High-Speed Imaging with Light-Deserializing Coded-Exposure-Pixel CMOS Image Sensor

CSC2529 Final Project Report

Instructor: David Lindell

Students: Jangwon Suh, Roberto Rangel

Abstract

The demand for high-speed, high-quality image sensors has surged due to modern applications requiring rapid adaptability to changing motion and lighting conditions. Conventional image sensors, commonly found in budget smartphones, face limitations with slower capture rates, resulting in motion blur and compromised image quality during fast action and sudden lighting shifts. While high-speed sensors can mitigate motion artifacts, their high cost and power requirements render them impractical for many applications. Recently, coded-exposure cameras have emerged as a viable alternative, enabling burst capture with low power by compressing high temporal resolution into single captures; however, it is desirable to achieve similar performance for continuous video capture. In this work, we propose a continuous compressive image acquisition system utilizing coded exposure with simultaneous exposure and readout control. This innovative approach achieves both flexible spatial and temporal resolution while maintaining continuous video capability. Our system operates at a native low-power readout of 30 frames per second (fps) and achieves an impressive effective output video rate of 480 fps with a 10x reduction in dead time compared to prior implementations. This results in satisfactory performance with seamless motion capture between consecutive frames, enhancing image quality and adaptability in real-time scenarios.

Introduction

Coded exposure was initially implemented using digital micromirror devices (DMDs), enabling users and computer vision systems to exercise considerable control over the radiometric and geometric properties of the imaging system. [1].

In addition to exploring variations in field of view, radiometric, and geometric properties, previous studies have investigated the temporal characteristics of various coded-exposure techniques, including their applications in deblurring [2]. In conventional single-exposure photographs, motion blur can occur due to moving objects or camera motion. However, coded-exposure techniques can rapidly open and close the shutter using a pseudo-random binary sequence during exposure, allowing the motion blur to retain decodable details of the moving subject.

Recent advancements have transitioned from DMDs to electronic exposure control at the CMOS pixel level. Coded-Exposure Pixel (CEP) cameras permit exposure programming at the individual pixel level [3]. Unlike global-shutter pixels that expose all pixels for the same duration, CEP cameras can partition each frame into multiple subframes (N subexposures), enabling selective light sensing within each subexposure. This capability allows for advanced multi-exposure imaging within a single readout frame, facilitating the analysis of scenes multiple times per frame with varying camera and illumination codes. This provides enhanced flexibility in how scenes are illuminated and how incoming photons are selectively captured.

Previously, coded exposure was applied to generate a compressive burst video output with locally varying speeds by assigning specific coded patterns to different motion regions [4]. While this strategy effectively balances high-spatial-resolution static areas with high-temporal-resolution moving areas, it still encounters limitations in continuous video output due to dead time.

Figure 1 provides a depiction of how a coded exposure camera operates. In the left section of the figure, we can observe the sensor's exposure phase, during which light integration is modulated through pixel masking. This masking process can be performed rapidly and multiple times, allowing the overall exposure period to be subdivided into smaller, programmable time units referred to as subexposures. For each of these subexposures, we have the flexibility to select specific pixels that will collect photocharges, enhancing control over how light is captured.

In this project, we utilize a sophisticated camera design where each pixel has two charge-collecting nodes, or taps. This feature allows the masking operation to direct charge integration to either of the two nodes, providing further customization in how light data is gathered since no light information is lost.

One application of this feature, explored in this project, is the creation of masks that adopt Bayer-like patterns to distribute exposure across selected subsets of pixels over time. This approach enables us to choose tiles of varying sizes (NxN), resulting in a total of N^2 subsets from the pixel array. Each subset captures different exposures of the scene throughout the exposure period, thereby allowing these subsets to be exposed at a rate N^2 times faster than the camera's overall exposure rate. However, this advantageous increase in temporal resolution comes with a trade-off: while enhancing the ability to capture rapid movements, it simultaneously reduces the spatial resolution by a factor of N.

A drawback of the existing coded-exposure camera is that the readout operates independently and does not align with the exposure phase. During the readout process, exposure effectively ceases, creating a temporal gap between full frames that is referred to as deadtime. Although this feature still permits burst imaging, it presents significant challenges for achieving continuous video capture, limiting the camera's effectiveness in dynamic filming scenarios.

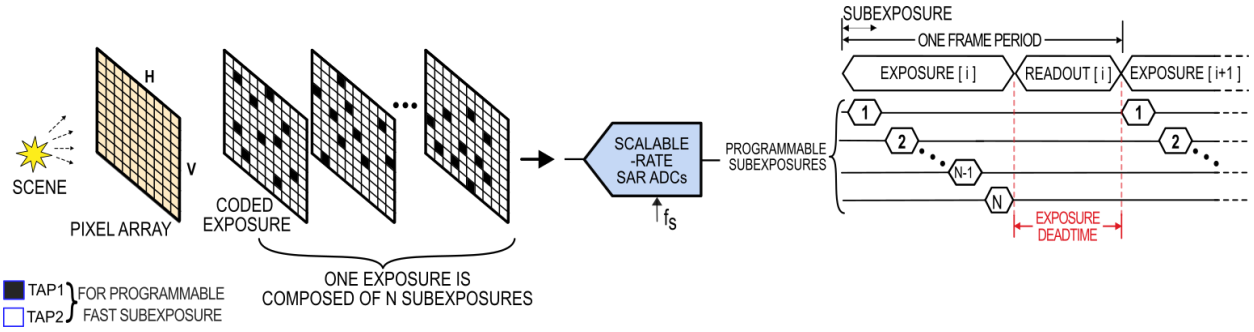


Fig. 1. Overview of a coded exposure camera operation. The left part illustrates the exposure phase, highlighting how pixel masking modulates light integration. The timing diagram demonstrates the subdivision of the overall exposure period into programmable subexposures, and the deadtime period during consecutive exposure phases. Source: adapted from [4].

