Project #3: Accelerating Computer Vision Techniques

Objective:

This exercise introduces you to accelerating computer vision techniques using the Intel FPGA SDK for OpenCL.

The Canny Edge-detection Technique:

In this project, we will implement a variation of the Canny edge detector, which is a widely-used edge-detection scheme in computer vision applications. Figures 1 and 2 show a sample image that is provided as the input to a Canny edge detector, as well as the resulting edge-detected output image.

The Canny edge-detection algorithm involves five stages which are applied to the input image in succession. The details of these stages are given below. As well, we will see how the sample input image from Figure 1 is transformed as it passes through each stage.

Figure 1: Original image.

Figure 2: Edge-detected image.

Stage 1: Grayscale Conversion

Figure 3 shows the state of our sample image at the end of the grayscale conversion stage. This stage converts the input 24-bit bitmap color image (8 bits each for red, green, and blue) into an 8-bit grayscale image. The grayscale value at each pixel is calculated as the average of the three 8-bit color values of the original image.

Figure 3: The sample input image after the grayscale conversion stage.

Stage 2: Gaussian Smoothing

Figure 4 shows the state of our sample image at the end of the gaussian smoothing stage. In this stage, a gaussian filter is used to smooth out the image, by modifying noisy pixels (pixels that are unlike their neighbouring pixels) to be more like their neighbours. Shown below is the 5 x 5 gaussian filter operation that is applied to the image. Note that the * denotes convolution, A is the original image, and B is the resulting filtered image. The effect of this operation is that each pixel gets assigned the weighted average value of the 5 x 5 grid of pixels surrounding each pixel.

$$B = 1/159 \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * A$$

Figure 4: The sample input image after the gaussian smoothing stage.

Stage 3: Sobel Operator

Figure 5 shows the state of our sample image at the end of the sobel operator stage. This stage overwrites each pixel with the overall intensity gradient at that pixel. To calculate the overall intensity gradient, the gradient is first calculated in the horizontal (Cx) and vertical (Cy) directions across the pixel, using the matrices below:

$$C_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * B \qquad C_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * B$$

The magnitudes of the two gradients are then added to calculate the overall gradient intensity value for each pixel, resulting in the image C:

$$C = 0.5|C_x| + 0.5|C_y|$$

In image C, which is the final output of this stage, the edges of the original image are highlighted as brighter pixels. Non-edges, which are areas with low intensity gradients, appear as darker pixels.

Figure 5: The sample input image after the sobel operator stage.

To illustrate the effect of the sobel operator, let us examine different image boundaries that may exist in an image as shown in Figure 6. For each 3 x 3 image shown, we can use the sobel operator to calculate the intensity gradient at the center pixel. Let us examine the vertical boundary example, where there is a boundary between darker pixels on the left side of the image, and brighter pixels on the right side. In this example, jCxj of the center pixel can be calculated as |{-1*2 + 0*98 + 1*181} + {-2*1 + 0*94 + 2*178} + {-1*6 + 0*91 + 1*184}| = 711. In the vertical direction, jCyj for this pixel is calculated to be 7. The intensity gradient for this pixel is high in the horizontal direction, and low in the vertical direction, which is to be expected along a vertical boundary. The total intensity gradient for this pixel, is 0:5 _ 711 + 0:5 _ 7 = 359 which saturates the 8-bit grayscale channel to value 255. The sobel operator would therefore detect this boundary as a strong edge, and the center pixel would become bright with a value of 255.

Vertical Boundary		
2	98	181
1	94	178
6	91	184

Horizontal Boundary		
188	185	181
86	94	91
4	3	3

Diagonal Boundary		
100	185	181
5	94	178
2	3	91

Non-Boundary		
137	145	129
133	140	138
129	130	131

Figure 6: Examples of boundaries in an image.

In the horizontal boundary case, we see that the vertical gradient is high (jCyj = 726) and the horizontal gradient is low (jCxj = 2). For the diagonal boundary case, the gradient is high in both directions with jCxj = 516 and jCyj = 552. Finally, for the non-boundary case, the gradients are low in both directions with jCxj = 4 and jCyj = 36. As the gradients are low, the center pixel would become dark with a value of 0:5 _ 4 + 0:5 _ 36 = 20.

Stage 4: Non-Maximum Suppression

Figure 7 shows the state of our sample image at the end of the non-maximum suppression stage. This stage aims to thin the thick and/or blurry edges that may have resulted from the sobel operator stage. Thick edges are problematic as many applications of edge detection benefit from the edges being as thin as possible. For example, to accurately calculate the surface area of an object, thin edges are desired as to not overlap with the surface. The non-maximum suppression stage thins the edges by removing the weaker (non-maximum) pixels of each edge, and keeping only the maxima. Figure 8 shows the effect of non-maximum suppression on a sample image containing a blurry vertical line. Notice that the vertical line, which is originally three-pixels wide, becomes one-pixel wide.

Figure 7: The sample input image after the non-maximum suppression stage.

0	41	134	45	0
0	43	135	46	0
0	35	136	41	0
0	41	132	35	0
0	44	131	41	0

Non-maximum Suppression →

0	0	134	0	0
0	0	135	0	0
0	0	136	0	0
0	0	132	0	0
0	0	131	0	0

Figure 8: The effect of nonmaximum suppression on a blurred vertical line.

