SIFTPGA

Evelyn Fu
Electrical Engineering and Computer Science
MIT
Cambridge, US
evelynfu@mit.edu

Shruti Garg
Electrical Engineering and Computer Science
MIT
Cambridge, US
sgarg@mit.edu

***Abstract*—In this paper, we present an implementation of the Scale-Invariant Feature Transform (SIFT) algorithm in System Verilog to be run on an FPGA. This algorithm, used in many computer vision pipelines, executes a number of computation heavy steps on many pairs of images to detect and match keypoints across video frames. We design a customized hardware system that parallelizes these tasks and accelerates the SIFT step of a vision pipeline that can be easily integrated into the rest of the vision pipeline in software. We also present our evaluation metrics and analyze the performance of our system compared to strictly software implementations. Our code can be found at https://github.com/shrutigarg914/SIFTPGA**

I. Introduction

Scale-Invariant Feature Transform (SIFT) is a classic and widely used method in computer vision for describing, detecting, and matching features between images of the same scene, taken at different angles and scales. It is often used as one step in the pipeline for many computer vision applications, such as object recognition, robotic mapping and navigation, image stitching, 3D modeling, gesture recognition, and video tracking. As engineers and researchers delve into larger-scale vision problems, it is important to keep computation times low in order to facilitate efficient iteration and improvement. In addition, computational efficiency also becomes a bottleneck for online systems, which often have to trade performance in order to keep up with live data. Therefore, we wanted to investigate designing custom hardware systems for foundational vision algorithms in order to speed up entire vision pipelines, focusing on SIFT.

A. Algorithm Overview

The main steps of the algorithm given an image are as follows, as described in the OpenCV docs [1]:

1) Initial detection of Scale-invariant extrema.
    a) Build the scale space or Gaussian pyramid for the image. Refer to Fig 1 for a visual of the process. The blur is increased along the row (the green arrow). The second last image in each row is down sampled, then used as the first image in the row below (blue arrow).
    b) Apply the Difference of Gaussian (DoG). That is, compute the difference of each individual pixel for pairs of adjacent images in a row. Note that this step might call for attention to representation of the floating point numbers and bits used, for the values of these differences are quite small for the majority of the image.
    c) Identify pixels whose DoG value is greater than all 26 of their neighbours (shown as green dots in Fig 2). These local extrema form our initial array of potential keypoints.
2) Refining Keypoints
    a) If the intensity of the location of the pixel is less than the threshold defined then this pixel is discarded.
    b) To remove edges, compute the Hessian matrix to compute the principal curvature.
3) Building Descriptors
    a) To add to the scale invariance, an orientation is computed for rotational invariance. To do so, compute the gradient magnitude and direction for

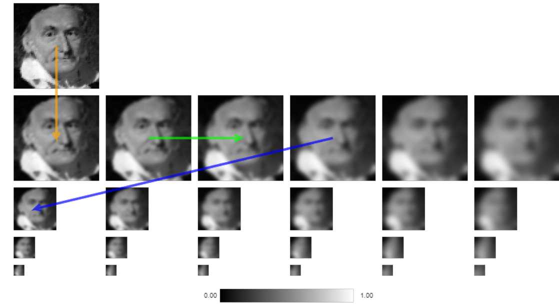

Fig. 1. Gaussian Pyramid. Image courtesy of [2]

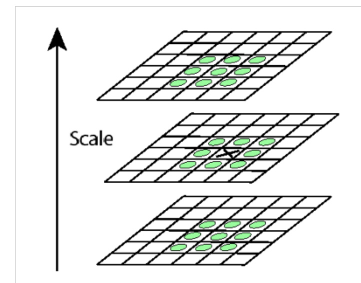

Fig. 2. Extrema in the Difference of Gaussian Pyramid. Image courtesy of [1]

