Quantum Juggling Trainer

Yue Chen Li, Toya Takahashi, Victoria Nguyen
Massachusetts Institute of Technology
77 Massachusetts Ave, Cambridge, MA 02139
yuecli@mit.edu, toyat@mit.edu, vkn@mit.edu

***Abstract*—This paper presents the design and implementation of Quantum Juggling Trainer, a system that aids users in practicing juggling patterns specified in Siteswap notation. Leveraging FPGA technology, the trainer processes real-time video input to track the trajectories of juggling balls and assesses the accuracy of the user's performance against a specified pattern. This paper discusses the technical challenges and design trade-offs in implementing the system, as well as ideas for future improvement.**

1. Introduction

Juggling is a skill that requires precision and timing, which can be improved by practicing complex patterns. By tracking ball trajectories and comparing them against patterns specified in Siteswap notation (Section 2.1), the Quantum Juggling Trainer provides real-time feedback to help users improve their juggling technique. We previously implemented live k-means clustering to track juggling in Python, and found the video feedback to be too slow. Hence, we decided to use the FPGA to efficiently track the juggling pattern in addition to evaluating it in real time.

Our design has three key components:

1. Pattern generation (Section 3)
2. Object tracking
   (a) Preprocessing (Section 4)
   (b) K-means (Section 5)
3. Pattern evaluation
   (a) Trajectory Simulation (Section 6)
   (b) Pattern evaluation (Section 7)

In section 2.2, we present a more in-depth description of the design. Moreover, in each of the sections listed above, we discuss the specific implementation. Finally, we present an overall evaluation of our results in section 8.

2. Background (Victoria)

In this section, we will explain the necessary background information and key details of our overall design.

2.1. Siteswap

Siteswap [1] is a notation for describing juggling patterns by encoding the sequence and timing of throws and catches. Each number in a siteswap pattern represents the number of beats a ball remains in the air before being caught and rethrown; hence, higher numbers correspond to higher throws. Odd numbers indicate throws that cross to the opposite hand, while even numbers represent throws that stay within the same hand. The pattern of throws described by a sequence can be looped indefinitely. For example, a "531" corresponds to a trick with three types of throws: a high throw (5), a low throw (3), and a horizontal pass (1). This notation provides a concise way to analyze and invent juggling patterns.

2.2. Overall Design

The block diagram for our overall design is shown in Fig 1, and an example video output is shown in Fig 2. The data flow is as follows: first, the input pattern for the trainer is specified using switches and buttons. The number of balls and preprocessed camera data (which generates the pink mask for the hands and cyan mask for the balls) are inputted to Object Tracking. This module then outputs the locations of the balls and hands (which appear as red and green crosshairs, respectively). The positions of the hands and pattern input, are passed to Trajectory Generation, which outputs a calculated trajectory to the standard video mux (which appears as moving red squares, rooted at green squares indicating initial hand coordinates), as well as the model coordinates of the balls to Evaluate Pattern. The number of balls and real coordinates from Object Tracking are also passed to Evaluate Pattern. Then, Evaluate Pattern compares the model and real coordinates to determine correctness. If the user is juggling correctly, the screen's border turns blue; otherwise it turns red (no juggling occurs in Fig 2, so the border is red).

2.3. Technical Challenges

One technical challenge is in meeting resource and timing constraints when implementing the k-means algorithm.

