

FPGA-based Feature Finding

1st Lidia Prokopovych

*Dept. of Electrical Engineering & Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
lidiap@mit.edu*

2nd Kiera Reed

*Dept. of Electrical Engineering & Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
kiera@mit.edu*

Abstract—We present a system that implements most of the components necessary to detect corner features in a camera using a corner-detection algorithm, as well as extract a feature encoder in the static window within the frame. This generates an output that can be viewed visually or sent downstream to a system that depends on feature detection, e.g. a simultaneous localization and mapping (SLAM) system for autonomous navigation in robots, or medical imaging. For the purpose of this system, we will be projecting the camera inputs onto a monitor and applying a neon-colored mask over pixels that represent corners for visualization. Corner detection is a nontrivial problem because of its large volatility due to noise; hence, many of our additional components and analyses were with the intention of creating as precise of a signal as possible, including Gaussian blur filters, non-maximum signal suppression (NMS), and image pyramid analysis. Feature extraction faces several challenges, including dependence on the orientation of a corner within a frame and balancing holding useful encoder information with not bulking memory or increasing latency.

Index Terms—Digital systems, Field programmable gate arrays, Computer vision, Feature extraction

I. SYSTEM OVERVIEW

This system uses an OV5640 camera to capture 25 fps, 8-bit grayscale video of its environment. The camera's output is read at 200 MHz; this data is immediately moved to a 100 MHz clock domain using a CDC FIFO in order to match the rest of the system.

The camera's output, once in the 100 MHz clock domain, is passed through two sequential 3x3 Gaussian blur filters (or, equivalently, a 5x5 filter) is passed to an image pyramid module. The image pyramid has an internal line buffer spanning 64 lines. It uses this line buffer's output to create three 16-line outputs, one at 1x scale, one at 1/2x scale, and one at 1/4x scale. This effectively downscales the input image in the vertical dimension. Additionally, the 1/2x scale output is marked valid only every second cycle and the 1/4x scale output is marked valid only every fourth cycle, so that downstream consumers only get 1/2x and 1/4x the pixels in the horizontal dimension as well. This module allows the system to run corner detection at three separate scales in parallel with minimal performance penalties.

The outputs from the image pyramid are sent to three corner detection modules. Each corner detection module creates a 7x7 pixel window buffer to feed to the corner detection algorithm. These buffers are small and rapid access is required, so they are stored in registers. The corner detection modules, which

use the the Features from Accelerated Segment Test (FAST) algorithm, classify pixels as keypoints and also output corner scores for the pixels. In our project checklist we originally planned to implement Harris Corner Detection, which is a more complicated but allegedly more accurate corner detection algorithm, but after observing surprisingly-successful results of FAST even with large amounts of noise, we chose to pivot and instead focus on creating a cleaner signal surrounding FAST.

The scores from the three corner detection modules are combined and sent to a non-maximum suppression (NMS) module in order to eliminate low-confidence features and create a less noisy output. The NMS module has an internal line buffer spanning 15 lines. It uses this line buffer to track the maximum values in any given 15 by 15 pixel window. If a pixel's feature score is equal to the maximum, its value is passed downstream; otherwise, the NMS module sets that pixel's feature score to 0. Since the NMS modules caches local maxima, it does not need to create a window buffer.

The output of the NMS module is then written to a DRAM frame buffer, which is used by the video output pipeline to visually display detected features. The frame buffer stores both pixel values and feature scores, which enables multiple output modes such as a mask over the raw camera input, a mask over a black screen, or only raw camera input.



Fig. 1: Output of system.

In parallel with the FAST-NMS pipeline, the system also computes 120-bit descriptors for the keypoints in the frame using the Binary Robust Independent Elementary Features algorithm in a 15x15 window. Finally, the system stores sequential BRIEF descriptors and computes the Hamming distance between them to estimate continuities between corners in subsequent frames.