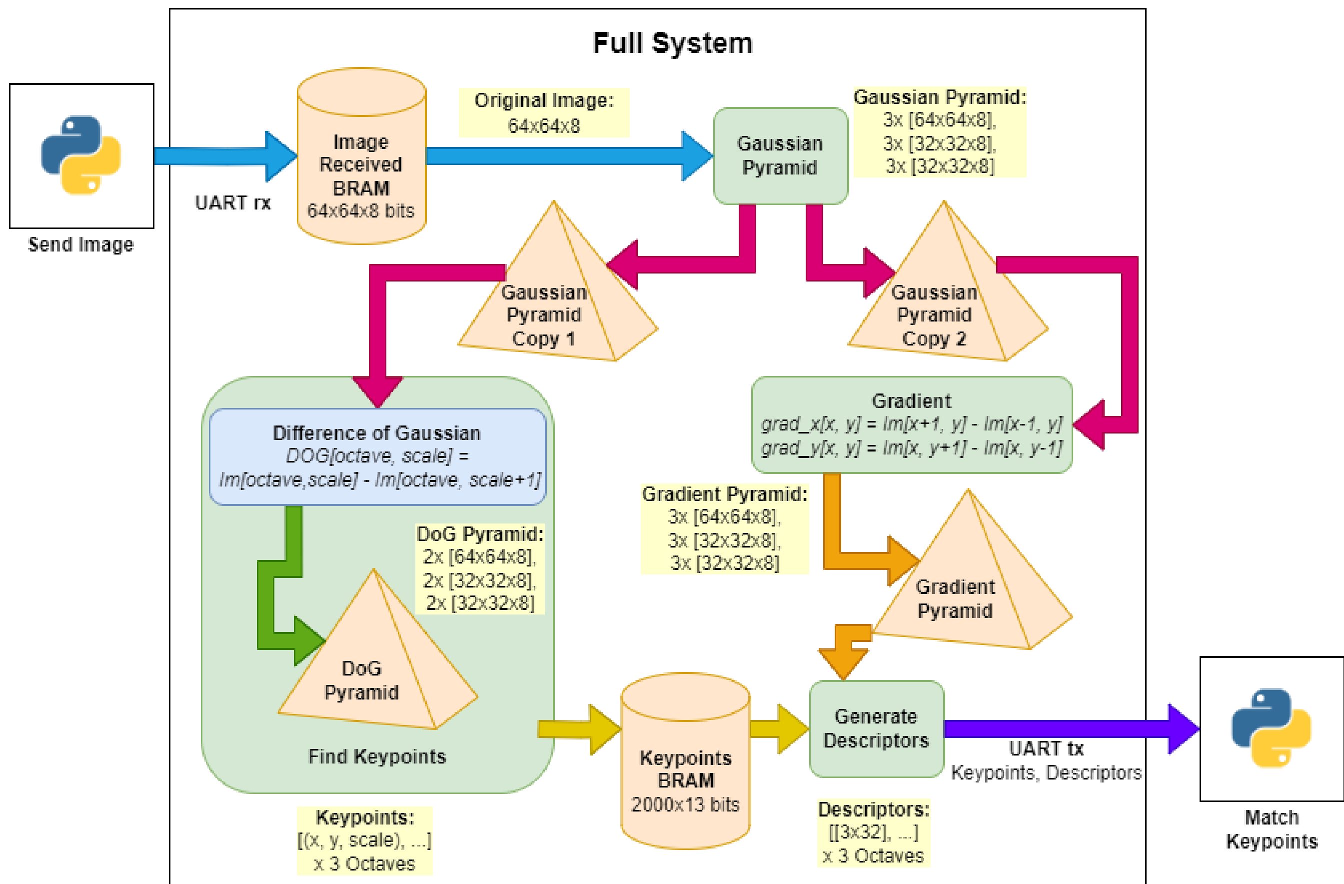

Fig. 3. High level overview of the system (Evelyn)

the neighborhood around each keypoint. Create a histogram with 36 bins for 360 degrees and add to each bin a value proportional to the magnitude of the gradient for all the pixels around a keypoint. The highest peak is used to calculate the orientation of the keypoint.

b) A 16x16 neighbourhood around the keypoint is divided into 16 sub-blocks of 4x4 size. For each sub-block, an 8 bin orientation histogram is created. These 128 bin values are represented as a vector to form a keypoint descriptor that can be compared across frames to stabililse matches.

4) Keypoint Matching
   a) Now with keypoints in both images, a nearest neighbour search associates matching keypoints between the frames.
   b) To improve matching, get rid of matches where the second closest match is too close (has a ratio of greater than 0.8) to the first one as it might just be the result of noise.

B. Design Specifications

There are certain fixed parameters in the system design chosen based on memory and time limitations:

1) Image Size: Inputs assumed to be 64 by 64 8 bit greyscale images.
2) Pyramid Size: We are using a fixed-size pyramid of 3 octaves and 3 levels of blur per octave.

II. High-Level System Overview

Figure 3 shows the setup of our system and its integration with an external computer, running a software vision pipeline, at a high level.

III. Communications (Shruti)

The data transfer between the computer and the system happens over serial UART. Though slow, UART was chosen for ease of implementation. There are two major use cases for this line of communication: during operation/evaluation and during debugging/development. There are two main forms of structures of data we will want to transmit: images and lists of values. For operation, the system needs to be able to receive a 128 by 128 greyscale image and transmit its calculated keypoints and matches. For debugging and testing during development, it would be useful to transmit images of different sizes from the FPGA board to the computer, and to receive images in return so we can verify our modules.

On the computer, the transmission of images and reception is done by python scripts. These scripts are relatively easily

