ADULI-LED

Auto-calibrating Display using Unconstrained Layouts of Individually-addressable LEDs

Preliminary Report

Noah Wiley
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
njwiley@mit.edu

Luc Gaitskell
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
lucg@mit.edu

***Abstract*—Custom lighting fixtures and displays can be simply and cheaply constructed with modern addressable LED strands. However, displaying images on these displays requires painstaking manual calibration based on the geometry of the arrangement, and could be near-impossible in unconstrained and non-uniform layouts. We present ADULI-LED, an FPGA based system to quickly calibrate similar displays with the use of a camera. We present a novel calibration algorithm aimed to effectively and quickly calibrate these displays and a comparison with more naive approaches. We implement specially designed hardware to execute this algorithm while simultaneously maintaining the timing for the sequential color data on the strand. Additionally, we provide hardware to visualize processes through HDMI and to display images and effects on the calibrated strands. In our evaluation, ADULI-LED can calibrate a string of 250 LEDs in just 8 camera frames or ~ 0.5s.**

***Index Terms*—Digital Systems, Field-Programmable Gate Arrays, Addressable LEDs, Video Processing**

I. Introduction

With the advent of cheap low power individually addressable LEDs like the WS2812b, arbitrarily long (1000+ LEDs) strips of addressable LEDs are widely used for a variety of lighting tasks. These chips require only one data wire per strip, and operate by reading the first 24-bits of color then passing subsequent data through. Their low cost and minimal wiring requirements make them particularly suitable for effects lighting.

Many existing systems are designed around microcontrollers like the ESP8266 and ESP32, supporting visually appealing effects and common connectivity for smart home systems. However, these systems are not well equipped for more demanding lighting configurations and I/O, and effects are limited to standard grid-like layouts. Additionally, the reliance on DMA for timing introduces a memory bottleneck on the maximum effect speed.

ADULI-LED enables appealing effects and images on arbitrary layouts of LEDs, by calibrating each pixel as a 2D coordiante on the viewing plane. While camera and LED I/O are possible on other platforms, microcontrollers suffer from low throughput and therefore long calibration and display latencies. A larger computer would also need to rely on another device, such as a microcontroller, to send the color data within the strict timing window for the WS2812b chips. As such our system is developed on an FPGA. This allows us to optimize image capture and memory movements by implementing our algorithm within the timing constraints to send data to a single strand pixel. The FPGA affords us the unique ability to sync our image capture and led strand display, reducing the number of unused video frames. Additionally, relatively low LED (~ 1000) count allows the use of BRAM onboard the FPGA. This BRAM is extremely close to our logic, greatly reducing time to store our calibration and eventual lookups.

A. Goals

- Provide effective calibration
- Display smooth camera output and simple effects on the calibrated display
- Sub-linear calibration time

We met our ideal goals for the final iteration of our project.

B. Anticipated Challenges

Given the restrictions of design for our FPGA, and the camera input data, we must:

- Detect LEDs uniquely from objects in the camera frame
- Robustly handle overlapping LEDs
- Precisely time camera frame capture
- Construct accessible lookups for determining LED colors.
- Operate within WS2812b protocol, as each LED color must be sent sequentially and before timeout.

C. Approach

To display correctly mapped colors to the LEDs, we must determine a relative location for each LED in the whole string. Our implementation determines which LED corresponds to a given camera pixel, during the calibration phase. Each LED simultaneously flashes its corresponding binary-encoded ID, and we detect it with the camera. This same pixel's color then is used to set the color of the corresponding LED during normal *display* operation.

II. Logical Design

At a very high level, ADULI-LED takes the camera feed as input and outputs to the led strand.

The LED strand is driven by the *LED Driver Module* which requests an LED ID, listens for color data, and sends this color data to the strand using the appropriate WS2812b protocol. The LED driver is given colors based on the requested LED ID by the *ID Shower* module in calibration mode and the *LED Color Buffer* module in display mode.

Calibration mode consists of a series of steps to collect and build up LED addresses. A single step involves displaying bits to the led strand, capturing a camera frame, and updating our calibration. These fine grained operations and timing are orchestrated by the *Calibration Step FSM*. This contains the calibration table BRAM and updates it appropriately at each step.