This project seeks to achieve a substantial reduction in exposure deadtime, allowing for continuous video capture with a seamless transition between full frames. This is accomplished by modifying the camera control firmware to permit simultaneous exposure and readout, which significantly decreases the time gap between exposure phases. We present experimental results that include a qualitative analysis of images captured using the same set of masks with both firmware versions, comparing the perceptible effects of deadtime in the video frames. Furthermore, we provide a quantitative analysis by measuring the absolute time intervals detected between exposure phases.

System Design

The following subsections detail the development of the firmware, software API, and capture & post-processing software.

Figure 2 illustrates the proposed operation of the sensor during the exposure and readout phases. On the left side of the figure, the Bayer-like masks applied during exposure segment the pixel array into tiles of size NxN, referred to as super-pixels. The timing diagram on the right outlines the exposure and readout operations.

Because one unit pixel per tile integrates photocharge during each subexposure, it requires N subexposures to fully expose a set of rows, with each of those rows separated by N-1 additional rows. Consequently, the proposed readout operation can start without waiting for all subexposures to finish. Instead, it waits for N subexposures to complete and then scans through the specific rows that were just exposed. This approach allows the readout to start earlier, in parallel with the exposure phase, thereby reducing significantly the time gap between exposure phases. This method utilizes a dual-port selection feature that allows for simultaneous exposure coding of certain pixel array rows while enabling readout for other rows [5].

In this work, we utilize masks of size 4x4, meaning that the readout of a specific set of rows initiates after every four subexposures. For instance, the first readout cycle will capture data from pixels in rows 0, 4, 8, and so on, while the second cycle will capture data from rows 1, 5, 9, …, and this pattern continues for N readout cycles until all rows are read, resulting in 16 subframes. Given that the full frame rate is 30 fps, the effective video capture rate from these subframes increases to 16 times that, or 480 fps. The sensor features a VGA-sized pixel array (640x480), meaning that the resolutions of the subframes will be divided by 4 during the tiling division, resulting in a final video output frame size of 160x120.

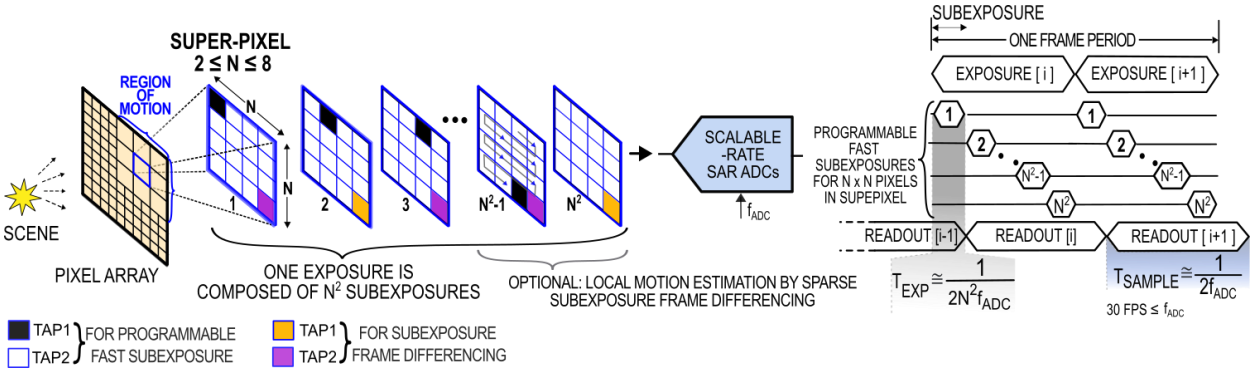


Fig. 2. Overview of proposed continuous high-speed operation. The left part illustrates the masking using tiles to divide the pixel array into super-pixel units. The timing diagram demonstrates the modified dynamics allowing simultaneous exposure and readout, which makes the deadtime negligible. Source: adapted from [4].

Camera Firmware Implementation

This section explores the implementation of a control system for the image sensor, utilizing an FPGA. The proposed camera firmware enables simultaneous exposure and readout leveraging the sensor's dual-port row driver [5]. Central to this design are the finite state machines (FSMs) realized in Verilog HDL, which govern the operation of both exposure and readout phases. The focus of this implementation is to achieve simultaneous image capture, as outlined in the included timing diagram, thus facilitating the continuous acquisition of high-speed video footage. In this discussion, we will present the architecture of the FSMs.

Figure 3 provides a simplified depiction of the state diagram for the FSM that controls the exposure process in the imaging system. This diagram illustrates the states and outputs. By default, it resets the photodiodes in preparation for the exposure phase. The next state takes care of global exposure of the photodiodes. During this process, global shutter operations are employed, allowing simultaneous exposure of all pixels. It is followed by a meticulous row-wise scanning procedure that masks certain pixels within each row.

Furthermore, at the conclusion of every four sub-exposures, the RO_start signal is asserted, instructing the readout module to proceed with reading the next set of rows.