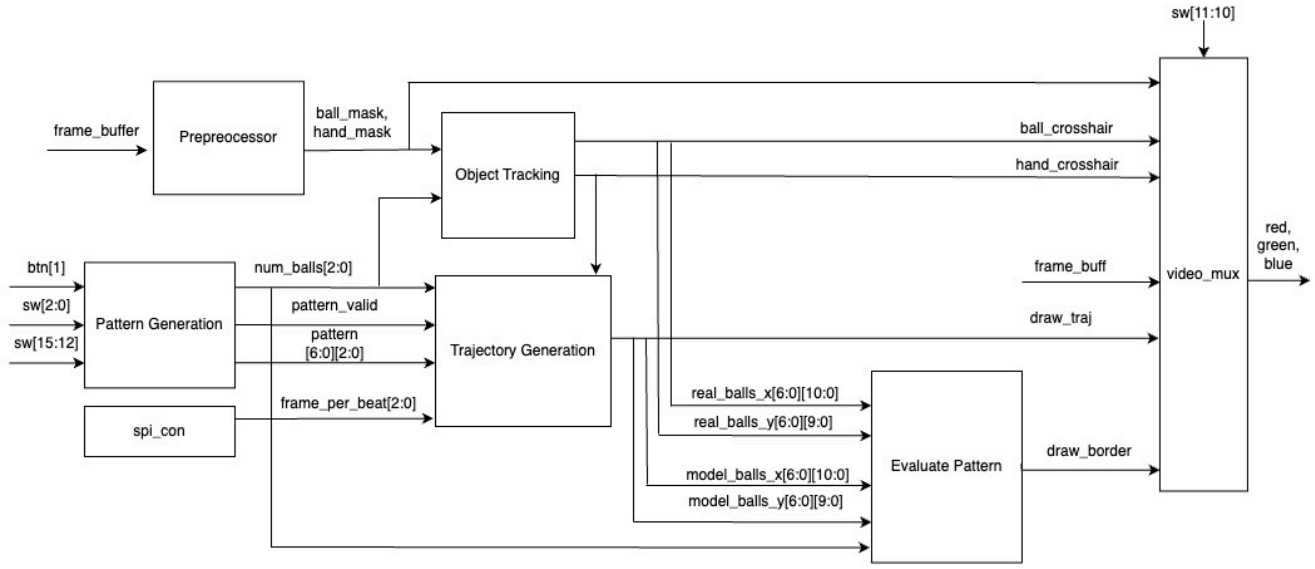

Fig. 1. Overview Block Diagram

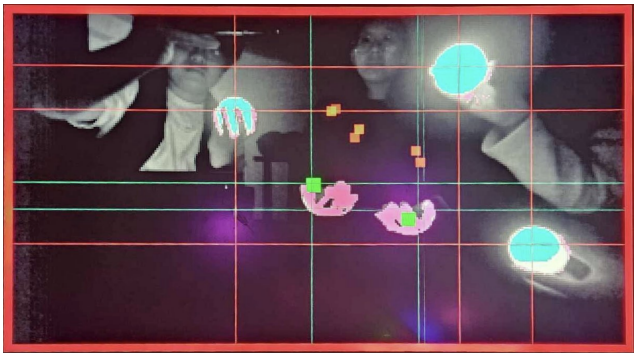

Fig. 2. Video output of Quantum Juggling Trainer: The balls are indicated by the cyan mask with a red crosshair tracking each one. The hands are indicated by the pink mask with green crosshairs tracking each one. Additionally, green squares indicate the initial position of the hands, which serve as endpoints for the trajectory, which is indicated by the moving red squares. The red border indicates the observed juggling is incorrect.

The algorithm requires us to store the whole frame across multiple iterations of centroid updates, and it takes a long time to process all the pixels in each iteration.

Additionally, we had to define a strategy for pattern evaluation. For example, we needed to determine a criteria for how many frames a pattern could deviate before being declared wrong, or how to judge the closeness of two trajectories. Algorithmically, we anticipated this would involve complex logic, and also careful tuning of various threshold parameters.

In the remainder of the paper, we explain our design choices and discuss workarounds.

3. Pattern Generation (Toya)

The pattern generation module outputs a valid siteswap pattern for downstream modules, switching between two primary states: the `INPUT` mode, where it accepts user input from the switches, and the `VALIDATE` mode, which checks the validity of the user-specified pattern (as detailed in 3.1). The input pattern is displayed on the seven-segment display, and if the pattern is invalid, an error message is shown. Additionally, the module calculates the number of balls required to perform the juggling sequence encoded by the validated pattern. This is done by averaging the throw numbers, which are combinationally summed and passed to a 6-bit divider module with constant latency. The output logic is pipelined to account for the division delay, ensuring synchronization of results.