In display mode, mapping information from the aforementioned BRAM and frame buffer data are stored in the *LED Color Buffer*. This contains another BRAM and logic to appropriately manage calibrated/not calibrated camera pixel color data and to drive the led strand.

Finally to provide a modularized and simple user interface, we implement a *Calibration Display Manager* FSM to further orchestrate calibration steps and display modes.

A cohesive block diagram is shown in Figure 5 and more detailed module descriptions are included below:

A. WS2812B Driver (Luc and Noah)

We have constructed a parametric WS2812B compliant driver in SystemVerilog. Although the LEDs are individually addressible, they must be written to sequentially, followed by a reset code to return to the first LED. The signal structure and data cascade is shown in Figure 1. As a result, our driver must maintain the state regarding which LED will have its color set next, to communicate this to the modules providing the requested color.

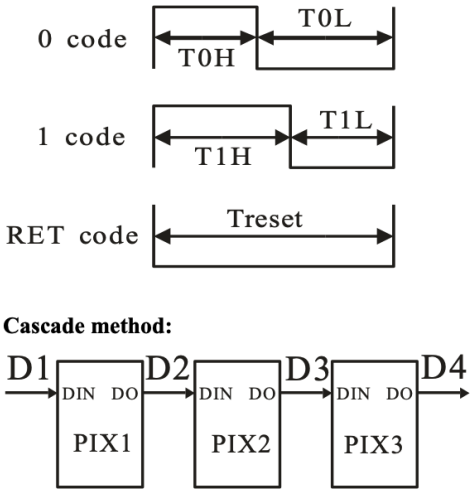

Fig. 1. WS2812B data codes and cascade structure. [1]

B. Calibration (Luc and Noah)

An optimal technique provides a robust and sub-linear time calibration sequence.

Our implementation runs $O(\log n)$ cycles to flash out each bit in each LEDs' ID, all at once. For this, Noah built an *LED ID Shower* module to take advantage of the inherent $O(\log n)$ encoding provided by the binary ID numbers. It interfaces with the LED driver and displays the ID bit colors appropriately. For each ID requested by the LED driver, the module drives a pixel red for 0 and blue for 1 based on the MSB *address bit number*. Red and blue were picked as their wavelengths are the most dissimilar aiding in better detection especially when using chroma red and chroma blue thresholds. This module outputs a signal *displayed frame valid* to indicate all pixels in the strand are displaying the correct colors and allows us to proceed with the camera capture.

The ID values accumulated at each pixel are shifted left, and then the LSB set to according to the pixel thresholding, reconstructing the LED ID in binary at each pixel. We therefore are able to detect which LED is in at a given pixel. The calibration is coordinated by the *Calibration Display Manager*, for iteration i from $0 \rightarrow \log_2(\texttt{NUM\_LEDS})$, is as follows:

1) Use LED *ID Shower* to display bit $\log_2(\texttt{NUM\_LEDS}) - i - 1$, as we build IDs from MSB first.
2) Collect and store calibration sample for that bit
3) Increment to next selected bit, and exit back to *display* mode once finished.

The individual calibration sampling is handled by the *Calibration Step* FSM, as shown in Figure 3, operates as follows:

1) Wait for signal to start calibration, usually triggered by LED driver finishing displaying the relevant bits.
2) Wait for camera to have captured new frame.
3) Threshold each pixel and accumulate new bit corresponding to that pixel into the pixel $\rightarrow$ LED ID BRAM.

Each LED is to display the corresponding bit by using pure red for for a 0 and blue for a 1. We then can use chroma red and chroma blue thresholding to detect these two colors in the image.

In order to modularize this operation, Luc built a *Shift Accumulate* BRAM, which can be easily validated in test benching outside of the LED calibration. This module allows the caller to provide a "summand" for a given address, for which the module:

1) Fetches the value at that address
2) Shifts the read result and appends the "summand" as the new LSB.
3) Writes the result back to that address.

We added further functionality for flagging pixels as invalid, in the case both or neither a 0 and/or 1 signal threshold is detected. Pipelining allows the module to support one request per-cycle, despite the two cycle delay on the initial read. The only limitation is that successive accumulation requests to the same address will cause data races, however, this does not occur in the video pipeline.