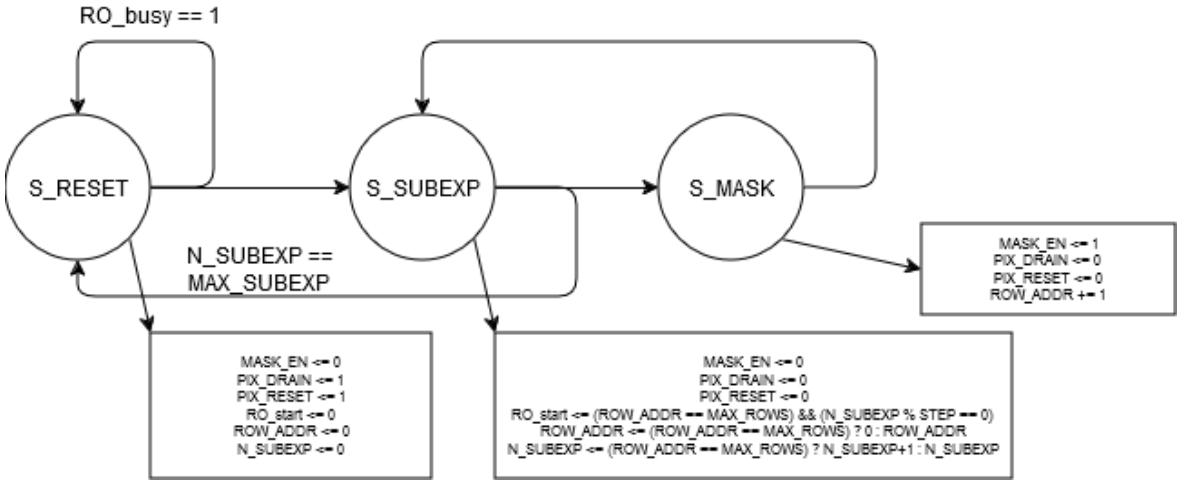


Fig. 3. State diagram of the FSM for exposure. It illustrates the sequence of states and outputs involved in resetting the photodiode, executing 16 sub-exposures using global shutter operations, and managing the transition from exposure to data acquisition via the RO_start signal.

Figure 4 illustrates the state diagram for the FSM that controls the readout phase. It features a "wait" state, which keeps the readout block in standby mode until a RO_start signal is received from the exposure FSM. Upon receiving this signal, the FSM initiates the conversion phase, during which it reads the rows corresponding to the last four sub-

exposures performed. This approach ensures that each row is read immediately following its exposure.

The conversion state is executed over 12 clock cycles, as this duration is necessary for the SAR ADC to convert the 12 bits of digital output. Once the conversion is complete, the S_OUTPUT state activates the output serializer, which transmits the converted bits generated by the ADC.

The FSM must manage the sequence of row addresses carefully to align with the pattern established in the exposure mask. In this instance, a Bayer-like pattern of 4x4 tiles is employed in the sub-exposure masks, resulting in a row scanning step of 4. For example, the first set of rows to be read will include 0, 4, and 8, among others. After completing the first set, the FSM returns to the wait state, preparing for the next set of exposed rows. At this point, the ROW_OFFSET variable is incremented, allowing the subsequent set of rows to be read as 1, 5, 9, and so on, continuing until all rows have been processed. For a 4x4 tile, this means that four readout cycles are necessary to read all the rows.

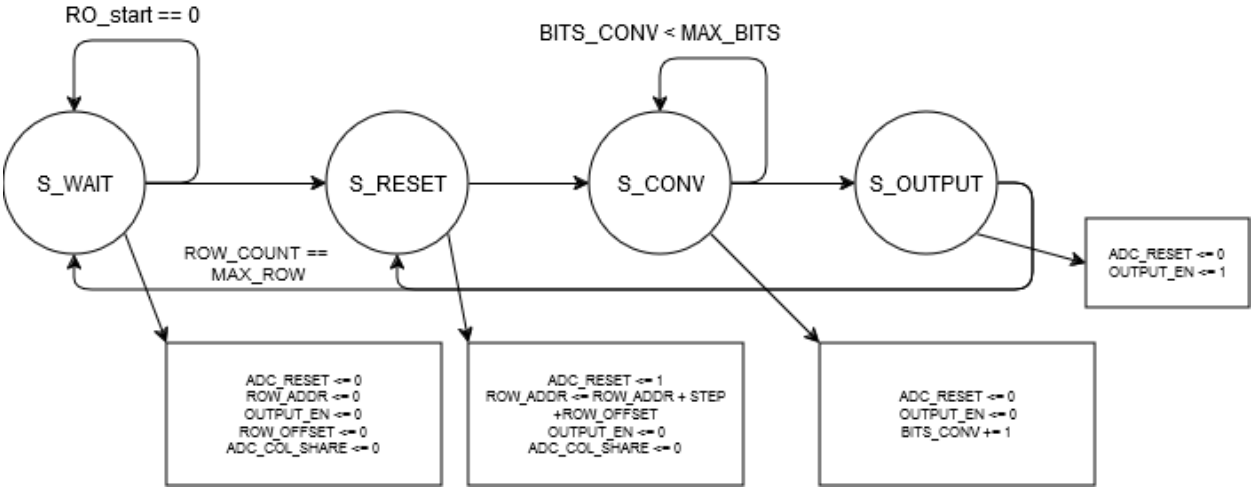


Fig. 4. State diagram of the FSM controlling the readout phase, illustrating the transition from the "wait" state to the conversion and output states, and detailing the sequential management of row addresses based on the Bayer-like 4x4 tile pattern.

Software API development

This subsection outlines the modifications made to the software API, particularly regarding the rearrangement of data during readout and the demosaicing algorithm for reshuffling compressed subframes. In this camera system, the Python programming language is utilized to develop software that controls camera operations, including the configuration of essential operating parameters, raw image acquisition, and subsequent post-processing of the obtained raw images.

In the application examined in this work, Python scripts are also employed to generate Bayer-like pattern masks that define the subexposure operation, and for the demosaicing of the captured encoded images.

