Optimizing for the OV7670 camera

Fabián Torres Álvarez, Karthik Ganesan, Prof. Bruno Korst - bkf@comm.utoronto.ca

Introduction

In this lab, you will be optimizing the Nucleo + OV7670 setup from Lab 5 to increase the frames per second (FPS) of the video stream sent from the Nucleo board to the computer. At the end of this lab document, you will find a list of items that you must show your TA when being marked during your in-lab demo.

1. Hardware Set Up

First, connect the OV7670 module to the Nucleo board in the same way you did for Lab 5. Using jumper cables, make the connections shown in Figures 1 and 2. The connections are also listed in the Table.

1.1 Sending Images Through Serial Port

When working with images, it is important to have a way to visualize the results. In this lab you will use the serial port to transmit image data from the Nucleo board to your computer, and visualize the images with the `serial_monitor_lab_06` program provided with this hand out. **NOTE:** This is not the same `serial_monitor` program from Lab 05; this version has extra functionality that we will use in this lab.
First, download `serial_monitor_lab_06.zip`, extract it and make sure you have permissions to execute the `.exe` file inside. Open a command line interface and run `.\serial_monitor_2.exe --help`. You should see the following:

```
> .\serial_monitor_lab_06.exe --help
Usage: serial_monitor_lab_06.exe [OPTIONS]

  Display images transferred through serial port. Press 'q' to close.

Options:
  -p, --port TEXT           Serial (COM) port of the target board
  -br, --baudrate INTEGER   Serial port baudrate
  --timeout INTEGER         Serial port timeout
  --rows INTEGER            Number of rows in the image
  --cols INTEGER            Number of columns in the image
  --preamble TEXT           Preamble string before the frame
  --delta_preamble TEXT     Preamble before a delta update during video
                            compression.
  --suffix TEXT             Suffix string after receiving the frame
  --short-input             If true, input is a stream of 4b values
  --rle                     Run-Length Encoding
  --help                    Show this message and exit.
```

Verify that you can run the new `serial_monitor` in a similar way to Lab 5. On your computer, run:

```
    .\serial_monitor.exe -p <Nucleo board COM port>
```

Use Keil μVision to send through serial port:

- String `"\r\n!START!\r\n"`.
- The character `0x00` 25,056 times (144×174).
- String `"!END!\r\n"`.

