FPGesture–Controlled Video Enhancement

Soojung Bae
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, U.S.A.
sjb565@mit.edu

Jorge Tomaylla Eme
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, U.S.A.
tomaylla@mit.edu

***Abstract*—We present a design for a Hand Gesture Controlled Video Enhancement device implemented in hardware in FPGA. Through a special glove with pink strips on the fingertips, we design a process to solve the blob detection problem, which aims to recognize all present fingertips, mark their center of mass, and encode their relative position into defined commands. These commands will then dictate the behavior of the video upscaling, which upscales the video resolution by a factor of four with bicubic interpolation method. Thus, with only hand gestures and two cameras, we can manipulate video output in exciting ways. Moreover, by fully serializing the computation in pixel-level, we demonstrate our implementation highly optimizes the usage of memory and computing resources on FPGA.**

***Index Terms*—Computer Vision, Field Programmable Gate Arrays, Video Enhancement, Blob Detection**

I. Introduction

The project is divided into two primary components: Hand Gesture Recognition and Video Enhancement. Hand Gesture Recognition encodes hand movement like pointing up or down to be correlated with commands like zoom out and shift. We tackle the hand gesture component by solving the blob detection problem where we uniquely label continuous groups of valid pixels, referred to as blobs. We then compute the center of mass of these blobs and send the centroid locations to the `hand_command` module to encode the relative positions x and y of each centroid - which corresponds to fingertips - to a useful command that will be fed to the video enhancement component.

In addition, Video Enhancement module upsamples the 128×128 sized low-resolution video stream to 512×512 high-resolution output on the fly. We utilize a bicubic spline algorithm for the enhancement, which in general outperforms bilinear and nearest-neighbor algorithms at the cost of computation complexity [1]. However, by pipelining the convolutions into smaller image patches scanning through the screen with HDMI's raster pattern, we greatly reduced the amount of memory and arithmetic units used on FPGA. Moreover, Video Enhancement module improves the color depth of the pixels; the upsampled image displays a smoother color transition with 2^{24} bit depth, whereas the original image frames from OV7670 camera has a bit depth of 2^{16}.

II. Physical Construction

A. Two AMD Spartan 7 XC7S50-CSG324A FPGAs

We isolate the implementation of Hand Gesture Recognition and Video Enhancement modules into two FPGA boards. This enables us to modularize our system into two primary components running independently. In addition, the boards communicate with each other with physically wired Pmodb ports.

B. Two OV7670 Camera Modules

Each FPGA board has a separate OV7670 Camera Module communicating via Pmoda ports. The camera outputs 30 fps 320×240 resolution video with 16-bit wide color depth.

C. A Glove with Color-Marked Fingertips

We used a thick, dark winter glove as the base, and sticker 5 pink strips with tape to the fingertips. The pink strips came from a well tested pink envelope whose threshold values had already been tuned. The glove's pink visibility was then tested against the camera output, as explained in Section III-A.

III. Hand Recognition

The Hand Gesture Recognition component comprises 5 main modules that allow us to perform crucial operations: filter out screen for valid pixels, downsample them in size to reduce BRAM usage, label connected groups of valid pixels, visualize their centroids, and transform these locations to hand commands. In addition, we make a glove prototype and perform initial tests on possible user test cases.

A. Glove Test Cases

In Fig. 3 we can see 4 recurrent patterns in user cases. In cases (a) and (b) we can see our ideal scenario where the camera output fully captures non overlapping continuous pink areas in the gloves' fingertips. Notice that (a) and (b) occur in a well-lit environment and there is a slight angling backwards in the palm plane and the vertical plane. In cases (c) and (d) we tilt the glove forward to see if the pink fingertips are still accurately captured. In (d) we used longer pink strips, so when tilting the glove, the camera outputs 2 blobs in the fingertips separated by a very thin row of non valid pixels. Similarly, case (c) has shorter pink strips that, when tilted, output gaps in the pinkness of some blobs. To solve this, we have two approaches: down sampling the pixels to eliminate the effect