II. I/O AND MEMORY USAGE (KIERA)

A. OV5640 Input

An OV5640 camera is used for video input, similarly to the labs. The camera is set up to generate 8-bit luminance output for a 1280x720 frame at 25 fps, which helps the system have a manageable amount of input data while maintaining a high resolution and acceptable frame rate.

B. Frame Buffer

The I/O for the DRAM frame buffer is handled by the frame builder and frame buffer reader modules. The former combines raw pixel data with the associated “cornerness” scores from FAST or the detected maxima from NMS, which it then sends to the frame buffer. The latter reads the frame buffer and generates the output pixel according to one of the following output modes (selected using `sw[5:4]`):

- 0) Score values (mapped to brightnesses)
- 1) Raw image only (for debugging)
- 2) Score mask over black screen
- 3) Score mask over raw image

C. Memory Usage

This design uses approximately 553 kilobits of BRAM (out of 2,760 kilobits available) and approximately 1.8 MB of DRAM (out of 128 MB available). Built-in modules are used to interact with BRAM and the open-source UberDDR memory controller, with additional wrapper modules, is used to interact with DRAM.

Most of the BRAM usage comes from the image pyramid, whose line buffers require 24 parallel BRAMs, 8 each with capacities of $8 \cdot 1280$, $8 \cdot 1280/2$, and $8 \cdot 1280/4$ bits. The remaining BRAM usage comes from the NMS module, whose line buffer requires 16 parallel BRAMs, each with a capacity of $16 \cdot 1280$ bits, and some filter modules, which require 8 parallel BRAMs, each with a capacity of $8 \cdot 1280$ bits.

DRAM is used exclusively by the frame buffer. The frame buffer stores a 1280 by 720 frame of 8-bit pixels associated with 8-bit scores, which requires $1280 \cdot 720 \cdot 16$ bits in total, or approximately 1.8 MB.

III. IMAGE PYRAMID (KIERA)

The image pyramid converts camera input into three streams of data: one at full resolution, one at half resolution, and one at quarter resolution.

This is accomplished by the means of three line buffers. Each has the same number of lines, but the lower-resolution ones are proportionally smaller and only receive valid data when the coordinates of that data are divisible by 2 or 4, respectively. This is illustrated visually in Figure 2.

Using three line buffers allows us to use smaller line buffers (7 lines instead of 28 lines), fewer parallel BRAMs in total (24 instead of 28), and smaller BRAMs (640 or 320 bytes per line for some), which combines to provide a significant resource saving. The previous implementation for this module used a large line buffer and downsampled its outputs, which caused resource utilization and timing issues. The switch to

multiple line buffers in parallel comes with a minor downside of complicating the propagation of `v_count` values for the three parallel line buffers, but that trade-off is well worth it.

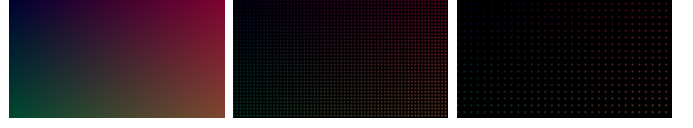


Fig. 2: What each line buffer “sees” in the image pyramid.

The latency of the image pyramid modules is about 90 us, which is the latency of a line buffer with a 7-line output. This is a tiny portion of the overall system’s latency and the benefit of multi-scale corner detection is worth the small delay.

IV. FAST CORNER DETECTION (LIDIA)

After receiving an appropriately-scaled window of 7×7 pixels, the FAST algorithm isolates a ring of 16 pixels around the center pixel and performs a series of comparisons between the pixels in the ring and the center pixel. If a continuous series of 9 pixels within the ring are all either lighter or darker than the center pixel by an intensity of at least `THRESHOLD`, then the pixel is classified as a keypoint (corner). FAST on its own has decent performance with a `THRESHOLD=32`, but after implementing a 5×5 Gaussian filter on the inputs to FAST, we were able to observe significantly better performance with even a threshold of 8.