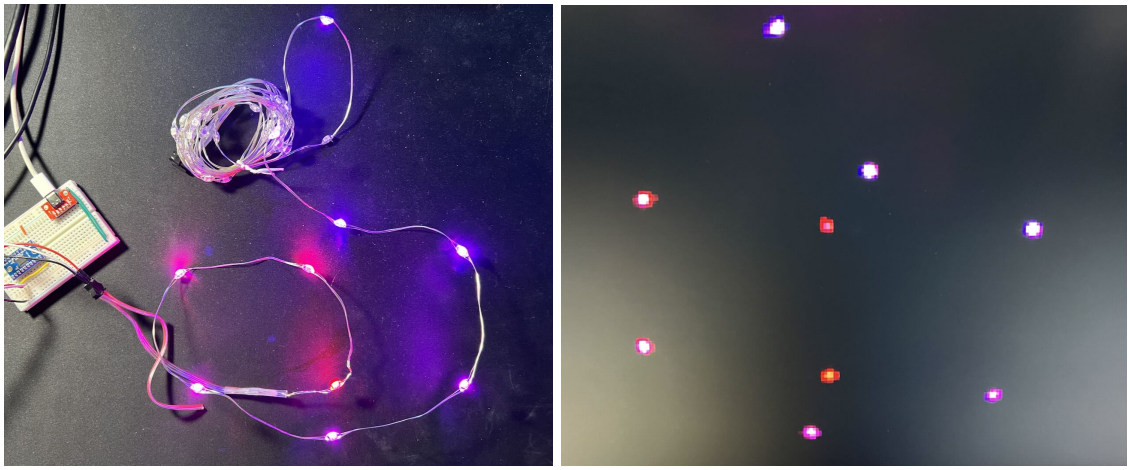

Fig. 2. Demonstration of setup and photo of HDMI display-out, using short OV5640 exposure time. Color data is preserved and not clipped.

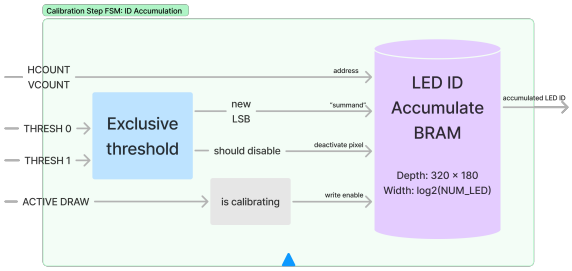

Fig. 3. ID Accumulation in Calibration Step FSM

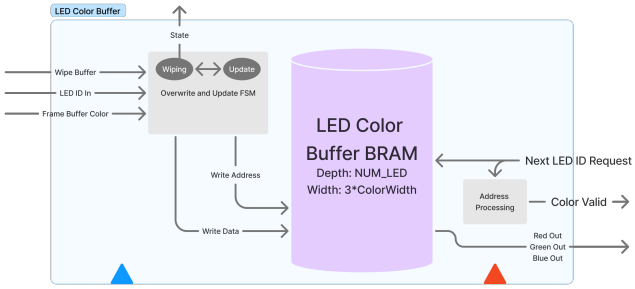

Fig. 4. LED Color buffer

This approach is significantly faster, allowing for a desirable logarithmic calibration time. However, it can fail for completely overlapping LEDs, as the captured IDs become bit-wise ORs and cannot be deciphered. In practice, LEDs that are not correctly detectable during calibration are most often obscured from view, and therefore should not be driven. Slight offsets between partially overlapping LEDs cause the opposing edges of their color regions to be detected properly, separately.

C. Displaying Camera Feed (Noah)

In order to display smooth videos on the calibrated display, we must output colors according the sequential LED ID numbers requested by the LED driver. Since our LED driver operates at very different timing than our camera, we need a buffer. Noah implemented this as the *LED Color Buffer* module which maps (LED ID $\rightarrow$ color). This is implemented with a dual port BRAM.

This module takes in 24 bit color data from the frame buffer and a corresponding LED ID from the Shift Accumulate BRAM. If the LED ID is a valid LED ID number and not disabled (Represented as an ID of all 1s), the module saves the color data to address row addressed by LED ID.