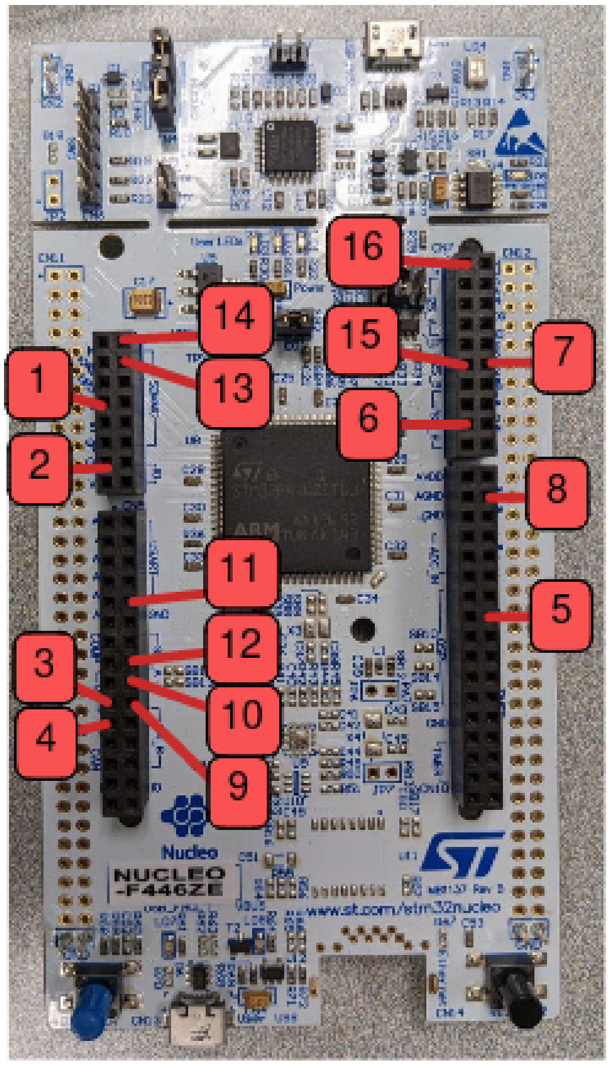

Figure 1

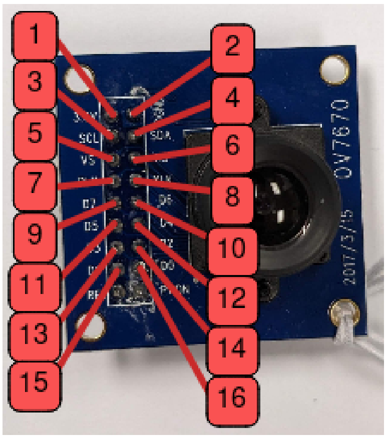

Figure 2

I/O Number	Camera I/O	I/O Label on Nucleo
1	3.3V	3V3
2	DGND	GND
3	SCL	I2C (top)
4	SDA	I2C (bottom)
5	VS	RX DO
6	HS	SPI_B (2nd from bottom)
7	PLK	D12
8	XLK	D6
9	D7	SAI_A (1st from bottom)
10	D6	SAI_A (2nd from bottom)
11	D5	USART (1st from bottom)
12	D4	SAI_A (2nd from top)
13	D3	SDMMC (2nd from top)
14	D2	SDMMC (1st from top)
15	D1	I2S_B (bottom)
16	D0	I2S_A (top)

Note the addition of the closing string `"!END!\r\n"`. This is needed because in this lab you will work with frames of varying byte sizes, and you need a specific byte sequence for `serial_monitor_lab_06` to identify the end of a transmission.
If your implementation is correct, a graphical window will open showing a black rectangle. Next, modify your code to output character `0xFF` instead of `0x00`. The `serial_monitor` should show a completely white rectangle. You may experiment with shades of gray, or alternating patterns.
When you can reliably send and visualize artificially generated images, you may proceed to the next section.

2. Baseline

Use your implementation from Lab 5 as a baseline. Modify it to start all frame transmissions with `"\r\n!START!\r\n"` and end them with `"!END!\r\n"`. Run `serial_monitor_lab_06` for at least 2 minutes and fill in in Table 1 with the typical, minimum and maximum FPS reported by the tool end of this handout. You may ignore the first few frames until reaching a steady state.

3. UART with DMA

One of the limitations with the Lab 5 implementation was that transmitting entire frames over the Serial Port is very time consuming. Function `HAL_UART_Transmit()` is a blocking function; the CPU does not move past that function until all data has been transmitted. As a result, the UART hardware sits idle for all the cycles that the main execution thread is not

running `HAL_UART_Transmit()`. We can improve this by using DMA transmit data through UART with non-blocking functions.
Modify your code to use `HAL_UART_Transmit_DMA()` instead. The signature of this function is the same as the blocking version, but no timeout parameter is needed. Note that the DMA transmit function can support much longer buffers than its blocking counterpart, and you can send an entire frame in a single function-call. Once you had modified your code, measure the FPS you get with `serial_monitor` and enter your results in Table 1.
You may check if the UART is free to begin a transmission with one of the following:

- By calling `HAL_UART_GetState(&huart3)` and checking that the result is `HAL_UART_STATE_READY`.
- From inside the IRQ handler with macro `__HAL_UART_GET_IT_SOURCE(&huart3, UART_IT_TC)`.

Try it yourself: When using UART through DMA, interrupts are triggered when a transmission is half-complete and complete. The interrupts are pre-configured to call specific callback functions if implemented by the user. Implement the following on main.c to pre-compute the next half frame as the current one is being sent:

- void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart)
- void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)

4. Compressing the data

Another way for us to increase FPS is to *compress* the data we are sending. Compression is a technique for reducing the size of data by removing redundant information. Images are highly amenable to compression as image-data contains significant amount of redundant information. Compression techniques belong to two types: 1) lossless compression, where all the information is retained but we reduce data size by removing only redundant information and 2) lossy compression, where we throw away some less-important data to reduce size. In this lab, you will implement one of each type of compression: *run-length encoding* (a lossless technique) and *data truncation* (a lossy technique).

4.1 Run-Length Encoding

Run-Length Encoding (RLE) replaces repeated copies of the same value in the data as a single value and a count. For example, the string: `AAAAAAAABBACCCAA` is represented in RLE as `A8B2A1C3A2`. Note that RLE does not always result in a shorter string than the original, for example the string `ABC` is encoded as `A1B1C1` which is twice as long as the original.
RLE is well suited for certain types of images such as computer icons, where adjacent pixels are likely to share the same shade or colour. However, RLE does not perform well for continuous tone images such as photographs coming from the OV7670 camera. This is because even though adjacent pixels might seem to be the same color, they differ slightly in the actual 8-bit pixel value, which limits the efficacy of RLE. However, even for frames coming from the camera, the upper-bits of each pixel are more likely to be the same between adjacent pixels, compared to the lower-bits. Therefore, to improve the effectiveness of RLE, we employ the lossy compression technique of *data truncation*.