3.1. Pattern Validation

Pattern generation includes a submodule for combinationally validating an input siteswap pattern. The validity of a siteswap pattern is determined by ensuring no two throws land at the same time. This is achieved by tracking a countdown for each throw in the pattern. Consider the a valid siteswap pattern “531”:

Beats:	1 2 3 4 5 6
Siteswap:	5 3 1 5 3 1
Countdown1:	5 4 3 2 1 0
Countdown2:	3 2 1 0
Countdown3:	1 0

For the first throw, countdown1 starts at 5 on beat 1 and decreases until reaching 0 on beat 5 which is when the ball

is caught. Simultaneously, at beat 5, countdown2 reaches 0, indicating the other hand is making a throw with a height of 3. In a valid siteswap pattern like "531," no countdown value coincides with another throw's height at any beat, except when a throw is initiated, ensuring proper sequencing and avoiding collisions. However, in an invalid siteswap pattern like "542", the countdown values matches at the same beat:

Beats:	1 2 3 4 5 6
Siteswap:	5 4 2 5 4 2
Countdown1:	5 4 3 2 1 0
Countdown2:	4 3 2 1 0
Countdown3:	2 1 0

To handle any pattern with a maximum length of 7 and up to 7 balls, two unpacked arrays of length 7 are defined. The first array, `countdown`, tracks the countdown for each throw, decrementing as illustrated above. The second array, `countdown_valid`, flags whether the corresponding countdown value was compared against the siteswap pattern. The module generates a validity signal, asserting it as high only when none of the masked countdown values match the siteswap pattern at any beat.

4. Preprocessing and Video Outputs (Toya)

To maintain high-definition video while processing Gaussian blurred images for speckle noise reduction, the video pipeline is divided into DRAM and BRAM pathways. The full-resolution 720p image is stored in the DRAM frame buffer and passed to the video multiplexer as the background image. In parallel, Gaussian blur is applied twice to the image before it is downsampled and stored in the BRAM frame buffer. The downsampled image undergoes image thresholding to create a mask identifying "ball-pixels," which serves as input for the k-means algorithm to pinpoint the locations of the juggling balls.

Experimental evaluation revealed that keying on luminance values, combined with the use of glowing juggling balls, yields the most accurate results. We also found that keying on the Chroma Red values for the gloves when they were set to be red yielded accurate results for tracking the locations of the hands. Despite both items being LED, the balls were significantly brighter than the gloves, so we were able to use luminance with a relatively high lower threshold to find the balls. Similarly, since the balls were so bright, we were able to set them to be colored pink so that when they were near the hands, they helped increase the Chroma Red value of the gloves.

Finally, the masked downsampled image and crosshairs marking the ball locations are then passed to the video mux, enabling optional overlay onto the high-definition video from DRAM. We improved the mux to support two separate sets of masks and crosshairs so that the balls and hands could be colored differently for more optimal visibility.

5. Object Tracking (Victoria)

To track the objects, we take the mask from the preprocessing module and apply the k-means clustering algorithm to determine the ball locations. The module takes inputs `k` (number of balls) and the initial guess for centroid coordinates, as well as parameter `MAX_ITERS` which specifies the number of k-means iterations (tuned to 20). We instantiate two of these modules: one for the hands and one for the balls.

The module operates as a state machine (Fig 3) with four states: `STORE` (section 5.1), `UPDATE` (section 5.2), `DIVIDE` (section 5.3), and `IDLE` (section 5.3).

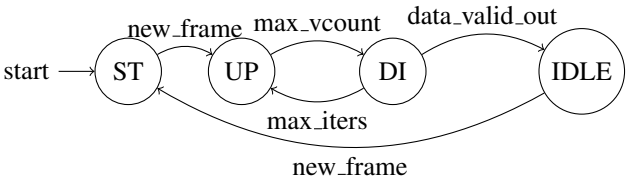

Fig. 3. K-means state diagram

5.1. Store

The module begins in the `STORE` state on reset. In this state, the mask is stored in BRAM memory at every four `hcount` and `vcount` within the frame. The memory consists of 5 BRAMs, each with a width of 64 and depth of 180, capable of storing one row of a 320×180 image across the BRAMs. With consecutive pixels in the same BRAM, it updates a locally stored 64 bits of data. Once `x_in` changes to exceed the domain of the current BRAM, the write enable signal for the current BRAM goes high and the 64 bits are cleared for the next BRAM's data. When the `new_frame` signal goes high, the module transitions to `UPDATE`.

One problem we encountered with storing the mask was that the `top_level` module would send the same pixel four times due to downsampling, which caused problems with writing the data to the BRAMs in our implementation. We resolved this by keeping track of the previous pixel so that we would only write when a new pixel came in.