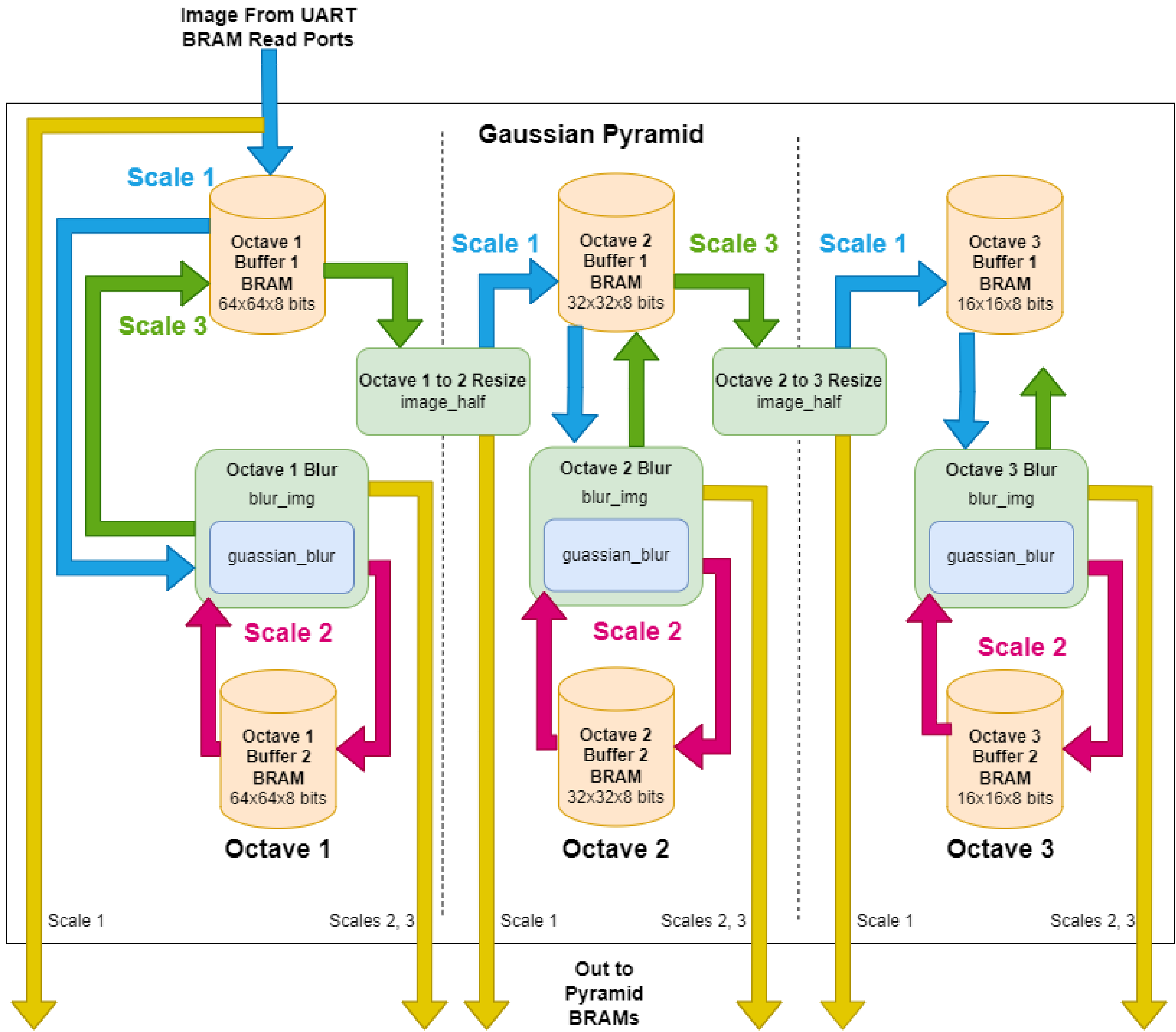

Fig. 4. Detailed diagram of the Gaussian Pyramid module (Evelyn)

modifiable to receive different outputs. This script does pre-processing such as resizing the image and converting it to greyscale. To do the latter, the script takes a weighted average of the red-green-blue pixel values to convert to greyscale: $0.2126R + 07152G + 0.0722B$ [3]. The weights account for perceptual luminance.

On the FPGA, the actual modules handling the protocols are in verilog and systemverilog based off of the ones generated by Manta [4]. These modules allow the user to define a parameter `CLOCKS_PER_BAUD`, which has been set to 50, and with the use of the 100MHz clock this leads to a 2MHz baudrate. There is a start and a stop bit with 8 data bits for a total 10 bit packet. (There are no parity bits). Therefore all of our data will be sent in 8 bit widths.

On top of these base modules, there are 3 variations of similar simple FSMs to transmit images, keypoints and descriptors when the respective BRAMs are ready. The images are sent pixel-by-pixel, each keypoint is sent as 2 8 bit values (one for each coordinate), and the descriptors are sent as 12 8 bit packets.

IV. Keypoint Detection

A. *Gaussian Pyramid (Evelyn)*

A detailed diagram of our design for the Gaussian Pyramid module can be seen in Fig 4. This module reads in data of the original video frame image and determines the pixel values for each image in the Gaussian Pyramid which is outputted to be stored to 9 pairs of identical BRAMs (18 total) at the top level, one for each image of the pyramid. We duplicate the BRAMs so they can be used simultaneously by the Difference of Gaussian and Gradient Pyramid modules. Within this module are 6 BRAMs meant to temporarily store intermediate images

within the pyramid, at each octave. We use these buffers since while computing the next image in a pyramid, we need to maintain the ability to read from the previous image while storing the new image. To create the images in the pyramid, we have one image blur module for each octave and one image resize module for each octave transition.

The Gaussian Pyramid design can be broken up into the following portions:

1) FSM: The Gaussian Pyramid module computes each image in the pyramid following an FSM, where the states are IDLE and one state for each image in the pyramid, named as O[octave]L[blur level]. When we have finished receiving the original image from UART, a start signal will be triggered, allowing the Gaussian Pyramid module to move from the IDLE state to O1L1. When this state transition is made, the module will begin reading from the BRAM that stores the original image and store this image, which is the octave 1 level 1 image, both in the first octave 1 buffer and in the BRAMs for this image in the pyramid at the top level.