The following code is utilized to create the subexposure masks, which are saved in a bitmap image file format. This bitmap file is read by the script responsible for communicating with the camera, which handles the transmission of data as a bitstream to the camera. The creation of the mask file is executed using the function **create_tiled_stack** to generate the numpy array for the mask, and the function **save_to_bmp** to produce a binarized bitmap file.

```
def create_tiled_stack(height, width, tile_size):
    num_layers = tile_size**2  # Total number of unique layers needed
    # Initialize an empty list to hold each full-layer array
    stacked_arrays = []

    # Create each layer by setting a "1" bit in the appropriate position within the tile
    for layer in range(num_layers):
        # Create a single tile-sized array
        tile = np.zeros((tile_size, tile_size), dtype=np.uint8)

        # Determine the position for the "1" bit within the tile
        bit_y = layer // tile_size
        bit_x = layer % tile_size
        tile[bit_y, bit_x] = 1

        # Replicate the tile to cover at least the desired height and width
        layer_array = np.tile(tile, (height // tile_size + 1, width // tile_size + 1))

        # Crop the array to the exact desired dimensions
        layer_array = layer_array[:height, :width]

        # Add this full-layer array to the stack
        stacked_arrays.append(layer_array)

    # Stack all the layer arrays vertically
```

```
    final_array = np.vstack(stacked_arrays)

    return final_array

def save_as_bmp(array, filename="burst_mask.bmp"):
    # Convert to an image format (multiplying by 255 to make it binary: 0 and 255)
    image = Image.fromarray(array * 255)
    image = image.convert("1")  # Convert to 1-bit monochrome
    image.save(filename, "BMP")


# Create mask using created functions
height = 480
width = 1024
tile_size = 4

mask_array = create_tiled_stack(height, width, tile_size)

filename = f"HS_mask_{tile_size}x{tile_size}.bmp"
save_as_bmp(mask_array, filename)
```

The camera API needed modifications to accommodate changes in the arrangement of readout data, as the pixel array rows are no longer read in sequential order. Previously, data for different rows was received sequentially. Consequently, adjustments in the code were necessary to adapt to the new row ordering. Below is a Python function used to generate a raw image with the correct row arrangement.

```
def rearrange_deadtime(img, step=4):
    height = img.shape[0]
    width = img.shape[1]

    img_rearranged = np.zeros((height, width))
    for offset in range(step):
        img_rearranged[offset::(2*step), 0::2] = img[(offset*height//step):(offset*height//step +
height//(2*step)), 0::2]
        img_rearranged[(offset+step)::(2*step), 0::2] = img[(offset*height//step):(offset*height//step +
height//(2*step)), 1::2]
        img_rearranged[offset::(2*step), 1::2] = img[(offset*height//step +
height//(2*step)):(offset*height//step + height//(step)), 0::2]
        img_rearranged[(offset+step)::(2*step),1::2] =
img[(offset*height//step+height//(2*step)):(offset*height//step+height//(step)), 1::2]
    return img_rearranged
```

The following Python function implements the demosaicing algorithm designed to reorganize pixel data into sequential subframes. This process utilizes the raw encoded image as input, along with the specified tile size corresponding to the Bayer-like mask.

```python
def reshuffle_deadtime(img, step=4):
    tile_size = step
    height = np.shape(img)[0]
    width = np.shape(img)[1]
    subimage_height = height // tile_size
    subimage_width = width // tile_size

    num_layers = tile_size**2  # Total number of unique layers needed

    # Initialize an empty list to hold each full-layer array
    imgs_reshuffled = []

    # Create each layer by setting a "1" bit in the appropriate position within the tile
    for layer in range(num_layers):
        # Create a single tile-sized array
        tile = np.zeros((tile_size, tile_size), dtype=np.uint8)

        # Determine the position for the "1" bit within the tile
        bit_y = layer // tile_size
        bit_x = layer % tile_size
        tile[bit_y, bit_x] = 1

        # Replicate the tile to cover at least the desired height and width
        layer_array = np.tile(tile, (height // tile_size, width // tile_size))

        # Use the binary pattern as a mask to extract the subimage
        subimage = img[layer_array == 1]

        # Reshape the subimage into a 2D array for visualization
        # To maintain visual proportions, compute the new dimensions
        subimage_reshaped = subimage.reshape((subimage_height, subimage_width))

        imgs_reshuffled.append(subimage_reshaped)

    return imgs_reshuffled
```

Software development for video capture

The following Python function is designed to perform post-processing of received raw images, specifically focusing on the calibration of black and bright levels.

```
def image_scale_deadtime(image, black_img, bright_img, gain=False, black=True, dynamic=False,
max_scale=3000, row = 480, col = 680, tab = 2):

    row, col, tab = 480, 680, 2
    max_val = 65535  # Equivalent to 2**16 - 1

    if gain:
        # Precompute bright_img and target medians
        y2_left = np.median(bright_img[:, 30:col - 10])
        y2_right = np.median(bright_img[:, col + 30:2 * col - 10])

        # Create y2 array using broadcasting
        y2 = np.zeros((row, col * tab))
        y2[:, :col] = y2_left
        y2[:, col:] = y2_right

        # Compute slope (m) and intercept (c)
        median_x1 = np.median(black_img)
        den = bright_img - black_img
        den[den == 0] = np.inf  # Avoid divide-by-zero
        m = (y2 - median_x1) / den
        c = median_x1 - m * black_img

        # Apply linear transformation and clip negative values
        img = m * image + c
        np.maximum(median_x1 - img, 0, out=img)

    elif black:
        # Black-level correction: Subtract and clip in-place
        img = np.maximum(0, black_img - image)

    elif dynamic:
        # Dynamic range scaling
        left_right = np.hstack((image[:, :col], image[:, col:]))
        min_lvl = np.percentile(left_right, 10)
        max_lvl = np.percentile(image, 90)

        # Scale and clip dynamically
        np.clip(image, min_lvl, max_lvl, out=image)
        img = max_val - ((image - min_lvl) * max_val / (max_lvl - min_lvl)).astype(np.uint16, copy=False)
        return img
```

```
    else:
        # Default: Use the image as-is
        img = image

    # Final scaling: Clip and normalize
    img = np.minimum(img, max_scale)  # Equivalent to np.clip(img, 0, max_scale)
    img *= max_val / max_scale
    return img.astype(np.uint16, copy=False)
```

