Hand Gesture Controller
Final Report

1st Tasmeem Reza
MIT EECS
Cambridge, MA, USA
rtasmeem@mit.edu

2nd Steven Reyes
MIT EECS
Cambridge, MA, USA
steverey@mit.edu

***Abstract*—We propose a system that tracks hand gestures and converts different gestures into different commands to control a PC, implemented on an FPGA with an attached camera. The hands are tracked with colored tags on fingertips, and the locations of the tags are determined via a multi-iteration k-means-clustering algorithm on a real time video feed. The FPGA is able to perform as much as 16 iterations of the algorithm for one frame of latency, utilizing techniques such as downsampling and parallelized computation to speed up calculations and reduce memory usage. Theoretically, the number of clusters shown, their distances, and their movements can be interpreted as different gestures to control some downstream media corresponding to certain commands and movements, e.g. numeric input, scrolling, and zooming. The command determined from the gesture can be transmitted to a PC via the PS/2 protocol. Through this project, we aim to demonstrate the feasibility of computing cluster means in real time, and that computer interaction is possible in the use case of numeric input.**

***Index Terms*—digital systems, field programmable gate arrays, cameras, object tracking, clustering, k-means, gestures**

I. Introduction (Steven)

With a working camera feed that gets displayed on an external monitor via HDMI, the main challenge of implementing hand gestures on an FPGA is the k-means clustering algorithm. This algorithm allows for finding the locations of cluster centroids by performing multiple iterations of cluster reassignment and cluster centering, provided that the number of clusters k is given. At each iteration, each point in a cluster is reassigned to the centroid closest to it, and after all points are processed, each centroid is repositioned to the center of mass of all points assigned to it. The difficult in implementing such an algorithm arises from the limited time and resources to do the calculations. The FPGA has limited memory, so it becomes necessary for the video data received from the camera to be immediately displayed to the screen. Due to this, the computation of the cluster locations has to be done within a limited timeframe, so that the latency between the video data and the cluster location data does not become noticeably high. This means the algorithm must be completed in the time it takes to display one or two frames of video. Furthermore, memory and computation constraints limit how much data is stored from previous frame, as the algorithm requires storing frame data to perform multiple iterations on the current frame.

Besides the constraints posed by the FPGA, the k-means algorithm may also get stuck in a local minimum, where it would not find the correct clustering. Two distinct clusters may be assigned to the same centroid if they are close enough, and if there are more centroids than clusters, some clusters may not be assigned any points, and so their locations cannot be updated and will always be too far from the actual cluster points. Additionally, the number of clusters present may be variable, as different gestures display different numbers of fingers shown.

To resolve these concerns, different techniques are employed. The video is downsampled in order to reduce the memory footprint of frame data to only 4 BRAMs. Multiple pixels can also be fetched from a BRAM at a time by storing them in rows of 2 pixels, halving the amount of time needed per iteration. For issues specific to k-means, it is augmented slightly by repositioning clusters with no assigned points onto points that are furthest away from their assigned centroids. Furthermore, clusters can be merged together if their distances are close enough within some threshold, so that a variable number of clusters are permitted.

II. System Overview (Steven)

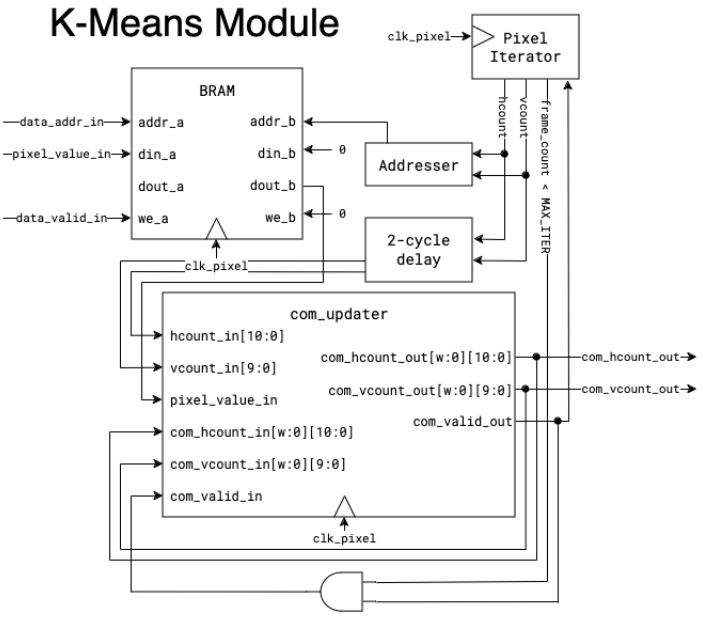

Fig. 1. Block diagram for the k-means clustering algorithm module.

