LIVE JUST DANCE IN HD: E
Final Report

Justin Chen
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, USA
jchen01@mit.edu

David Lee
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dlee888@mit.edu

***Abstract*—We present a pose extractor and comparer using a 2D-to-1D skeletonization algorithm in addition to a green screen mask, noise reduction convolution, and custom scoring algorithm, all implemented on an FPGA. Our entire pipeline including the skeletonization algorithm is designed to run with minimal latency and output a complete comparison between a baseline model and the human pose within two frames of video. All outputs and scores are fed through HDMI and are capable of being overlaid on top of a pipelined HD camera feed.**

***Index Terms*—Digital systems, pose extraction, Field programmable gate arrays, skeletonization**

I. Overview

A. Problem Summary

The problem of pose extraction is complex, with most modern solutions requiring deep learning technologies and large GPUs to run in real time. In this project, we explore the possibility of running a simpler pose extraction algorithm, one that can run in real time on our given Field Programmable Gate Array (FPGA). Specifically, we explore whether or not it is possible to use this algorithm for our purpose of *pose comparison*, the problem of determining a heuristic that given two poses outputs a number that quantifies how similar the poses are.

It is challenging to come up with and implement a solution fast enough to run in real time yet provides enough information to be able to extract a meaningful score. Our presented algorithm finds some middle ground in this tradeoff, where the image is downsampled and the pose extraction algorithm is able to run within several passes of the frame. Furthermore, with the use of a snappy two-iteration algorithm, we are also able to define a meaningful comparison metric that is able to run within one pass of an input pose.

B. Construction

The construction of our system consists of our FPGA connected to an OV5640 camera through PMOD pins and an external monitor through an HDMI cable.

Our pose extraction algorithm consists of the following main parts. Firstly, the FPGA is connected to a camera that outputs HD 1280×720 video. The output is simultaneously fed into a DRAM frame buffer in addition to logic that will check for whether or not each pixel is part of a green screen background.

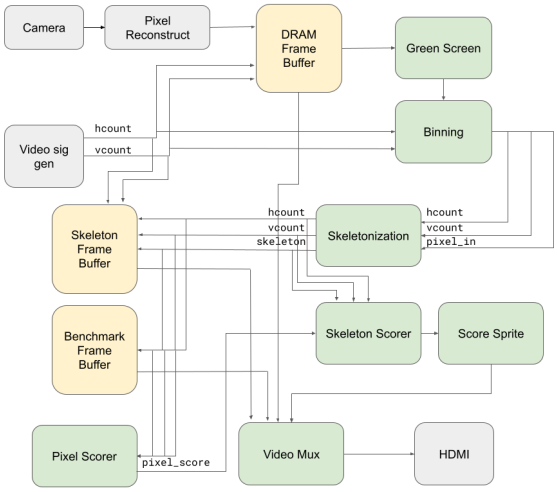


Fig. 1: Block diagram overview

Secondly, the masked camera output will be binned down to a resolution of 320×180, where it will be passed into the skeletonization module. Finally, the skeletonization module will take the blob passed in and run a skeletonization algorithm to output a skeleton that represents the pose of a human in the camera frame. This pose is then upscaled and overlaid on top of the HD video feed.

Following that, we also implemented a pose comparison feature. The press of a button allows the current skeleton on display to be stored as a "benchmark skeleton". After a benchmark skeleton is stored, later skeleton frames are scored against this benchmark skeleton in real time and a numerical score is displayed on the screen.

A simple summary of this in block diagram format is shown in Fig. 1.

II. Pose Extraction

A. Green Screen Detection (Justin)

Our algorithm begins by extracting a human blob from a green screen background. To do this, we take data outputted from the frame buffer and use their RGB values to determine whether or not the pixel is "green enough".

Our first iteration of used RGB values and created threshold maxima for the red and blue values ($\approx$ 0x70), and a threshold minimum for the green value ($\approx$ 0xA0). Following testing, it was found that a simple threshold would not suffice due to inconsistencies in the saturation and value of the green color depending on the external lighting.