Additionally, the following function addresses adjustments related to brightness, contrast, and gamma, enhancing the overall visual quality of the images.

```
def adjust_img_deadtime(img, alpha, beta, gamma, black_level, bright_level):
    # Rescale image intensity range to [0, 1]
    final_frame_scaled = np.interp(img.astype(np.float32),[0, 65535],[0, 1])

    # Apply contrast and brightness adjustment
    final_frame_scaled = final_frame_scaled*alpha + beta

    # Apply gamma correction (optimize if gamma is integer)
    if gamma != 1:
        final_frame_scaled **= gamma

    # Apply black and bright level adjustment
    final_frame_scaled = np.interp(final_frame_scaled,[black_level,bright_level],[0,1])

    return final_frame_scaled
```

Multithreading

While the previously presented methods aim to reduce the deadtime in the image sensor to negligible levels between consecutive exposure phases, one challenge persists. The Python software that manages camera operations for both image acquisition and post-processing introduces its delays, especially during the post-processing stage. To tackle this issue, we performed timing profiling on the Python code to pinpoint bottlenecks and subsequently restructured the code to facilitate multithreading. This allowed us to separate the image acquisition and buffering processes from the post-processing functions. Thus, aiming to achieve continuous live capture at a rate of 30 fps, aligned with the sensor's frame rate.

Figure 5 demonstrates the reduction in frame period achieved through multithreading. In this configuration, Thread 0 handles the image readout and stores it in the buffer, while

Thread 1 retrieves the image from the buffer, processes it, and displays the processed image on the screen.

To maintain data integrity, we implemented a buffer lock, ensuring that only one thread can access the buffer at any given time. This prevents the accumulation of raw images from Thread 0 in the buffer, which can occur due to differences in processing speeds between the two threads. Such accumulation could result in delays in image projection on the screen.

By leveraging multithreading, we can significantly reduce the total frame period, as illustrated in the figure. Under optimal conditions, where the readout and post-processing durations are equal, the total frame period can be reduced by as much as half.

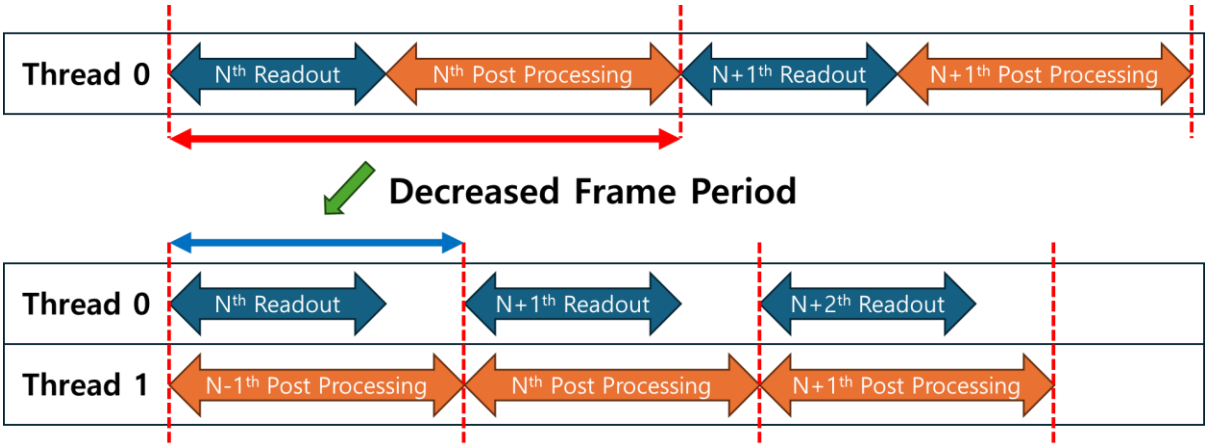


Fig. 5. Illustration of reduced frame period through multithreading, where Thread 0 manages image readout and buffering, while Thread 1 processes and displays the image.

Following is the Python code for implementing the multithreading operation.