A block diagram of the k-means algorithm module is shown in Figure 1. The first step is to store the pixels in the mask that are part of the clustering process into BRAMs, as the algorithm refers to it over multiple iterations. There are two separate sections of memory, where one is written to with fresh mask pixels of the current frame, and the other is read to perform the actual k-means algorithm on. Only the reading memory module is shown in the block diagram. Whenever the HDMI protocol signals a new frame, the roles of each memory module are swapped.

The pixels are downsampled by 4 times in both dimensions, so that there are 16 times fewer pixels to process through per iteration. This mainly allows multiple iterations of the algorithm to be performed in the time span of reading and saving one frame of pixels. A Python simulation of the algorithm shows that only 5 iterations are needed until centroids converge, while this method allows at most 16 iterations to be performed in a single frame.

The main driver of the algorithm is the pixel iterator module that sequentially generates the pairs of coordinates (`hcount`, `vcount`) of all down-sampled pixels, and makes a read request for each one. The fetched values as well as the properly pipelined `hcount` and `vcount` values are fed into the `com_updater` module, which determines the assignments of clusters and accumulates running totals to be divided later for obtaining the center of mass coordinates. After all the values have been fetched from memory, the pixel iterator pauses and waits for the `com_valid_out` signal, indicating that the `com_updater` module has finished performing division. The video signal generator also keeps track of how many iterations have been performed so that it stops issuing new read requests when the maximum has been reached. At this point, the centroids' locations are properly outputed with a valid out signal.

From this, the centroids are merged together, as explained later in the paper, to determine the correct number of clusters. With ten fingers, the number of clusters could be anywhere from 0 to 10. This number is then used to issue numeric input commands to a computer via the PS/2 protocol, which allows simulated keyboard presses to be transmitted. A state machine is maintained to keep track of the current digit, where 0 clusters indicate the start of a digit, and 1 to 10 each correspond to a digit to be typed (with 10 clusters for the digit 0). The input commands are only sent at every state transition from 0 to another digit. This allows any string of digits to be typed.

III. K-Means Augmentation (Steven)

There are two main modifications necessary to make the k-means algorithm more reliable: repositioning centroids with no assigned points and increasing the number of clusters. When implemented naively, the algorithm has a tendency for centroids too far from the points' locations to not be assigned any points at all. In this case, there is no new valid position, since the newly calculated coordinates would require dividing 0 by 0. If the centroid's position is not updated, however, future iterations might not assign new points to it, as the other centroids get closer and closer to the actual points.

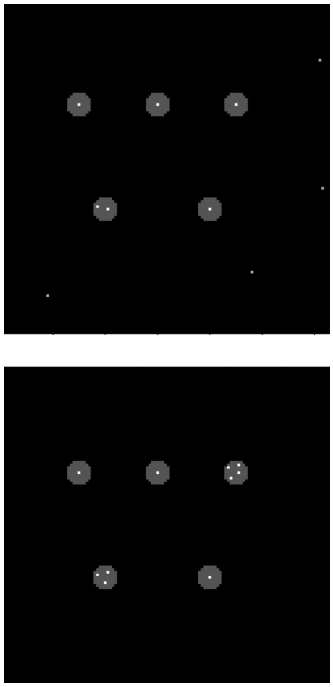

Fig. 2. Repositioning empty centroids, before (top) and after (bottom) modification, with the grey area representing cluster points, and the white points representing centroids. The floating centroids around the sides in the top figure are repositioned to have assigned points in the bottom figure.

To resolve this, the centroids with no assigned points are repositioned on top of the point that has the maximum distance from its assigned centroid, which is the method that the k-means algorithm of the `scikit-learn` library uses to resolve empty clusters (1). In practice, this resolves the issue of empty clusters, as shown in Figure 2.

One other issue that arises is that it's still possible for centroids to be responsible for multiple clusters, as shown in Figure 3. Increasing the number of centroids resolved this problem, so that each cluster has at least one centroid. This solution was specifically chosen, since clusters are also to be merged together eventually to account for a variable number of clusters. In this system, the number of centroids was fixed at $k = 16$. After implementing these two changes, every cluster consistently gets assigned at least one centroid in the Python simulations.