The output side utilizes the other port to respond to requests from the LED driver, looking up data from the BRAM, scaling appropriately, and generating a properly synced color valid signal.

Finally, random or left over data in the color buffer BRAM can cause distracting colors to be displayed on un-calibrated pixels. To fix this, an overwrite state was added to the FSM, with a wipe input, to write zero to all of the addresses of the BRAM before a new calibration is used.

Sending the full 24-bit color to a pixel takes 30μs, and therefore gives us up to 3000 cycles per LED ID request from the LED Strand driver. This uniquely allows us to use a BRAM frame buffer and eliminates the need for queues.

D. Camera Configurator (Luc)

We found that the brightness of the LEDs caused the camera sensor's pixels to saturate, which reduces the ability to get accurate color information during calibration. By manually modifying the camera exposure time, the LED detection can be greatly improved.

Tuning this also helps easily filter out unnecessary background information, while maintaining the LED color data. The HDMI output of the camera pointed at LEDs is shown in

Figure 2. Note that this is operating without thresholding, and is the direct readout of the camera.

In order to configure the camera during operation, we built a configuration module to overwrite the camera register BRAM with values based on user-selected switches when requested. On reset of the camera, the camera registers are then sent over I2C, with the updated configuration.

Using the OV5640 datasheet [2], we determined the corresponding *Auto-Exposure Control* (AEC) and *Auto-Gain Control* (AGC) registers. We added the following to the end of the provided `mode_180_320_25fps.mem` configuration from Kiran:

- 0x350[0-2]: Exposure setting
- 0x350[A,B]: Real gain setting
- 0x3503: Control AEC and AGC manual or automatic

When triggered, the configurator overwites these additions to the initial camera BRAM as requested.

E. Displaying Test Pattern (Luc and Noah)

Due to noise in the camera, we opted to also provide test patterns for performance evaluation. For modularity, we opted to add a multiplexer to choose the pixel source on the camera side of the framebuffer BRAM. This allows pattern generators, such as a vertically scanning white line, to inject its own data into the camera framebuffer, which ultimately is driven out to the LED strings. This is controlled by user-operable switches.

We also include patterns that avoid the camera and calibration pipeline. Our sequential LED pattern, drives each at a time, in order, and clearly illustrates the inherent ordering of the LEDs in our unconstrained strands.

F. Clock Crossing (Luc and Noah)

In order to properly cross clock domains between the *Calibration Display Manager* and the *Calibration Step* FSM, we had to set Vivado to increase the tolerance for timing violations between the 100MHz and pixel clocks. Unfortunately, this was unavoidable as the *Calibration Display Manager* must also communicate with the ID shower, which operates off the 100MHz clock.

To acommodate, we only send single-bit triggers, for multiple cycles, which then trigger actions that run for multiple cycles before using the value again. This avoids metastability from being interpreted as multiple triggers. To avoid arbitrary counters, we have single-bit acknowledgments, allowing the triggering module to hold it high until the triggered module indicates its state has changed to in-progress.

This acknowledgment takes the form of two wires signaling: (1) idle and (2) in-progress.

The triggering module sends the trigger until signal (2) is high, and waits until signal (1) is high again. This prevents the multiple $0 \leftrightarrow 1$ transitions on a single wire due to metastability from causing transitions that inadvertently skip states.

III. Physical Design

Our work utilizes the RealDigital Urbana board, with a AMD Spartan-7 FPGA. We additionally utilize:

- WS2812b LED strands: sequentially addressed LEDs used to display content.
- OV5640 Camera Module: used both for calibration and input data for displaying on the LED display.
- HW-221 level shifter: drives the 5V logic from the 3.3v logic of the Urbana board.

Although our current implementation drives the LEDs as a single strand, this can be trivially replicated to drive multiple strands at once. Shorter strands allow for higher refresh rates, if desired.

IV. Evaluation

Due to the nature of the project, our evaluation is mostly qualitative, judging the look of the patterns, with respect to the HDMI output of the camera feed. During operation, we noticed that content with high contrast was significantly easier to recognize. It also helped reduce the noise at each pixel in the frame. As a result, we often opted to keep the camera in fast-exposure mode, and use bright targets, such as an tablet screen.