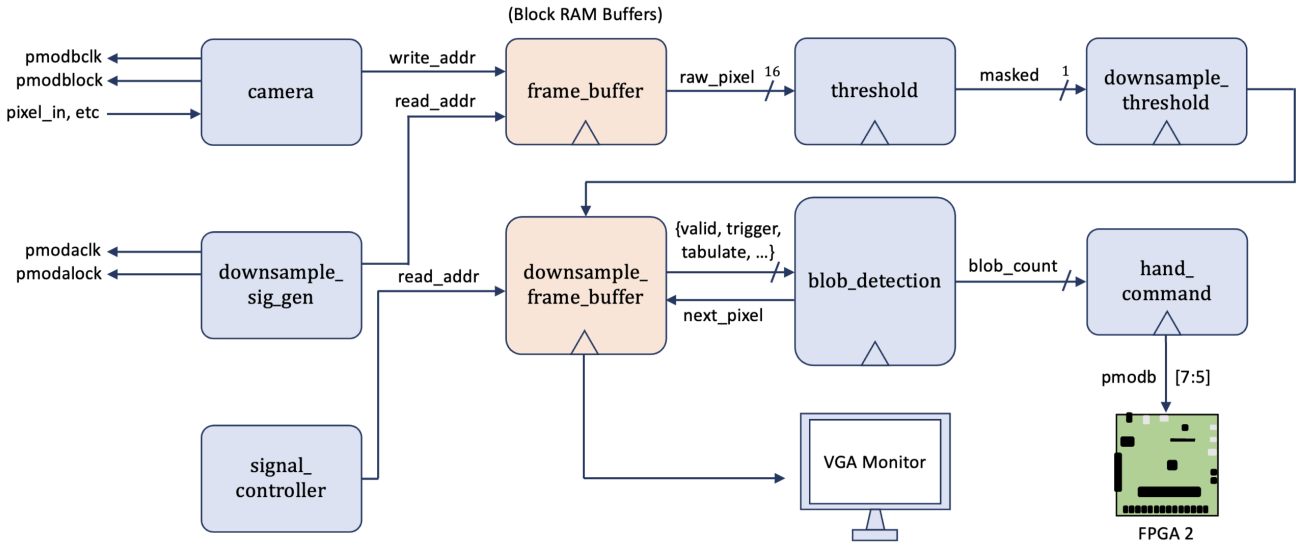


Fig. 1: **Hand Recognition High-Level Block Diagram**.

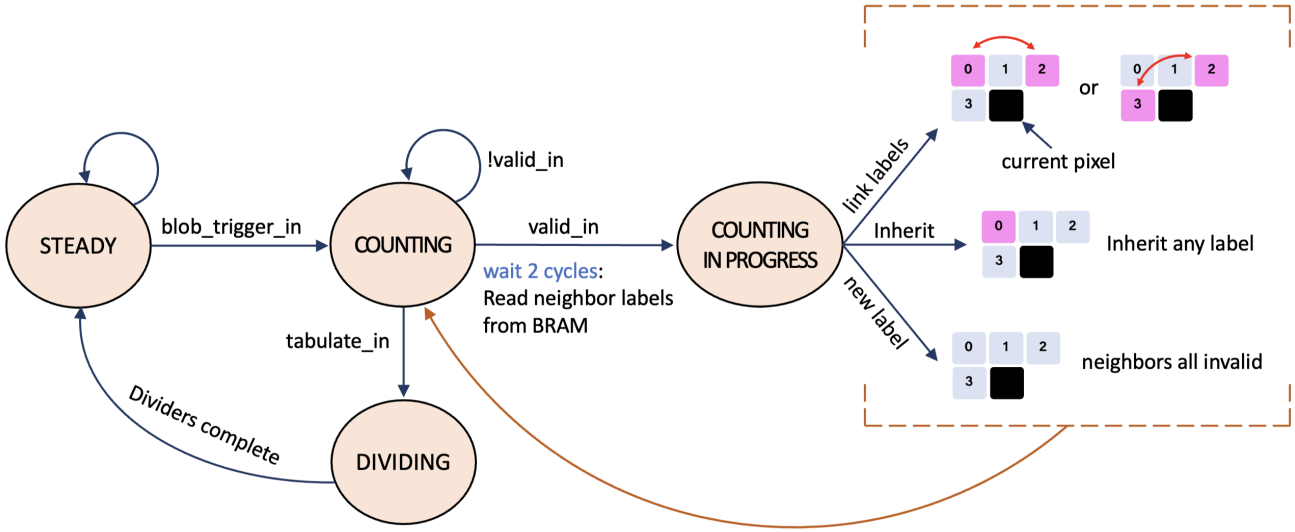
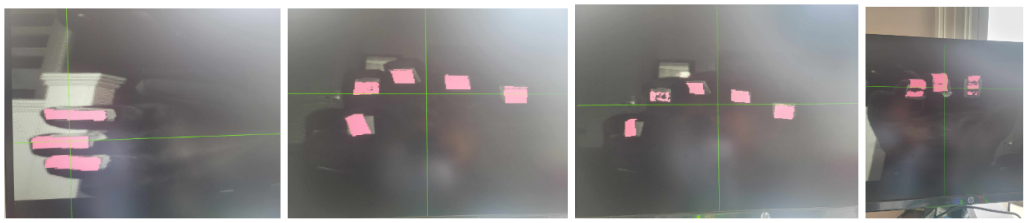