4.2 Data truncation

We truncate the frame received from the camera by discarding the 4-least significant bits (LSBs) from each pixel. Doing so limits the resolution of our image (i.e., from $2^8 = 256$ to just $2^4 = 16$ values per pixel), but produces larger areas of the frame that have the same value (such as the background). Modify your code to drop the 4 LSBs from each pixel, and pack them so that each byte transmitted through UART contains the information of two pixels (one pixel in the 4 MSBs, one pixel in the 4 LSBs) as shown on Fig. 3.
Run the serial monitor with the flag `--short-input` to view an image streamed using this format. Measure FPS and fill in your results in Table 1. If your implementation is correct, the image will be coarser, but frames per second will roughly double.

4.3 Combining techniques

Modify your code to implement RLE, and send the encoded buffer through serial port. Each byte transmitted through serial port will consist of a pixel value in the 4 most-significant bits, and the number of consecutive pixels with the same value in the 4 least-significant bits as shown on Fig. 4. As we only use 4 bits to keep count of the number of repeated pixels, a sequence of repeated pixels longer than 15 should be sent as a separate byte.
Run the serial monitor with the flags `--short-input --rle` to view an image stream this format. Measure FPS and fill in your results in Table 1. A correct implementation should be able to achieve FPS higher than 1.0 in the typical case.

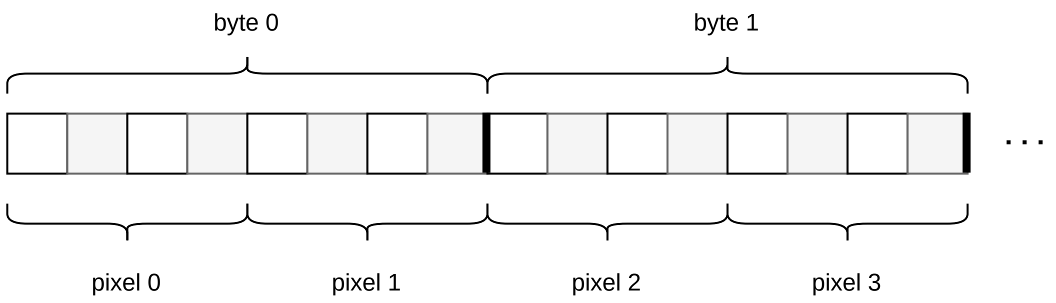

Figure 3. 4-bit pixels diagram.

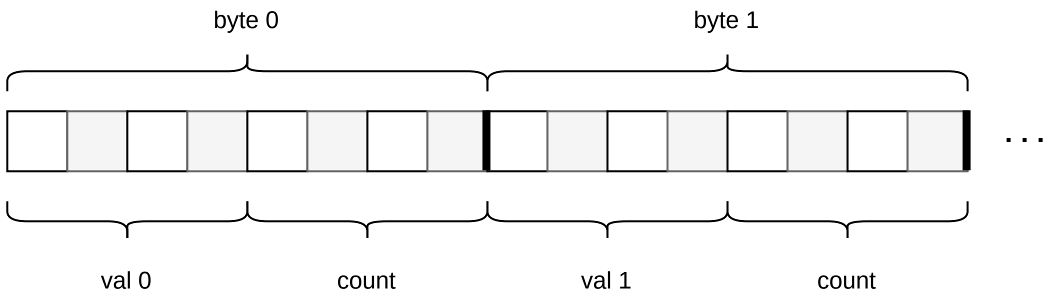

Figure 4. 4-bit pixels RLE diagram.

5. Video Compression Algorithm

So far you have focused on optimizing the transmission of individual frames, and you have taken advantage of the spatial correlation between pixels. Because the frames are part of a video, there is one remaining dimension we use for compression: time.
Pixels in consecutive frames of a video are typically highly correlated. In other words, we expect pixels in one place in one frame (i.e., frame A) to be the same or very similar to the same pixels in the next frame (i.e., frame B). We can take advantage of this to send a smaller transmission with the changes instead of full frames. If we calculate the per-pixel differences between frames A and B, we expect to get a lot of 0s. This makes this difference much more efficient to compress using RLE.
Implement a function to calculate the difference for every pixel between the current frame and the one before it. You must still use data truncation to only consider the 4 most significant bits for each pixel. Once you have this array of differences, compress it using RLE and send the result through the serial port. When sending just the differences, start the transmission with `"\r\n!DELTA!\r\n"` instead of `"\r\n!START!\r\n"` to tell the serial monitor that this transmission contains a partial update of a frame. You cannot keep sending just differences as doing this for too many frames will lead to a major drop in quality. Therefore, once every N frames, you should send a full frame. For $N = 5$, measure the FPS you achieve and fill in your results in Table 1.
Observe how the frame rate changes if you capture a moving object from a static image. Why do you think this happens? You should be ready to answer questions from your TA about this behaviour if asked.