Once we have finished reading every pixel of the image, we transition to O1L2 and trigger the octave 1 blur image module to start. This blur image module will read from the first octave 1 buffer and directly write the blurred image our to the second octave 1 buffer and to the according BRAMs at the top level.

Once the blur module signals that it is complete, we transition to O1L3 and similarly trigger the blur module, except this time we will read from the second octave 1 buffer and write to the first octave 1 buffer and out to the according BRAMs at the top level for this image.

Once the blur module signals it is done again, we transition to O2L1 and trigger the octave 1 to octave 2 resize module, which reads the last blurred image from octave 1 out of the first octave 1 buffer and writes to the first octave 2 buffer and out to the according BRAMs at the top level.

The same flow of blur, blur, resize, is continued throughout the rest of the states. We use combinational logic based on which state we are in to switch which BRAMs the submodules read from and write to. Since we need the previous image in a pyramid to construct the next, this process must be done sequentially.

2) Blur Image: The Blur Image module uses a Gaussian Blur submodule, which computes the blurred value at a given pixel in the image by computing a weighted average of the pixels in the 3x3 neighborhood around that pixel. The code for computing this weighted average was taken from [5] and modified slightly. The Blur Image module loops through each pixel in an image and collects the 9 pixels in the 3x3 neighborhood, taking 18 cycles total for 9 reads. Once these pixels are collected, the Gaussian Blur submodule, which is pipelined to take 3 cycles, is triggered. The result of the blur is written out to the blurred image BRAMs at the original pixel coordinate. When the blur for each pixel has been calculated and written out, it pulls the done signal high for one cycle to tell the FSM to continue.

3) Image Half: The Image Half module uses nearest neighbor resizing to halve the size of the last image in each octave to use as the first image in the next octave. It iterates through every other coordinate value, (x, y) in the larger image and writes the value of that pixel to the address for (x/2, y/2) in the downscaled image. Once it has finished with the last even address of the first image, it will pull the done signal high for one cycle to tell the FSM to continue.

B. Gradient Pyramid (Evelyn)

The gradient pyramid is stored as 18 BRAMs at the top level. Since we need to know the gradients in the x and y direction to determine orientation for descriptor generation, we use 1 BRAM for the x gradient and 1 for the y gradient for each image in the Gaussian Pyramid. The Gradient module reads from one image BRAM and writes the x gradient and y gradient for each pixel out to the corresponding x and y gradient BRAMs. Therefore, we have 9 instances of the Gradient module in our top level to calculate each gradient image in parallel. The Gradient modules are triggered to start once the Gaussian Pyramid module signals that it has completed. Inside the gradient module, we use an FSM that starts in the IDLE state and when triggered, iterates over all the (x, y) coordinates in each image, reads the pixels adjacent on the x axis to our desired coordinates in the next two states, calculates the gradient in the next state and saves it to the x gradient BRAM, then repeats with the y axis. The gradient is calculated according to the following equations:

$$\Delta x = (p_{x+1,y} - p_{x-1,y})/2$$

$$\Delta y = (p_{x,y+1} - p_{x,y-1})/2$$

Once all coordinates are iterated through, we return to the IDLE state and pull the done signal high for one cycle.

C. Keypoint Finding (Shruti)

This module takes the gaussian pyramid, and writes keypoints for that pyramid in a BRAM. More specifically, this module first takes the difference of successive gaussians in each octave, giving us 6 BRAMs. Then it goes through and writes the coordinates of all the points in each of these three octaves that are local extrema (either greater than or less than all of their neighbours). A general overview is given in Figure 5. Each of the submodules is explained in further detail below.

*1) **Difference of Gaussian (DoG)**:* Since the Difference of Gaussian is calculated between blur layers of the same octave, we parameterize this module for the size of the image at each octave. It is a simple module that reads the same address from both the BRAMs it is given and writes their difference in a new BRAM. As each level only has 3 images, we need 2 instances of DoG per octave. Each DoG These are chained such that the second DoG modules will start when the previous ends to handle reading from the middle blur level's BRAM for both DoGs in the same octave. For a scaled up version of the system, we can set these up such that there are two sets of DoG modules such that all DoG modules in that set read from unique BRAMs and therefore we can parallelise the generation of the Difference of Gaussian BRAMs.