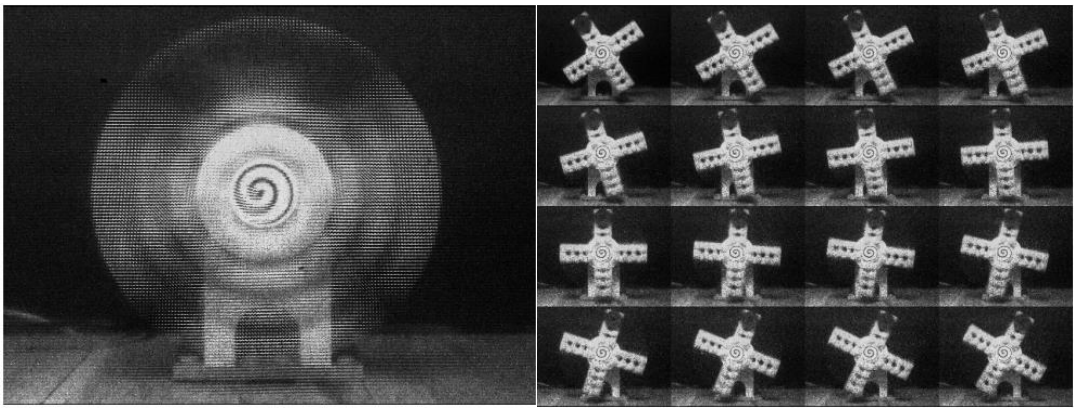
```
# Start threads
reader_thread = threading.Thread(target=image_reader, daemon=True)
processor_thread = threading.Thread(target=image_processor, daemon=True)
reader_thread.start()
processor_thread.start()

# Keep main thread alive
try:
    while reader_thread.is_alive() and processor_thread.is_alive():
        time.sleep(1)
except KeyboardInterrupt:
    print("Exiting...")
    t7.close()
    cv2.destroyAllWindows()
```

Experimental Results

For this experimental analysis, we chose an object that produces periodic rotational motion to effectively observe the sensor's behavior during its subexposures. Our objective is to validate the results obtained from full-frame transitions and analyze the implications of the remaining deadtime in the imaging process.

Figure 7 illustrates the imaging outcomes for a single full frame. In Figure 7(a), the raw output, captured at 30 fps, exhibits distiguished artifacts in areas of motion due to the encoding with 16 subframes. Figure 7(b) presents the image after the application of the demosaicing algorithm, which integrates the 16 sequential images into a cohesive high-speed sequence at 480 fps, effectively capturing the motion details.



(a) (b)

Fig. 7. Imaging outcomes for a single full frame. (a) Raw output at 30 fps showing artifacts in motion areas due to 16 subframe encoding. (b) Demosaiced image showcasing a cohesive high-speed sequence at 480 fps, effectively capturing detailed motion.

To assess the impact of reducing deadtime during continuous video capture, we can compare consecutive full frames by analyzing the last subframe of one frame and the first subframe of the subsequent frame. Ideally, we aim for the motion displacement in active regions to correspond with the displacement observed in intra-frame transitions.

Figure 8 presents the rearranged outputs from three sequential full frames, with a focus on the transition between the last subframe of one frame and the first subframe of the next. This analysis demonstrates that the motion displacement during these crucial moments mirrors that of intra-frame transitions, underscoring the effectiveness of reducing deadtime to maintain continuity in motion representation.

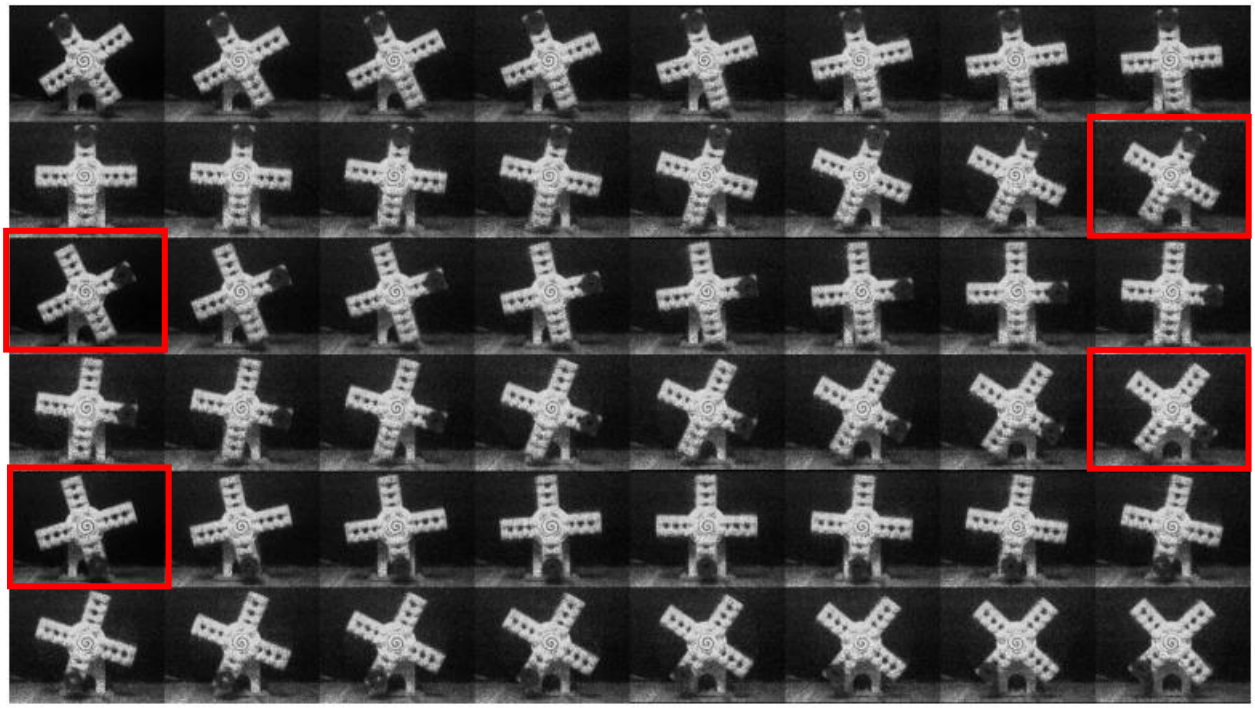


Fig. 8. Analysis of transition between the last subframe of one frame and the first subframe of the next for 3 consecutive frames, illustrating how reducing deadtime preserves motion continuity.