Fig. 2: **Blob Detection Algorithm**.

of the gaps, and further adjusting the upper and lower threshold values.

B. Top Level Pipeline

Fig. 1 displays the top level flow of the blob detection component and the multiple modules that help calculate the blobs' centroids as well as how we send the information to the second FPGA for video upscaling. Our camera saves the video in a frame buffer, which is then thresholded for valid pink values. We then apply a 5x5 filter that outputs 1 if within this 25 pixels there are at least 10 valid pixels. THis reduces the frame from the original 320x240 to 64x48. The justification for this downsampling is that in the original frame we need to process 76,000 pixels to classify into 5 blobs, which takes more cycles to compute as well as double the BRAM, as we would exceed the BRAM height limit. Now, with a downsampled buffer, we only need to process 3072 pixels, easily stored in half the BRAMS, and requires only 2 cycles per processed pixel to update the blob groups and running sums. In addition, the `signal_controller` module regulates the interaction of the integral modules; for instance, when to trigger the blob detection algorithm, when to wait for a fresh downsampled frame, and what values to read from the downsampled buffer. The blob detection output is then printed on the LEDs for visualization purposes and is also processed in the `hand command` module to be encoded into readable commands. WE then send these commands via SPI transmitter to the other FPGA. We only need to send 3 bites of information so SPI controller is sufficient for cross communication.



(a) (b) (c) (d)

Fig. 3: **Glove Strips**. Resulting PSNR for this sample is denoted beneath each sample.

C. Downsampling

As noted in the glove test cases, there are instances of scattered single valid pixels in the environment that are not from the pink fingertips. These occur as the reflection of some pink object nearby or not a tight enough thresholding for valid target colors. Furthermore, when we incline the gloves forward by an angle greater than $45°$, blobs often divide into 2 blobs separated by a thin row of non valid pixels. To address these issues, we incorporate a down sampling module that takes 5x5 pixels and stores a 1 if there is a majority of valid pixels (at least 10) or 0 otherwise. This filters out most isolated pixels as well as keep blobs connected if they are subtly separated. Since we reduce the number of pixels by a factor of 25, we also need much less BRAM capacity. In fact, we came up with ways to delete some unnecessary BRAMS by replacing them with running x sums, y sums, and total number of pixels in each blob.
As explained before, the downsampled buffer ports are controlled by the `signal_controller` module.

D. Signal Controller

The `signal_controller` manages the interaction between the `blob detection` module and the downsampled frame buffer. We trigger the blob detection algorithm - and therefore the reading of the downsampled buffer values - when we reach the vcount value 320. This is because with every new frame, we need to scan through the original sized frame to apply downsampling. Additionally, this module generates the x_in and y_in valued to read from the downsampled buffer, and it alerts the `blob detection` module when this downsampled frame read has finished, which is integral to triggering the DIVIDING state in the algorithm state machine. Finally, the `blob detection` module sends a "next pixel ready" signal to the signal controller, which indicates that the downsampled pixel has been properly labeled and all relevant BRAMs and sums have been updated, so we are ready to analyze the next downsampled pixel.

E. Threshold

The `threshold` module takes in 2 8-bit numbers and outputs 1 if the pixel we are looking at falls within the threshold boundaries.In context to our project, we first filter the camera through the Chroma Red channel to target the bright pink of the glove's fingertips. We then pass in the values 0xF0 and 0x20 as upper and lower thresholds. As seen in the glove test cases, sometimes there are isolated pixels or a blob is bisected by a single row of non valid pixels, so a further tight tuning of the threshold values will increase the accuracy of the camera's valid pixels.

