EEE 591
Real-Time DSP Systems

Project 1
Real Time Image Processing

Due Date: March 19th, 2024

Jayaram Atluri
ASU ID: 1228699721

Professor: Dr. Chao Wang

Introduction:

This project aims to apply real-time image enhancement techniques to live video streams, providing hands-on experience with DSP applications. Through tasks like Thresholding, Gray Level Quantization, Gray Level Transformations, and Histogram Equalization, we gained practical skills in MATLAB and hardware interaction. This project bridges theoretical knowledge with real-world applications, enhancing our' understanding of image processing
Below are the contents of deliverables in the order as follows:

1. Tables

- **Thresholding**
  - Global thresholding
  - Band thresholding.
  - Semi thresholding
- **Gray Level Quantization**
  - Quantization Levels (128, 64, 32, 16)
- Gray Level Transformations
  - Transformation 1
  - Transformation 2
  - Transformation 3
- **Histogram Equalization**
  - Histogram Equalization Observations

2. Screenshots

  - Global thresholding
  - Band thresholding.
  - Semi thresholding
  - Gray Level Quantization
  - Gray Level Transformation

3. Codes

- **Thresholding Functions**
  - Global Thresholding C
  - Band Thresholding C
  - Band Thresholding C
  - Band Thresholding Hybrid (Grad Students Only)
  - Semi Thresholding C
  - Semi Thresholding Hybrid
- **Gray Level Quantization Functions**
  - Gray Level Quantization C
  - Gray Level Quantization Hybrid
  - Modified MATLAB script to reconstruct quantized image for display.
- **Gray Level Transformation Functions**
  - Gray Level Transformation 1 C
  - Gray Level Transformation 1 Hybrid
  - Gray Level Transformation 2 C
  - Gray Level Transformation 2 Hybrid (Grad Students Only)
  - Gray Level Transformation 3 C (Extra Credit)
  - Gray Level Transformation 3 Hybrid (Extra Credit)

- Histogram Equalization Functions
  - Calculate Histogram C
  - Calculate Histogram Hybrid

- Map Levels C
- Map Levels Hybrid
- Transform Image C
- Transform Image Hybrid

1. Tables:

Thresholding:

Task	Parameters	Observation and MATLAB Plot
Global Thresholding	Threshold: 80	The output image is binary, with the background and brighter regions turning white, and the subject and shadows turning black. The output histogram shows two distinct peaks corresponding to the black and white regions in the image.
Band Thresholding	Threshold1: 80, Threshold2: 240	The band thresholding has isolated the mid-tones. The subject's face and shirt are white, indicating these pixel values fall within the specified band. The output histogram is expected to show a concentration of pixel values within the threshold band.
Semi Thresholding	Threshold: 80	Semi thresholding darkens areas below the threshold, keeping brighter areas unchanged. The subject's face is visible against a darker background. The output histogram will likely show a reduction in lower-intensity values.

Gray Level Quantization:

Quantization Levels	Degradation in Image Quality	Missing Values in Histogram
128 (shift right by 1)	Yes, slight degradation with less detail, especially in finer textures.	Yes, there are fewer histogram peaks, indicating reduced gray levels.
64 (shift right by 2)	Yes, moderate degradation with more pronounced loss of detail and contouring effects.	Yes, histogram peaks are more spaced out due to fewer gray levels.
32 (shift right by 3)	Yes, significant degradation with coarse quantization visible, resulting in a blockier appearance.	Yes, histogram shows significant gaps, indicating many gray levels have been combined.
16 (shift right by 4)	Yes, severe degradation with a very blocky and abstract appearance, losing most details.	Yes, the histogram is very sparse, with most original gray levels missing.

Gray Level Transformations:

Transformation	A	B	Observation
Transformation 1			Image inversion is observed, dark regions become light and vice versa, enhancing details in dark areas.
Transformation 2	70	100	With A<B, the dynamic range is expanded in the lower gray levels, enhancing details in darker regions.
Transformation 2	100	70	With A>B, the dynamic range is compressed, potentially improving visibility in highlights.
Transformation 2	70	70	With A=B, no transformation is applied; the image remains unchanged.
Transformation 3	70	100	With A<B, the transformation might enhance mid-tones while keeping darker and lighter regions largely unchanged.
Transformation 3	100	70	With A>B, there might be an aggressive contrast enhancement effect, particularly affecting mid-tone regions.
Transformation 3	70	70	With A=B, the transformation could create a high-contrast image where mid-tones are either significantly darkened or lightened.

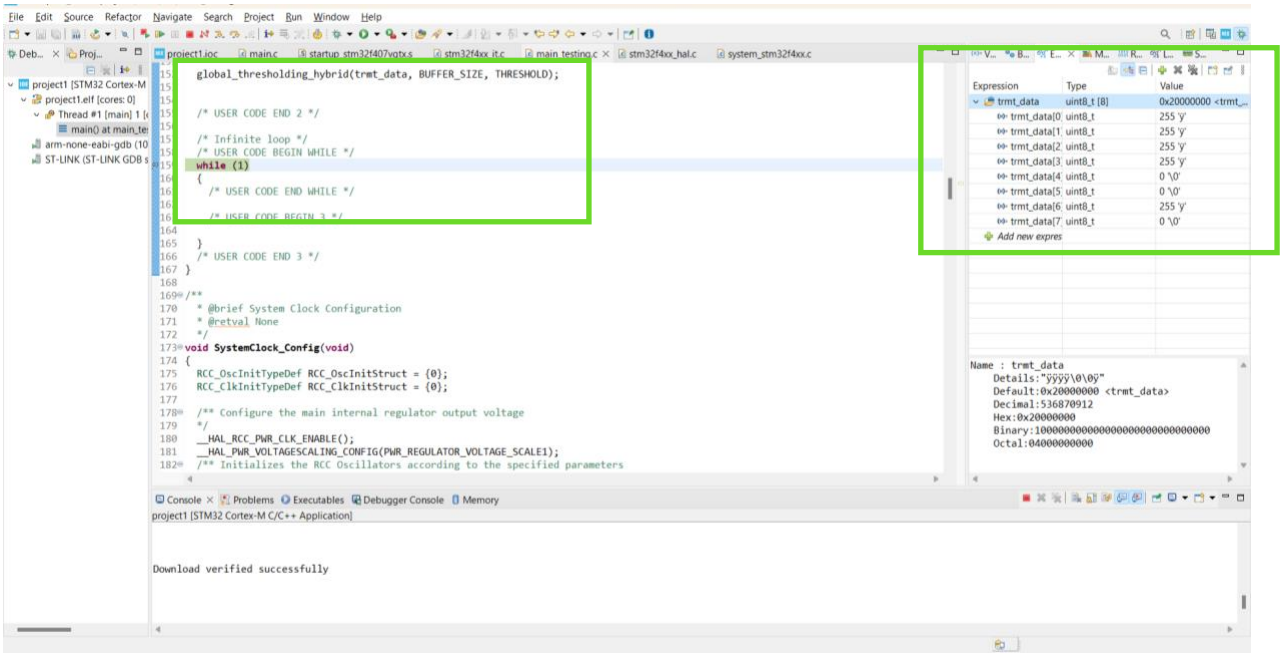
Histogram Equalization Observations:

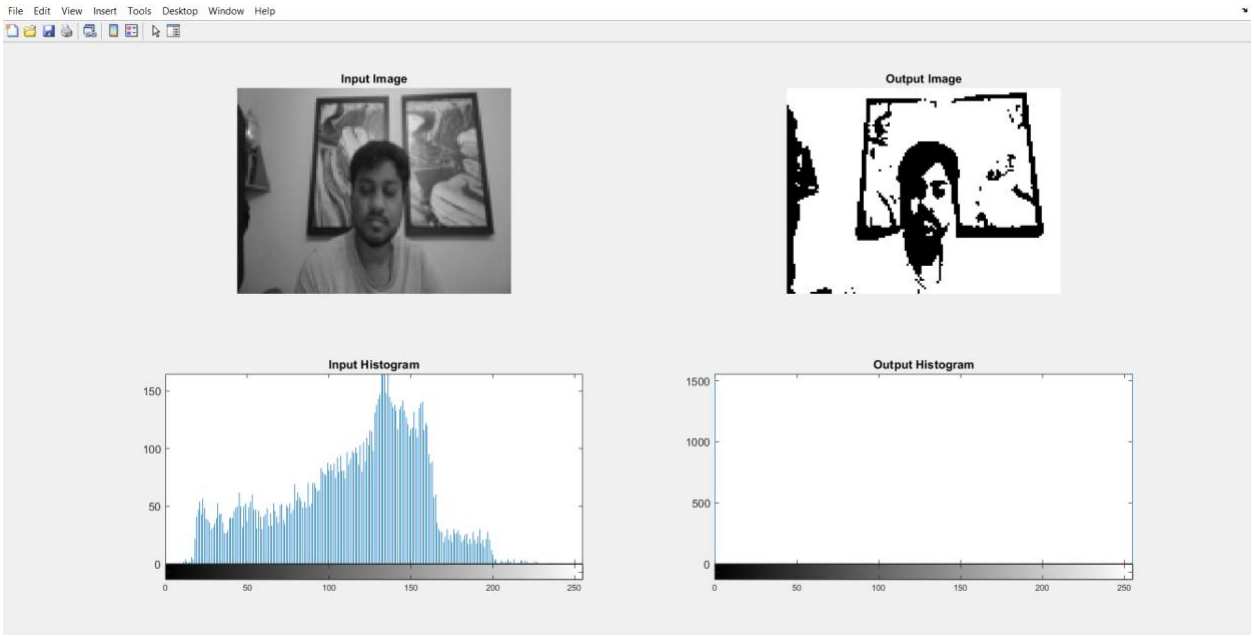
Equalization	Observation
Histogram Equalization	The image that has been processed has shown improved contrast. The features that were previously difficult to distinguish, such as edges or textures, are now more visible and clear. The distribution of pixel intensity values across the histogram range is now more uniform, which leads to an improved visual perception of the image. Darker regions may have been brightened, and brighter areas may now show more detail.