Figure 9 illustrates the outcomes of the same experiment conducted using the old firmware, revealing a marked deadtime effect that significantly impacts performance. The moments of transition from one complete frame to the next are highlighted similarly to those in Figure 8, drawing attention to these critical intervals. In contrast to Figure 8, the old firmware exhibits a pronounced displacement following each frame reading, which hampers its effectiveness. While using this firmware produces adequate results for high-speed burst imaging, it ultimately limits the sensor's application for continuous video capture. This is mainly due to the discontinuous nature of the video output, which risks losing visual information during the gaps between full frames.

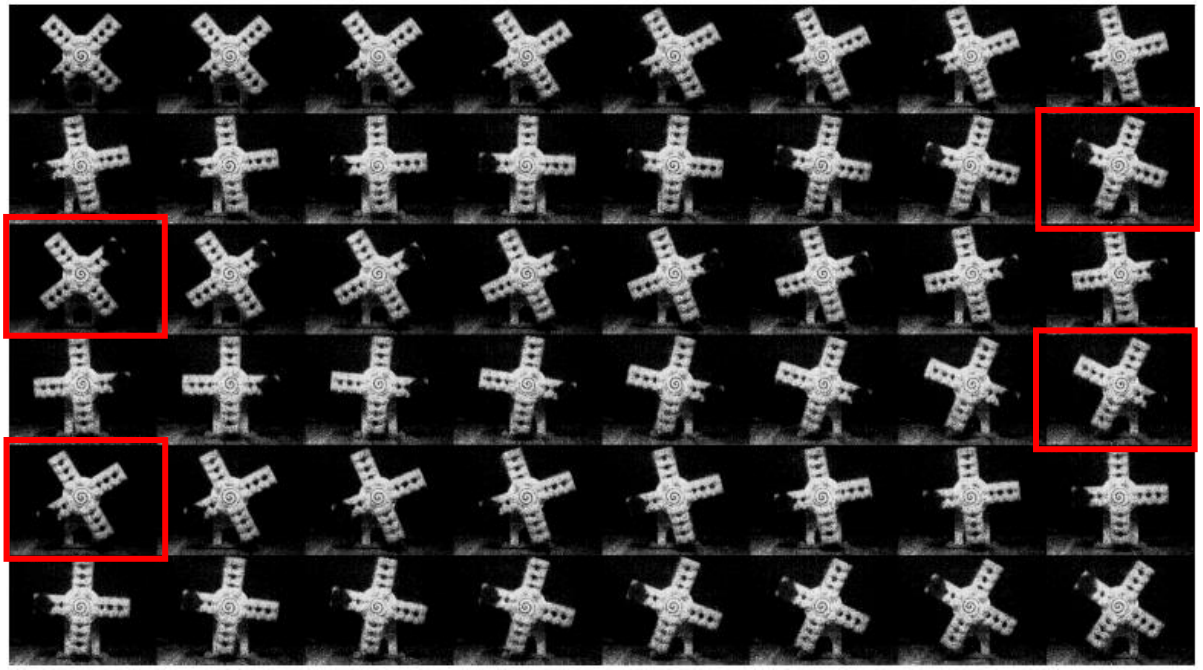
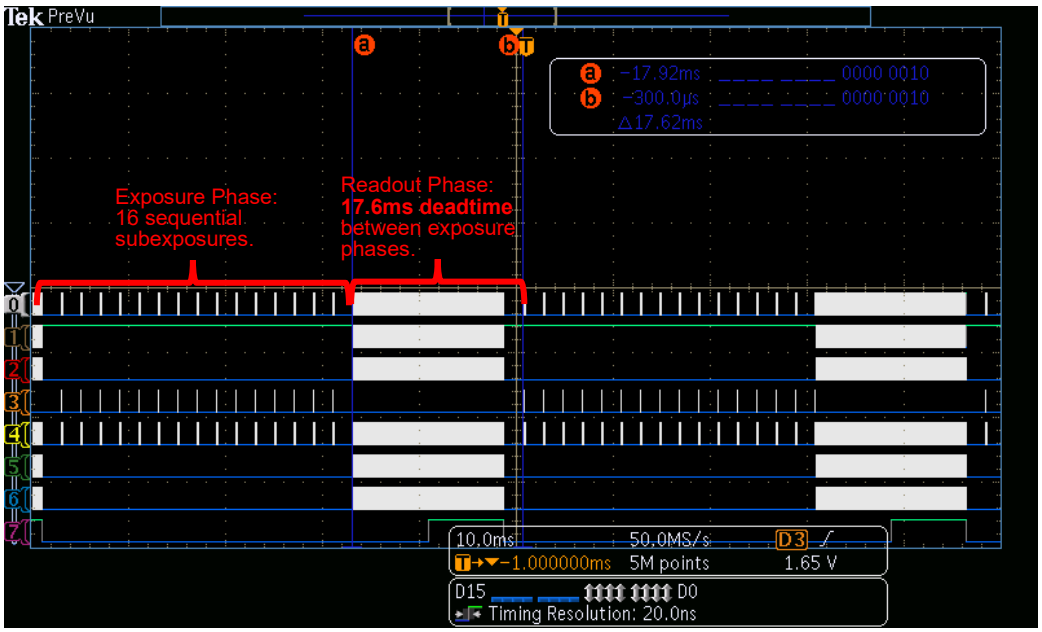


Fig. 9. Subframes captured for 3 consecutive frames using the old firmware, highlighting the significant deadtime effect and pronounced frame displacement that affects continuous video capture capability.