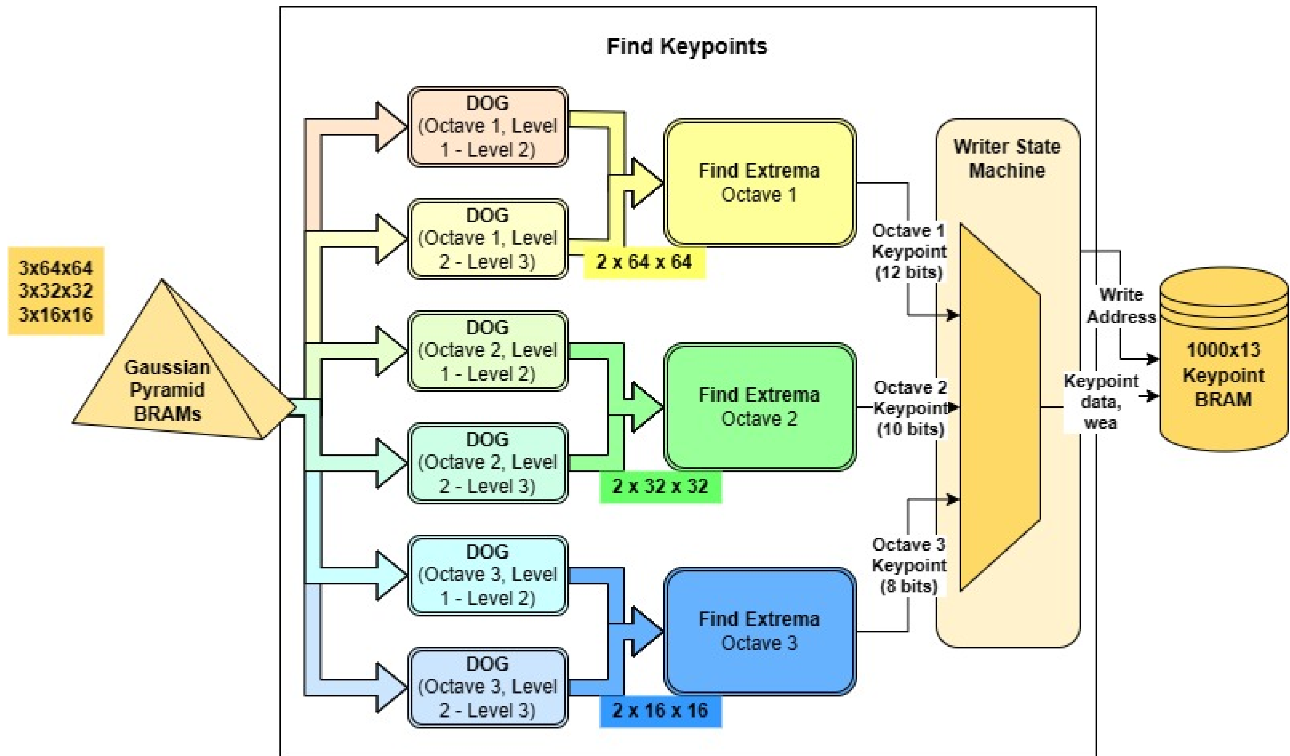

Fig. 5. An overview of the module for finding keypoints (Shruti)

*2) **Find Extrema**:* In a more general case, this submodule should take a voxel of 3 by 3 from each octave's stack of difference of gaussians to check if its center is an extrema (as shown in Figure 2. Since we have 2 difference of Gaussian images in each octave, this requirement reduces down to needing to look at the 8 neighbours in the same BRAM, the corresponding pixel in the other BRAM and its 8 neighbours (for a total of 18 pixels being considered at each checking step). Note that the pixels on the edges at each scale of the images (so pixels with less than 9 immediate neighbours) are not checked for extrema as they will not provide us enough information to be a keypoint. This module is mostly implemented as an FSM to read pixel values and output keypoints as shown in Figure 6, but the actual check for a pixel being an extremum is performed with a constant cominational logic once the appropriate values are loaded in. Also, since there are only 2 images being considered, the module checks and writes keypoints for both images simultaneously.

When started this module first loads the 18 pixels centered around (1, 1). Once, loaded if the combinational logic indicates the pixel in one or both BRAMs is a keypoint, the module will indicate to its parent module to write the keypoint(s) it outputs at that cycle. It will then shift the voxel over by one column (i.e. discard the values from the leftmost pixels, shift the voxel values to their left neighbours, and then load in the values for the rightmost pixels in the voxel), and repeat the check and write. We do this shift to reduce the number of cycles we spend waiting for reads espescially when we already have 12 of the 18 values we need for the next check. However, when the voxel hits the right edge of the picture, we will need to move down a row which means we will have to load in more than just the rightmost values. We choose to go back to the start of the next row and load all 18 values (this allows us to reuse the logic from the first starting state in the FSM). Thus, we go through the entire DoG image and indicate whenever we find a keypoint.

3) Keypoint BRAM: The entries into the BRAM are in the form `{x, y, l}` where x and y are the pixel number along the x or y axis, and l is a one bit representing which of the two DoG BRAMs the keypoint is from in the given octave. To identify which octave keypoints are from, we write a 0 value in the BRAM after each octave's keypoint finder is done and before the next octave's keypoint finder starts. Looking for this 0 value when traversing the BRAM then can tell us which octave the keypoint belongs to.

Here it is worth noting that the same point in the image will be represented by different coordinates at different octaves, since the images scale down. So if we parameterize the keypoint modules by their size (as we have done), the bit width of the coordinates will be different. The way we have handled this currently is to simply index into different parts of the data, and handle each octave's keypoints separately when transmitting them. In retrospect, a much cleaner and easier approach would have been to simply pad the coordinates so they were the same width before concatenating them.