In addition, the “cornerness” score of a keypoint is computed by taking the sum of differences between the ring and center pixels.

We originally intended for FAST to be computed on all three levels of the image pyramid and then combined; however, we found that the corner detection functionality of the 2x and 4x did not provide any meaningful information on top of the 1x version, and even contained false positives. For this reason, we did not include the image pyramid in our final pipeline, although viewing the pipeline through either of the two other downsampling factors can be toggled using ‘`sw[14]`’.

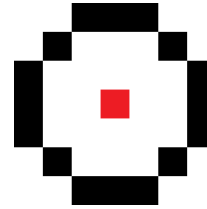


Fig. 3: Pixel circle used for FAST.

FAST has the potential to produce multiple corner markers in a neighborhood of pixels due to intensity gradients. To create a crisper corner detection signal, our system feeds the output scores from FAST into a non-maximum suppression (NMS) module to filter out weaker signals and maintain only the strongest. In our working implementation of the system, connecting the FAST and NMS pipelines in series produced frame errors that we were not able to resolve. However,

“sw[5:4]“ can be used to toggle between viewing the working results of FAST and NMS, with “score inputs” for the latter being pixel intensity values), as well as raw camera inputs.

The FAST module operates on a window buffer created from the output of the image pyramid. Since this window is only 7x7 pixels, the latency of FAST is just $\lceil 7/2 \rceil$ cycles, or about 40 ns. This ensures that detected corners are available to downstream modules as quickly as possible.

The FAST module takes $1280 \cdot 720 \cdot 10$ ns to process a full frame, or about 9.21 ms. This is comparable to software implementations such as OpenCV (Python), which our testing determined to take between 5 ms and 22 ms per frame, depending on various factors and varying even while using the same inputs. However, the software implementation has to process the entire frame before providing any outputs, so its latency for any point in the frame is also 5-22 ms. This proves that our implementation is comparable with software implementations in terms of latency, as well as providing outputs sooner (due to the sliding-window implementation) and having a constant time to process a complete frame, making it easier to integrate with other hardware that relies on corner detection.

V. NON-MAXIMUM SUPPRESSION (KIERA)

The non-maximum suppression module acts like a filter, only passing through values that are a local maximum within a set region (by default, a square 15x15 window). Additionally, the NMS module has support for a built-in threshold function in order to remove the need for a dedicated threshold module.

Non-maximum suppression ensures that spurious corner detections (i.e. those under the minimum threshold or less confident than others in their vicinity) don’t affect the system’s output. This is not obvious visually when using FAST scores, since detected corners are often farther apart than the NMS window spans anyway, but its effect is very visible when used in conjunction with noisier filters, such as those from Lab 7, as shown in Figure 4.

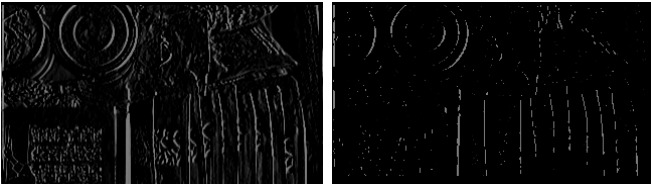


Fig. 4: NMS cleaning up the Lab 7 edge detection filter.

As another module that is based on line buffers, the latency of NMS depends on the latency of the line buffers. Since it uses a 15-line buffer, the latency of the NMS module is therefore approximately 192 us. This is a negligible delay when compared to the time it takes to process an entire frame, which is approximately 9.21 ms.

