Anglerfish: A Novel Pacing System for Swimmers with Stereo Camera Tracking

1st Brian Li
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
brianli@mit.edu

2nd Viveca Pannell
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
vlpannel@mit.edu

***Abstract*—The Anglerfish system is a swimming training tool that aims to help swimmers achieve their pacing targets by displaying a light emitting diode (LED) light target for them to follow underwater. The system provides accurate pacing via the "speed" of the moving target, it also records athlete lap times and slows down the target to allow an athlete who has fallen behind to catch up to the target and resume following the pacing target. This function requires knowledge of where the swimmer is located in the pool, which is estimated using a stereo camera setup. The Anglerfish system is currently in development, and we have created a prototype that implements all of its main features (LED pacing target, split times, and swimmer location detection) on a pair of field programmable gate arrays (FPGAs). We present a prototype of this pacing system, having implemented a stereo image matching algorithm for distance detection, FPGA control over an LED strip, and simultaneous and real-time image data collection from two cameras to implement the image processing in real-time.**

***Index Terms*—FPGA, image processing, LED, neopixel, SPI, stereo camera**

I. Physical Construction (Brian)

The hardware components of this project include:

- 2 OV7670 Cameras that are spaced 2.5" apart that collect stereo image data
- 5 meter WS2811 LED NeoPixel strip which is used to make the light target for the swimmer to follow
- IR TX/RX modules used for low data rate link between FPGAs to send swimmer's time stamps
- 2 Urbana FPGAs

II. Stereo Image Processing (Brian)

We focused our project on the aspect of stereo image processing that is most computationally intensive: identifying distance (also referred to as disparity) between corresponding pixels in left and right images. An example of a disparity map is shown in Figure 2. Once we have the disparity map, it is easy to calculate the distance associated with a given pixels by using information about the geometry of our system setup. For this matching problem, we decided to implement an algorithm called Sum of Squared Differences (SSD).

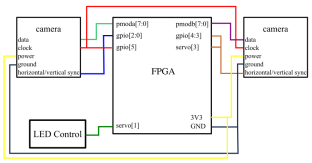

Fig. 1. FPGA pinouts, adjusted to handle two cameras and LED strip signal

In essence, SSD takes a block of pixels in the left image and a block of pixels in the right image, whose sources are offset by some given distance, and subtracts the blocks, squares the differences, and then sums everything together to come up with a similarity metric. The lower this metric, the more similar the blocks are to each other. Note that the size of the blocks, n, is the most important parameter, as it affects not only the performance of the algorithm but also the hardware we configure on the FPGA as detailed in the sections below. By repeating this calculation for blocks of pixels and changing the offset distance, d, by one pixel each time, we can find the d at which the SSD is minimized. We then save this d into a block random access memory unit (BRAM) and repeat for all pixels in the image. This data then allows us to look up the distance of the swimmer assuming we can identify their swim cap using masking and a center-of-mass (COM) calculation.

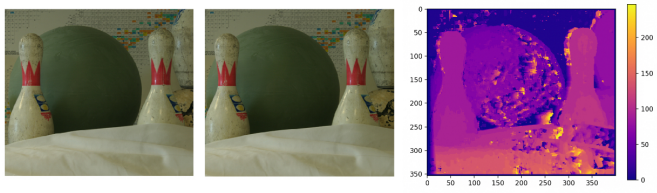

Fig. 2. Left: left camera view, Center: right camera view, Right: disparity map from SSD

Report not published. Submitted to 6.205 staff 13 December 2023. *(Corresponding authors: Brian Li, Viveca Pannell.)*

The authors are with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA (email: brianli@mit.edu; vlpannel@mit.edu).

This article has supplementary downloadable material available at https://github.com/brieonbread/anglerfish.git, provided by the authors.