2. Screenshots:

Function Screenshots:

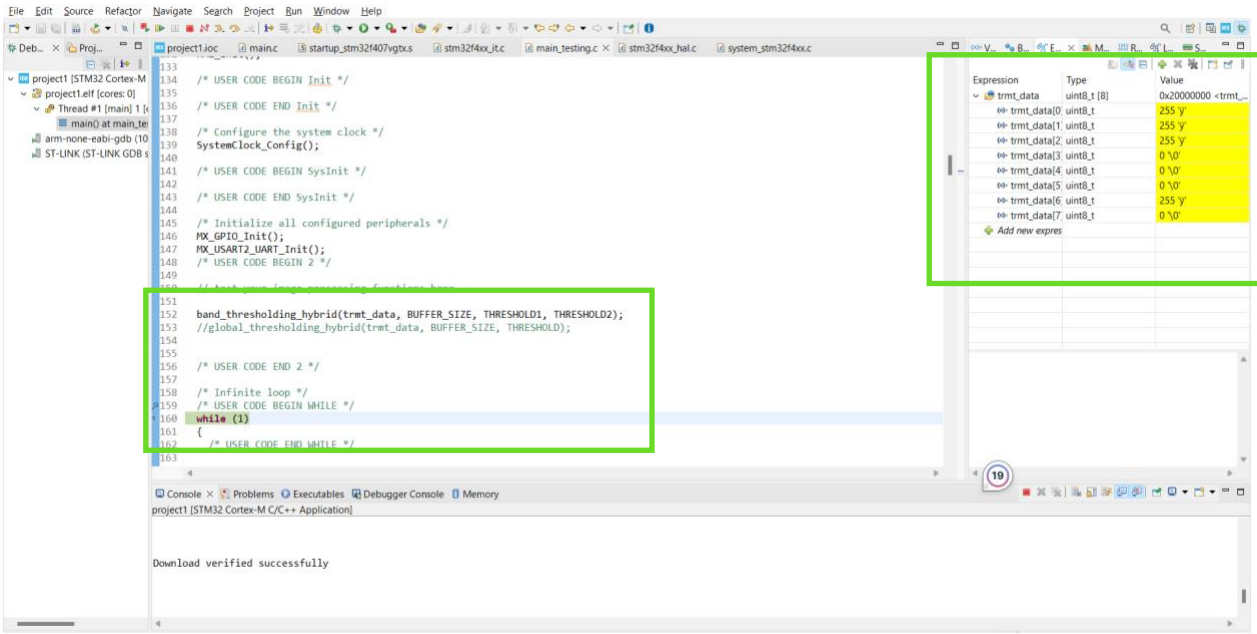
Global thresholding:

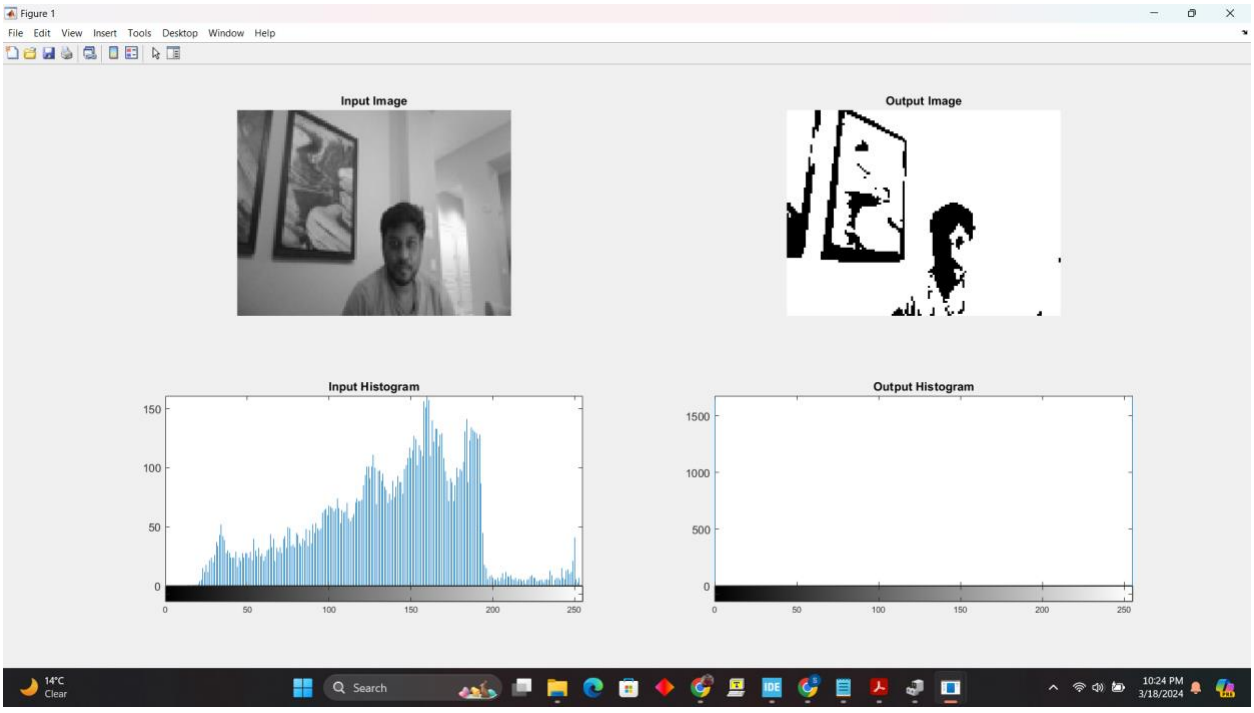




MATLAB plot

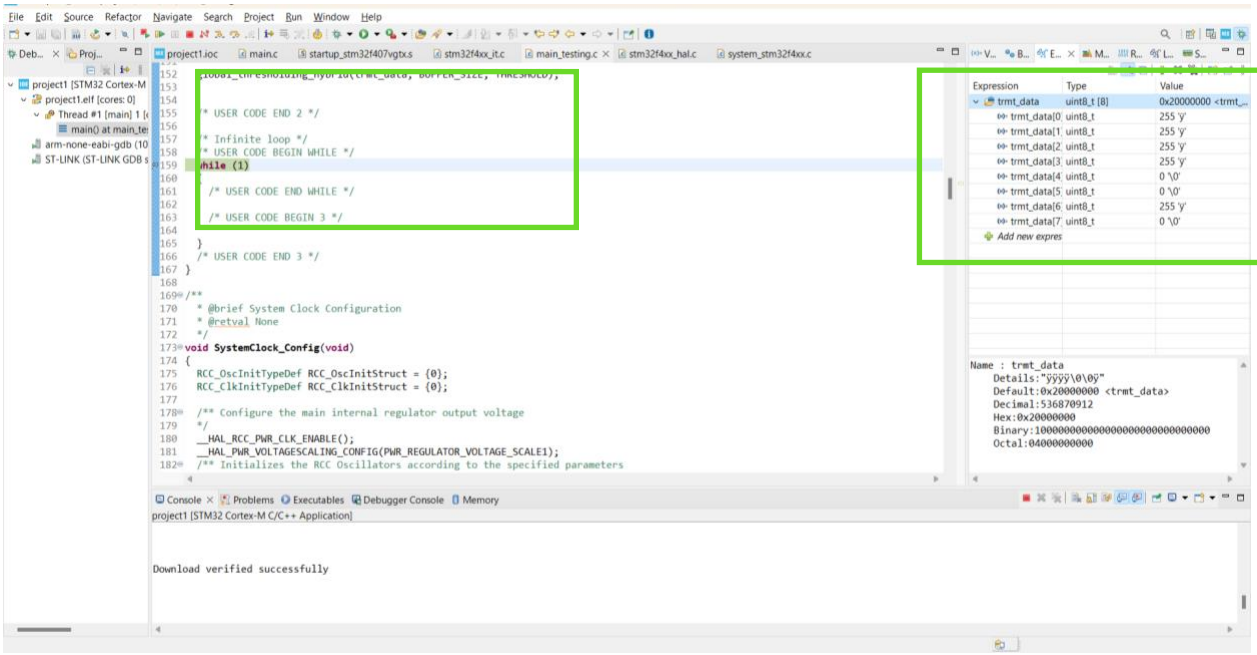
Band thresholding:

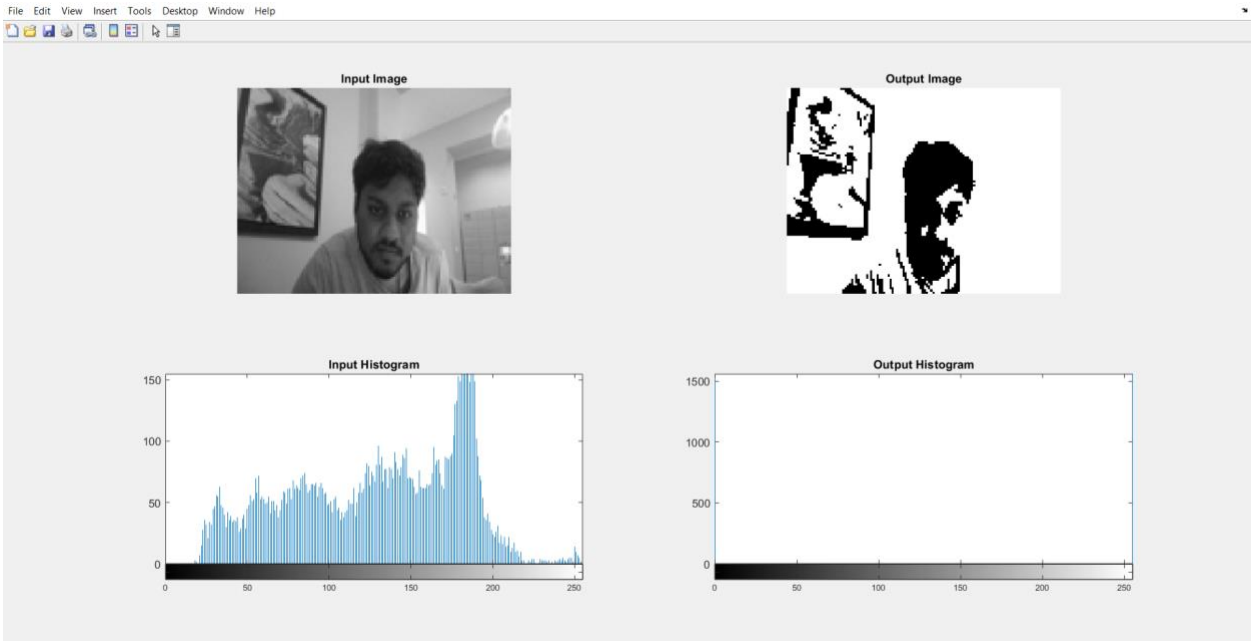




MATLAB plot

Semi thresholding:

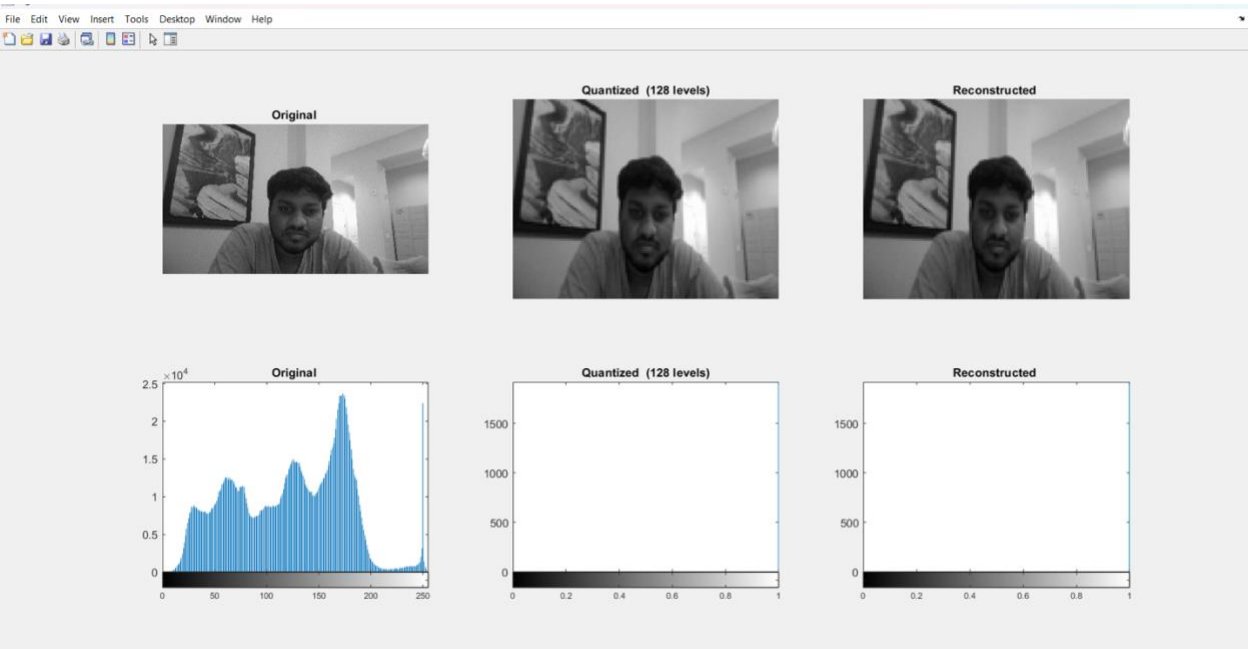


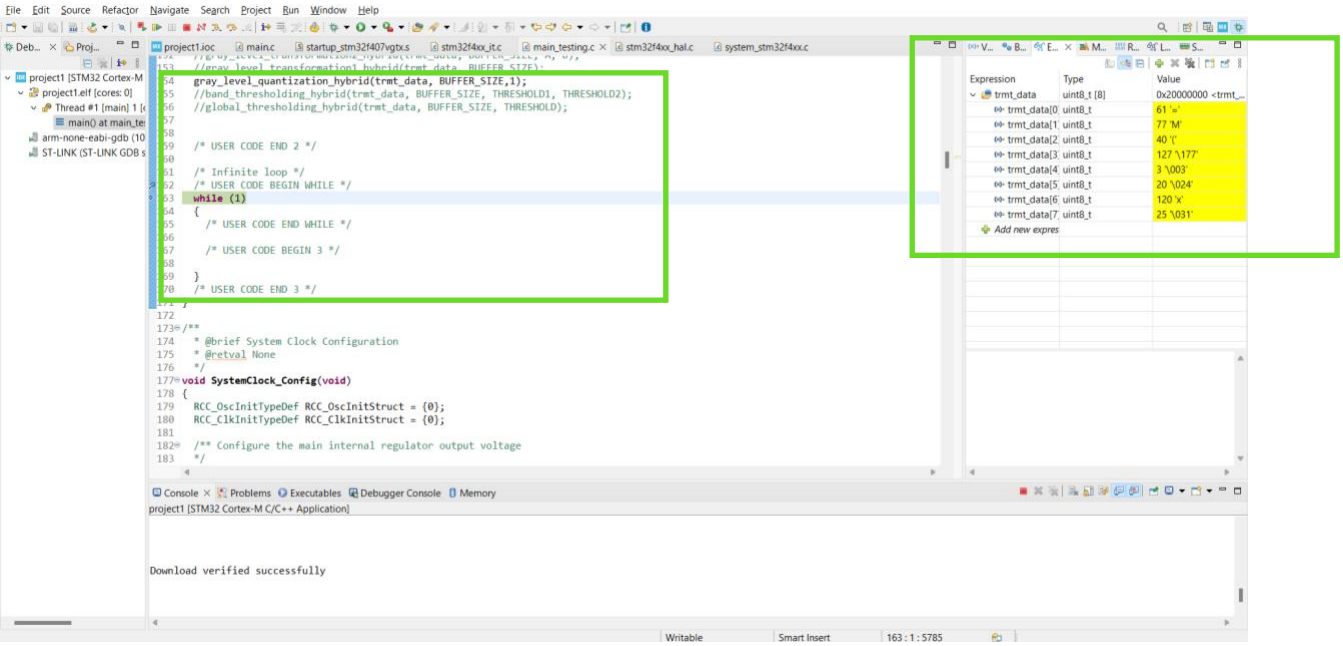


MATLAB plot

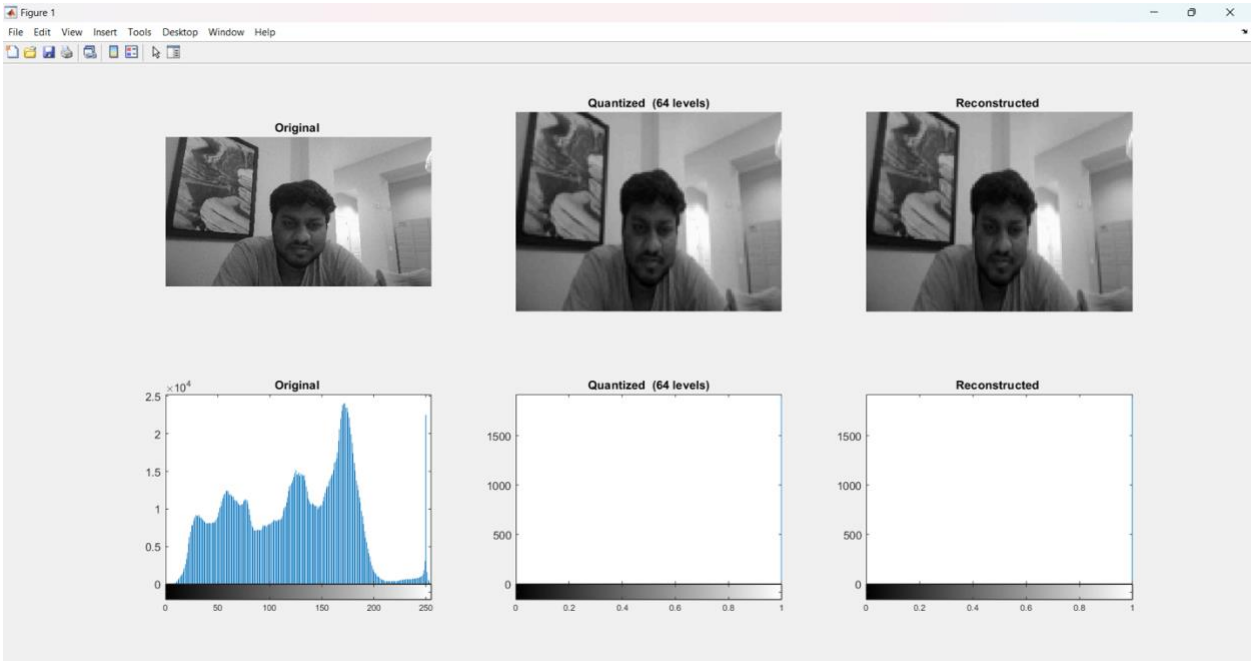
Gray Level Quantization:

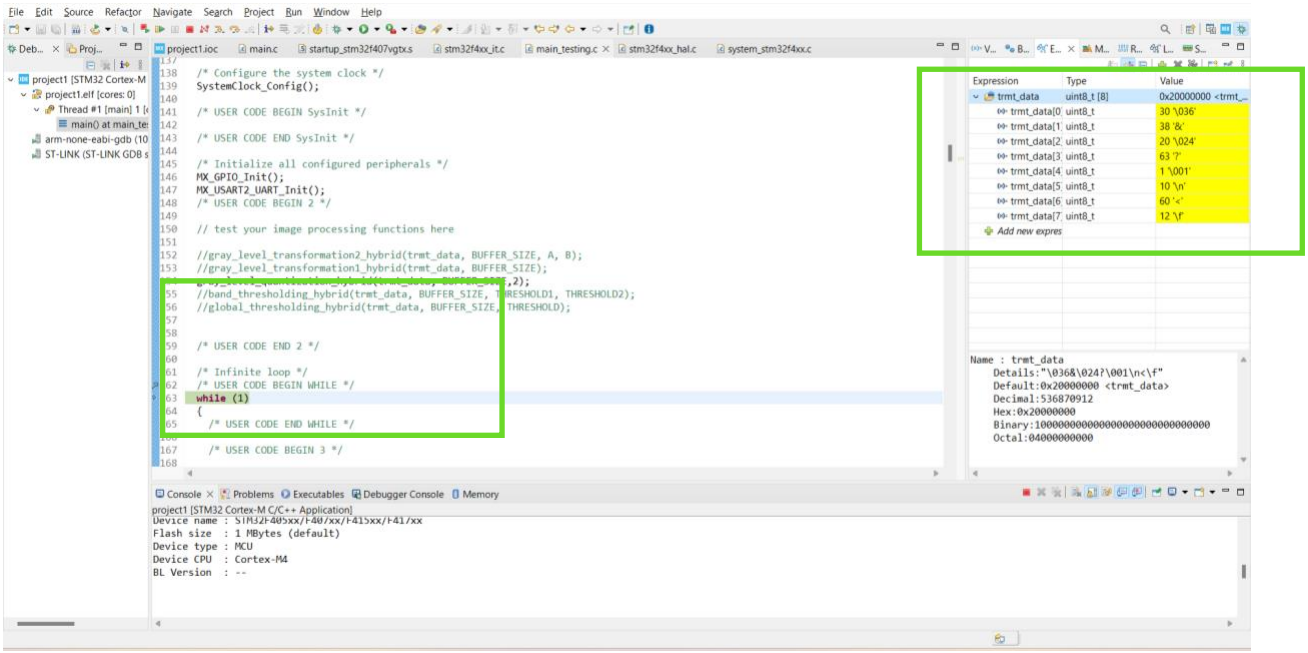
128 (shift right by 1):



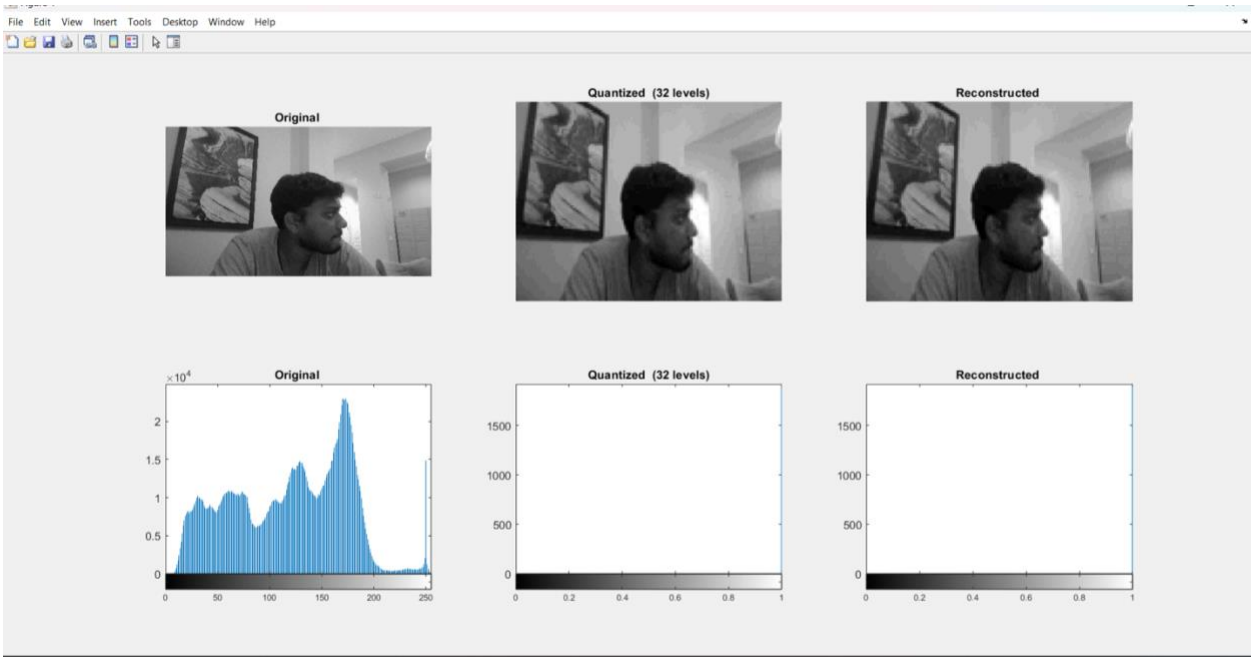


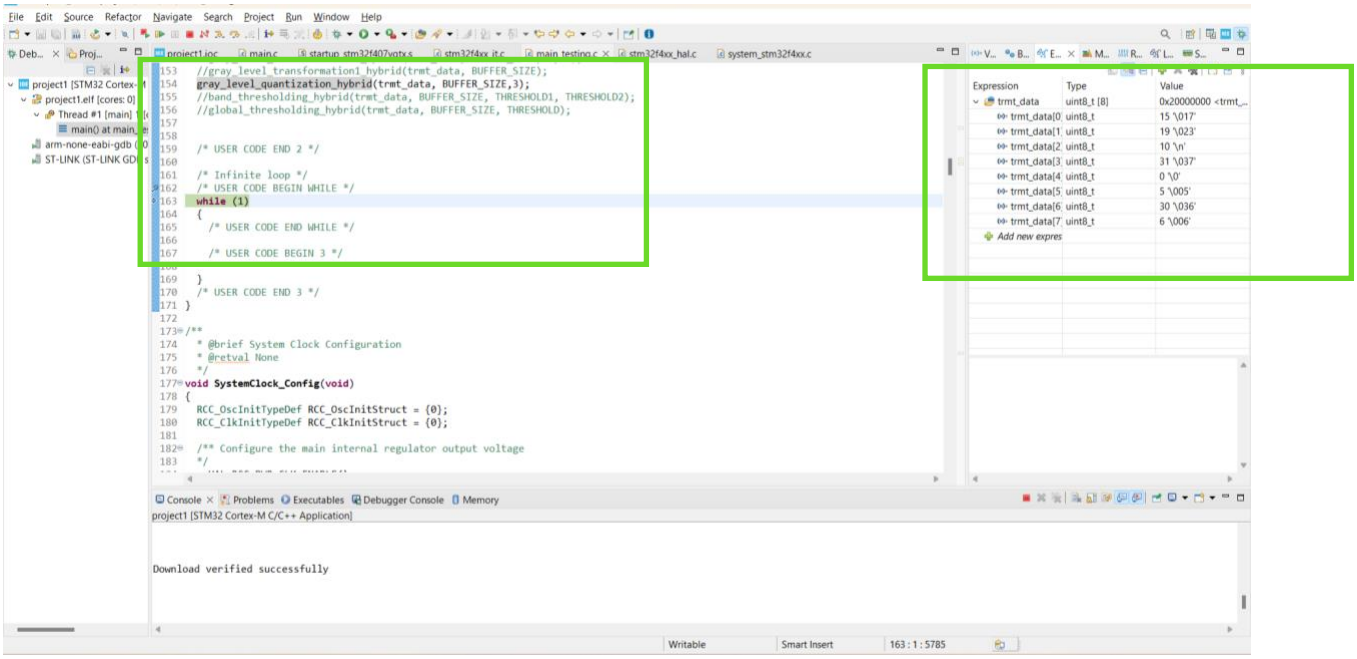
64 (shift right by 2):