IV. Memory Footprint (Tasmeem)

Our k-means algorithm makes extensive use of BRAMs. Since we are downsampling the image by a factor of 4, the size of image that we need to store is 320×180 for a 720p resolution. This requires us to store $320 * 180 = 57600$ bits of data since we only need to store a 0 or 1 bit for each pixel. Unfortunately, a single BRAM can only store 36 kBits of data, whereas we need roughly 57 kBits. Hence, we decided to use two BRAMs store the mask for the entire frame. In our design,

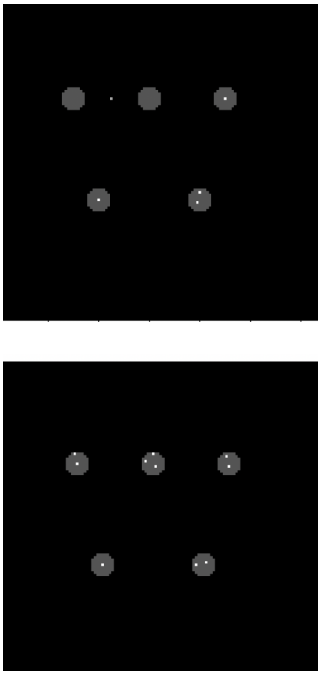

Fig. 3. Increasing centroid count, before (top) and after (bottom) modification. Here, the number of centroids are increased from 5 to 10.

this is abstracted away as `frame_memory` module which has the capacity of two BRAMs.

The k-means algorithm needs to read the entire frame each iteration, and our implementation performs around 16 iterations per frame. Therefore, we instantiate two `frame_memory` modules. One module stores the mask from the previous frame and is kept stable (no writes are issued to it for the current frame). The mask from the current frame is written to the other `frame_memory` module. The k-means algorithm always operates on the previous frame because the data is kept stable for the entire frame. When the controller module receives signal for a new frame, it switches the roles for the two `frame_memory` modules, as such, the `frame_memory` module that stored the previous frame now writes data from the current frame, and the module that wrote pixel data from the current frame is now the module has stores data from the previous frame. The idea is similar to rolling buffers. The whole process requires 4 BRAMs in total.

One important detail that is worth mentioning, the BRAMs we instantiated in the `frame_memory` module has a depth of 2 bits. So every read instruction fetches two consecutive bits of the image.

V. K-Means Iteration (Tasmeem)

The system allows for 16 iterations of k-means per frame, and testing has shown that 5 iterations are needed for convergence. However, the centroids of the current frame being processed are initialized with those of the previous frame. In practice, only one iteration is needed to stabilize the centroids to the current frame's data.

In each cycle we can read two consecutive bits of the image. We can use this to our advantage because k-means algorithm can operation on different parts of the image and we can later merge the information to calculate the clusters for the entire image. Hence, we have two `accumulator` modules that computes $\sum_{\text{pixel}(x,y) \text{ belongs to cluster}j} |x - x_j^{\text{cluster}}|$ and $\sum_{\text{pixel}(x,y) \text{ belongs to cluster}j} |y - y_j^{\text{cluster}}|$ for all clusters j. One `accumulator` module operates on the even pixels and the other one operates on the odd pixels. Then finally we merge the results from the two `accumulator` modules to calculate the final coordinates of the clusters by computing their center of masses. This allows us to compute one iteration of k-means algorithm in roughly $\frac{320 \times 180}{2} = 28800$ cycles. Each frame takes $1650 \times 750 \approx 1.2$ million cycles in the HDMI. Even if we perform 16 iterations on each frame, it will only take $16 \times 28800 = 0.46$ million cycles. So we can comfortably finish calculating the center of masses before the start of the next frame.

The center of mass calculation requires a division operation module to divide the total sum of coordinates by the total counts for each cluster. The given module from lab assignments does repeated subtraction of the divisor from the dividend, which can be too slow if the dividend is as much as 320×180, and the divisor is only 1, taking on the order of tens of thousands of cycles. The implementation was improved to a base-2 long division algorithm, where each cycle determines one bit of the dividend, MSB first, taking only 32 cycles.

VI. Cluster Merging (Tasmeem)

Since we are counting the number of fingers, we do not know the exact number of clusters we would have in our image. Our idea to solve this problem is to run k-means algorithm on 8 clusters. Since we have 5 fingers, we can say with high certainty that every cluster will be assigned to one finger.

Now we essentially have a graph theory problem, where we need to construct a graph such that cluster i and cluster j has an edge if and only if they were assigned to the same cluster. Once we create this graph, we can simply run any graph traversal algorithm to find the number of connected components which should be the same as number of fingers on the frame.

