FPGATrack: Spatial Drawing System

Jack Gray
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA

Christopher Mejia
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA

***Abstract*—We present FPGATrack, a hardware solution leveraging two FPGAs and two cameras to track objects in 3D space. Each FPGA receives input from a camera and calculates the position of the object in their respective 2D views, then one FPGA will combine those two positions to construct the position of that LED in 3D space. This position is then be rendered in a 3D scene on a monitor through HDMI and the trajectory tracked to make "spatial" drawings. The user is able to review drawings and rotate the view to see the depth component. Running on an FPGA allows for faster, parallel processing of camera inputs compared to traditional microcontroller or CPU-based solutions. The system benefits from the FPGAs ability to handle multiple high-speed data streams simultaneously, enabling real-time calculations with minimal lag.**

***Index Terms*—Digital systems, FPGAs, Object tracking**

I. Physical Layout

The set up itself consists of two FPGAs seated at 90 degrees from each other, each with an OV5640 camera, and a view of the object in the center. They are connected together via two wires connected to pins on their respective PMODB ports.

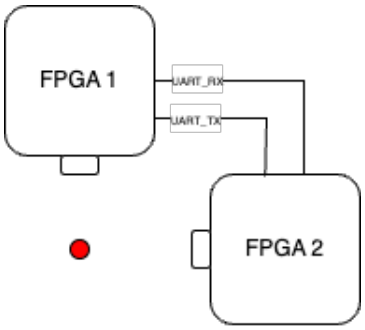

Fig. 1. Physical Layout

II. Centroid Tracking & Communication

A. Initial Single Point Tracking

The initial tracking system utilizes modified lab hardware, with the camera configured to capture frames at 320x180 resolution at 25FPS. The video feed is processed through some of lab developed modules. First, the camera output signals are inputted to our pixel_reconstruct module, followed by conversion to the YCrCb color space using the rgb_to_ycrcb module. Since the tracking system is specifically designed to detect red objects, it exclusively utilizes the chroma red (Cr) channel from the converted output. The chroma red output

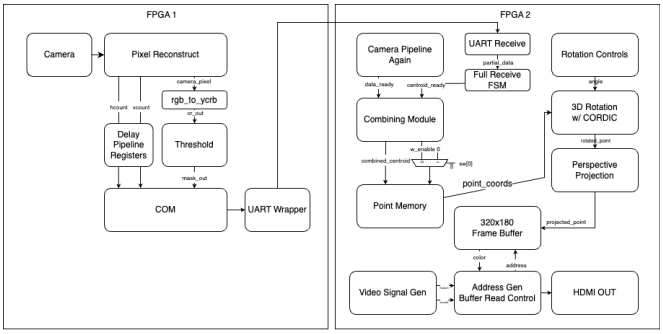

Fig. 2. Initial Full Stack

of the hcount and vcount that is currently being processed is passed through our lab threshold module that checks if the pixel is red. The signal out of the threshold module is used as the valid_in signal to our COM module to include the corresponding hcount and vcount from our pixel_reconstruct in the running sum for the COM calculation.

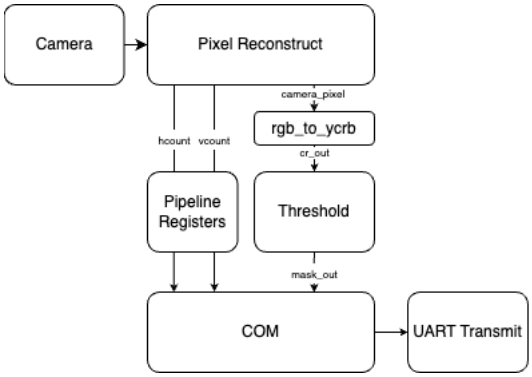

Fig. 3. Initial Single Point Tracking Layout

The center of mass from our lab used a divider that took an indeterminate amount of cycles. To amend this and improve performance, we implemented a divider that uses a non restoring division algorithm to complete 32 bit division in 34 cycles. The module processes the dividend and divisor across multiple clock cycles following a three-state finite state machine with states (IDLE, DIVIDING, DONE). The algorithm operates by iteratively shifting and comparing values, subtracting the divisor when possible, and building the quotient one bit at a time. More details on the algorithm can be found here.