5.2. Update

During the `UPDATE` state, we read back the data stored in our BRAMs to update the centroids. Although 64 pixels are read during each BRAM read request, only 2 pixels are processed at a time. For each of the pixels being processed, we calculate the distance to the current centroids. The `minimum` module then finds the minimum distance

between each pixel and any of the first `num_balls` centroids. The k-means module subsequently calculates, for each centroid, the sum of the x-coordinates, y-coordinates, and total mass of the points closest to it. These sums are used to update a running total for each centroid. Once all pixels have been updated (when the vcount equals the height), we proceed to the `DIVIDE` stage.

One problem we encountered here was the amount of operations and space it took to calculate these distances and running sums made it impossible to do entire rows, or even large chunks of them in parallel. After many attempts to maintain a small number of cycles against a negative WNS and overutilized LUT, we decided that it was infeasible to complete a sufficient number of iterations of k-means within our vertical sync period. Thus, we ended up only computing small widths of pixels in parallel at a time and dropping every other frame in order to comfortably fit each iteration of k-means within the resources available on the FPGA.

5.3. Divide and Idle

In the `DIVIDE` state, these sums are divided by the total mass to compute the updated centroid coordinates. The process iterates between `UPDATE` and `DIVIDE` for `MAX_ITERS` cycles. After completion of the iterations, the `data_valid_out` signal is set high, and the module returns to `IDLE` to process the next frame of pixels. The centroids don't change between this stage and when the module returns to `STORE` on a `new_frame` signal, and the corresponding centroids are displayed by crosshairs on the video output.

6. Trajectory Simulation (YC)

After a valid pattern is specified, the trajectory simulation module generates the locations of the balls and (to be implemented) hands frame by frame using an idealized parabolic motion model. The calculated locations were fed downstream to the video mux which overlays it onto the camera feed.

6.1. Ball Logic

Given a siteswap pattern, the first task is to determine which ball is being thrown at each beat. This was achieved by maintaining a 'queue' with that information for the next 7 beats, including the current beat. When we tick a beat, we updated this structure (1) by shifting every value right once, since the ball to be thrown at beat b is the ball to be thrown at beat $b+1$, and (2) by handling the ball that is thrown in the current beat separately. Update (2) required setting the queue at the $b-1$-th beat to the current ball.

Aside from the identity of the ball, other auxiliary information including the start time of the throw, which hand the ball is thrown from, and the type of the throw was also stored. These were maintained using arrays and wrap-on counters that cycle between the two hands and the throws in the Siteswap pattern.

6.2. Kinematics

For a given frame, the locations of the balls were calculated using kinematics equations. The x and y components of their motion were considered independently and we assumed they do not experience drag. In all calculations, we used pixels as the unit for distance and frame count as the unit for time.

$$x(t) = x(t_0) + v_x \cdot (t - t_0) \tag{1}$$

$$y(t) = y(t_0) + v_y \cdot (t - t_0) + \frac{1}{2} \cdot \tilde{g} \cdot (t - t_0)^2 \tag{2}$$

To determine $x(t)$ and $y(t)$ for a given ball using equations 1 and 2, we set $x(t_0)$ and $y(t_0)$ based on the starting hand and t_0 to the frame count when a ball is thrown. To reduce overlap between balls, $x(t_0)$ is set to $x_{hand} \pm s$, where s is a small offset parameter. We also read out the v_x and v_y for the corresponding throw from a small pre-calculated table that is detailed in section 6.3. $\tilde{g}$ is a hard-coded parameter tuned empirically.

6.3. Pre-calculations

Since a k-throw in Siteswap lasts k beats, we defined the airborne time of a k-throw to be $k \cdot frame\_per\_beat$, where $frame\_per\_beat$ is an adjustable input for scaling the height of the trajectories using a potentiometer. Changing this value allows the juggler to adjust the speed of their tosses, as well as for their standing in different positions relative to the camera. The potentiometer was implemented as demonstrated in class, though we only took the first 3 bits of the data we read because of the sensitivity of the trajectory to our effective gravity. Since we are also using a camera, we wired the potentiometer to the Pmod B ports.