F. Blob Detection Module

This is the module in charge of solving the blob detection problem. It is a flow diagram with 3 states: WAITING, COUNTING, and DIVIDING.

1) WAITING: We cannot begin the algorithm right away at the start of a new frame because we must first wait for the original 320x240 frame to be properly downsampled. This state keeps the algorithm dormant so that it does not start reading pixels from the downsampled buffer until we have made sure the downsampled buffer BRAM has been properly updated, at which point we transition to the COUNTING state.

2) COUNTING: Fig. 2 illustrates the flow of the blob detection algorithm. In short:

- We read an incoming downsampled pixel p, is it a valid threshold value? If not, we do not update any BRAMs or sums and we wait for the next pixel. If yes, we continue.
- We will write the pixel in the VALID BRAM as a valid pixel.
- We then proceed to check the valid bits of the neighbors of the current pixel. Specifically, the 3 pixels above the current pixel and the one directly to the left. If no valid neighbor pixels are found, we can start a new label and put the current pixel in the next available BLOB BRAM. If there is only 1 type of valid neighbor label, the current pixel will inherit that label, updating both the VALID and BLOB BRAMs. Lastly, if there are 2 valid neighbor labels, it means that the currency pixel is the intersection between 2 existing blobs. Therefore, we will inherit the smallest label and link the 2 blobs in an array so that we avoid rewriting all pixels' labels. In the actual implementation, we arbitrarily choose the label of the neighbor to the left.

3) DIVIDING: Once we finish going through the downsampled pixels, the signal controller will trigger an end of downsample frame signal which will trigger the DIVISION state. The division module takes an undetermined amount of cycles so there is a state machine regulating the process to move on only when we are done calculating all centroids. An

important detail, and source of stressful hours of debugging, is that we must make sure that all labels that were linked throughout the COUNTING state are properly combined into 1 blob and that the linked labels' center of mass are not calculated. In the implementation, we created an array that maps label to label if they are merging. We limited this array to only map to 1 value. This means, when DIVISION is triggered, we look into this array, add the x sum, y sum, and pixel total to the first label, and stop the calculation of the label that was linked to by the original label. A clear problem with this is that the second label could in turn be linked to another label, which in turn could be linked to the next one and so on. We did not think we would encounter such cases based on our user cases, but it turns out a pixeled slope in the shape of y = x performs badly. This is because the shape of the blob is a stair case, and when encountering a new step, the algorithm will give the step a new label, which will be then linked to a previous label. The lack of linked list handling in the algorithm yields to erroneous blob counts for such blobs (output of 5 when there is truly 1 poorly slanted blob). This is a feature needed for a more robust performance in the future. //

Once we are done calculating all blobs' centroids, we output the x and y locations of the blobs' centroids, as well as a 5 bit valid blob array that indicates if the ith blob was present in the frame. .

G. Crosshair

This module allows us to visually evaluate the accuracy of the center of mass calculation for each blob. It displays one thin green cross pattern in the screen per label detected. In our design we have defaulted the maximum number of blobs or labels to be 5. In practice, we found it more helpful to link the valid blob array output to the LEDs and 7 segment display for easier debugging and examination of blob test cases.

H. Hand Command

The valid blob array output is sent to this module for hand encoding. For simplification, we encode 5 commands:
0 : NOTHING
1 : UP
2 : DOWN
3 : LEFT
4 : RIGHT
5 : NEXT FILTER
We sum the number of blobs present in the screen to calculate this result, although more mathematical formulas are encouraged for more specific commands in future iterations.
This data is then send to the next FPGA for video enhancement through an SPI controller. We do special wiring of the camera device and pmodb and pmoda ports to facilitate communication between both FPGAs. This can be shown in the video explanation for the project https://www.youtube.com/watch?v=FiGxE-KXj5g.

IV. Video Enhancement

The Video Enhancement module consists of several submodules that enable serializing the bicubic convolution of the entire frame into a series of convolutions of 4×4 image patch. Specifically, 4×4 image patch scans through the frame in left-to-right, top-to-bottom order, calculating the arrays of upscaled pixels on each step. The resulting upscaled pixels are buffered in 4 Block RAMs, which are then retrieved whenever requested by the HDMI display signal.