This stack works to track one red object in the FPGAs view. The COM that is calculated by this hardware is in the range of 320x180, as a result one coordinate is 9 bits wide and the other is 8 bits wide. The final output of the stack is then 17 bits wide. This pipeline is replicated on both FPGAs, but one of them transmits its results to the other via UART.

B. Multiple Point Tracking via K-Means

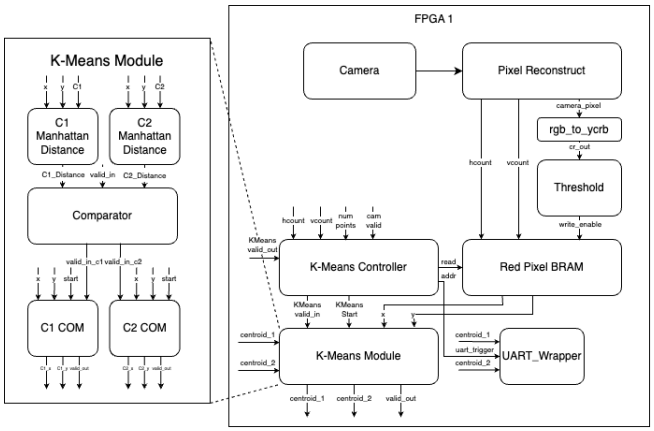

Fig. 4. K-Means Two Point Tracking

A hardware implementation of the K-means algorithm would replace the current center of mass calculation in the camera pipeline. The design used BRAM to store points identified as red through thresholding, then perform multiple iterations of K-means clustering during the vertical blanking period of the camera stream. All calculations will happen between frames and output to either the local logic or UART logic at the start of the new frame. This BRAM is 17 bits wide and 14,400 entries deep. We assume that our objects won't take up more than 1/4th of the frame so the relevant points we need to save should be lower than that ((320x180) / 4 = 14400).

Each time the treshold output goes high, the write address increases by one and a write request is issued. This address is reset to 0 at the end of frame. There is also a running count of the number of pixels written to this BRAM that is used by the K-Means Controller. The initial centroids are one close to the top left of the screen (50, 30) and the other close to the bottom right (270, 150).

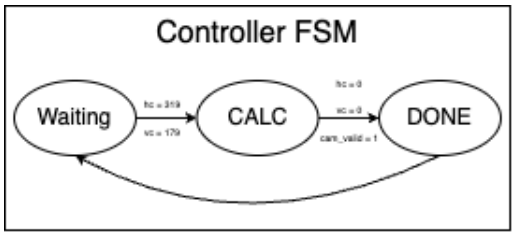

Fig. 5. K-Means Controller FSM

The K-Means controller is a 3 state FSM that will control the addressing into the BRAM and start calculations of the K-Means module. It will begin in WAITING where it is waiting for the end of the camera frame (hcount = 319, vcount = 179) to transition to CALCULATE state and begin K-Means. In the calculate state, it will wait for the first valid pixel of the next frame to stop calculations, move to DONE, and trigger the UART transmission. The outputs in these stages are as follows.

WAITING:

- All outputs 0

CALCULATING:

- read_address begins at 0 and increases by 1 every cycle until it reaches num_points. This only rolls back to 0 after reaching its max when the K-Means module valid_out goes high meaning we are ready to start a new iteration.
- The K-Means valid_in will go high two cycles after an address change is made (a read request is put through to the BRAM) to ensure the request has been resolved.
- The K-Means start signal will go high two cycles after the read request at addr = num_points - 1, meaning we have gone through all the relevant points and clustered them.

DONE:

- All outputs 0 except UART_Trigger which will go high to start transmission of the current centroids.