A. Obtaining Simultaneous Stereo Image Data from OV7670 Cameras (Viveca)

In order to enable the central FPGA (which performs the stereo image processing, camera data collection, and split time output) to simultaneously use two cameras as well as an infrared (IR) receiver, the pinouts used for integrating the OV7670 board with the FPGA's communications had to be altered, and the two cameras had to share one clocking pin while they used up all available pmod and gpio pins. Our new assignment of pinouts allowed the FPGA to simultaneously use two different cameras, using a switch to select which camera's data was being streamed to a display via high definition multimedia interface (HDMI) communication. In accordance with advisor feedback about the importance of rising clock edges, we specifically chose a pin assignment such that the servo pins on the FPGA (which, due to being connected to a larger resistance than other pins cause weaker rising edges) to be used for the horizontal sync signal of the second camera rather than the shared clock or any other signals that may require a sharp rising edge.

B. Stereo Vision Testing and Debugging Tools (Viveca)

In order to help with the development of the camera pipelines and stereo vision depth sensing modules, we developed a module to allow an FPGA to completely transmit the contents of an entire BRAM onto a computer, communicating this data via the universal asynchronous receiver-transmitter (UART) protocol. The module, given an enable signal, sequentially queries data entries in a BRAM and send them via the FPGA's UART transmission output. This way, a computer that is receiving serial data and is connected to the FPGA's UART connections can read the data that the BRAM readout module gives it.

The module is governed by a simple FSM. The FSM shifts between states IDLE, INIT, SET DATA, and SEND DATA. In each state respectively, the module: waits for a signal to begin dumping data and ensures that it has not already sent all data, requests the first data entry from the BRAM and waits for that data to be output (since BRAMs take two cycles rather than one in order to produce valid and correct output), requests the next data entry from the BRAM while saving the current data to a register, and waits for the UART transmission module to signal that it is done before sending the data and transitioning back to SET DATA or INIT (if done sending all data).

This version of BRAM readouts was a vast improvement upon its previous incarnation as a camera readout via manta[1], a Python-based hardware debugger. It is much faster to directly communicate via UART.

C. Memory Architecture (Brian)

According to its datasheet, the Urbana FPGA has 120 BRAMs for a total of 4.2 Mb of on onboard storage. Each image from our cameras is 320 by 240 pixels, which is roughly 0.5 Mb for each image for 8 bits per pixel, so we had enough storage onboard to have one BRAM frame buffer from each camera. For our memory architecture, we decided to use two dual-port BRAMs with 12800 entries total (320*40), where each entry corresponded to a set of n pixels. With each pixel represented by 8 bits, each entry required $8n$ bits of data. Figure 3 shows an example when $n = 6$. By storing all of the pixels required for a row of SSD calculation in a single word, we were able to reduce the number of cycles where we needed to wait in order to get data from the BRAMs and take advantage of the spatial structure of the SSD calculation. In addition to the left/right frame buffers, we also made use of a single port BRAM with 76800 entries (320*240), with each entry corresponding to the disparity calculated for that pixel. The maximum disparity is 240 (the number of columns in the frame) and thus each word in this BRAM is 8 bits wide. Overall, our memory architecture ended up using 96.00% of our BRAM resources.

Site Type	Used	Fixed	Prohibited	Available	Util %
Block RAM Tile	72	0	0	75	96.00
RAMB36/FIFO*	72	0	0	75	96.00
(RAMB36E1 only)	72				
RAMB18	0	0	0	150	96.00

TABLE I
OUR BRAM UTILIZATION ON THE FPGA ACCORDING TO THE POST-SYNTHESIS REPORT

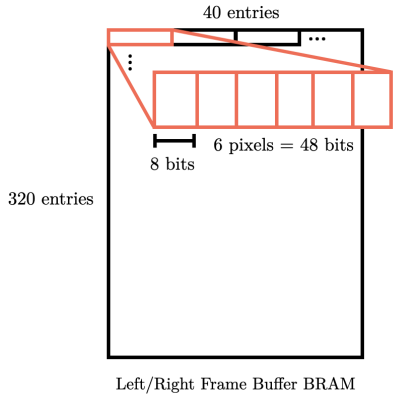

Fig. 3. Image data stored in 2 BRAMs where each word correspond to a set of n pixels