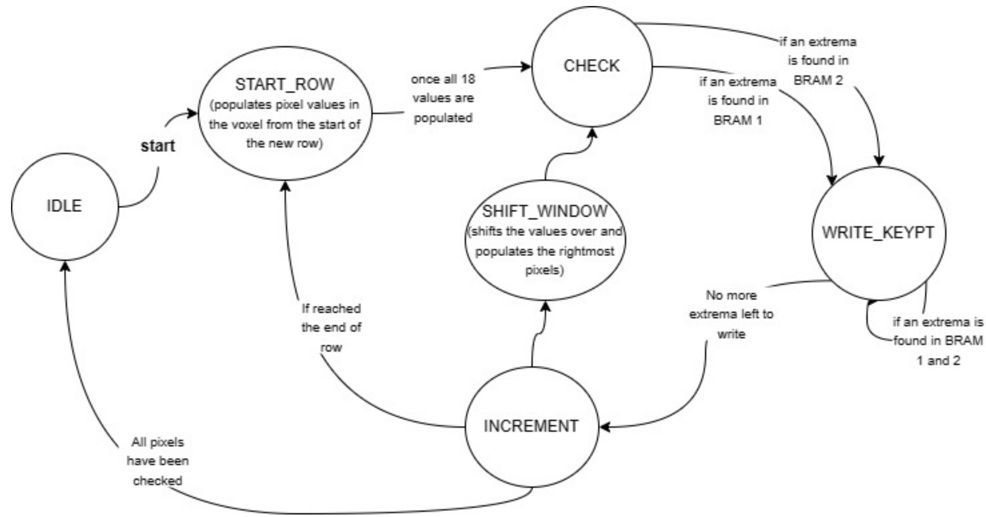

Fig. 6. FSM checking for extrema given two DoG BRAMs (Shruti)

D. Descriptor Generation

This module relies on both the keypoints and the gradient BRAMs. For each keypoint, the module will take a 4 by 4 pixel grid and for each quadrant generate a histogram representing what the gradients look like in that patch.

1) Orientation (Evelyn): This submodule is used by the Histogram submodule to determine which "bin" the orientation of the gradient at a given pixel is in. We categorize the orientations into 8 bins, which split the 360-degree orientation space into 45 degrees each. See figure 7 for a visual.

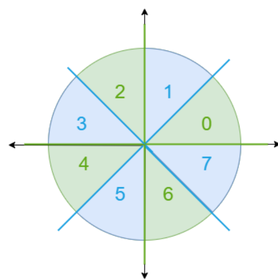

Fig. 7. Orientation Bins

We determine the bin using the separate x and y values of the gradient from the gradient pyramid. Due to the nature of how we section the orientation space, we can generally use the table I to compute the bins without having to compute the angle directly:

$x > 0$	$y > 0$	$abs(x) > abs(y)$	bin
T	T	T	0
T	T	F	1
F	F	T	2
F	T	F	3
F	F	T	4
F	F	F	5
T	F	F	6
T	F	T	7

TABLE I
ORIENTATION DECISION TABLE

Some exceptions to this table are made to handle the boundaries of the bins, where we follow the rule that at a boundary, we fall counterclockwise.

2) Histogram (Evelyn): The Histogram submodule takes in an (x, y) coordinate and valid in signal and determines a histogram describing how many pixels in the patch of 4 pixels where it is the upper left corner fall into each bin. This uses an instance of the Orientation submodule. Since up to 4 pixels in a patch can fall into each bin, each of the 8 bins requires 3 bits, resulting in a 24-bit output. This histogram and a valid out signal are outputted once all the orientations are computed for the patch.

3) Descriptor BRAM: The BRAM is structured such that the entries are the width of a histogram (24 bits wide) and four entries in the BRAM correspond to a full descriptor. These can then be transmitted entry by entry with each entry being sent in 3 packets of 1 byte.

4) Generating the descriptors (Shruti): This module was again an FSM, that went through the list of keypoints and for each keypoint writing a histogram for each of the four patches in the Descriptor BRAM. For each keypoint, it determines where the top left corner of the 4 x 4 pixel grid will lie. Generally, this will be 2 to the left and 2 above the keypoint (such that the keypoint is the left corner of the lower right patch). However, for edges (when the keypoint is in the second row or column), the grid starts at the edge or corner. Given this top left corner of the grid, we can now cycle through each patch by adding/subtracting 2 to give the histogram submodule the top left corners of the 4 patches to generate the 4 histograms. The module keeps track of how many 0s it has seen and stops writing descriptors once it has seen 3 (which indicates we have reached the end of the third octave's keypoints).

In addition to cycling through all the keypoints, a major role of this module is also to route the correct BRAMs to the

histogram submodule. There are 6 sets of 2 gradient pyramids to choose from when evaluating the histogram, and putting this logic away in this module instead of top level or in the histogram itself keeps the layout cleaner. Given the octave state we are in and the level bit, we use combinational logic outside the FSM to do this routing of data.

5) Current State of the System: Here we will note that this particular module does not work on hardware yet. All the submodules are test benched and work in simulation. However, when combined with the previous modules and put on hardware, the descriptors sent back to the computer are too little and nonsensical. Both team members have spent sad long hours (on the scale of about 20 combined) debugging the system to go from ”what is even happening” to ”we think the module is reading wrong values from the Keypoint BRAM and terminating early, and our sending module seems like its changing what is actually stored on the BRAM.” If we had more time we would probably finish writing a bigger testbench that tests the connections between the previously tested system, and honestly maybe bin the current module and re-write it with cleaner design choices (such as reformatting the keypoints BRAM, or even splitting it into three different BRAMs).