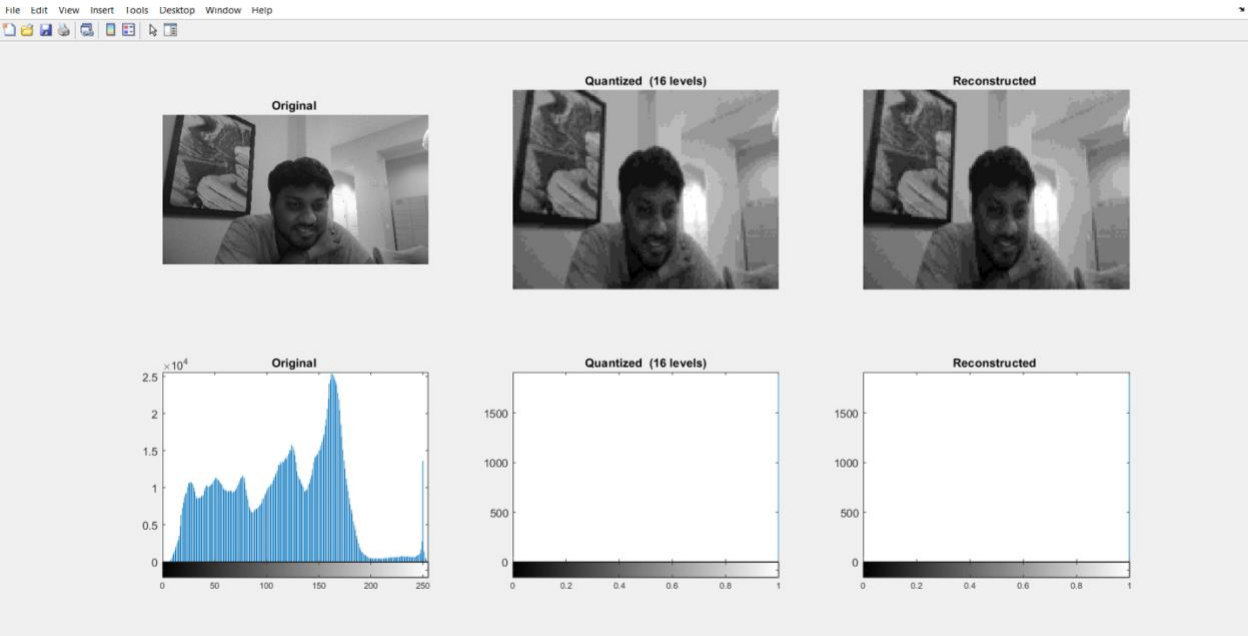


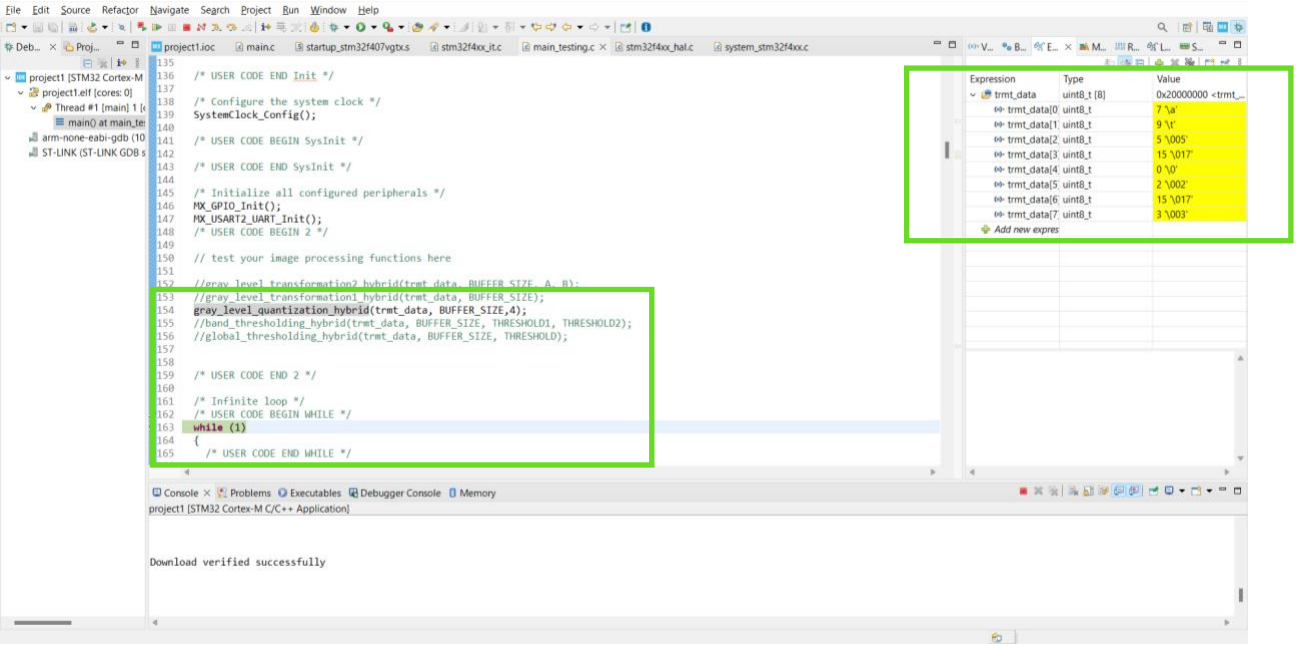
32 (shift right by 3):





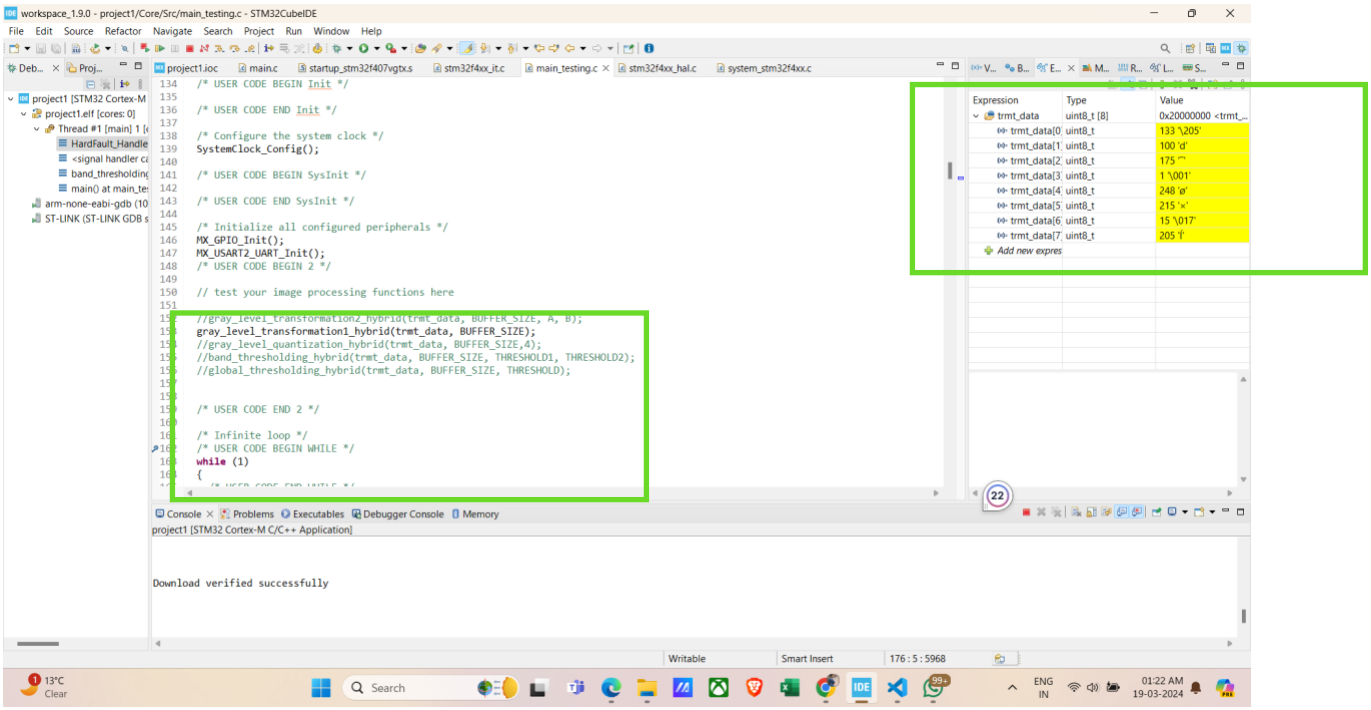
16 (shift right by 4):