The K-Means module takes in the relevant (x,y) output by the BRAM and the current estimation of the centroids. It computes the Manhattan distance of each point to the centroid and control the valid_in for each COM based on which distance is smallest. This should lead to the running sum of each COM module containing a distinct cluster. The start signal from the controller starts the COM calculations which will output our centroids and a valid_out signal. The valid_out signal is used to start a new iteration if we are still in calculate. But the instant the controller sees the new frame start coming in, it will trigger the UART to output whatever our current estimates are.

These modules and the full stack was implemented. The Kmeans module was validated in simulation (was able to converge on two red circles in subsequent test images with different random initializations), but we were unable to fully get it working on hardware before the project deadline. Regardless, the relevant modules and the new top-level for FPGA_1 are provided in our Github Repo in a folder called FPGA_1_KM.

C. UART Communication & Combining Centroids

On FPGA_1 the 17 bit result of the camera stack is then sent to the awaiting FPGA via the UART_Transmit module from the labs. To do this we created a UART_Wrapper that takes in 17 bits and sends it in 3 bytes. Each byte's top two bits are used for alignment (Byte 0 has top two bits 00, Byte 1 has 01, etc) so the receiving system, whether it be a computer for debugging or the other FPGA, can window the received bytes appropriately.

There is also a pseudo-handshake protocol implemented where FPGA_1 will not begin to transmit unless FPGA_2

sends it a particular byte (0xAA) to signal the start of a transmission. This is also to help with the alignment of bytes that are being received. So that FPGA_1 does not start transmitting bytes when FPGA_2 is not ready and that FPGA_2 can start windowing and arranging bytes on the clean start of a 3-byte transmission.

FPGA_2 receives the 3-byte transmission via uart_rx_wrapper a similar wrapper of UART_Transmit. The wrapper waits for three bytes and ensures that the transmission's alignment bits are correct. If the alignment bits are not what it expected, the module waits until it sees another byte with top bits 00 to start building the 17 bit centroid coordinate again.

The combine_centroids module merges 2D coordinate data from two sources: local calculations (x, y) and UART input (y, z). Using a two-state FSM (WAIT_FOR_INPUTS, AVERAGING), it synchronizes and processes incoming data. In WAIT_FOR_INPUTS, the module independently latches coordinates when local_valid_in or uart_valid_in signals are received, allowing new data to overwrite previously latched values. Once both sources have provided valid data, the module transitions to AVERAGING state, where it computes the mean of the two y-coordinates using the non restoring divider module. When division completes, the module outputs the combined 3D position (local x, averaged y, uart z) with a valid signal, then returns to WAIT_FOR_INPUTS state, resetting ready flags for the next coordinate set.

D. Interpolation

With just the centroids displayed the output can look quite disconnected when the tracked object is moved across the frame. To amend this problem we decided on interpolation between the centroids. This module takes in the coordinate outputs from the combine_centroids module and its ready signal. Its outputs are coordinates in between the previous centroid and the newest centroid that are to be displayed.

In the case of the first centroid, the module only keeps track of it to keep it as a reference centroid for the next centroid output. No interpolation happens when receiving the first centroid. After this point is received each centroid received afterwards will be the target destination for our output coordinates in the module. This is accomplished by incrementing each output by one in the direction of the target destination. Once the output coordinate is equal to the destination coordinate the interpolation is done. The destination coordinate will then become the previous coordinate and the process can repeat from there.

This is not a good interpolation technique if we are trying to interpolate between points that are very far apart because the line may not have the slope that would be expected. We do not come across this problem in our scenario because the centroid output rate is fast enough that the points are never very far apart. This can give the viewer the illusion that there isn't even any interpolation between the centroids.

The centroid inputs into this module should be coming in at around 25 per second. This is slow enough that our interpolation will be done before another centroid is found. We do not need to worry about the case when we receive a new input before finishing.

III. Trajectory Memory & Display Pipeline

There are two modes DRAWING and REVIEWING which are controlled by sw[0] on the FPGA. In DRAWING mode, as the object moves its trajectory gets drawn on the display and in REVIEWING the trajectory can be rotated. The combine_centroid module outputs three 8 bit wide signals centroid_x, centroid_y, and centroid_z and a valid_out signal. In DRAWING mode, the concatenation of these signals are stored in our dual port point_memory BRAM. This BRAM is 26 bits wide (9 bits for x + 8 bits for y + 9 bits for z) and 2250 entries deep. Our camera outputs a frame every $\frac{1}{25}$th of a second, using this as an approximation of how fast a centroid will get output out of combine_centroids and with an aim to store about 90 seconds of a trajectory, we chose to store 2250 points in our BRAM.