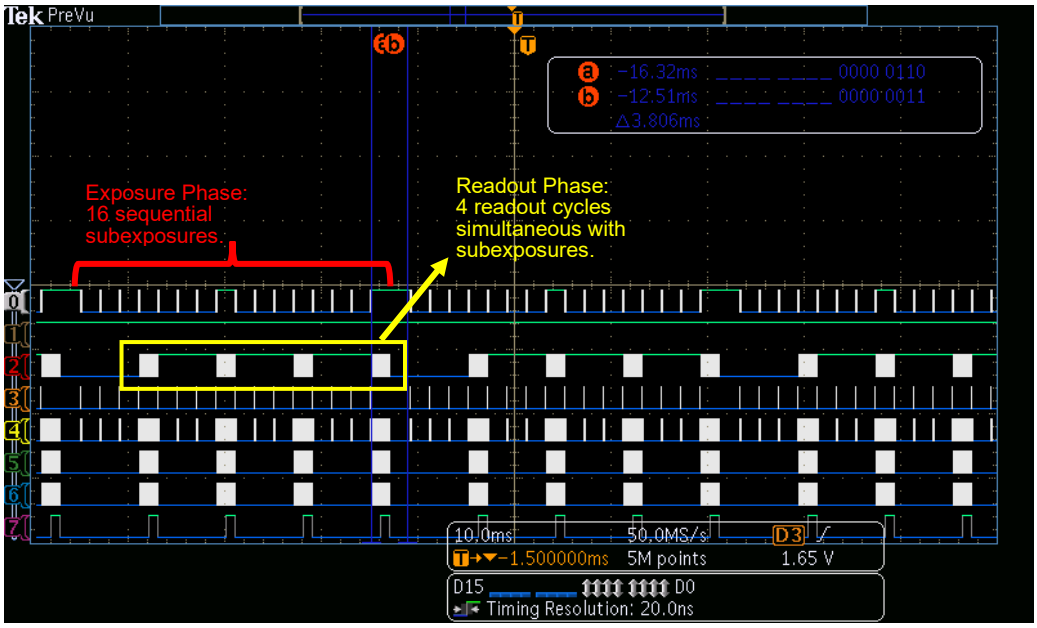
Figures 8 and 9 offer valuable qualitative insights into the effects of reducing deadtime, but establishing a quantitative comparison is crucial for further analysis. We employed an oscilloscope to monitor control and output signals from the sensor, enabling us to accurately assess deadtime in both the current and previous firmware versions.

Figure 10 demonstrates oscilloscope captures for the old firmware (a) and the new firmware (b). The new firmware clearly showcases the simultaneous operation of the readout and exposure phases, while the old firmware highlights that the duration of readout following exposure significantly contributes to deadtime, resulting in 17 milliseconds between frames instead of the expected 2 milliseconds.

In contrast, the new firmware reduces the additional wait time to just 1.8 milliseconds beyond the expected 2 milliseconds.



(a)



(b)

Fig. 10. Comparison of oscilloscope captures for the old firmware (a) and new firmware (b). The new firmware demonstrates significantly reduced deadtime during the readout and exposure phases, achieving a 1.8 ms delay compared to the old firmware's 17 ms.

Table 1 compares the delays of the previous and current systems, highlighting deadtime during sensor operation and software delays for raw image acquisition and post-processing. Since these two software tasks run concurrently on the new implementation, the longer delay (post-processing) determines the total frame period. Improvements in minimizing delays have reduced the total frame period to under 33 ms, allowing the system to achieve 30 frames per second.

Table 1. Comparative delay analysis. Unit is milliseconds.

Item	Previous	Current	Ratio
Sensor deadtime	17.6	1.8	0.102
Raw image acquisition	34.8	17.2	0.494
Post-processing	57.5	30.5	0.530
Total	**109.9**	**32.3**	**0.294**

Conclusions

The proposed system successfully addressed the challenges of simultaneous exposure and readout in coded-exposure imaging, showcasing significant advancements in reducing deadtime and improving continuous video capture capabilities. By leveraging innovative firmware modifications, optimized APIs, and multithreading, the system demonstrated substantial improvements in frame rates, spatial-temporal resolution, and motion representation. Experimental results highlighted a reduction in deadtime to nearly 1.8 milliseconds, achieving seamless video continuity, as validated through oscilloscope measurements and quantitative comparisons with previous firmware.

Future work could explore further optimizations, such as enhancing the scalability of the coded-exposure algorithm for more complex motion scenarios and implementing real-time adaptive coding strategies. Additionally, integrating the system with machine learning models could improve its application in dynamic environments, such as real-time video resolution scaling using generative AI.

Bibliography

[1] Nayar, Shree K., Vlad Branzoi, and Terry E. Boult. “Programmable Imaging: Towards a Flexible Camera.” *International Journal of Computer Vision* 70, 2006.

[2] Raskar, Ramesh, Amit Agrawal, and Jack Tumblin. “Coded Exposure Photography: Motion Deblurring Using Fluttered Shutter,” 2006.

[3] N. Sarhangnejad *et al.*, “Dual-Tap Computational Photography Image Sensor With Per-Pixel Pipelined Digital Memory for Intra-Frame Coded Multi-Exposure,” *IEEE J. Solid-State Circuits*, vol. 54, no. 11, 2019.

[4] R. Rangel *et al.*, “23,000-Exposures/s 360fps-Readout Software-Defined Image Sensor with Motion-Adaptive Spatially Varying Imaging Speed,” in *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, 2024.

[5] R. Gulve *et al.*, “Dual-Port CMOS Image Sensor with Regression-Based HDR Flux-to-Digital Conversion and 80ns Rapid-Update Pixel-Wise Exposure Coding,” in *2023 IEEE International Solid- State Circuits Conference (ISSCC)*, 2023.