V. Evaluation

A. UART

We relied heavily on our UART framework to evaluate the function and accuracy of our system. Since we were working with images, it was helpful to visually see the results of our modules. The system currently identifies keypoints that visually make sense and resemble the keypoints found by python. For example, the keypoints lie in major regions that would help identify the image and its subjects. Please refer to the Appendix for the visuals on our results.

B. Python Scripts

To evaluate the accuracy of our code, we wrote python mirrors to our SystemVerilog modules as we were testing each of the units individually. Comparing the results from the FPGA with the results from the scripts often helped us debug the system. These functions then can be put together for a working system in python.

Additionally, Evelyn wrote a python script to calculate keypoint matches using nearest neighbors and use OpenCV's libraries to visualize matches between keypoints in two images. We tested this on keypoints generated with our python mirror. An example of matching using the data from our python mirrors can be seen in the appendix. Also, Shruti wrote code to send an image and receive the keypoints and descriptors from the FPGA that we would later merge the matching and visualization code into.

C. Speed

We wanted to evaluate the speed of our system, and try to optimise it, but it fell outside the scope of the time we had. The idea was that we will have the FPGA keep track of the time passed since an input image is fully received and the output corresponding to that image is ready to be transmitted. These timing counters will also be useful (and necessary) when we pipeline the system in addition to just evaluation. Depending on the magnitude of the value, timing information can be sent back over UART or displayed on hardware in binary using the small LEDs. This can then be measured against how long c++ code takes to run similar keypoint generation and matching processes on similar images. The code we were intending to benchmark againt is this repositor for FPGA-Video-Processing, on GitHub [5].

VI. Discussion

A. Space Constraints

While designing and implementing this project, we had to revise our original goal to accommodate a limited number of BRAMs and space within each BRAM. For example, we realized that one 64x64 image almost entirely takes up all 36k bits in a single BRAM. Additionally, a lot of our design was oriented around optimizing for speed, since we assumed we could later buy external memory to deal with memory limits. For example to be able to calculate the descriptors and gradients in parallel, we wanted to duplicate our Gaussian Pyramid storage, and to avoid having the calculate the gradients for each pixel every time it appeared in a patch around a keypoint, we computed the gradients while the keypoints were being evaluated and stored them, also taking up multiple BRAMs. Therefore, given we only have 72 BRAMs to use, we had to limit our original goal of using 4 octaves and 4 blur layers per octave, with an original image size of 128x128 to instead using a 3x3 pyramid and an original image size of 64x64. Given how small these images are, we also could not use the usual descriptor size for SIFT of 128 entries, which looks at a 16x16 pixel patch around each keypoint. Instead, our descriptors are only 32 entries long and looks at 4x4 patches. Given these sacrifices, our total BRAM count theoretically comes to: 1 - original image, 18 - Gaussian Pyramid, 6 - Gaussian Pyramid Buffers, 18 - Gradient Pyramid, 12 - DoG, 1 - Keypoints, 1 - Descriptors = 57 BRAMs. It is worth noting that because of bugs we could not fix in the descriptor generation, many BRAMs are optimized out and only 37 BRAMs are in use after compiling.

B. Future Work

We learned a lot while working on the project, including a lot about how not to design things. Firstly, we would fix the bugs with descriptor generation. Next, we would implement keypoint matching on the FPGA. Finally, we didn't get around to optimizing our design for time, so we would want to pipeline it. We would also probably want to use external memory to be able to handle larger images and get more descriptive results.

References

[1] "OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform)." *Opencv.org*, 2020, docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html.

[2] Weitz, Edmund. *SIFT - Scale-Invariant Feature Transform*, 2016, weitz.de/sift/.

[3] Brandon Rohrer. "How to Convert an RGB Image to Grayscale." *139. Signal Processing Techniques*. https://e2eml.school/convert_rgb_to_grayscale (accessed 22 November 2023).

[4] Fischer Moseley, "Manta: An In-Situ Debugging Tool for Programmable Hardware," 2023.

[5] Yu George, FPGA-Video-Processing, (2021), GitHub repository, https://github.com/georgeyhere/FPGA-Video-Processing