A. Upscale Signal Generator

As upscaled pixel values are constantly rewritten and buffered on the Block RAMs, it is important to control the timing to separate buffer read and write operations. Therefore, `upscale_sig_gen` module inputs the current HDMI control signal (horizontal/vertical coordinate, active draw bit, etc.) and returns (1) valid signal which initiates BRAM writes, (2) first read address to access frame buffer for image filtering, and (3) second read address to access filtered frame buffer for image upsampling. Controlled by the valid signal, image filtering and upsampling only takes place when HDMI is not actively drawing pixels on the monitor.

B. Image Filter

`image_filter` module demonstrates how arbitrary kind of 3×3 image filters can be integrated into our design. Since image filters may contain both positive and negative coefficients, the problem arises from clipping the output into the right value since merely using two's complement would not differentiate value overflows and underflows. Thus we divide each filter into two non-negative filters each containing only positive or negative components. The result is obtained by simply subtracting and clipping the output of two filters, which will be stored in `filtered_frame_buffer` as shown in Fig. 7. Our implementation includes 3×3 image patches for Gaussian blur, Gaussian sharpening ($f = 0.5$), and Sobel derivative (∇_x, ∇_y) filters.

C. Pixel Shifter

As shown in Fig. 5, `upscale_sig_gen` module scans through the `filtered_frame_buffer` in column-major order to construct 4×4 image patches for convolution. In addition, since the size of the four-times upscaled frame (1280×960 pixels) exceeds the HDMI display size, the module shifts the offset read address in accordance with the control signal from the hand recognition module.

Once the pixels are retrieved from the camera's frame buffer, the serialized pixel values are stored in `pixel_shift` module which acts as a secondary buffer. As shown in Fig. 5, when 4×4 patch scans through the frame, it is sufficient to only replace the first and last column to form the next patch. Thus we utilize a FIFO buffer, `pixel_shift`, that stores 16 most recent pixel inputs and outputs a valid bit every four cycles whenever the valid image patch is ready.

D. Bicubic Interpolation

In the next pipeline stage, 4×4 image patches are upscaled both in spatial dimension and color depth. As we use bicubic spline algorithm, each output pixel is a result of multiply-accumulate between input pixel array and the predetermined

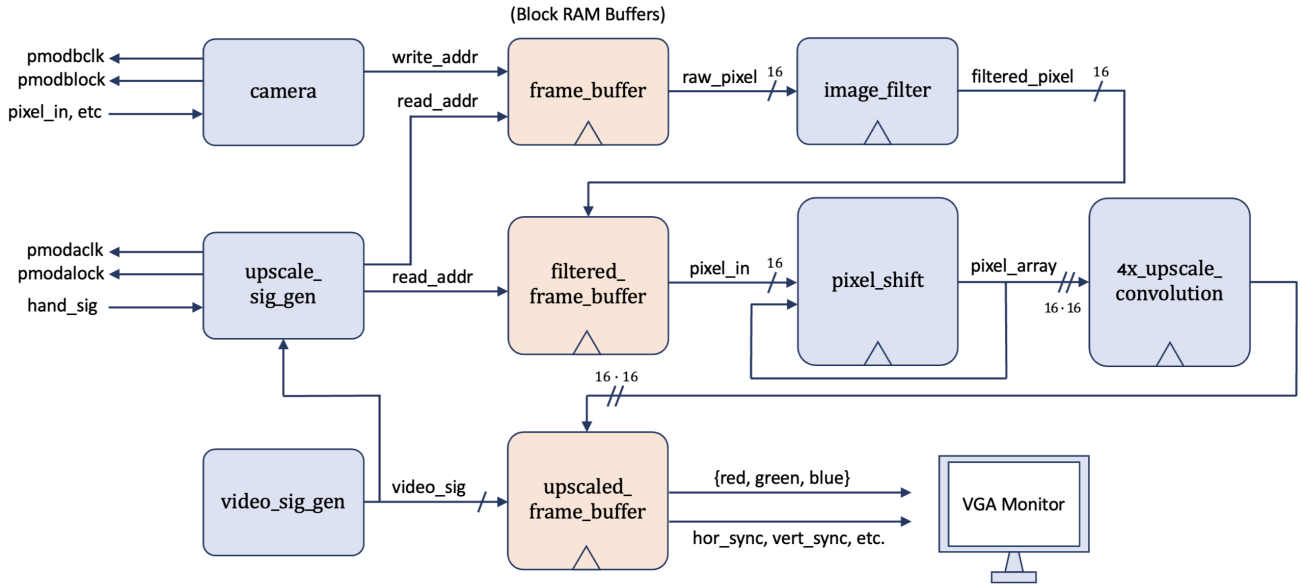


Fig. 4: **Video Enhancement High-Level Block Diagram**. Output pixels are encoded and serialized in accordance with the HDMI specification before connected to a VGA monitor. Buffers for storing temporary frames are colored in orange.