The addressing logic (could have been separated into its own module) in top_level handles the drawing and reading addresses for storing and retrieving points. This logic manages two independent address counters for the two ports of the BRAM. Port A sequentially stores incoming centroids and interpolated points, incrementing the write address and the total point count when the interpolation module is active. After reaching 2250 points, the write address wraps back to zero, and begins overwriting old data.

The read addressing is a bit more complicated. When the first point is written (number of points increases to 1) the module initiates the first read request by raising an issued_read signal that passes through a 4 stage pipeline before use. This ensure that the write request was propagated and that the read request was resolved before triggering the rotation and projection module. Subsequent read requests are put through and the read address is increased when the rotation and projection module signals that it is done. This prevents throwing away data by only increasing the address whenever the consumer is ready for new data. The read address resets to 0 when it either reaches the end of the BRAM, the display mode is switched, or the angle is changed in review mode.

The output of the rotation and projection module is then written to a 320x180 frame buffer BRAM that is 57600 entries deep and one bit wide. The HDMI signal generation modules from the labs are then used to read from this BRAM and create the appropriate HDMI signals. This frame buffer is cleared on reset and whenever the angle input is changed. This is done by just cycling through every address in the BRAM and writing 0s to it.

IV. Rotation Controls

A. Sine and Cosine Calculation

The angles for the display's vertical-axes rotation are controlled using the FPGA's btn[1] to increment and btn[2] to decrement. The angle is a 32 bit number using the binary angle measurement system. In this system, the angle in degrees is

represented by $b \times \frac{360^\circ}{2^{32}}$, where b is the value of our binary angle. We are selecting the current axis of rotation using sw[1] and sw[2] on the FPGA. The speed of our angle selection will complete a full rotation in around 16 seconds. This can be changed via a controlled parameter. We also plan to make some set angle configurations that can reset the frame to different perspectives.

These angles are then used in the CORDIC module. The CORDIC algorithm works as a binary search algorithm that finds an angle with approximately the same slope as the desired angle. This can be accomplished by keeping a table containing $\arctan(\frac{1}{2^{-i}})$ for i from 0 to 30. Using these values a binary search can be conducted to find the approximate angle. An X value and Y value are initialized before the algorithm starts. Y is set to 0 and X is set to 2^{15} to represent a starting angle of 0. After the algorithm finishes, the output will need to be shifted by 15 bits to correct for this starting value. The binary search is conducted using only bit shifts and additions. Once the search concludes the cosine and sine values will have converged to a value that is approximately 1.6468 times the correct values. This can be corrected using a combination of bit shifts and additions in order to scale the value down. More information on the CORDIC algorithm can be found here.

With the module up and running, we can currently get cosine and sine values within 0.01 of the actual value of cosine and sine. This could be improved by using more bit shifts to scale the value down from 1.6468. Combining our angle incrementing system and our CORDIC module, we can obtain our rotation matrices for the projection of our system.

As of now, the module only takes in an angle and a set value of X and Y. We would like to alter the algorithm to take in custom X and Y values that will allow us to bypass a later multiplication. The plan is to use our coordinate as an input and allow its values to be rotated using CORDIC. We will need to scale the input by 2^{15} like before, but we will shift the value back down for our output that bypasses the multiplication step.

B. Rotation & Projection Module

The rotation is achieve by taking the cosine and sine output from our CORDIC module and combining it with the xyz coordinates we have combined from our two FPGAs. First, the coordinates are shifted to the perspective that the center of the 3D space will be the origin. As is the system sees the corner of the 3D space as the origin. We can make this correction by subtracting 160 from our width values and 90 from our height value. The module also implements the rotation matrix multiplication to rotate the values buy the set angle. This can be thought of as multiplying by

$$R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

This matrix controls the vertical axis of rotation. After performing the matrix multiplication the module shifts the output by 15 bits. This corrects the scale of our cosine and sine output from the CORDIC module. After the rotation has been calculated, values are shifted back to the original perspective with the origin in the corner of the 3D frame. This is done by adding 160 and 90 to the width and height values respectively.