APPENDIX
FIGURES FROM OUR RESULTS

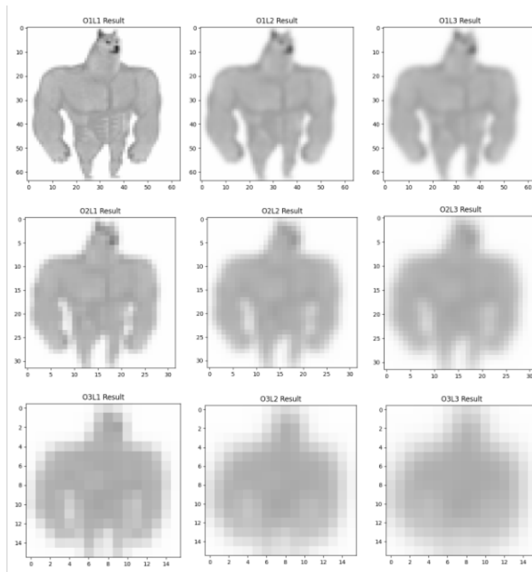

Fig. 8. Gaussian Pyramid on Buff Doge

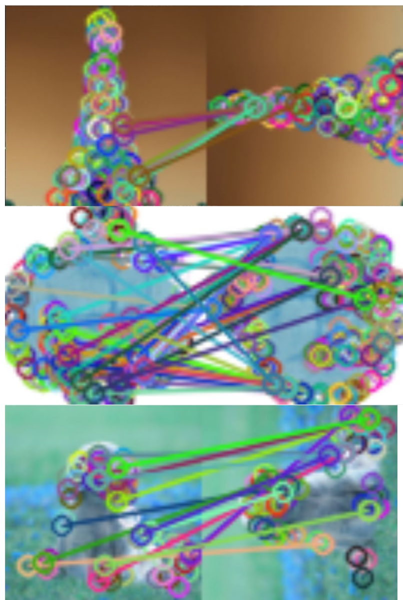

Fig. 9. Python matching script

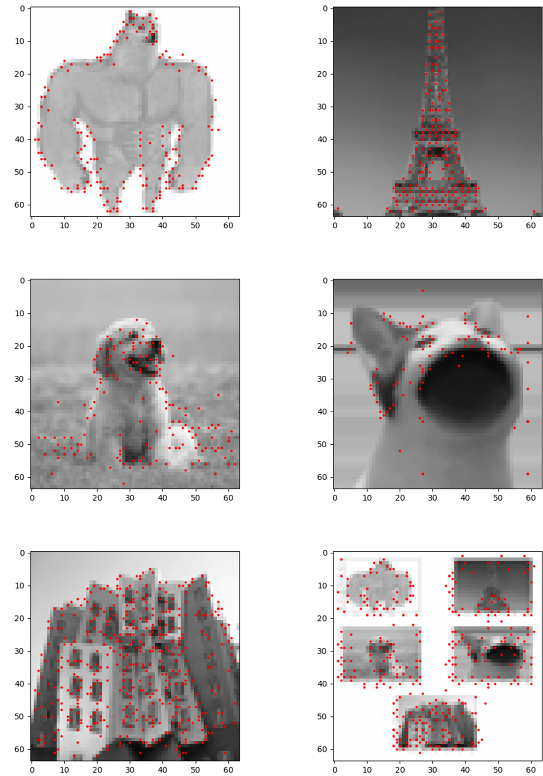

Fig. 10. Keypoints on a few different images

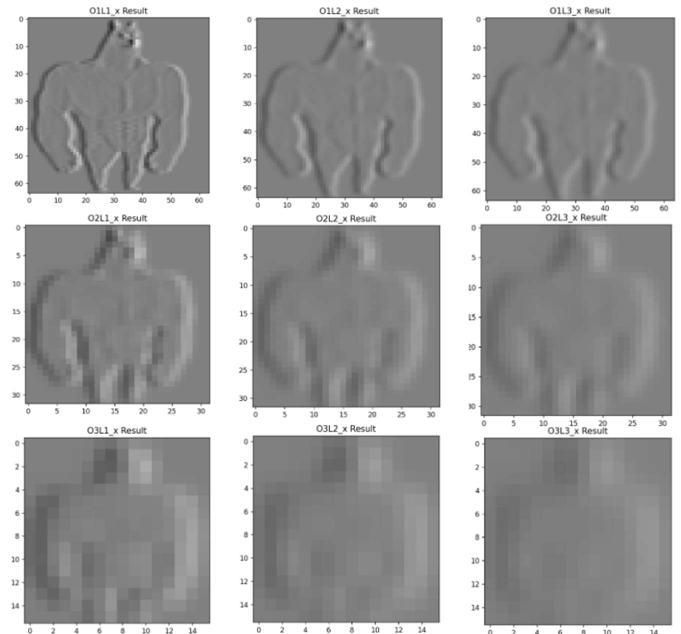

Fig. 11. Gradient Pyramids (x axis)

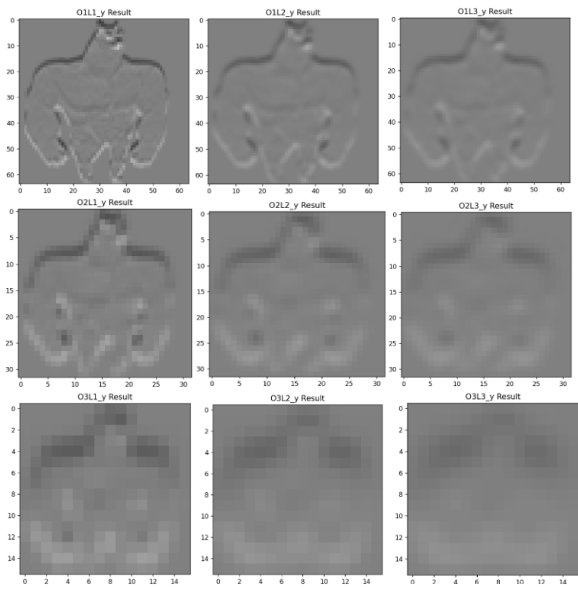

Fig. 12. Gradient Pyramids (y axis)