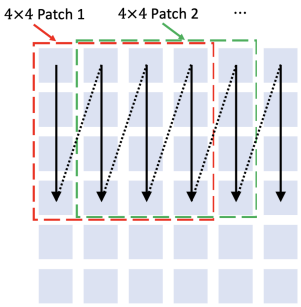


Fig. 5: Pixel values read from frame buffer in the raster pattern depicted with black arrows, where 4×4 patch for convolution is formed every four cycles.

kernel coefficients. Theoretically, this requires 16 different kernels for the 16 upsampling locations, whereas the total number of arithmetic operations required will be:

$$N_{\text{output pixels}} \times (N_{\text{addition}} + N_{\text{multiplication}}) = 16 \times (16 + 16) = 512. \quad (1)$$

Although the number of operations is greatly reduced owing to pipelining, it is intractable to transport the logic on Spartan7 FPGA which at most provides 120 DSP48 blocks for complex multiplications. In addition, floating point arithmetics will double the data width of each pixel from 16 bits (camera output) to 32 bits floating point without gaining any benefit on the accuracy.

Therefore, several design choices are made to resolve such issues: First, we approximated floating point operations to integer multiplications. As the image is upsampled by a factor of four, kernel coefficients are rational numbers with denominators expressed as the power of two. Thus non-integer operations are replaced by integer operations and bit shifts in the final stage. Second, integer multiplications are further replaced with bit shifts and chained additions. For instance, we utilized the equivalence $n \times 385 = (n << 8) + (n << 7) + n \ (\forall n \in \boldsymbol{N})$. Finally, the sum of 16 output values of multiplications is calculated through multiple pipeline stages to meet the clock constraints. As a result, `bicubic_interpolation` consists of four pipeline stages and utilizes only two DSP48 blocks.

E. Upscaled Frame Buffer

As the output image is upsampled four times larger, a basic FIFO structure can't resolve the timing mismatch of production and consumption of pixels. The essential difference from the situations using ordinary buffers is that our upsampling module produces four rows of pixels simultaneously. Thus we decided to use four dual port Block RAMs to store each row of pixels on each Block RAM.

V. Evaluation of Blob Detection

The entire code can be found in the following git repository: https://github.com/sjb565/Fpgesture-controlled-video-enhancement.

A. Centroid Visualization

The video explanation of this project showcases the performance of the blob detection module. For a small number of pixels we accurately label blob counts, but for bigger numbers, the algorithm fails to correctly merge labels and blobs together, especially in the staircase blob case. The hand commands were accurately deterministic and sent consistent information to the Video enhancement module, as displayed in the video.

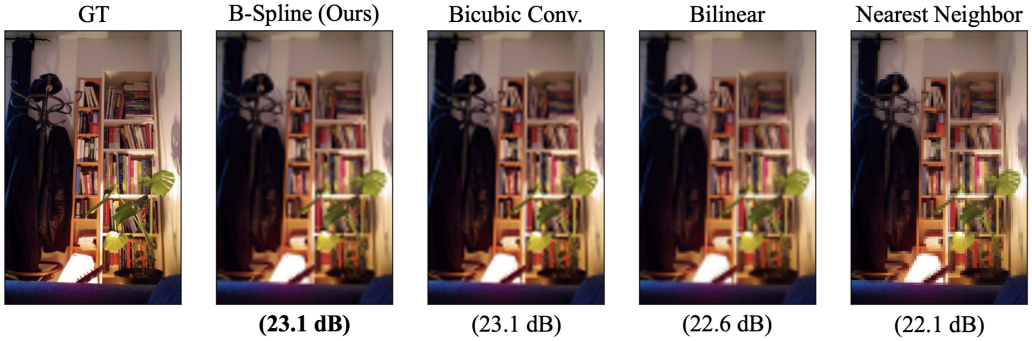


Fig. 6: **Comparison of upsampling methods**. Resulting PSNR for this sample is denoted beneath each sample.