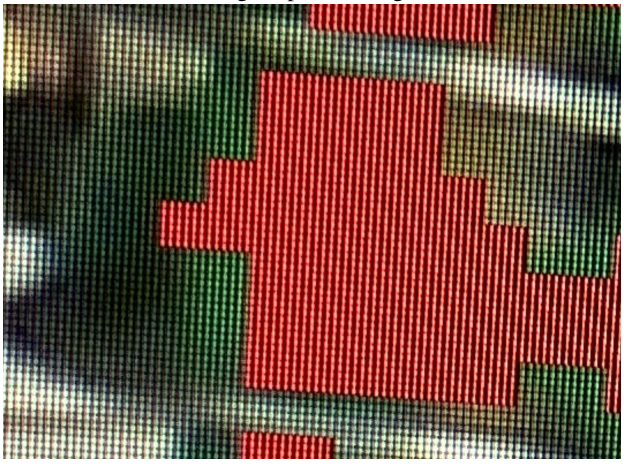
Our second iteration utilized the YCbCr family of color spaces. Similar to our first iteration, we define 3 thresholds, each in 4-bit or 5 bit resolution space that are then each extended to 8 bits by zero-padding. Luminance (Y) is filtered by a minimum threshold, while chroma blue (Cb) and chroma red (Cr) are filtered by a maximum threshold. A sample mask over a green notebook is displayed for visual detail.

Component	Switches	Optimal Threshold Value
Luminance (Y)	[7:6],[3:1]	$> 8'b0$
Cb	[15:12]	$< 8'b1000\_0000$
Cr	[11:8]	$< 8'b0111\_0000$

TABLE I: YCrCb controls and threshold values.

The performance of masking by YCbCr is suitable to our lighting and fabric. In future works seeking to improve performance, the possibility of using HSV style color spaces could also be explored. We predict HSV style color spaces to have less noise and better low-light performance than what we currently have implemented.

(a) Pre-binning output of the green mask



(b) Post binning with noise reduction and down sampling

Fig. 2: Zoomed in input and output of the binning module.

B. Binning (Justin)

Binning serves two purposes – noise reduction and down scaling. To create a viable skeleton, any noise should be significantly eliminated as to not create any stray features that the algorithm may consider as part of the human. Further, down scaling was preemptively included, as the skeletonization algorithm runtime in the worst case is $O(wh \cdot \min(w, h))$, where w and h are the width and the height of the input image, respectively.

An example of the binning module in action can be seen in Fig. 2.

Input to the binning module is 1-bit data from the green screen threshold informing whether the pixel at a particular coordinate location should be ignored and included. Each of these pixels is stored in one of 4 horizontal line buffers (block ram). As the fourth line buffer is written to, we read from all 4 line buffers and tally the number of human pixels (1's) versus green pixels (0's). Whichever one is the majority is outputted by the binning module alongside a new, downscaled (320x180) coordinate position for the pixel.

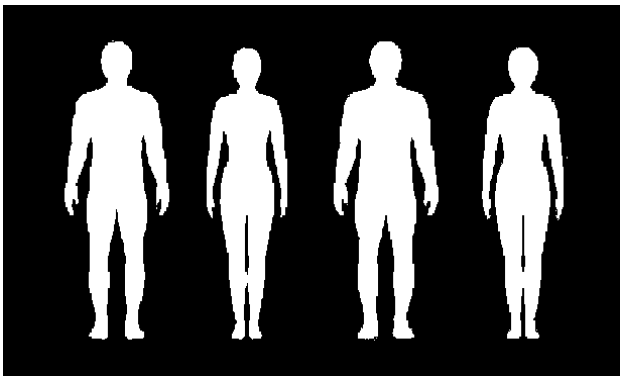
C. Skeletonization (David)

For skeletonization, we use the algorithm presented in [1]. We chose this algorithm because it strikes a good balance between being able to run within a fixed number of frames on our resource-limited FPGA, while also being able to provide us with enough information to make a reasonable score of our pose.

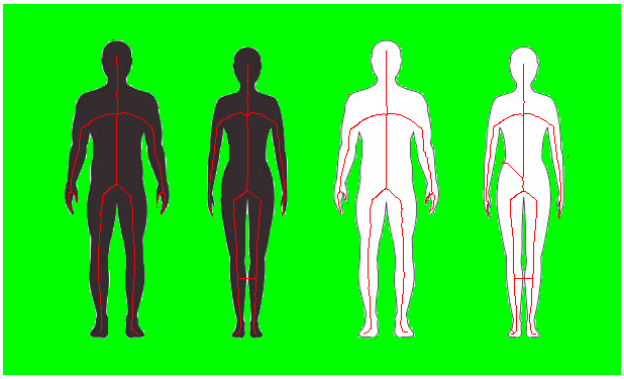
The input to this algorithm is a binary image (i.e., with all pixels equal to either 0 or 1), containing some number of continuous objects. The output of the algorithm is an image of the same resolution, consisting of a "skeleton", where each continuous object in the original image has been thinned down to width 1.

