GPIO threads. The user application tasks the camera thread to take a picture, sends it to the inference thread, and instructs the GPIO task to control the LED.

2.2 Software Application Structure

The application source code is stored in the `src` folder, while the model and model scripts are located in the `model` folder. The `src` folder contains all necessary source files to run the application, which is mostly code that can be modified by the user to achieve a different behavior before or after the inference. The `model` folder contains the model and all the utilities to run the model. This code is generated and is not usually modified by the user.

The following is the folder structure of the application:

```
repository
  |
  |--- model
    |--- model_unoptimized.tflite
    |--- convert.py
    |--- convert.yaml
    |--- model.utils.cpp
  |--- src
    |--- mapfile.xc
    |--- user_main.c
    |--- user_inference.cpp
    |--- user_camera.xc
    |--- user_gpio.c
 |--- CMakeLists.txt
```

2.3 Model Preparation

This section contains the model and the utilities to export the model. The model is trained using the Tflite_micro framework, and is converted to an XCORE optimised model using the xmos_ai_tools.

Unoptimized model

This is the TensorFlow Lite model after int8 post-training-quantization. This model is not optimized for the XCORE architecture and will be converted to an XCORE model in the next stage.

For more information regarding model quantization, please refer to TensorFlow post-training quantization documentation tf-pt-quantization.

Model export

In order to convert the unoptimised TensorFlow Lite model to an XCORE model, the `xcore-opt` tool will be used. To simplify the conversion a Python script is provided (`convert.py`). This script takes the unoptimized TensorFlow Lite model and optimizes it to run on an XCORE. This script reads a configuration file `convert.yaml`, and generates a `.cpp` file that will be compiled by the tool chain to provide an inference function to the programmer. The script also produces the headers for the model and the optimised TensorFlow Lite model.

Model utils

These are utilities for both treating the input and output of the model, and checking the size of both the input and output tensor of the model.

This example collects all output information of the model in different formats. It encapsulates this output in the following structure.

```
// Struct containing all the outputs of the model
typedef struct
{
    int8_t  output_int8;
    float   output_float;
    uint8_t output_label; // this one is computed
    float output_probability; // this one is computed
} model_output_t;
```

The `int8_t` output does not have a meaningful value by itself, it is the output of the model. The final layer of the model utilizes a sigmoid function, which is designed to map the probabilities of a binary class into a range between 0.0 and 1.0 (`float32` output). The conversion between int and float is done by a dequantization process. The label ID is the class that the model predicts the input belongs to. The probability output reports the probability of the input belonging to the class. The calculation of the probability is done using a threshold value, which is 0.5 by default.

For example if the model produces a sigmoid output of 0.039, this means the model estimates a 3.9% chance of the input belonging to class 1 and a 96.1% chance of it belonging to class 0. Given a threshold of 0.5 for classification, we calculate the deviation as 0.5 minus the sigmoid output. Thus, the deviation is 0.5 - 0.039 = 0.461. The Certainty Percentage is then computed as (0.461 / 0.5) × 100% = 92%. This indicates that the model is 92% confident that the input belongs to class 0.

This information is collected in a single structure so that all the data can be easily transmitted between threads.

```
void model_chan_out_result(
  model_output_t* output,
  streaming_chanend_t c_inference) {
  s_chan_out_buf_byte(c_inference, (uint8_t*)output, sizeof(model_output_t));
}

model_output_t model_chan_in_result(
  streaming_chanend_t c_inference) {
  model_output_t result;
  s_chan_in_buf_byte(c_inference, (uint8_t*)&result, sizeof(model_output_t));
```

(continues on next page)

(continued from previous page)

```
  return result;
}
```

To ensure structured and controlled communication of information across cores, a similar approach is taken for the model size, even though it may seem excessive. This approach serves to illustrate a general method for handling information in a systematic manner.