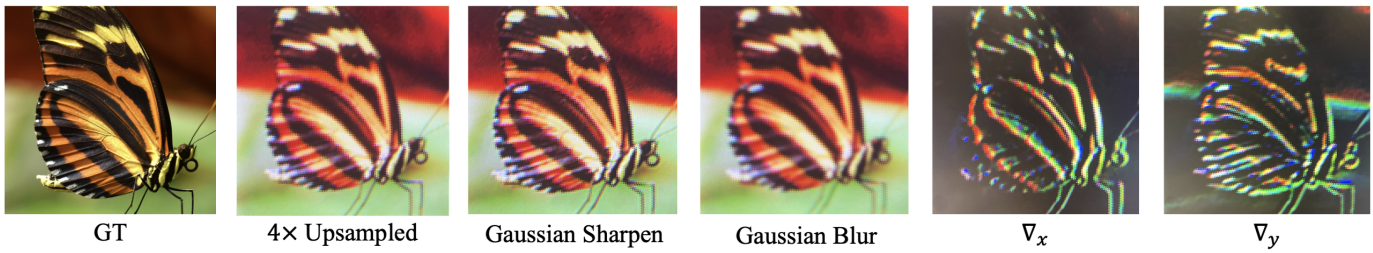


Fig. 7: **Sample Images of Hardware Implementation Results**. The photos of upsampled results are taken with external camera device (not directly from the VGA monitor). The image filters (e.g., ∇_x, ∇_y) are applied independently on each RGB color channel.

VI. Evaluation of Video Enhancement

A. Software Verification on Real-Life Data

As our upsampling implementation uses hard-coded numerical coefficients as parameters, it is essential to test and compare the performance of the method with reliable software implementations. To test on real-life data, we extracted 35,000 random samples from the ImageNet Dataset and scaled down the samples' sizes to 512×512 [2]. Note that each method's performance relies heavily on the choice of the dataset, and hence ImageNet Dataset with real-life objects and scenes is used for this experiment. In addition, we compared our hard-coded numerical kernels with the upsampling methods from the OpenCV Python Library [3].

TABLE I: Performance of Upsampling Methods

Interpolation Method	**PSNR** (dB)
B-Spline (Ours)	25.63
Bicubic Convolution	**25.79**
Bilinear	25.04
Nearest-Neighbor	24.23

* $\alpha = -0.75$ for Bicubic Convolution

As shown in Tab. I, our upsampling method shows similar level of Peak Signal-to-Noise Ratio (PSNR) compared with the OpenCV's bicubic interpolation method and outperforms Bilinear and Nearest-Neighbor algorithms. The sample result of the experiment can be found in Fig. 6.

B. Hardware Realization

Fig. I represents the sample results from the hardware implementation of our system. Due to the external lighting condition, upsampled images have different color temperature and brightness compared to the digital ground truth image.

TABLE II: Resource Utilization

Type	**Site Type**	**Used**	**Available**	**Util [%]**
Slice Logic	Slice LUTs	6852	32600	21.02
	Slice Registers	8362	65200	12.83
DSP	DSP48E1	2	120	1.67
Memory	RAMB36	48	75	64.00
	RAMB18	5	150	3.33

TABLE III: Block RAM Allocation

Buffer Type	**Data Width**	**Depth**	**RAMB36**	**RAMB18**
Frame Buffer	16 bits	320×240	48	-
Filtered Buffer	16 bits	131×4	-	1
Upscaled Buffer	24 bits	512×4	-	4

Tab. II shows the summary of our system's resource utilization. As described in Section IV-D, the video enhancement module utilizes 21% of available LUTs while using only two DSP blocks, by substituting bit shifts for multiplications. On the other hand, our system utilizes 67.33% of available Block RAMs. The details of memory allocation are presented in Tab. III, which indicates most of the resources are allocated

to `frame_buffer`. The realization of actual components for video manipulation suffice with only five RAMB18 blocks.

For the system's latency, our system performs upscaling at the rate of 60 fps since its operations are synchronized with an HDMI signal. Specifically, hardware signals besides the TMDS signal run on a 74.25 MHz clock rate, or 13.468 ns clock cycle. The timing requirement is satisfied with an 8.839 ns data path delay.

REFERENCES

[1] D. Han, "Comparison of commonly used image interpolation methods," in *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*. Atlantis Press, 2013/03, pp. 1556–1559. [Online]. Available: https://doi.org/10.2991/iccsee.2013.391

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[3] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.