The algorithm involves iterating over the entire image multiple times. Each iteration, pixels that satisfy a certain criterion (based solely on the 3×3 grid of pixels centered around it) are removed from the blob and set to 0. After many iterations of this algorithm, we will arrive at a state where no more pixels can be removed. In this case, the algorithm is done and we can output the resulting image.

An example of the behavior of the algorithm (simulated using a python program) can be seen in Fig. 3



(a) A binary image to be inputted to the skeletonization algorithm



(b) The output skeleton overlaid in red on top of the input image

Fig. 3: Simulated input and output of the skeletonization algorithm.

To implement this, we opted to use a BRAM frame buffer to store the input image and the resulting skeleton. The decision to use BRAM made sense as it was relatively fast and easy to implement, and was feasible because of the small image (320×180) and the small bit depth (only 1 bit was needed for each pixel). Additionally, we implemented four rolling line buffers, three to read the last 3 rows of the image, and the fourth to write the next row into the buffer. Storing the last three columns of the frame buffer output enabled us to correctly calculate the new image resulting after each iteration.

The module was implemented to continuously run iterations until no more pixels could be removed. Since the time it takes to process each frame is variable and extra frame buffers would be resource intense, we decided to not pipeline the module but instead have a busy flag that dictated whether or not new inputs would be accepted. Empirically, the module seems to skeletonize an image once every 1-3 frames.

We chose to write the module in this manner because it takes an unknown time to calculate a skeleton, and the theoretical maximum number of iterations a skeletonization could take is 179. This happens when the input is the complete frame, in which case the algorithm will remove exactly one line per iteration. However, in practice, we find that the number of iterations is usually much less. Therefore, a fully pipelined solution would likely require much more extraneous resources (needing a new frame buffer for each different image in the pipeline) and also have a larger latency.

III. Scoring and Display

For the sake of clarity in this section, let us refer to the skeleton that we are comparing to as the "benchmark skeleton", and the skeleton of the human as the "trial skeleton". To score a trial skeleton on how accurately it replicates a benchmark pose, we present a unique method that utilizes our skeletonization computation and balances computational time and memory.

A. Heat map

The scoring method is divided into two main steps. The first of which is to populate and maintain a 2D *heat map* (array) that stores the Manhattan distance of every pixel to the nearest point on the benchmark skeleton. An example of what this would look like is displayed in Figure 4. The maximum distance that the array needs to store is

$$\text{MAX-DISTANCE} = 320 + 180 = 500\text{px}.$$

Thus our heat map maintains a 320 by 180 frame buffer, with each entry necessitating

$$\lceil \log_2(\text{MAX-DISTANCE}) \rceil = 9 \text{ bits}.$$

A quick calculation gives us an estimated memory usage of roughly 20% of our BRAM allowance. To populate the entries of the heat map, we introduce a two-pass algorithm.

4	3	2	1	2	3	4
3	2	1	0	1	2	3
3	2	1	0	1	2	3
2	1	0	0	0	1	2
3	2	1	0	1	2	3
2	1	0	1	0	1	2
3	2	1	2	1	2	3

Fig. 4: Example calculation of a skeleton heat map.

B. Two-pass Algorithm (David)

Our method of scoring requires no off-board computation and is intended to directly compare the similarity between two skeletons. We do this by using a two-pass method of finding the distance to the nearest benchmark skeleton pixel for every trial skeleton pixel, and summing up the total distance. To save on computation, we use the manhattan distance. The direct

algorithm requires only two passes which are described as follows:

FORWARD PASS

Initially, we initialize a distance array $d(x, y)$ to be 0 for foreground pixels and ∞ otherwise.

Then, we iterate through the image, starting from the top left corner until the bottom right corner. For each pixel (x, y), the distance $d(x, y)$ is updated using the following formula:

$$d(x,y) = \min \begin{cases} d(x,y), & \text{current pixel value} \\ d(x-1,y)+1, & \text{left neighbor} \\ d(x,y-1)+1, & \text{top neighbor} \end{cases}$$

BACKWARD PASS

After the forward pass, the distance $d(x, y)$ is refined by processing the image in reverse order, starting at the bottom right corner and iterating until the top left corner. The update formula is:

$$d(x,y) = \min \begin{cases} d(x,y), & \text{current pixel value} \\ & \text{after forward pass} \\ d(x+1,y)+1, & \text{right neighbor} \\ d(x,y+1)+1, & \text{bottom neighbor} \end{cases}$$

Following just two iterations, our heat map is fully populated and may be stored for as long as needed i.e. until another benchmark pose wants to be used for comparison. To notify the module to create a heat map of a new benchmark skeleton, a button (specifically button 3) is pressed, triggering a *flag* (a sort of valid/invalid boolean) that only lowers once a complete skeleton has been inputted into the algorithm.