For $k > 0$, we calculated the horizontal and vertical velocity components using kinematics equations. v_x is calculated from equation 4 using a divider module by fixing the horizontal distance according to equation 3. For v_y, we observed that the $v_y = 0$ at the top of the parabola to derive equation 2, which was calculated only using multiplications and a right bit shift.

$$distance = \begin{cases} x_{right\_hand} - x_{left\_hand} & x \text{ is odd} \\ 2 * offset & x \text{ is even} \end{cases} \tag{3}$$

$$v_x = \frac{distance}{k \cdot frame\_per\_beat} \tag{4}$$

$$v_y = \begin{cases} 0 & x = 2 \\ \frac{1}{2} \cdot g \cdot (k \cdot frame\_per\_beat) & \text{otherwise} \end{cases} \tag{5}$$

Note that $k = 0$ and $k = 2$ are special cases that were handled separately. While $k = 0$ was defined as a throw with no balls and the velocities' values do not matter, it could cause division by zero error in equation 4. On the other hand, $k = 2$ was defined as a non-throw in which the ball is simply held, hence v_y is forced to be 0 in equation 5. As the divider modules used in these calculations required a variable number of cycles, we pre-calculated these velocities in a separate stage before the main calculations of the ball and hand trajectories.

6.4. State-machine

Using the above parabolic motion model, the trajectory generator module was implemented as a state machine (Fig 4) with three states, `IDLE`, `INIT`, and `TRANSMIT`. `IDLE` is the reset state to clear the module's outputs or trigger a recalculation of the trajectories in module inputs (e.g. Siteswap pattern) are changed. When a valid pattern was received, the state machine transitions to `INIT`, which corresponds to the state where the calculations in section 6.3 are performed. To simplify the implementation, a separate divider was used for each possible type of throw, yielding 7 dividers. After all the dividers finished their calculations, the state machine finally transitions to `TRANSMIT`, which corresponds to the state where the calculations in sections 6.1 and 6.2 are performed. Only in this state does the state machine have valid outputs, and the state machine remains in this state indefinitely unless it is reset.

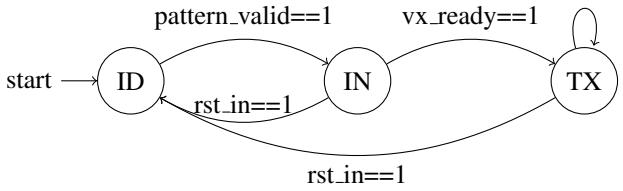

Fig. 4. Trajectory state diagram

7. Pattern Evaluation (YC)

To evaluate a user's juggling pattern, we compared it against the calculated model juggling pattern. First, we perform a frame-by-frame evaluation that checks if the error between the tracked and calculated balls is within some threshold. After that, we determine a juggling pattern to be correct by aggregating the frame-level judgments over a fixed time window.

7.1. Graph Modeling

The key challenge in this module is determining the correspondence between tracked and calculated balls. To approach this problem, we begin by formally defining the 'best' assignment of tracked balls to calculated balls as the assignment that minimizes the error of the assignment. This error is also the value that we compare to a threshold to determine if a frame is 'valid', i.e. the tracked balls are close enough to the calculated balls. To penalize big deviations for any ball and avoid needing to compute square roots, we define this to be the sum of the square of the Euclidean distances between each pair of matched balls. Then, the error can be computed by taking the sum of squared differences in x and y-coordinates for each pair of balls.

With this definition, we can now model the problem as an instance of minimum-cost maximum bipartite matching[1]. Here, the graph is a complete bipartite graph with two sets of vertices being the tracked and calculated balls; the weights of each edge are the costs of matching the two corresponding balls.

Fortunately, this is a classic problem that can be solved by the **Hungarian Algorithm**, also known as the **Kuhn-Mankres algorithm**. The outputs of this algorithm are the matching and the cost of the matching, the latter of which can be used directly[2].

7.2. Frame-level Trajectory Evaluation

In our frame-level evaluation module, we first explicitly construct the full adjacency matrix combinationally. Then, we mechanically translated a reference C++ implementation of the Hungarian algorithm [2] into SystemVerilog. As all calculations of weights and potentials in the algorithm can involve negative numbers, the corresponding vectors must be declared as signed.