The graph construction is based on the following heuristic. If cluster i and cluster j are assigned to the same finger, there must be a pixel which is close to both clusters. In other words, we define a pixel (x, y) is *noisy* for cluster i and cluster j if

$$|d((x,y),(x_i^{\text{cluster}}, y_i^{\text{cluster}})) - d((x,y),(x_j^{\text{cluster}}, y_j^{\text{cluster}}))| \leq 1$$

where d is the Manhattan distance metric function. For each pixel, we can easily find i and j by computing the closest and second closest clusters from it. This can be implemented as a reducer in the register transfer level (RTL). Hence we can construct the graph by iterating through the pixels in the BRAM one more time and for each noisy pixel adding an edge between the closest and second closest clusters.

Once we have constructed the graph, we find the number of connected components by a simple $O(k^3)$ algorithm where k is the number of clusters (also the number of vertices). Since k is small, this does not pose any overhead for us.

VII. PS2 Module (Steven)

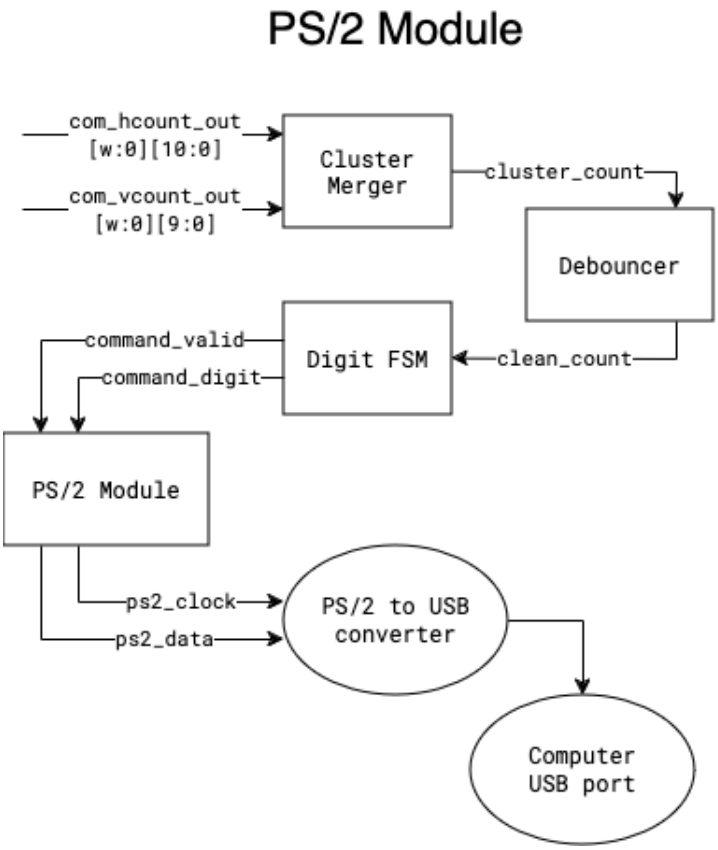

Fig. 4. A block diagram for the pipeline to convert cluster locations into a command corresponding to typing a digit of the number of clusters after merging. The rectangular shaped modules indicate FPGA modules, while the circular ones indicate physical hardware.

A summary of the pipeline for determining and sending the command through PS/2 to the computer is shown in Figure 4. For the purposes of typing a digit based on the number of clusters in the video feed, a glove is modified so that a small piece of pink paper is taped onto each finger tip. When the system is configured to only detect pink pixels, the number of fingers shown to the camera corresponds to the number of clusters the FPGA can count.

However, since the pink squares may have different levels of illumination, the raw number of clusters after merging can still flicker between numbers adjacent to it. To prevent this, the count is fed through a debouncer module to get a clean signal of the number of clusters detected, with the threshold debouncing time set to 13 milliseconds. Afterwards, the count is sent over to a finite state machine to interpret the series of cluster counts as keyboard press commands. Since we have 11 possible states, corresponding to 0 to 10 clusters, the 0 cluster is designated as the start of a new digit to be typed, while the other 10 correspond to the end of typing of its corresponding digit, with 10 clusters corresponding to digit 0. Then, the digit typing command is sent only on state transitions from 0 clusters to some number of clusters. This allows for the typing of the same digit consecutively, instead of only typing when digits change. Furthermore, this also prevents further flickering of digits not caught by the debouncer, so that any change between two nonzero cluster counts do not send a command.