In order for more quantitative analysis, we were able to use the display pattern in section II-E for more predictable basline for performance analysis. For a dense arrangement of LEDs, we noticed around 5 of the 250 LEDs were mis-identified.

A 98% accuracy seems adequate for our fast one-shot calibration implementation. The naive approach is to illuminate each LED individually, and use a technique such as COM to determine its position in the frame. This is susceptible to other light sources, while the requirement for consistent and exclusive blue / red illumination in ADULI-LED virtually eliminates these problems. Additionally, the individual calibration requires a single frame per-LED, which would take at least 10 seconds for the string we demonstrated. Such linear scaling would become intractable for larger strings, while the $\log$ scaling in ADULI-LED would still take around a second for 10-million LEDs.

A. Module Performance

Due to clocking the modules off strictly timed clocks, such as the pixel clock, all of our modules are designed to process one item per clock cycle.

However, certain actions, such as the shift accumulation in section II-B, require multiple cycles to operate and are pipelined accordingly. We maintain a properly pipelined video datapath, adjusting synchronizing flip-flops to account for the additional delay due to the calibration module. There are now 10 cycles between the video signal generator and the HDMI output. The WS2812b are relatively slow, compared to clock of the FPGA and, as mentioned in section II-C, we have around 3000 cycles before we must have the next LED color ready.

Due to the relatively relaxed latency tolerance as humans, we do not notice the tens of nanoseconds of delay in either the HDMI or LED control.

B. Resource Consumption

Our design heavily utilizes BRAMs, to store:

1) $320 \times 180 \times 16$b frame buffer
2) $320 \times 180 \times \log_2(\texttt{NUM_LEDS})$b pixel to LED map
3) $\texttt{NUM_LEDS} \times 16$b LED to color buffer

According to Vivado, we used sixty percent of the available BRAM. Our approach to clock crossing is document in section II-F.

V. Discussion and Further Work

A. Improving Accuracy

By combining the naive approach and our robust ID $\log(n)$ approach, we may be able to increase accuracy for improperly calibrated LEDs. If an LED was not found during the calibration process, then the naive approach could be a more definitive test, calibrating this LED individually using COM.

B. Improving Pixel Display Colors

Since the LEDs on the strands are so large, even with a low resolution camera, many camera pixels map to the same LED pixel. In our current implementation and given the raster pattern of our camera and frame buffer, LED i receives the color in bottom right most camera pixel that was mapped to LED i. While we still achieved cohesive video throughput with this, it may be interesting to explore an averaging approach for the color of an led. If able to bound the maximum number of camera pixels that map to the same LED, a similar approach to the *Accumulate Shift* BRAM may be used to average color values assigned to each LED.

C. Reusing Frame Buffer / Using DRAM

While our design and usage of BRAM comfortably builds given the resources our AMD Spartan-7 FPGA, the usage of DRAM may enable some new features. The number of LEDs is usually on the order of thousands of pixels, and does not drive the main memory usage. Our frame buffer and shift accumulate BRAM are similar in size and very large, driving our memory usage. We may be able to achieve a higher camera/hdmi resolution by using a larger shared frame buffer for both calibration and display, by running the calibration logic directly on the camera input. In addition to adding complexity, this removes our video feed during calibration which is useful for debugging and is interesting to watch during calibration. To remedy this, DRAM may be used to support higher camera/hdmi resolutions. A higher resolution may allow for better color averaging as described above and possibly better performance with tightly packed led strand configurations.

D. Reducing Number of Clock Domains

Our implementation uses 3 different clocks (camera clock, hdmi pixel clock, and 100MHz general clock). While we excitedly explored this as an exercise working with various clock domain crossings and to offer flexibility for different led driver modules, we recognize that our current system may have been implemented with just the camera clock and hdmi clock. Multiple clock domain crossings required careful experimentation with when we sent signals between the calibration manager and the calibration step FSM and the number of bits we sent. A less flexible but more optimized system may allow for even tighter timing resulting in a faster calibration.