D. SMAC Engine Architecture (Brian)

In order to perform the SSD calculation for the block of pixels in the left and right images, we proceeded by calculating the SSD for the first row in the block, then the second row in the block, and so on. To do this, we first implemented a 48-bit SMAC engine, which takes in two 48-bit numbers. Each set of 8 bits corresponds to a pixel, thus each SMAC engine takes in 6 pixels to process at once. For each set of pixels, the engine takes their difference, squares the difference, and then accumulates the result in a flip flop as shown in Figure 4 Thus, each SMAC engine allows us to compute the SSD metric for six pairs of left/right pixels in one clock cycle. We furthermore took advantage of our memory architecture

[1]https://fischermoseley.github.io/manta/

to compute the SSD metric for an entire block in one clock cycle by placing n SMAC engines in parallel. This is shown in Figure 4. As they progress down each row, the SMAC engines accumulate the SSD metric down each column. At the end of this process, the system can sum together the values in the flip flops across all the SMAC engines in order to get the SSD metric for the given set of left/right blocks. An important note is that while in theory (and if we had more time) we could calculate the SSD metric in a single clock cycle, in practice we added registers to break up some of the combinational logic and avoid timing issues. Thus, each calculation actually takes 3 cycles to complete.

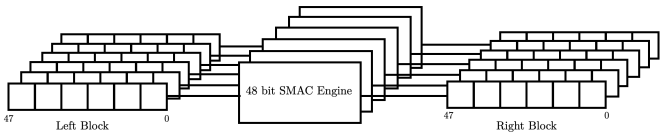

Fig. 4. 48 bit SMAC Engine, our implementation uses n 48-bit SMAC engines in parallel

E. Putting it all together (Brian)

In order to implement stereo matching, we also required a few additional components. In order to compute the disparity for a given pixel, we needed to calculate SSD for multiple sets of blocks where the right block has various offsets compared to the left block. This meant that pixel values could get reused multiple times. To avoid having to read the same pixel values multiple times from the BRAMs, which would introduce additional cycles of delays, we also implemented a set of temporary buffers. There are four buffers total, two wired with the left image buffer and two wired to the right image buffer. Each buffer contains 6 registers, which each stores one word of size $8n$ bits.

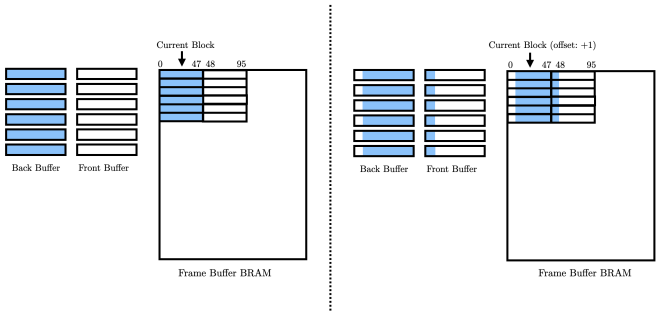

Fig. 5. Most blocks don't fit within a single set of words so we employ a front and back temporary buffer so we can access all data needed for these blocks without constantly reading from BRAM

The reason we needed two temporary buffers for each image buffer (as opposed to just one) was because, in most cases, the pixels in a given block didn't fit exactly in a single word. Rather we needed to read two words from the BRAMs in order to get the complete set of data. There is an example of this in Figure 5. In order to update these buffers as the pixels positions change, we wrote a separate module that takes in the current positions of the pixels we are looking at, and retrieves the correct data from the left and right image BRAMs. This turned out to be a bottleneck in our stereo pipeline. For a nominal update, based on the memory architecture we chose, it takes a minimum of 6 cycles to update the temporary buffers for the right front buffer only (note that the right pixel is constantly being shifted/updated in the SSD algorithm) in addition to an extra cycle for transferring the values from the front buffer to the back buffer (since we don't need to get these from BRAM again). However, in an effort to simplify some of the higher level control logic, we decided update both front and back buffers each time, which means it takes 12 cycles to fully update the temporary buffers. In retrospect, we could have chosen a slightly different memory architecture to finish updating buffers. This is discussed in the following section.