Once a forward and backward pass has been completed, the same module can also handle queries to the heat map. By sending in horizontal and vertical coordinates of pixels with the *flag* lowered, the module queries it's internal frame buffer and returns the corresponding distance exactly three cycles later.

The module was implemented so that the forward pass is done simultaneously while the skeleton is being inputted. This decreases latency and removes the need for a line buffer, since we do not need an additional read port for the pixel we are currently calculating. The backward pass was implemented using the same frame buffer with an additional line buffer to be able to access the pixel above the current pixel being iterated.

C. Skeleton Scorer (Justin)

We use a skeleton scorer (as opposed to an individual pixel-distance-scorer) to repeatedly compare trial skeletons, that are constantly being generated by the skeletonization module, to the benchmark skeleton stored in our heat map. To do so, we organize our module into three states: IDLE, COMPUTING, OUTPUTTING.

In the IDLE stage, we reset all the variables and prepare the next trial skeleton. This stage is crucial because output from the skeletonization and heatmap modules is relatively uncontrolled. Thus, without an IDLE stage, the skeleton scorer would be unable to detect when to start a new comparison. An IDLE stage was also helpful in controlling a single-cycle valid out signal described in the OUTPUTTING stage.

The scorer transitions to the computing stage upon receiving a valid input of horizontal and vertical counts of zero. For every pixel that is received we minimize needless computation by considering all distances further than 31 pixels to be equivalent (e.g. a distance of 99 pixels would be truncated to 31). It is important to note that this cannot be done in the heat map module, as the two-pass algorithm requires real, uncapped distance for it to work properly. Afterward, the distance is converted into a *score* ranging from 0 to 7 by applying an integer division of 4 (e.g. a distance of 11 translates to a score of 2). The singular pixel score is tallied into a cumulative skeleton score. A worst-possible score equal to 7 times (the # of pixels in trial skeleton) is calculated in parallel. Computed in combinational logic, the final score of a trial skeleton depends on the relationship between the skeleton score and the max score, and also ranges from 0 to 7.

Finally, our OUTPUTTING state sends a single cycle high signal that updates a score on screen. OUTPUTTING sequentially transitions back to the IDLE state immediately after one cycle.

D. On Screen Visuals (Justin)

After completing thorough pipelining, our entire scoring system can output a score on the scale of ten times per second. These scores sometimes experience jitters, so to improve readability we use an event counter to output only one score for every ten that we calculate.

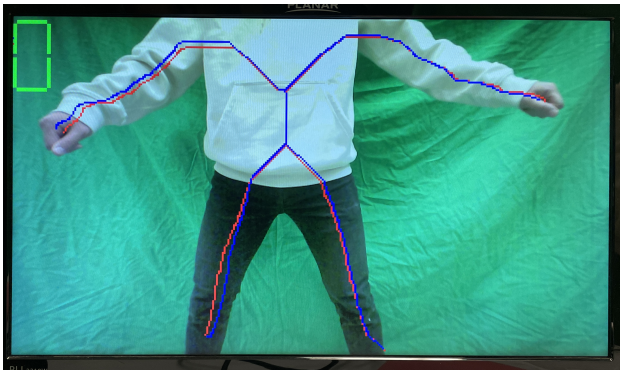
To physically display the score, we chose to implement a score sprite that functions as a seven-segment display formed from seven block sprites. Based on the final score, specific block sprites are activated. In addition, worse (larger) scores appear redder, while better (lower) scores appear green. An example of this in action can be seen in Fig. 5.

We implement an additional frame buffer in our top-level code to store the benchmark skeleton that shows the user the pose that they must attempt to mimic. This skeleton is blue and it's write enable signal is attached to the aforementioned flag used to indicate when a new benchmark skeleton would be entered in for heat map generation.