The main challenge in the translation was the presence of multiple nested loops; other parts of the code can be more easily written as chunks of combinational logic. Since the algorithm takes a variable number of cycles, it was implemented as a state machine. The trick was to write each loop as one would do in assembly code. Each while or do-while loop was given labels in the C++ code for the body and the loop-condition check, and each for loop was given labels in the C++ code for the initialization, body, update, and loop-condition check. Then, each remaining unlabeled chunk of code was also assigned a label, and all the labels were converted into states. Finally, appropriate transitions were constructed based on the control flow of the C++ code. Note that an initial state INIT and final state ANS were also included. This naive translation of the algorithm yields a total of 15 states and the entire module was implemented in $\approx$ 200 lines of SystemVerilog code.

7.3. Aggregated Evaluation

Since a pattern can be interpreted as a periodic function, the user's juggling was judged to be correct if at least a cer-

[1] In fact, it must be a perfect matching!

[2] The former would have been useful too, had pattern inference been developed.

tain number of frames were correct in a fixed time window. This was achieved by storing a counter of the number of correct frames in the past N outputs of the frame evaluator. To update the counter, we used a size N queue to maintain the history. Finally, the counter was compared to a threshold number of frames to decide if observed juggling pattern replicates calculated juggling pattern satisfactorily.

8. Evaluation (Toya)

In this section we further discuss our implementation and evaluate its performance on latency, throughput, and resource utilization.

8.1. Deliverables

We delivered on our core goals stated in section 1: our project correctly reads and validate patterns, and it correctly generates trajectories from the hands positions and validate them against the path of the balls. While we originally planned to shift the trajectories to match the hand positions, our final implementation allows the user to fix hand placement by pressing a button. This was a practical decision motivated by the fact that a juggler's hands would ideally circle around fixed points, and has the advantage of being computational cheaper. Moreover, we also found our initial goal of fitting the entire algorithm into one vertical sync period to be infeasible, leading us to drop every other frame for k-means.

8.2. Latency and Throughput

Since the results of our modules were visualized in every frame, we evaluated latency and throughput based on the number of frames required for calculations to complete. The primary bottleneck is the `UPDATE` and `DIVIDE` states of the k-means module, which take 2 frames to perform 20 iterations for convergence. During the `UPDATE` state, the module requires 35 cycles per iteration to determine the closest centroid for each pixel and sum the x,y, and total masked pixel counts, processing all 5 BRAMs across 180 rows. The `DIVIDE` state then uses 14 cycles per iteration to compute the means for each cluster. With 20 iterations, these steps total $(35{\cdot}5{\cdot}180{+}14){\cdot}20 = 630{,}280$ cycles, well within the full frame duration of $1650 \cdot 750 = 1{,}237{,}500$ cycles. Since we output video at 60 fps, the k-means modules track the balls and hands at 30 fps. While more k-means iterations could fit within 2 frames, empirical testing in Python confirmed that exceeding 20 iterations does not significantly improve results.

8.3. Resource Utilization

BRAM We utilized 51.5 out of 75 BRAM tiles (68.7%), with 24 tiles dedicated to the camera frame buffer for storing the downsampled 320×180 image. Thus, the most effective way to reduce BRAM usage would be to exclusively use DRAM for image storage. This change would require moving the Gaussian blur module to after the unstacker and adding pipelining to align the blurred image with the original camera feed. Additionally, a smaller optimization could save 5 additional BRAM tiles by sharing the BRAMs used for k-means between its instantiations for the balls and the hands.

LUT We utilized 58.04% of slice LUTs and 17.92% of the slice registers. Significant contributions came from the DDR3 MIG, k-means, pattern evaluation, and trajectory generation modules, each adding approximately 4,000 out of 32,600 logic LUTs. Initially, we attempted to perform k-means during the display's vertical sync period by summing all 64 BRAM values in parallel. This required storing $\mathrm{clog}_2(\sum_{n=256}^{319} n) = 15$ bits for the sum and $\mathrm{clog}_2(64) = 6$ bits for the total number of masked pixels across 7 clusters. Additionally, to compute the sum combinationally, we stored intermediate binary sums in a 64-element array, resulting in $64 \cdot 7 \cdot (15 + 6) = 9{,}408$ bits. In addition, 64 instantiations of the minimum module was required to find the closest centroid for each pixel. Together with the large number of bits used for storage, this caused LUT overutilization errors. To resolve this, we spread the k-means iterations over two frames and limited the summation to two pixel values per cycle.