VI. FRAME ENCODING (LIDIA)

A. Descriptor Generation

In order to identify consistencies in features between frames, the keypoint windows must be encoded into easily-comparable descriptors. We used the Binary Robust Independent Elementary Features (BRIEF) algorithm for this purpose due to its ease of comparison using the Hamming distance. However, since keypoints in the camera frames are subject to discrepancies due to changes in the frame angle, keypoint windows must first be converted into rotation-neutral forms using a rotation matrix applied to every coordinate in the window. The process of computing the rotation matrix is as follows:

- Compute the intensity centroid (c_x, c_y) of a 31x31 pixel window centered at the keypoint as:

$$c_x = \frac{\sum x \cdot I(x, y)}{\sum I(x, y)} \quad \text{and} \quad c_y = \frac{\sum y \cdot I(x, y)}{\sum I(x, y)}$$

- Compute the orientation as

$$\theta = \arctan \frac{c_y}{c_x}.$$

- Compute the rotation matrix as

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

The three steps above can be simplified to produce rotation matrix

$$R = \lambda \begin{bmatrix} 1 & -\gamma \\ \gamma & 1 \end{bmatrix}$$

where

$$\gamma = \frac{c_y}{c_x} = \tan(\theta)$$

and

$$\lambda = \sqrt{\frac{1}{1 + \gamma^2}}$$

Computing λ as above requires two divides performed sequentially, which creates a huge latency bottleneck in the computation. Instead, by rearranging factors, we can obtain

$$\lambda = \sqrt{\frac{c_x^2}{c_x^2 + c_y^2}}$$

which allows us to compute γ and λ in parallel, thus more efficiently utilizing the FPGA’s resources.

Because c_x and c_y are on similar orders of magnitude, inputting them directly into the `divide_sv` module to compute γ would generate an extremely imprecise quotient. As an easy workaround, we left-shifted c_y by eight bits and treated the quotient as a fixed-point value with eight fractional bits. In simulation, this produced tangent values corresponding to orientation angles that were within one degree of the expected outputs. Likewise, we shifted c_x^2 by eight bits before computing the leading multiplier. After performing the square root, the leading multiplier is left with only four fractional bits.

The 256-bit descriptor is generated by comparing a predetermined pattern of 256 pairs of pixels within a 15x15 window around the keypoint. For each pixel, the rotation matrix is applied to generate the rotation-invariant pixel:

$$\begin{bmatrix} h_count^* \\ v_count^* \end{bmatrix} = R \begin{bmatrix} h_count \\ v_count \end{bmatrix}.$$

In hardware, we implemented this as:

```
new_hc << 12 = multiplier · (hc << 8) - γ · vc)

new_vc << 12 = multiplier · (γ · hc + vc << 8))
```

The descriptor is generated using pixel values corresponding to the post-rotation `h_count` and `v_count` used in place of the pre-rotation pixel values.

We note that, due to integer truncation in hardware, the rotation matrix may not generate the rotated pixels with full accuracy. To account for this and also to reduce noise disturbance, we used the blurred version of the frame generated during corner detection. Even with these modifications, we found that the descriptor generation was quite noisy, and even when the camera was covered with a finger (e.g. all pixels were black or at least very dark), the descriptor values still flickered. When we added a parameter `DIFFERENCE_THRESHOLD` with the comparisons, we observed better robustness against noise at the expense of slightly worse precision.

Unfortunately, while we were able to successfully integrate the λ and γ calculations in our system and implement BRIEF, we were unable to debug the memory accesses required to rotate the pixel values to create the rotation matrices.

B. Comparing Frame Keypoints

Given the descriptors of two keypoints, the difference score between them is computed using the Hamming distance, or

$$d = \text{popcount}(A \text{ XOR } B).$$

If the distance between two descriptors is less than 25, then they are labeled as a match. In our system, a match is denoted as a neon-blue mask over the image pixel.