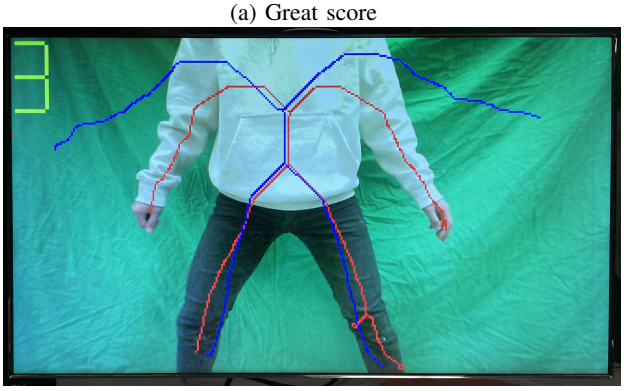
IV. RESULTS AND DISCUSSION

A. Evaluation

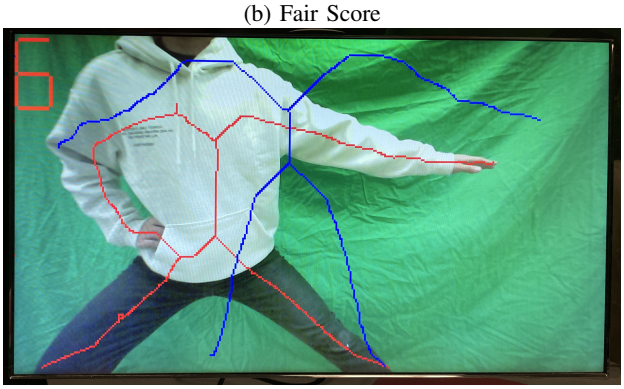
In the worst case, our skeletonization module requires 179 iterations to complete. This would give a latency of $179 \times 320 \times 180 = 10310400$ pixels. Since we are running the pixel clock at 74.25 MHz, this corresponds to $0.14\,\text{s}$, or just over 4 frames of our 30 FPS video. In reality, the algorithm should run much faster, as the input blob is usually a relatively small fraction of the screen and therefore requires fewer iterations. Based on video observation of the project in action, the skeletonization is observed to output a skeleton every 1-2 frames under normal conditions.



(a) Great score



(b) Fair Score



(c) Terrible Score

Fig. 5: A story of pose comparison told by three photos.

The two-pass algorithm implementation has a latency of roughly 320×180 cycles after the skeleton is fully inputted. This is because the forward pass is done while the skeleton is inputted, and so afterwards only the backward pass remains. Therefore, the latency of this module is well under 1 frame of delay.

Some additional overhead is incurred due to the need to store and read skeletons from a frame buffer. This is because the skeletonization algorithm generally does not finish computation in an integer number of frames. After we finish a skeleton, we must store it in a frame buffer until the HDMI output is ready to output the next frame. Conversely, we must wait until a new frame is received from the camera before we can start computing the next skeleton.

As for memory usage, the majority of our memory usage comes from the pixel scorer module, which needs to store a buffer of 320×180 distances, each with bit depth 9. We calculated that this would take up roughly 20% of our available BRAM. This is confirmed by the post synth util report, where the module takes up 18 out of the 28 used RAMB36 blocks. Other components that used significant amounts of BRAM were the frame buffer for the skeleton and the benchmark skeleton, as well as the internal frame buffer of the skeletonization module. Each of these stored a 320×180 frame of bit depth 1. In total, our code used roughly 40% of our available memory.

All of our modules run on the HDMI pixel clock, so our design must meet the timing requirement enforced by our 74.25 MHz pixel clock. Based on Vivado's timing summary, we were able to meet this requirement, with 0.403 nanoseconds of slack.

B. Implementation Insights and Retrospective

Looking back, we learned some important lessons during the development process:

- Think about proper pipelining when first implementing modules. The effects of poor pipelining are not always negligible. It's much easier to write it correctly the first time then to debug pipelining issues.
- Be careful and watch out for subtle issues such as clock domain crossing or assigning to a variable both sequentially and combinationally. These issues can be very hard to spot and have very unintuitive side effects.

V. Appendix A: Source Code

VI. Appendix B: Resource Usage Reports

Below is a dump of relevant resource utilization reports:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	6807	0	0	32600	20.88
LUT as Logic	6306	0	0	32600	19.34
LUT as Memory	501	0	0	9600	5.22
LUT as Distributed RAM	484	0			
LUT as Shift Register	17	0			
Slice Registers	5874	0	0	65200	9.01
Register as Flip Flop	5832	0	0	65200	8.94
Register as Latch	42	0	0	65200	0.06
F7 Muxes	31	0	0	16300	0.19
F8 Muxes	0	0	0	8150	0.00

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	31	0	0	75	41.33
RAMB36/FIFO*	28	0	0	75	37.33
RAMB36E1 only	28				
RAMB18	6	0	0	150	4.00
RAMB18E1 only	6				

References

[1] T. Y. Zhang and C. Y. Suen. 1984. A fast parallel algorithm for thinning digital patterns. Commun. ACM 27, 3 (March 1984), 236–239. https://doi.org/10.1145/357994.358023

[2] https://www.reddit.com/r/JustDance/comments/1eiiptc/just dance not real dancing/