If we are able to improve the CORDIC algorithm, we will be able to bypass this entire module and use just the CORDIC module to control our axes of rotation.

Initially, we thought we needed to divide by the depth component of our cooridantes to project onto the screen but this caused many issues and fine-tuning of constant multiples. In reality, we didn't need to do this as we're looking at a 320x180x320 space from a 320x180 viewport. So, the projection is just taking the x, y values.

V. Evaluation

Qualitatively, the latency of our system is very low. When drawing, we are able to see the points fairly quickly and there is minimal lag between a movement and the corresponding drawing. Our camera produces new frames every 1/25 of a second and our logic is much faster than that.

On our peripheral FPGA we used only one RAMB18 and 2 DSP blocks. Our processing and drawing FPGA used 5 RAMB36, one RAMB18, and 6 DSPs. This could be optimized by having one unified buffer for the point memory and the frame buffer. Instead of writing into a frame buffer, we could rotate all the points in place (in the same BRAM) and key in with an hcount and vcount when getting hdmi signal. This is just a rough idea, and needs more thought into the scheme for keying into point data. As it stands there isn't much optimization to be done, the BRAM limits are directly tied to just how much data you want to store.

Clocked at 200MHz, we must make sure we meet all timing constraints at double the clock speed than we used in lab. Our system was able to do this with a WNS of 0.629 on FPGA_1 and 0.295 on FPGA_2.

Our final design is able to use both cameras to track an object and draw the trajectory on the screen. We are also able to rotate around the vertical axis and see the depth data we collect, giving us a better view of the 3D trajectory. We were able to meet the requirements for our commitment level project. We made progress on some of the ideal goals like our K-Means, but were unable to debug the hardware in time. With minimal changes we could definitely include rotation controls about all 3-Axis, and perhaps drawing in different colors.

A video of a recorded trajectory being rotated (before interpolation was implemented) can be found here.

VI. Insights

Working through this project we were able to come up with a few insights:

- UART modules seem like sure you can just change the modules to send X bits instead of a byte, but this will leave you unable to debug on your computer. Instead, just send things in byte sized chunks so you can actually test things. Also, windowing and aligning multiple transmission is really important. You need some synchronization

mechanism, your sending could be going fine but you may just be receiving/windowing wrong and you spend time looking for the issue in the wrong place :D.

- We were worried about the centroid parts coming in from the local and the uart not being synchronized and pairing together XYs and YZs from wrong times, but it turned out to not be that big of an issue. We think making the peripheral FPGAs hardware fairly light made this not that big of an issue.
- The CORDIC algorithm turned out to be incredibly efficient and saved us from using a lot more resources in order to store cosine and sine values. It is a very well pipelined and pretty accurate algorithm that allowed us to trust its outputs were fast and correct.
- Figuring out a way to visualize hardware outputs before your visual pipeline is done is really helpful. We just did UART to our laptops and displayed points in a pygame window to validate stuff on hardware after simulation.

VII. Contributions

Jack worked on the angle pipeline and interpolation between points. Chris worked on the point collection, transfer between FPGAs, display, and the KMeans pipeline even though it didn't end up working on hardware :(. Both authors wrote the evaluation, rotation/projection section, and the conclusion. In the report, Jack wrote about the interpolation and the sine/cos calculation via CORDIC. Chris wrote about the camera pipelines, communications, and display pipelines. Code can be found here.

VIII. Acknowledgments

We would like to acknowledge Lucas DeBonet. While he was not able to complete the project with us, he helped us brainstorm and come up with the idea of our project. We would also like to thank our mentor Kiran Vuksanaj for providing insight and helping with the design and implementation of our system. And one last thank you to Joe for teaching one of the best classes at MIT.