VII. DESIGN EVALUATION

A. Original Project Goals and Extent of Completion

1) Minimum Viable Prototype:

- Demonstrate real-time corner detection using FAST at 25 frames per second (MET)
- Demonstrate robustness against high-frequency noise (MET)
- Offer multiple visual output modes, including at least corner-over-video overlay and "only corners" (MET)
- Demonstrate correct storage of video and score data in BRAM line buffers, with proper boundary behaviors (MET)

2) Goal:

- Demonstrate real-time corner detection using Harris corner detection at 25 fps (NOT ATTEMPTED)
- Demonstrate more accurate corner detection using parallel detection modules with scaled-down inputs (COMPLETED*)
- Prove that this FPGA implementation is faster than an equivalent one in Python (MET)

*did not improve accuracy

3) Stretch Goal:

- Demonstrate real-time corner detection at 60 fps (NOT ATTEMPTED)
- Demonstrate encoding of frame corner features using BRIEF descriptors (ATTEMPTED, PARTIALLY MET)
- Demonstrate storing of previous frame descriptors in DRAM (NOT ATTEMPTED)
- Compute and demonstrate object permanence based on Hamming similarities between descriptors of sequential frames (MET)

Our minimum viable product was implemented and worked about as well as we would have expected our goal to work (hence why we omitted implementing Harris corner detection). We also implemented the other parts of our goal, although didn't use all of them because they did not improve the performance of our system (e.g. image pyramid). Some of our stretch goals were attempted and/or partially met.

Some of our planned features worked but did not work very well when integrated in the FAST pipeline, such as NMS and the image pyramid. On the other hand, some of the features we did not originally intend to implement worked very well, such as the double Gaussian blur and FAST threshold. Overall, we held a very experimental approach to our project, trying different techniques to improve our goal of creating a noise-robust system, and switching gears when our empirical results didn't match our expectation.

Similarly, we also implemented part of BRIEF but found that it didn't work very well with the cameras we were using, although it did work in simulation (`brief_in_python.py` and `test_descriptor_builder.py`). For example, in simulation the algorithm was unresponsive to a black screen (e.g. no corners, no variation in pixel values), but if I covered the camera with my finger to produce a black screen, BRIEF in hardware did not behave in the same way. We were able to implement rotation-invariant brief in simulation (including computing the necessary scalar multipliers and variables), but were unable to debug its integration into the overarching system.

VIII. CONCLUSIONS AND FUTURE WORK

In this report, we demonstrate that an FPGA with limited hardware can be used to perform corner detection with relatively high robustness against noise. We present an empirically-driven approach to reducing noise and maximizing accuracy in the system using techniques such as

image pyramids, blur filters, and scaling manipulations. We optimize for latency and camera frame rate, using modified versions of algorithms such as FAST and BRIEF. We present a system with comparable accuracy to well-established software APIs such as `opencv-python` while being exclusively on hardware.

IX. CONTRIBUTION DETAILS

The modules are noted with their primary contributor above.

Lidia planned out the overall system pipeline and fine-tuned the algorithms used by conducting research and trial-and-error simulation experiments.

Kiera created Figures 2-5 in this report.

Both authors contributed to the report and video in an equal capacity.

X. ACKNOWLEDGEMENTS

We thank Prof. Joe Steinmeyer for his fantastic teaching and incredible support throughout the semester, Monica Chan for project supervision and guidance, Kiran Vuksanaj for developing the 6.205 OV5640 toolchain, and all the rest of the 6.205 staff for the tons of debugging help and making the lab a pleasant environment to work in.

XI. REFERENCES

- 1) Vibhakar Vemulapati and D. Chen, “FSLAM: an Efficient and Accurate SLAM Accelerator on SoC FPGAs,” Dec. 2022, doi: <https://doi.org/10.1109/icfpt56656.2022.9974562>.
- 2) R. Liu, J. Yang, Y. Chen, and W. Zhao, “eSLAM: An Energy-Efficient Accelerator for Real-Time ORB-SLAM on FPGA Platform,” arXiv.org, 2019. <https://arxiv.org/abs/1906.05096> (accessed Dec. 11, 2025).
- 3) “OpenCV: FAST Algorithm for Corner Detection,” docs.opencv.org. https://docs.opencv.org/3.4/df/d0c/tutorial_py_fast.html
- 4) OpenCV, “OpenCV: ORB (Oriented FAST and Rotated BRIEF),” docs.opencv.org. https://docs.opencv.org/4.x/d1/d89/tutorial_py_orb.html
- 5) “OpenCV: BRIEF (Binary Robust Independent Elementary Features),” docs.opencv.org. https://docs.opencv.org/3.4/dc/d7d/tutorial_py_brief.html