2.4 Top level Main

The top level main program is stored in a file `mapfile.xc` which serves as the entry point of the application. It declares the channels and starts the threads. As stated earlier, the model runs on tile[0], and the high-level application runs on tile[1].

```
int main(void)
{
  // Channel declarations
  streaming chan c_inference;
  chan c_gpio;

  par{
    on tile[0]: user_inference(c_inference);    // Run the model on tile[0]
    on tile[1]: user_gpio(c_gpio);              // Run the GPIO on tile[1]
    on tile[1]: user_camera();                  // Run the camera on tile[1]
    on tile[1]: user_main(c_inference, c_gpio);  // Run the high level app on tile[1]
  }
  return 0;
}
```

In this code there are 2 channels declared:

- c_inference: This streaming channel is used to send the image to the model and receive the result. The use of a streaming channel allows the model to continue processing the data without waiting for the entire image to be received. This approach prioritizes faster processing over waiting for data transmission.
- c_gpio: This channel is used by the high-level application to send the LED value to the GPIO thread.

Now each thread will be explained in more detail.

2.5 Main Program

The main program is stored in `user_main.c` This is the previously mentioned high-level API, that will interact with the camera and the GPIO.

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

  // get model size (input/output)
  model_size_check_t model_size = model_chan_in_size(c_inference);
  size_print(&model_size);

  // assert all sizes match
  assert(model_size.input_size == image_size);
  assert(model_size.output_size == EXPECTED_OUTPUT_SIZE);
```

(continues on next page)

(continued from previous page)

```
// initialize camera
camera_init();
```

The code above allocates memory for the image first, it gets the model size, and checks if the input size matches the image size. It finally initializes the camera.

```
{
  // grab a frame
  printf("Requesting image...\n");
  assert(camera_capture_image(img_data) == 0);

  // Update LED state
  output_status = led_values[result.output_label];

  // set output
  chan_out_word(c_gpio, output_status);
```

This part will capture an image from the camera, and send it to the model. After computing the result, it will send the result to the GPIO thread.

2.6 User Inferencing Thread

The code for the user inferencing thread is stored in `user_inference.cpp`

This thread is responsible for running the NN model. The code snippet below represents a while loop that waits for an image to be sent to the model and then runs the model. After running the model, it computes the result, and sends it back to the high-level application.

```
// model initialization
printf("Model init\n");
model_init((void *)nullptr);
while(1)
{
  if(s_chan_in_byte(c_inference))
  {
    // Receive the image
    s_chan_in_buf_byte(c_inference, (uint8_t *)data_ptr, model_size.input_size);

    // Run the model
    printf("Model invoke\n");
    model_invoke();

    // Compute the output
    TfLiteTensor* output = model_output(0);
    model_output_t result = model_output_map_binary(output, 0.5);

    // Return the result
    model_chan_out_result(&result, c_inference);
```

Note that the model is initialized before the while loop. The model is initialized with a `nullptr` because the model does not need to read the weights from flash. Besides, `data_ptr` is the pointer to the input tensor of the model. The `model_invoke()` function will run the model, and store the result in the output tensor.

2.7 Camera Thread

This thread will be responsible for capturing the image from the camera, and sending it to the high-level application. It uses 4 threads.

```
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, so the first thread will receive the packets from the camera. The second thread will handle the packets and send them to the ISP. The third thread will process the image, and the fourth thread will control the sensor. The user does not need to modify this code.

2.8 GPIO Thread

This thread will be responsible for turning on the LED if the logo is detected. The SELECT_RES block will wait for a value to be sent from the high-level application, and then turn on the LED according to the received value.

```
void user_gpio(chanend_t c_gpio)
{
  // PORT_LED and PORT_BUTTON are defined in the xn file
  port_t p_leds = PORT_LED;
  uint32_t value = 0;
  port_enable(p_leds);

  SELECT_RES(
    CASE_THEN(c_gpio, led_handler))
  {
    led_handler:
      value = chan_in_word(c_gpio);
      printf("Received value: %lu\n", value);
      port_out(p_leds, value);
      continue;
  }
}
```

If there is some change in the channel c_gpio, the value will be read, and the LED will be turned on according to the value. The port corresponding to the LED is defined in the XCORE-VISION-EXPLORER.xn file. The port_out will output the value to the mentioned port.