Try it yourself: How many partial frames can you send before you notice that your video quality starts degrading? How much does the content of the video (i.e., static video vs. video with lots of movement) affect this number N? Can you think of a way to automatically calculate N based on the calculate differences?

6. (Optional) Measuring Power

So far, we have not looked at some important aspects of embedded system design such as power consumption. We will now describe how you can measure the power consumed by the Nucleo platform so you can factor power into your design decisions. The Nucleo board provides a way to measure its own current consumption by exposing two headers connected in series with its power supply. You can access them by removing the IDD jumper highlighted on Fig. 5. **Note that while the jumper is off, the CPU has no power supply and nothing will run. Take special care to not misplace or lose the jumper.**

Use two jumper cables to connect to each of the IDD header pins. Connect both jumper cables with the leads of a through-hole 1.1 Ω resistor as shown on Fig. 6. Once you know the voltage drop across this resistor, you can calculate the current through the circuit. Since the Nucleo platform uses a supply voltage of 3.3V, you can now calculate the power consumed by the platform. When used like this, this type of resistor is called a *shunt resistor*. Ideally, we want to use as small a resistor value as possible so that the resistor itself does not affect our measurements. Once you are done, be sure to replace the jumper across the IDD header pins.

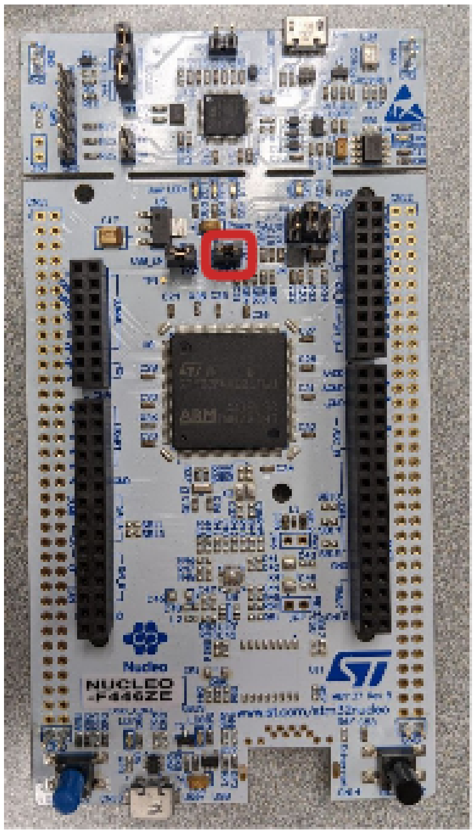
Figure 5

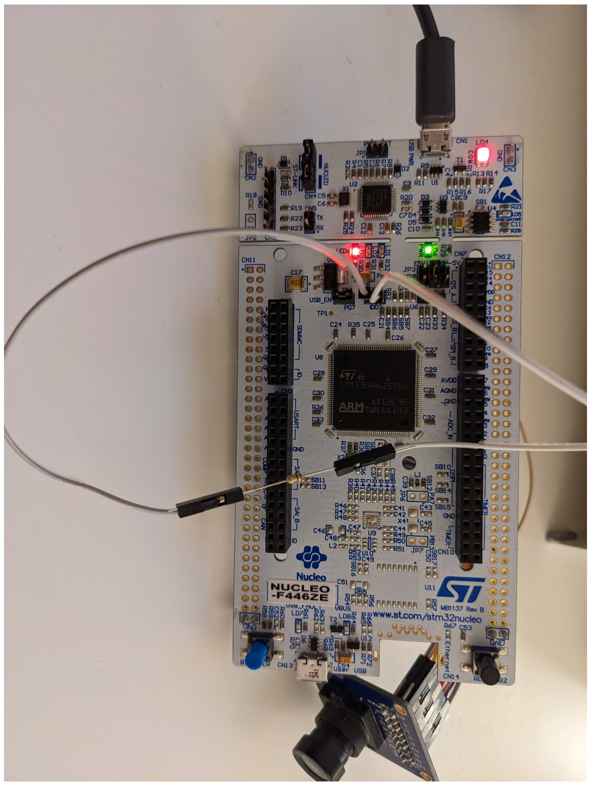
Figure 6

7. In-lab Demo

During your lab session, you must show your TA the following:

- **Demo 1:** Show the camera streaming a video of the baseline implementation from Lab 5.
- **Demo 2:** Show the camera streaming a video with 4-bit pixels and Run-Length Encoding.
- **Demo 3:** Show the camera streaming a video with the video compression algorithm.

Report your results in Table 1.

Table 1. Frames per second results.

	Min. FPS	Typ. FPS	Max. FPS
Baseline			
UART with DMA			
4-bit pixels			
4-bit pixels with RLE			
Video Compression Algorithm			

For full marks, your images should show little or no screen tearing.
Your TA may ask questions about your solution. Be prepared to explain your design decisions and your code.