Finally, we implemented a top level state machine to coordinate between the various modules involved in carrying out the SSD algorithm. The states are described below:

- IDLE: default state
- NEW FRAME: resets a bunch of registers, counters, etc. in preparation for start of new disparity map calculation
- UPDATE CENTERS: contains logic that updates the (x, y) counters for left/right images depending on where we are in the calculation (i.e. if a row is finished, set x = 0 and $y = y + 1$
- UPDATE BUFFERS: uses updated (x, y) counters in order to grab the corresponding entries from left/right image BRAMs and save in temporary buffers
- CALCULATE: begins SSD calculation using parallel 48-bit SMAC engines for the left/right blocks
- UPDATE DISPARITY: checks if SSD is less than the smallest SSD so far and updates the disparity offset accordingly
- SAVE: we are done with disparity calculation for this (x, y) calculation, so save result to BRAM

The transitions between the states are illustrated in Fig 6

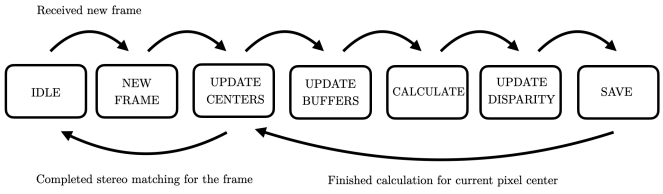

Fig. 6. Overview of our stereo algorithm implementation FSM states and transitions

F. Evaluation of SSD Algorithm (Brian)

Because it is difficult to debug the SSD algorithm on the FPGA, we wrote a "ground truth" Python implementation of the SSD algorithm to allow us to quickly test how well the algorithm worked on various images and parameters (i.e. block sizes). To test if our implementation on the FPGA worked as

expected, we loaded a left and right test image, ran the algorithm, and then retrieved the values from the BRAM, storing the disparity map to our computer for checking. Ultimately, this module proved to be a tradeoff between memory and timing The lower bound for the number of cycles for our SSD system, without breaking the image up into chunks, is $320 * (1 + 2 + ... + 240) = 9254400$ calculations. Assuming a clock rate of 100 MHz and 1 cycle per calculation, this works out to roughly 0.1 seconds for a complete disparity map calculation on a 320x240 stereo image.

One idea for we have to decrease the number of cycles spent updating temporary buffers is redesigning our BRAM architecture. Instead of having a single BRAM for the left side and a single BRAM for the right side, we could instead use a set of interleaved BRAMs as illustrated in Fig 7. This would allow use to retrieve all values necessary to update buffers in a single clock cycle, resulting in significant speedup.

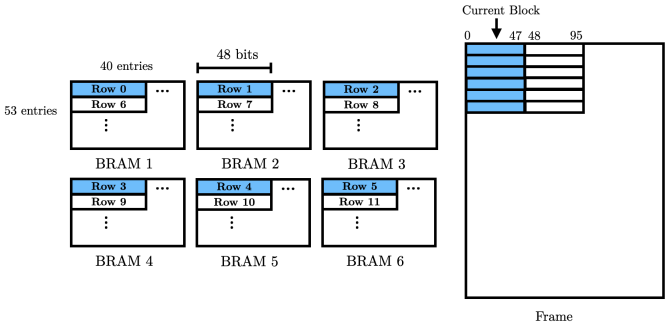

Fig. 7. Set of proposed interleaved buffers when $n = 6$, Tradeoff between increased memory/indexing complexity vs. cycle time

III. Implementing WS2811 LED Driver (Brian)

The NeoPixels we used for this project work in a cascade fashion where each integrated circuit (IC) in the line of pixels receives 24 bits of data, then reshapes the remaining data and passes it to the next ICs down the line as shown in Figure 8.