XII. APPENDIX A: BLOCK DIAGRAM

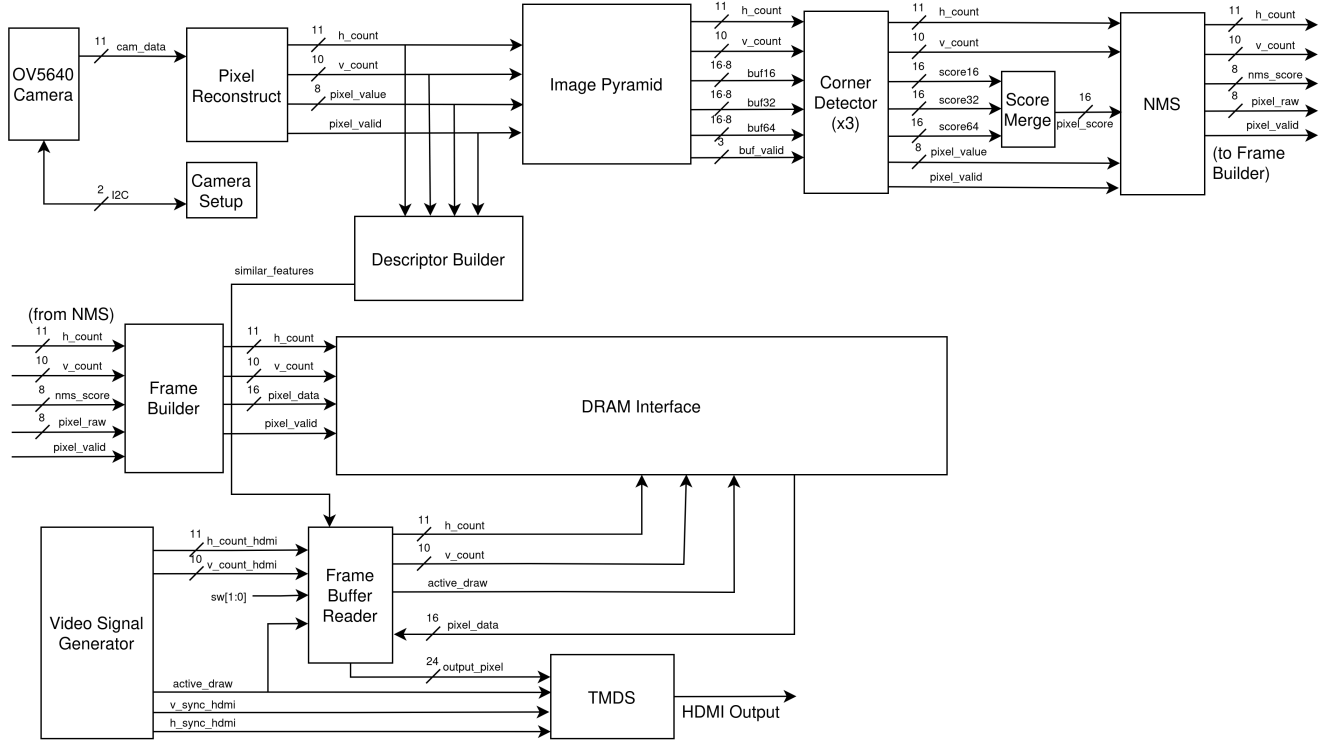


Fig. 5: High-level block diagram of the FFF system.