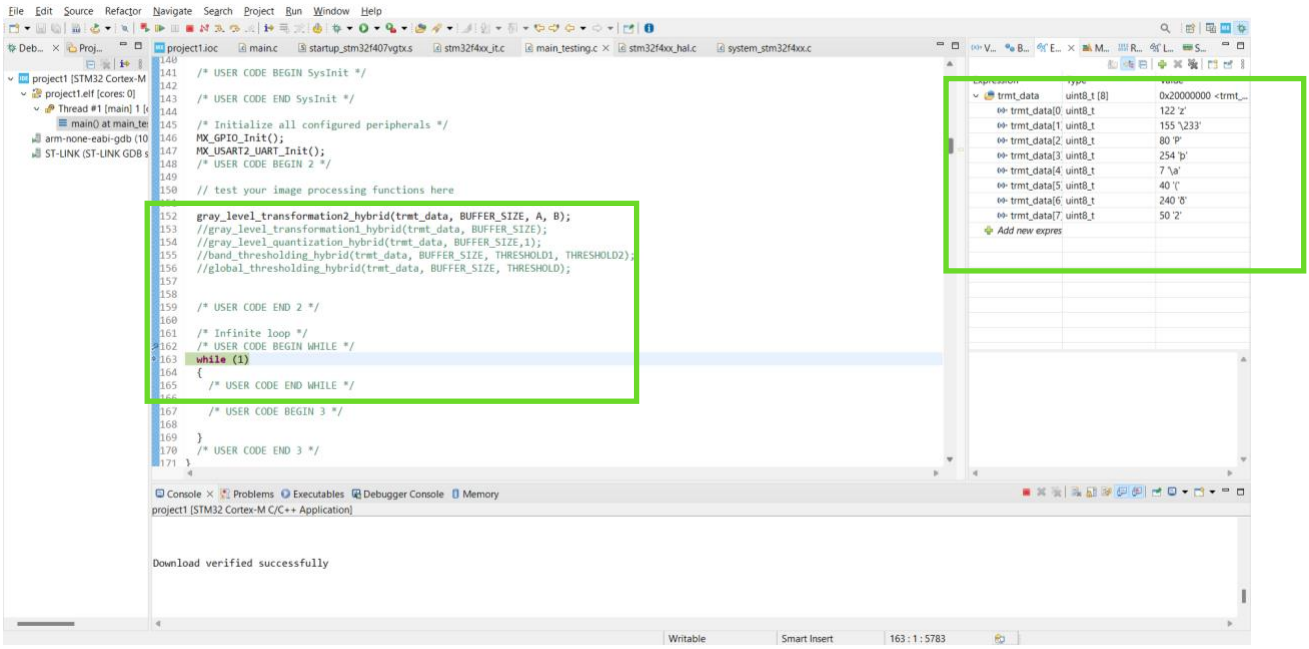


Gray Level Transformation:

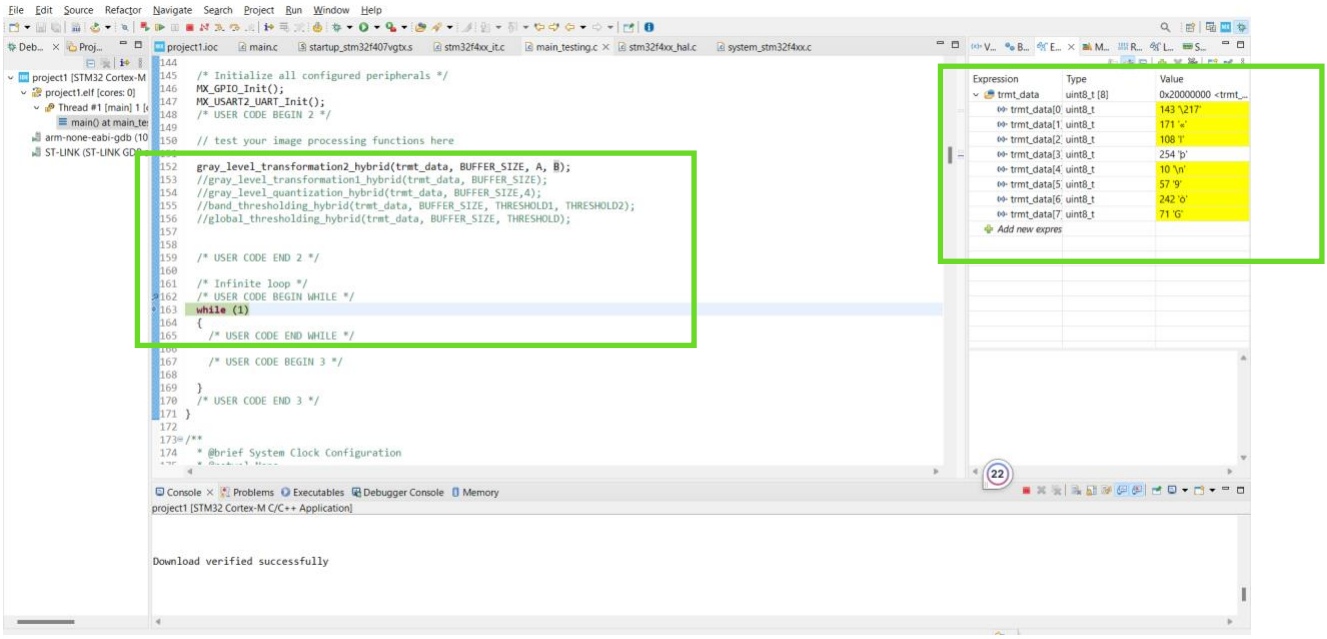
Transformation 1:



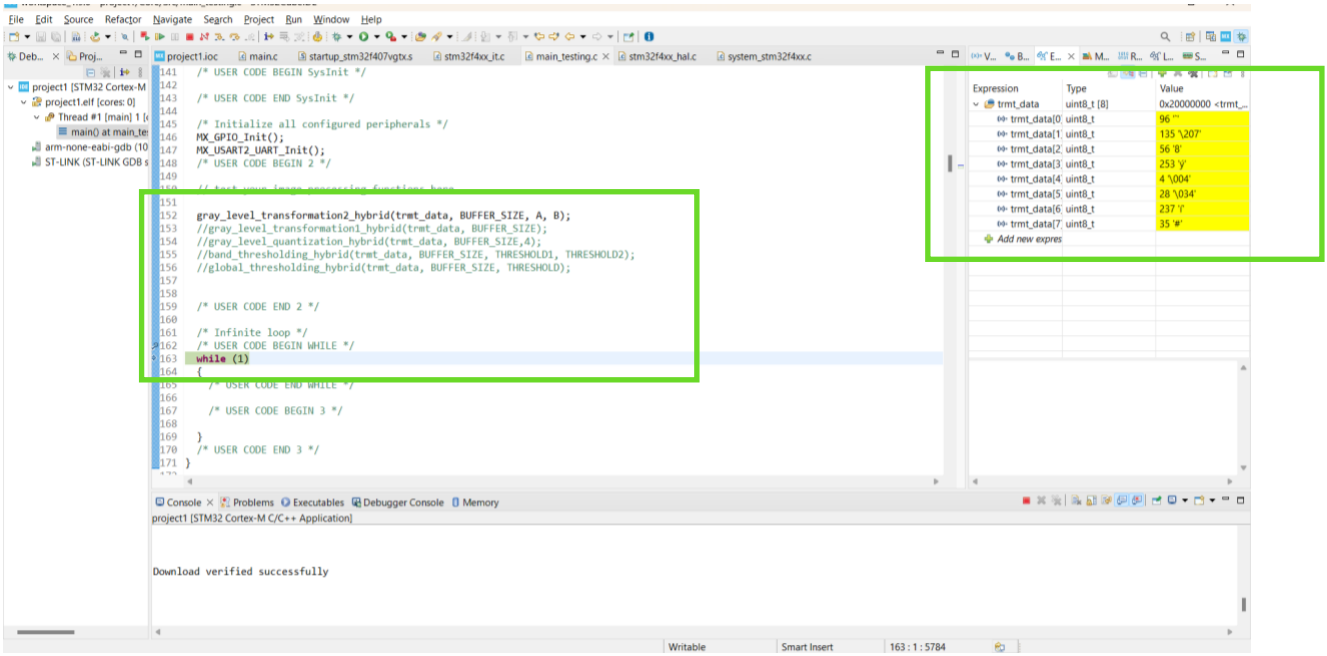
A=B=70:



A=100, B=70:



A=70, B=100:



3. Codes:

Thresholding Functions:

Global Thresholding C:

```
void global_thresholding_c(uint8_t *x, uint32_t size, uint8_t threshold) {

for(int i=0;i<size;i++) { if (x[i]>=threshold) {

x[i]=255; }

else {

x[i]=0; }

} }
```

Band Thresholding C:

```
void band_thresholding_c(uint8_t *x, uint32_t size, uint8_t threshold1, uint8_t threshold2) {
    for (int i = 0; i < size; i++) {
        // Check if the pixel value is within the threshold range
        if ((x[i] >= threshold1) && (x[i] <= threshold2)) {
            x[i] = 255; // Set the pixel to white if within the range
        } else {
            x[i] = 0;   // Set the pixel to black otherwise
        }
    }
}
```