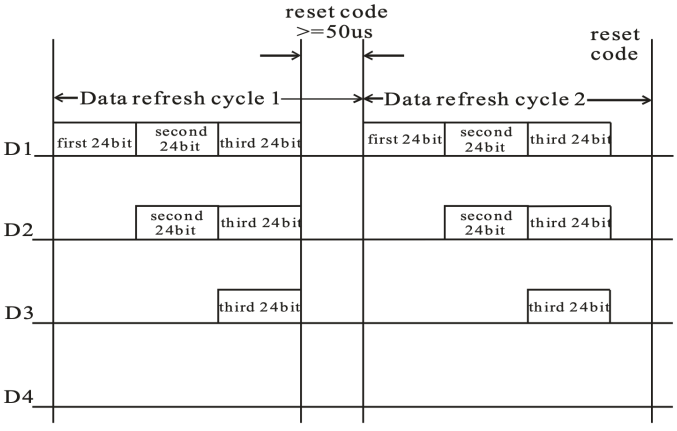

Fig. 8. Data passed to ICs on LED strip in cascading fashion

A. Packet Structure (Brian)

Each 24-bit packet contains 8 bits for each color red, green, and blue arranged in the following manner: packet = $[R_7, R_6, ..., R_0, G_7, G_6, ..., G_0, B_7, B_6, ..., B_0]$.

B. Timing Requirements (Brian)

There were 3 different symbols we could send to the LED strip: 0, 1, and reset. The difference between symbols 0 and 1 came from how long the signal was held high before going low as shown in Figure 9. A reset symbol simply holds the signal low for a certain number of cycles, which allows the LED lights to latch and display the new colors. The timing requirements are summarized in Table II.

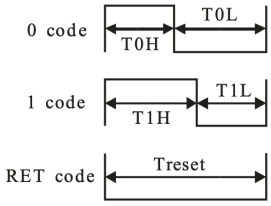

Fig. 9. There are 4 symbols we need to transmit T0H, T1H, T0L, T1L, TLL

Symbol	Parameter	Min	Typical	Max
TOH	0 code, signal high duration (ns)	350	500	650
T1H	1 code, signal high duration (ns)	50	1200	1350
T0L	0 code, signal low duration (ns)	1850	2000	2150
T1L	1 code, signal lowduration (ns)	1150	1300	1450
TLL	latch, signal low duration (ns)	6000	-	-

TABLE II
TIMING REQUIREMENTS FOR WS2811 NEOPIXEL LEDS

C. State Machine (Brian)

We made use of 2 state machines to implement sending commands to the LED strip–a major finite state machine (FSM) and a minor FSM. The major state machine contains logic that controls the whole led strip. It is used to assign a color to each LED and is in charge of generating the signal to refresh the LED strip. It uses 4 states: IDLE, START BLOCK, IN BLOCK, and END BLOCK. A block corresponds to the sequence of bits needed to change a single LED (note that technically each IC on the strip is connected to 3 LEDs which all display the same color). Thus, if we wanted to control 10 LEDs, we needed to send a sequence of 10 blocks followed by TLL, the latch signal, as listed in Table II. In order to generate the sequence of bits within a block, our minor state machine used 4 states: IDLE, RECEIVED INPUT, TRANSMIT 0, and TRANSMIT 1. When it receives a signal that a new RGB value is ready, it reads in the values stored in the RGB value register. If the most significant bit (MSB) is a 1, it enters the TRANSMIT 1 state, and if it is a 0, it enters the TRANSMIT 0 state. It continues parsing the RGB value register by shifting the bits to the left by one on each iteration, reading the MSB, and entering the appropriate state to generate the bit until all 24 bits have been transmitted.