DSP We utilized 98 out of 120 DSPs, with the majority allocated to the pattern evaluator module (72 DSPs) and the trajectory generator module (21 DSPs). The pattern evaluator required DSPs to perform 49 parallel 11-bit multiplications and 49 parallel 10-bit multiplications when computing squared Euclidean distances between pairs of juggling balls. Although Manhattan distance would have reduced DSP usage, we chose Euclidean distance for its higher sensitivity in pattern evaluation, which is critical compared to other modules like k-means that can effectively use Manhattan distance. To address DSP overutilization, we restructured the k-means module, replacing multiplication-based calculations for the summed x and y values with simpler addition, which increased LUT usage but alleviated DSP constraints. This adjustment effectively balanced resources. The trajectory generation module, requiring 21 DSPs, used them for computing uniformly accelerated motion equations.

8.4. Timing Requirement

WNS Our final build achieved a WNS of 0.292 ns, with the critical path located at the camera registers. While this indicates potential for more aggressive use of combinational logic per cycle, such optimization would not provide significant benefit, as our throughput is constrained by the video frame rate.

9. Reflection

Although we were able to achieve our commitment and goals for the project, there were some aspects of the project that we were not able to complete. Thus, some avenues for further improvements include:

- **Pattern inference**: Our main stretch goal which we were unable to accomplish was pattern inference, i.e. given the observed juggling, the FPGA determines what pattern is being juggled. Unfortunately, we faced unexpected challenges in managing resource utilization, making that goal outside the scope of this project. However, we note that our current implementation of pattern evaluation accomplishes half the task of this goal by accumulating approximately enough data from the juggling to judge the pattern, and we could build upon it to actually infer the pattern instead.
- **Potentiometer**: Another point of improvement would be our use of the potentiometer. Due to us requiring an integer number of frames per beat and the limited number of pixels per frame, we did not have a lot of flexibility in the frames per beat as determined by the potentiometer. We feel it would be a more effective use of the potentiometer if we allowed for floating point frames per beat, which would facilitate a more fine grained tuning of the juggling height/speed for the user.

Additionally, there are many aspects of digital system design that we learned throughout the development of this project:

- **Resource management**: One of our greatest struggles was managing our LUT and DSP usage. This was particularly emphasized during the implementation of k-means, when we had to concede to dropping alternate frames. We were eventually able to develop a better intuition on how much is too much; for example, it is infeasible to process 64 pixels at once in k-means. This also taught us a lesson in optimization, as we had to develop more efficient algorithms to handle repetitive tasks or dodge multiplication.
- **Test cases**: one lesson that this project really emphasized was the importance of good test cases. Our specifications left a lot of opportunities for bugs, which many of our initial test cases did not catch properly. For example, our original k-means test case did not properly consider the context in which the algorithm was being used in. Rather than downsampling an image to feed pixels from, we directly fed in pixels from an image of reduced dimensions, which left us oblivious to the bug described in section 5.1.

10. Contribution

We detail the contributions from each of the members below. All the members contributed in the initial planning of the design as a whole, in particular the timing and memory details given in the preliminary project design.

- **Yue Chen Li**: YC was responsible for coding the trajectory simulation (section 6) and pattern evaluation (section 7) modules. As the original inspiration for the project, he was also responsible for testing and demoing the project by juggling, as well as giving insights into the reasonableness of its evaluation criteria.
- **Toya Takahashi**: Toya was responsible for coding the pattern generation (section 3) and preprocessing (section 4). The first required him to further research Siteswap notation in order to implement the state machine. He additionally played a large role in debugging the k-means module and negative WNS.
- **Victoria Nguyen**: Victoria was responsible for coding the k-means (section 5) module as well as reading the potentiometer input into the trajectory module to achieve a reasonable frames per beat. She additionally edited together the final video.

11. Thanks

Finally, we would like to thank all the people who helped us along the way:

- Our instructor, Joe Steinmeyer, for his lectures, lab code, advice on our project, and teaching 6.205
- Kiran for advice, especially for tracking objects
- All the TAs and LAs who helped us debug in OH
- German House for lending us their TV and Shin for lending us his monitor

Source code

References

[1] Siteswap Notation Used in Juggling Lab. [Online]. Available: https://jugglinglab.org/html/ssnotation.html

[2] Hungarian algorithm. [Online]. Available: https://cp-algorithms.com/graph/hungarian-algorithm.html