Band Thresholding Hybrid (Grad Students Only):

```
__attribute__ ((naked)) void band_thresholding_hybrid(uint8_t *x, uint32_t size, uint8_t threshold1, uint8_t threshold2) {

    __asm volatile (

        "PUSH {r4, r5, r6, r7, lr}\n\t"      // Save registers and the link register

        "MOV r4, #0\n\t"                     // Initialize loop counter r4 to 0

        "MOV r5, #255\n\t"                   // r5 will hold the value 255

        "MOV r6, #0\n\t"                     // r6 will hold the value 0

        "band_loop: CMP r4, r1\n\t"          // Compare loop counter with size

        "BGE band_exit\n\t"                  // If counter >= size, exit loop

        "LDRB r7, [r0, r4]\n\t"              // Load the pixel value into r7

        "CMP r7, r3\n\t"                     // Compare pixel with upper threshold

        "BHI next_pixel\n\t"                 // If pixel > threshold2, go to next_pixel

        "CMP r7, r2\n\t"                     // Compare pixel with lower threshold

        "BLO next_pixel\n\t"                 // If pixel < threshold1, go to next_pixel

        "STRB r5, [r0, r4]\n\t"              // Else, pixel is within range, set to 255

        "B increment_counter\n\t"            // Skip to increment_counter

        "next_pixel: STRB r6, [r0, r4]\n\t"  // Pixel is outside range, set to 0

        "increment_counter: ADD r4, r4, #1\n\t"// Increment loop counter

        "B band_loop\n\t"                    // Continue loop

        "band_exit: POP {r4, r5, r6, r7, pc}\n\t" // Restore registers and return

    );

}
```

Semi Thresholding C:

```
void semi_thresholding_c(uint8_t *x, uint32_t size, uint8_t threshold) {
    for(int i = 0; i < size; i++) {
        if(x[i] < threshold) {
            x[i] = 0; // Set to zero if the pixel value is below the threshold
        }
    }
```

Semi Thresholding Hybrid:

```
__asm volatile (
    // x -> r0, size -> r1, threshold -> r2
    "PUSH {r4, r5, r6, lr}\n\t"       // Save registers and the return address on the stack.
    "MOV r3, #0\n\t"                  // Initialize loop counter r3 to 0.
    "MOV r5, #0\n\t"                  // r5 holds the value to set for pixels below the threshold.
    "loop_semi_thresh: CMP r3, r1\n\t" // Compare loop counter (r3) with size (r1).
    "BGE exit_semi_thresh\n\t"        // If r3 >= r1, exit loop.
    "LDRB r6, [r0, r3]\n\t"           // Load the current pixel value into r6.
    "CMP r6, r2\n\t"                  // Compare current pixel value (r6) with threshold (r2).
    "BLT thresholding\n\t"            // If current pixel is less than threshold, branch to thresholding.
    "B continue\n\t"                  // Otherwise, continue to the next pixel.
    "thresholding: STRB r5, [r0, r3]\n\t" // Store zero in pixel value if below threshold.
    "continue: ADD r3, r3, #1\n\t"    // Increment loop counter.
    "B loop_semi_thresh\n\t"          // Go back to the beginning of the loop.
    "exit_semi_thresh: POP {r4, r5, r6, pc}\n\t" // Restore registers and return.
);
```

Gray Level Quantization Functions:

Gray Level Quantization C:

```
void gray_level_quantization_c(uint8_t *x, uint32_t size, uint8_t shift_factor) {
    for(int i = 0; i < size; i++) {
        x[i] = x[i] >> shift_factor; // Quantize pixel by shifting
    }
}
```

Gray Level Quantization Hybrid:

```
__attribute__ ((naked)) void gray_level_quantization_hybrid(uint8_t *x, uint32_t size, uint8_t
shift_factor) {
    __asm volatile (
        "PUSH {r4, r5, lr}\n\t"           // Save registers
        "MOV r4, #0\n\t"                  // Loop counter
        "quantize_loop: CMP r4, r1\n\t"     // Compare counter with size
        "BGE quantize_exit\n\t"             // Exit if done
        "LDRB r5, [r0, r4]\n\t"             // Load pixel into r5
        "LSR r5, r5, r2\n\t"               // Logical Shift Right by shift_factor
        "STRB r5, [r0, r4]\n\t"             // Store back the quantized pixel
        "ADD r4, r4, #1\n\t"                // Increment counter
        "B quantize_loop\n\t"               // Loop back
        "quantize_exit: POP {r4, r5, pc}\n\t" // Restore registers and return
    );
}
```

Modified MATLAB script to reconstruct quantized image for display:

```
close all; clear all; % Close figures, clear variables
s1 = serialport('COM4', 115200); % Setup serial comms
configureTerminator(s1, "CR/LF");
flush(s1);
cam = webcam; % Initialize webcam
frame = snapshot(cam); % Capture frame
original_image = rgb2gray(frame); % Grayscale conversion
original_image_resized = imresize(original_image, [96, 128]); % Resize image
% Send image data
imageData = original_image_resized(:)';
write(s1, imageData, "uint8");
% Wait for return data
while (s1.NumBytesAvailable < numel(imageData))
    pause(0.1);
end
% Read and reshape quantized data
quantizedData = read(s1, s1.NumBytesAvailable, "uint8");
quantized_image = reshape(quantizedData, size(original_image_resized));

% Define quantization level and reconstruct image
selected_level = 16; % Example for 16 levels,change from 64,32,14,128
shift_factor = log2(256/selected_level);
reconstructed_image = bitshift(quantized_, shift_factor);
```

```
% Display images
figure;
subplot(2, 3, 1); imshow(original); title('Original ');
subplot(2, 3, 2); imshow(quantized, []); title(['Quantized (' num2str(selected_level) ' levels)']);
subplot(2, 3, 3); imshow(reconstructed, []); title('Reconstructed ');
% Display histograms
subplot(2, 3, 4); imhist(original); title(' Original Image');
subplot(2, 3, 5); imhist(quantized); title(['Quantized Image (' num2str(selected_level) ' levels)']);
subplot(2, 3, 6); imhist(reconstructed); title('Reconstructed ');
% Cleanup
clear('cam'); % Release webcam
clear s1; % Close serial port
```

Gray Level Transformation Functions:

Gray Level Transformation 1 C:

```
void gray_level_transformation1_c(uint8_t *x, uint32_t size) {
    for(uint32_t i = 0; i < size; i++) {
        x[i] = 255 - x[i]; // Invert pixel
    }
}
```

Gray Level Transformation 1 Hybrid:

```
__attribute__ ((naked)) void gray_level_transformation1_hybrid(uint8_t *x, uint32_t size) {
    __asm volatile (
        "MOV r2, #0\n\t"                // Counter
```

```
        "MOV r3, #255\n\t"              // Inversion constant
        "gl_trans1_loop: CMP r2, r1\n\t"  // Compare counter with size
        "BGE gl_trans1_exit\n\t"        // Exit loop if counter >= size
        "LDRB r4, [r0, r2]\n\t"         // Load byte
        "SUB r4, r3, r4\n\t"            // Subtract from 255
        "STRB r4, [r0, r2], #1\n\t"     // Store byte and post-increment address
        "ADD r2, r2, #1\n\t"            // Increment counter
        "B gl_trans1_loop\n\t"          // Loop back
        "gl_trans1_exit: BX LR\n\t"     // Exit
    );
}
```

Gray Level Transformation 2 C:

```
void gray_level_transformation2_c(uint8_t *x, uint32_t size, uint8_t a0, uint8_t b0) {
    uint8_t a2 = 255 - a0;
    uint8_t b2 = 255 - b0;
    for(uint32_t i = 0; i < size; i++) {
        if(x[i] <= b0) {
            x[i] = (uint8_t)(((float)a0 * x[i]) / b0); // Scale up to a0 for x <= b0
        } else {
            uint8_t p = x[i] - b0;
            x[i] = (uint8_t)(((float)p * a2) / b2) + a0; // Scale from a0 to 255 for x > b0
        }
    }
```

Gray Level Transformation 2 Hybrid (Grad Students):

```
__attribute__ ((naked)) void gray_level_transformation2_hybrid(uint8_t *x, uint32_t size, uint8_t a0, uint8_t b0) {

    __asm volatile (

        "PUSH {r4-r9, lr}"                  // Save registers

        "MOV r6, #0"                        // Loop counter

        "MOV r5, #255"                      // 255 constant for calculations

        "loopg2: CMP r6, r1"                // Compare counter with size

        "BGE exit2"                         // Exit if done

        "LDRB r4, [r0, r6]"                 // Load current pixel value

        "CMP r4, r3"                        // Compare with b0

        "BGE greater_than_b0"               // Branch if greater than b0

        // If less than or equal to b0

        "SUB r7, r5, r2"                    // r7 = 255 - a0

        "SUB r8, r5, r3"                    // r8 = 255 - b0

        "SUB r9, r4, r3"                    // r9 = current pixel value - b0

        "MUL r7, r7, r9"                    // r7 = (255 - a0) * (current pixel value - b0)

        "UDIV r7, r7, r8"                   // Divide r7 by r8

        "ADD r7, r7, r2"                    // Add a0

        "STRB r7, [r0, r6], #1"             // Store result and post-increment address

        "ADD r6, #1"                        // Increment counter

        "B loopg2"                          // Loop back

        // If greater than b0

        "greater_than_b0: "

        "MUL r4, r4, r2"                    // Multiply by a0
```

```
        "UDIV r4, r4, r3"                   // Divide by b0
        "STRB r4, [r0, r6], #1"             // Store result and post-increment address
        "ADD r6, r6, #1"                    // Increment counter
        "B loopg2"                          // Loop back
        "exit2: POP {r4-r9, pc}"            // Restore registers and return
    );
}
```