D. Putting it Together (Brian)

In order to be able to change position of the light target to match the swimmer's pace, we implemented a top level module with the following states: IDLE, FORWARD, REVERSE, LATCH, and TRANSMIT. We wanted to generate a new cascade on each iteration where all LEDs remain off except for the LEDs corresponding to the target. Once the target has reached the end of the LED strip, we enter the REVERSE state where the target switches directions in order to complete the lap. To change the speed of the target, we can included a register that tells the state machine how many cycles it should stay in the LATCH state (minimum of 50000 ns, or 5000 cycles), which effectively changes the rate at which the cascade is refreshed. For debugging purposes, we connected the register to switches 3-0 on the board.

E. Evaluation (Brian)

We tested our implementation of the WS2811 driver extensively in test benches using various sequences of RGB values and numbers of LEDs and compared them to what we expected. In addition, to verify that we could control the speed of the target LED accurately, we used a stopwatch to measure the target's speed visually. The result agreed with our simulations.

IV. Inter-FPGA Communication (Viveca)

The Anglerfish timing system also utilizes a second FPGA, to be placed at the far end of the pool in order to time swimmer lap times, as stereo vision depth analysis becomes less accurate over longer distances. The two FPGAs communicate with one another to keep track of swimmer lap times, with the central FPGA housing critical functions such as central timekeeping, LED pacer control, and stereo vision image processing, and the peripheral FPGA signalling when a swimmer completes a lap based on image data from its camera pointed at the end of the pool.

The communication is one-way, with the peripheral FPGA utilizing an IR LED to send out signal for a short amount of time when it detects that a swimmer has completed a lap (with no signal being sent by default). The central FPGA detects a new burst of the IR signal in order to calculate lap times and display them on the FPGA's seven-segment display. The IR signal is a 38 kHz emission (when signalling high) to match the desired signal frequency of our IR receiver (which is also a demodulator).

The secondary FPGA acts as a motion gate, performing a subtraction of its current image's pixels (as they are received from the camera, not saved to BRAM) from its original image (saved to BRAM, collected before the user manually switches on switch 0 on the FPGA). It tracks the center of mass of the pixels with large a large discrepancy from the original image and send the central FPGA a new burst of IR signal when this calculated center of mass changes direction (meaning the swimmer has completed the lap and is returning the other way).

This simple mode of communication allows for somewhat accurate split time recording, as IR transmission is relatively fast compared to a swimmer (thus, IR signal reaches central FPGA before swimmer is too far away from the wall) and COM calculation is not complicated (thus it is not too temporally far away from the true center of mass from the swimmer), albeit it is not accurate to the hundredths of a second as originally intended due to a frame rate of 30 frames per second and a lack of interpolation methods to more precisely calculate the time at which the swimmer completed the lap.

At the time of the submission of this paper, the usage of the peripheral FPGA in Anglerfish timing operations was temporarily abandoned in favor of greater focus on the central FPGA. This was mainly due to two reasons: first, because the overall system had not yet been adapted for the 25m pool (hence, the image processing was deemed sufficient for our prototype's limited length), and second, because the directionality of the IR LED we were using required high precision aiming from the IR LED to the IR receiver, which we did not believe was necessary for this project.

V. Conclusions and Lessons Learned

Overall, we were able to meet our goals of having a functional stereo matching algorithm implemented and a LED target pacer using the LED strip. We implemented simultaneous usage of two cameras fed into image processing computations on the FPGA, and at the same time implemented FPGA control over our LED strip. Over the course of this project's development, we also learned many things about the nature of FPGA projects, algorithm design, and what we could implement in the future.

Before taking on this project, we had little experience with handling arrays and memory in Icarus Verilog (iverilog) and Vivado. Throughout the course of this project's development, we were able to adapt to handling such data, adapting our testbenches and developing other methods of debugging in order to "read out" our arrays and BRAMs.

From this project's development, we gathered more ideas for how to further develop the Anglerfish timing system, perhaps taking it beyond the prototyping stage someday. Although our current design was specifically created for 6-by-6-pixel block correlations, we would like to eventually parameterize our modules such that the hardware implementation of SSD could be generalized to use any n-by-n sized block. Additionally, although we deemed the peripheral FPGA/motion gate unnecessary for this version of prototyping, we would like to include the second FPGA in a second iteration of the system.