Note: In this board, PORT_LED is a 32-bit port, so the value will be a 32-bit value. If the user is using different peripherals, the value may be different. The recommended way in this case would be to read/peek first, modify the needed value, and then write the port modified values.

3 Building and Running the Application

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

Please refer to the Application Note `AN02017` regarding how to get started using the `XCORE.AI Vision Development Kit` and install the required software before proceeding to build the application.

Once, the environment is set, run the following command from the top-level directory:

```
# python reqs
# note: create and activate venv first
cd an02005
pip install -r requirements.txt
# build
cmake -G "Unix Makefiles" -B build
xmake -C build
# run
xrun --xscope bin/app_xmos_logo_camera.xe
# (Optional) The application can also be flashed into memory
xflash --target-file XCORE-VISION-EXPLORER.xn bin/app_xmos_logo_camera.xe
```

After running the demo, the camera will start capturing images and sending them to the model. If the model detects the XMOS logo, the LED will turn green, and the probability of the detection will be printed.

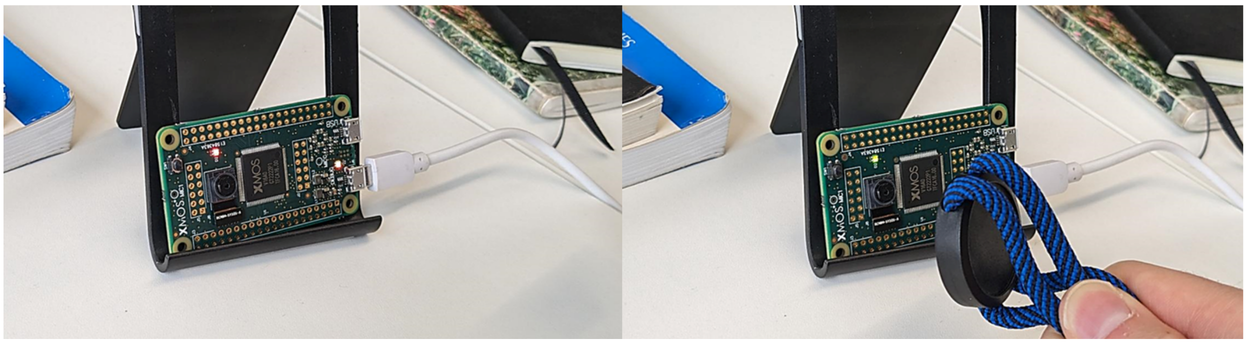

Fig. 3.1: Demo output for the `XCORE.AI Vision Development Kit`, when the logo is shown, built-in LED turns green.

Console Output example:

```
------------------------------
Model results:
Raw (int8): -118
Dequant sigmoid prob: 0.039062
Label ID: 0
Probability: 0.92
------------------------------
```

As mentioned above, when connected to a host, the device prints further information regarding the detection. It prints the raw output (int8), the dequantized sigmoid probability (float), the label ID, and the output probability. The printed data match the structure of the `model_output_t` discussed in the *model utils section*, and a more detailed explanation of them can be found in the same section.

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

4 Resource Usage

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

- Tile[0]: 1/8 threads used.
- Tile[1]: 6/8 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	476556	91
Tile [1]	524288	111744	21

As we can see the model is using *476556* bytes of memory, and the application is using *111744* bytes of memory. 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 below is a table with the performance of the camera and the model running on the `XCORE.AI Vision Development Kit`.

Function	TICKS	MS	FPS
Capture+ISP	16064737.00	160.65	6.22
Model+Dequant	1643829.00	16.44	60.83
Others	240073.00	2.40	416.54
Total elapsed	17964394.00	179.64	5.57

We can see that the model is taking only 16.44 ms to run. The camera takes 164.78 ms to capture, process and send the image. The total time is 183.62 ms, which means that the whole application runs at 5.45 FPS.

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.