When the digit is determined, the digit is converted into its corresponding scane code. This scan code consists of two parts: the make code and the break code (2). The make code corresponds to the key being pressed, while the break code corresponds to the key being released. Specifically, the make code consists of a byte of data, while the break code corresponds to two bytes, the byte `0xF0`, and then the same byte as the make code. The three bytes are sent with a number of cycles separating them, with each needing 11 bits to communicate, the start bit of 0, the byte itself, an odd parity bit, and the stop bit of 1. These bits are sent on a slower clock of around 13 kHz, which is a speed compatible with the PS/2 protocol. With the FPGA running at around 74.25 MHz, the PS/2 module maintains an internal counter looping every $74,250,000/13,000 \approx 5,700$ cycles, with half the time being low and half the time being high. This simulates the clock signal being sent to the PS/2 clock cable. The protocol samples at every falling edge, so the data is only updated at the rising edge on the PS/2 data line.

Finally, the two PS/2 signals are outputted through PMOD pins on the board, which can be connected to a PS/2 cable port, along with its ground wire. This can either be plugged into a computer with an already existing PS/2 port, or into a USB adapter to work with more modern computers. This adapter needs an extra circuit that can inform the computer that the device is a keyboard, costing around $7.

VIII. Evaluation and Closing thoughts

In `cocotb` simulations of the memory writing and reading, and k-means modules, each frame allowed for around 20 iterations to be done, which is greater than the expected 16, because of the extra v-sync region that was initially unaccounted for. Nevertheless, the fact that consecutive frames were not that different from each other allowed even only one iteration to get reliable results. The latency of the centers of masses are only behind one frame, which for this use case is negligible, since the user only changes the number of clusters after confirming a digit has been typed. In this timeframe, a single frame does not matter.

With the memory used being only 4 BRAMs, and the system only needing one iteration for convergence, the downsampling could be reduced to only 2x instead of 4x, which allows for more accurate cluster counting, as there would be less aliasing.

Initially, the division module posed a timing constraint, with the initial implementation of binary searching the dividend by multiplying it with the divisor. The multiplication was between two 64-bit integers, and was the main culprit to

WNS becoming negative. However, with the new long division module, WNS gets to around 0.5 nanoseconds.

In practice, the counting system works quite reliably, and from our own testing, the right digit is typed around 90 % of the time. This allowed for some use cases such as typing a phone number. Overall, the project was a success, reaching our proposed ideal goal. For further additions to this project, mouse scrolling would be the next use case we would implement. This only needs to keep track the location of $k = 1$ centroid, and keep track of its horizontal and vertical velocities. The magnitude would determine the number of scroll commands sent to the computer, via the Z-counter of the PS/2 mouse protocol (3).

With regards to implementation, we successfully programmed and tested each of the combined memory module, accumulator module, and `com_updater` module. However, integrating them all into `top_level.sv`, proved to be a challenge, as there is not much testing you can do with `cocotb`. We tried to probe the values of the variables by printing their values on the seven-segment display, but this ultimately did not aid our debugging process. What ended up working was integrating each module into `top_level.sv` one at a time. We will definitely keep this principle in mind when working with future projects with FPGAs or any other hardware, when printing out intermediate values isn't supported.

Steven worked on the initial Python testing, k-means algorithm implementation and integration, and the PS/2 protocol implementation and wiring. Tasmeem also worked on module integration, and worked on cluster graph construction and connected component counter modules for cluster merging. Both contributed in writing the reports and presentations, and Tasmeem conceptualized the block diagrams, while Steven converted them into computer-generated diagrams.

ACKNOWLEDGMENTS

The authors would like to thank Joe and the course TAs and LAs for their unwavering support throughout this project. This project would not have been possible without the lab assignments they built that served as the foundation for this project's conception.

REFERENCES

[1] L. Li, "K-Means Clustering with scikit-learn — towardsdatascience.com." https://towardsdatascience.com/k-means-clustering-with-scikit-learn-6b47a369a83c, 2019. [Accessed 28-11-2024].

[2] A. Chapweske, "The PS/2 Mouse/Keyboard Protocol." https://oe7twj.at/images/6/6a/PS2_Keyboard.pdf, 2003. [Accessed 11-12-2024].

[3] A. Chapweske, "The PS/2 Mouse Interface." http://web.archive.org/web/20040604043149/http://panda.cs.ndsu.nodak.edu/∼achapwes/PICmicro/mouse/mouse.html, 2001. [Accessed 11-12-2024].