Gray Level Transformation 3 C (Extra Credit):

```
void gray_level_transformation3_c(uint8_t *x, uint32_t size, uint8_t a, uint8_t b) {
    for(uint32_t i = 0; i < size; i++) {
        // Example transformation logic:
        // If the pixel value is less than 'a', set it to 0 (black).
        // If the pixel value is greater than 'b', set it to 255 (white).
        // Otherwise, scale it linearly between 'a' and 'b'.

        if(x[i] < a) {
            x[i] = 0;
        } else if(x[i] > b) {
            x[i] = 255;
        } else {
            // Scale between 'a' and 'b'. Adjust this part to match the desired transformation.
            x[i] = (uint8_t)(((x[i] - a) / (float)(b - a)) * 255);
        }
    }
```

Gray Level Transformation 3 Hybrid (Extra Credit):

```
__attribute__ ((naked)) void gray_level_transformation3_hybrid(uint8_t *x, uint32_t size, uint8_t A, uint8_t B) {

    __asm volatile (

        "PUSH {r4-r8, lr}"                  // Save registers and the link register

        "MOV r4, #0"                        // Loop counter

        "MOV r7, #255"                      // Constant 255

        "SUB r8, r7, r2"                    // 255 - A

        "SUB r7, r7, r3"                    // 255 - B

        "SUB r7, r7, r8"                    // 255 - A - B

        "MOV r8, #2"                        // Constant 2

        "MUL r8, r8, r3"                    // 2 * B

        "SUB r7, #255, r8"                  // 255 - 2 * B

        "transform3_loop: CMP r4, r1"       // Compare counter with size

        "BGE transform3_exit"               // Exit if done

        "LDRB r5, [r0, r4]"                 // Load current pixel value into r5

        // First condition

        "CMP r5, r2"                        // Compare pixel with A

        "BLE less_than_A"                   // If pixel <= A, branch to less_than_A

        // Second condition

        "CMP r5, #255"                      // Compare pixel with 255 - B

        "BHI greater_than_255_minus_B"      // If pixel > 255 - B, branch to greater_than_255_minus_B

        // If A < pixel <= 255 - B

        "SUB r5, r5, r2"                    // pixel - A
```

```
        "MUL r5, r5, #255"                // (pixel - A) * 255
        "UDIV r5, r5, r7"                 // Divide by 255 - A - B
        "ADD r5, r5, r3"                  // Add B
        "STRB r5, [r0, r4], #1"           // Store result and post-increment address
        "B increment_counter"             // Skip to increment_counter
        // If pixel <= A
        "less_than_A: "
        "MUL r5, r5, #255"                // pixel * 255
        "UDIV r5, r5, r2"                 // Divide by A
        "STRB r5, [r0, r4], #1"           // Store result and post-increment address
        "B increment_counter"             // Skip to increment_counter
        // If pixel > 255 - B
        "greater_than_255_minus_B: "
        "SUB r5, r5, #255"                // pixel - (255 - B)
        "ADD r5, r5, r8"                  // Add (255 - 2 * B)
        "STRB r5, [r0, r4], #1"           // Store result and post-increment address
        "increment_counter: "
        "ADD r4, r4, #1"                  // Increment counter
        "B transform3_loop"               // Loop back
        "transform3_exit: POP {r4-r8, pc}"   // Restore registers and return
    );
}
```

Histogram Equalization Functions:

Calculate Histogram C:

```
void calculate_histogram_c(uint8_t *image, uint32_t *hist, uint32_t size) {
    for (uint32_t i = 0; i < 256; i++) {
        hist[i] = 0; // Initialize histogram
    }
    for (uint32_t i = 0; i < size; i++) {
        hist[image[i]]++; // Count occurrences of pixel values
    }
}
}
```

Calculate Histogram Hybrid:

```
__attribute__ ((naked)) void calculate_histogram_hybrid(uint8_t *image, uint32_t *hist, uint32_t size) {
    __asm volatile (
        "PUSH {r4-r7, lr}\n\t"              // Save used registers and link register
        // Initialize histogram to zero
        "MOV r3, #0\n\t"                    // Set histogram index to 0
        "hist_zero_loop:\n\t"
        "STR r3, [r1, r3, LSL #2]\n\t"      // Zero the histogram bin
        "ADD r3, r3, #1\n\t"                // Increment histogram index
        "CMP r3, #256\n\t"                  // Check if we've reached 256 bins
        "BLT hist_zero_loop\n\t"            // Loop if not done
        // Calculate histogram
        "MOV r4, #0\n\t"                    // Set image index to 0
        "calc_hist_loop:\n\t"
        "CMP r4, r2\n\t"                    // Compare image index to size
        "BGE hist_done\n\t"                 // If done, branch to end
        "LDRB r5, [r0, r4]\n\t"             // Load pixel value from image
        "LDR r6, [r1, r5, LSL #2]\n\t"      // Load current bin value from histogram
        "ADD r6, r6, #1\n\t"                // Increment bin value
        "STR r6, [r1, r5, LSL #2]\n\t"      // Store updated bin value back to histogram
        "ADD r4, r4, #1\n\t"                // Increment image index
```

```
        "B calc_hist_loop\n\t"              // Loop back
        "hist_done:\n\t"
        "POP {r4-r7, pc}\n\t"               // Restore registers and return
        :
        : "r" (image), "r" (hist), "r" (size)
        : "r3", "r4", "r5", "r6", "memory"
    );
}
```

Map Levels C:

```
void map_levels_c(uint32_t *hist, uint8_t *mapping_table, uint32_t size) {
    uint32_t sum = 0;
    for (uint32_t i = 0; i < 256; i++) {
        sum += hist[i];
        mapping_table[i] = (uint8_t)((sum * 255) / size); // Cast the result to uint8_t
    }
```

Map Levels Hybrid:

```
__attribute__ ((naked)) void map_levels_hybrid(uint32_t *hist, uint8_t *mapped_levels, uint32_t size, uint16_t levels) {
    __asm volatile (
        "PUSH {r4-r7, lr}\n\t"              // Save used registers and link register
        "MOV r4, #0\n\t"                    // Initialize sum to 0
        "MOV r5, #0\n\t"                    // Loop counter for levels
        "map_loop:\n\t"
        "CMP r5, r3\n\t"                    // Compare counter with levels
        "BGE map_done\n\t"                  // If done, branch to end
        "LDR r6, [r0, r5, LSL #2]\n\t"      // Load histogram bin value
        "ADD r4, r4, r6\n\t"                // Add to sum
        "MOV r6, r4\n\t"                    // Copy sum to r6
        "MUL r6, r6, #255\n\t"              // Multiply by 255 (L-1)
        "UDIV r6, r6, r2\n\t"               // Divide by total number of pixels
        "STRB r6, [r1, r5]\n\t"             // Store mapped level
        "ADD r5, r5, #1\n\t"                // Increment loop counter
        "B map_loop\n\t"                    // Loop back
```

```
        "map_done:\n\t"
        "POP {r4-r7, pc}\n\t"                // Restore registers and return
        :
        : "r" (hist), "r" (mapped_levels), "r" (size), "r" (levels)
        : "r4", "r5", "r6", "memory"
    );
}
```

Transform Image C:

```
void transform_image_c(uint8_t *image, uint8_t *mapping_table, uint32_t size) {
    for (uint32_t i = 0; i < size; i++) {
        image[i] = mapping_table[image[i]]; // Apply mapping to each pixel
    }
}
```

Transform Image Hybrid:

```
__attribute__ ((naked)) void transform_image_hybrid(uint8_t *image, uint8_t *mapping_table, uint32_t size) {
    __asm volatile (
        "PUSH {r4-r6, lr}\n\t"               // Save registers
        // Loop over the image to apply the mapping table
        "MOV r4, #0\n\t"                     // Counter for loop
        "transform_loop:\n\t"
        "CMP r4, r2\n\t"                     // Compare counter with size
        "BGE transform_end\n\t"              // If end of image, exit loop
        "LDRB r5, [r0, r4]\n\t"              // Load the current pixel value
        "LDRB r5, [r1, r5]\n\t"              // Get the mapped value from the mapping table
        "STRB r5, [r0, r4]\n\t"              // Store the transformed value back to the image
        "ADD r4, r4, #1\n\t"                 // Increment the loop counter
        "B transform_loop\n\t"               // Loop back
        "transform_end:\n\t"
        "POP {r4-r6, pc}\n\t"                // Restore registers and return
    );
}
```