Stage 5: Hysteresis

Figure 9 shows the end result of the hysteresis stage. The goal of the hysteresis stage is to remove pixels that do not belong to an edge and weak edges altogether. This stage uses two user-defined thresholds: the high threshold and the low threshold. The hysteresis algorithm examines each pixel to determine whether:

1. the pixel exceeds the high threshold, or
2. the pixel exceeds the low threshold value and there exists at least one adjacent pixel (horizontally, vertically,
or diagonally) that exceeds the high threshold.

If at least one of the two criteria are met, the pixel is preserved. Otherwise, the pixel is removed by turning it black.

Figure 9: The sample input image after the hysteresis stage.

Task 1:

Write a C++-language program that implements the five stages of the canny edge detector as described in the preceding sections. Start with the skeleton code provided in /design_files/part1/, which contains functionality for loading and storing 24-bit color BMP image files. Once a 24-bit color image is loaded into memory, your code should transform the pixels according to the five

stages, then store the resulting edge-detected image. The skeleton also contains code that measures the runtime of your program, which we will use to compare with the OpenCL version. Test your program on the sample BMP images provided in /design_files/

Task 2:

Before we proceed with implementing a Canny edge detector using OpenCL, let us devise an efficient memory architecture for storing the pixel values as they undergo transformations at each Canny stage. Recall that you have two types of memory at your disposal: global memory outside the FPGA, and local memories inside the FPGA. The host program provides input data to the accelerator by placing it in global memory. Since global memory accesses are slow, your accelerator should cache the required pixels into local memory before using them. The question then, is how to configure the FPGA's local memory resources to best cache the pixels.

To determine the best memory configuration, let us consider the pixel usage pattern for box operations. Note that the Gaussian smoothing, Sobel operator, non-max suppression, and hysteresis stages of the detector are all types of box operations, as they work on a box (or frame) of pixels to determine each output pixel. Figure 10 depicts a 3x3 box operation (such as the 3x3 sobel operation) on three adjacent pixels of a 10-pixel-wide image.
Notice that there is significant overlap (6 pixels) in the frames of successive operations.

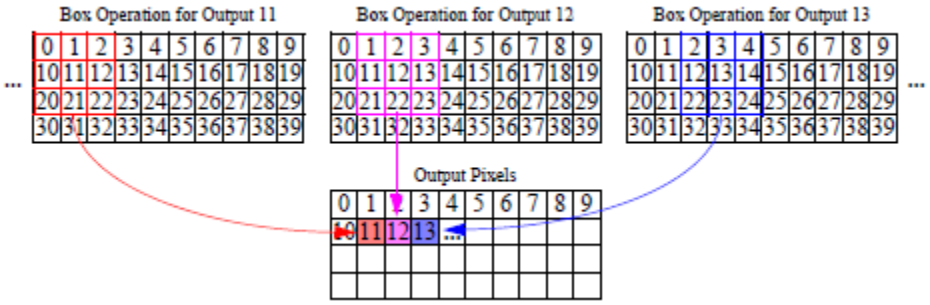

Figure 10: A 3x3 box operation on three adjacent pixels of a 10-pixel-wide image.

To take advantage of the overlap in successive frames of box operations, we will use the FPGA's local memory resources to construct the shift register design shown in Figure 11. The figure shows the shift register design for a 3x3 box operation that works on 10-pixel wide images, but the idea can be extended for arbitrary box sizes and image widths. Once a sufficient number of pixels have been loaded, the shift register provides a new 3x3 frame for the box operation at every cycle by simply shifting in a new pixel and shifting out the oldest pixel which is no longer be used.

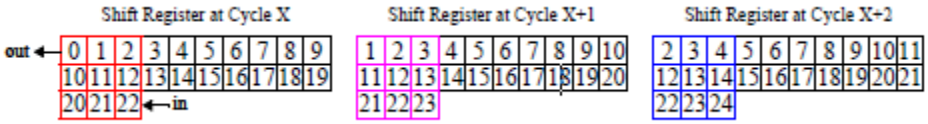

Figure 11: The shift register for 3x3 box operations on 10-pixel wide images providing 3 frames over 3 cycles.

How many shift registers will you require for your canny edge detector? How large should each of these shift registers be, given that your circuit works on images that are 720 pixels wide? How many pixels must be loaded into each shift register before the corresponding box operation can start? Using such a shift register design requires you to zero pad the input image before shifting in its pixels, to properly operate on the boundaries of the image.
Why is zero padding necessary, and what is the necessary padding size for a given box size?

Task 3:

Create an OpenCL application that implements the five stages of the canny edge detector. Use shift registers to hold the pixels as they undergo the Canny stages, and apply the box operations on the pixels while they are in flight through the shift registers. The shift registers should be the minimum length necessary to hold required pixels. When operating in its steady state, your accelerator should have a throughput of one pixel per clock cycle. The accelerator is considered to be in its steady state when its shift registers are full. For simplicity, do not zero pad the boundaries of the input image and accept some error along the boundary edge pixels. Your application must be able to operate on images that are 720 pixels wide with variable height. Start with the skeleton code provided in /design_files/part3.
The shift-register design methodology requires the use of Intel FPGA SDK for OpenCL's ability to infer shift registers from your OpenCL code. This feature is described in Section Inferring a Shift Register of the document Intel FPGA SDK for OpenCL Programming Guide.