VI. Acknowledgments

We would like to thank Joe Steinmeyer, the 6.205 TAs, and other course staff for all their assistance, support, and for making 6.205 a very interesting and enjoyable class!

References

[1] https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf
[2] https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf

Appendix

A. Source code

[redacted]

B. Overall block diagram

The overall block diagram can be seen in Figure 5

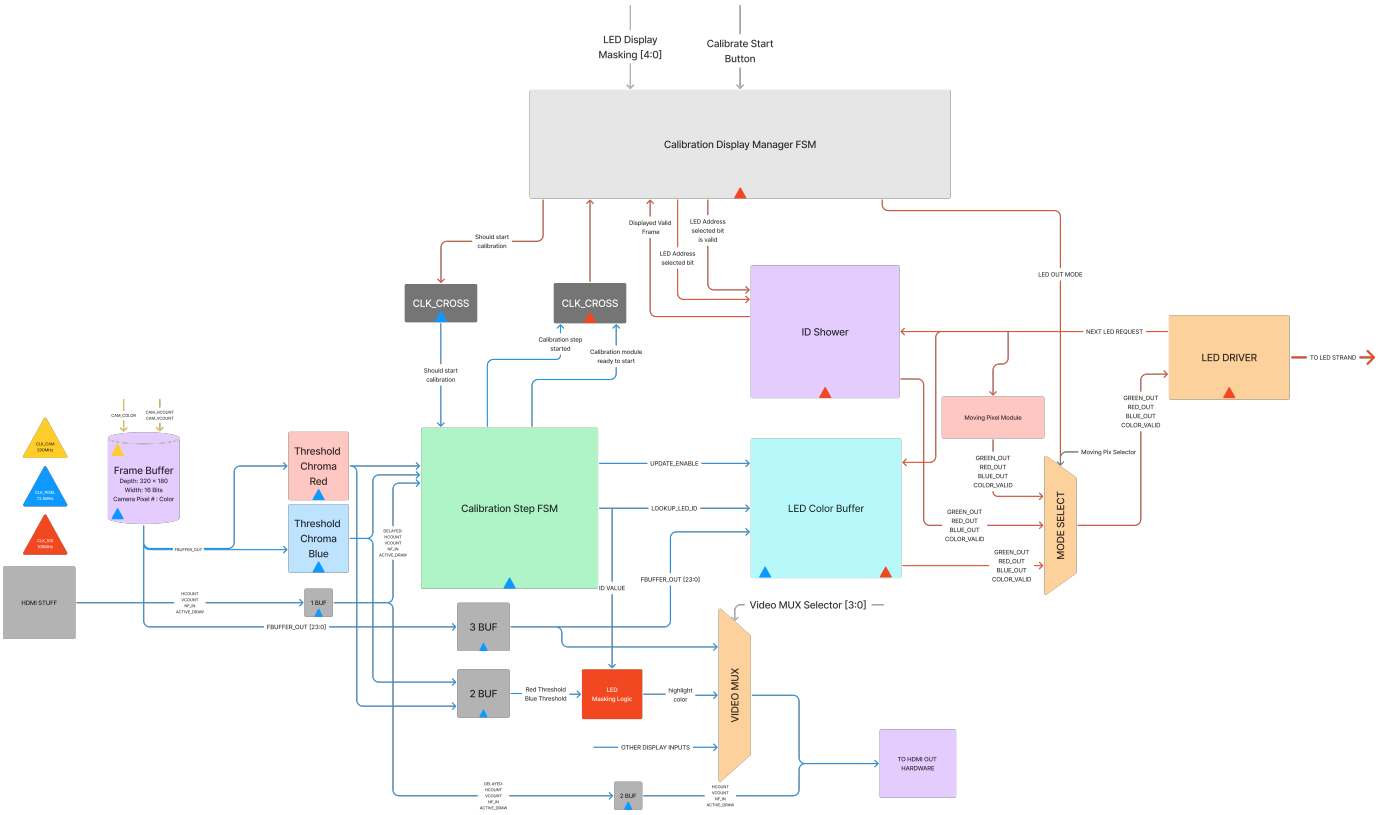

Fig. 5. System Block Diagram detailing modules used for calibration, LED strand driving, and HDMI output