EE5332 : Mapping Signal Processing Algorithms to DSP Architectures
Edge Detection in Images

Goutham R EE15B027 | Akshay Sethia EE15B072 | Alfred Festus Davidson EE15B073

Problem Formulation

We note that the major bottleneck in image processing is the time taken to transfer the image from memory into the processor. This becomes a major issue when image processing algorithms are implemented in a way that does not have proper data access patterns.

Further, we are constrained by the memory resources available in the hardware accelerator/FPGA and hence, storing the entire image in the accelerator and then processing it is not possible.

Therefore, we have decided to implement standard image processing functions (by implementing Canny Edge detection) such that they all stream data from one module to the next. This allows for minimum memory requirements on the accelerator and the maximum throughput is now only limited by the memory access time.

Edge Detection Objective

- Input: RGB Colour Image
- Output: Image with 1 bit pixels, which indicate whether or not an edge is present at that pixel
- Applications
  - Used as a pre-processing step for neural networks. Reduces the image size for further processing
  - Extract useful data from the image, independent of image noise / capture conditions
- Algorithm Constrains
  - Low error rate
  - Accurate estimate of edge location
  - Given edge must be marked only once

The Canny Edge Detection Algorithm

We have chosen to implement the Canny Edge Detection algorithm. It consists of the following steps.

Convert to Grayscale

- The input RGB image is first converted to grayscale. There are standard coefficients for this conversion.
- Output Grayscale Pixel Value $= 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$

Gaussian Filter

A Gaussian filter is applied on the image to smooth it and reduce the noise present in the image. This is required as this algorithm detects edges based on derivatives, and hence is sensitive to image noise.

$$\text{Gaussian Filter : } \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

- The kernel used is a standard 5×5 Gaussian Filter used in the OpenCV library.
- The filter coefficients have been scaled and rounded off to integer values for easy computation

Gradient Calculation

The gradient is calculated to find the edge intensity and direction. Sobel filters are used for this.

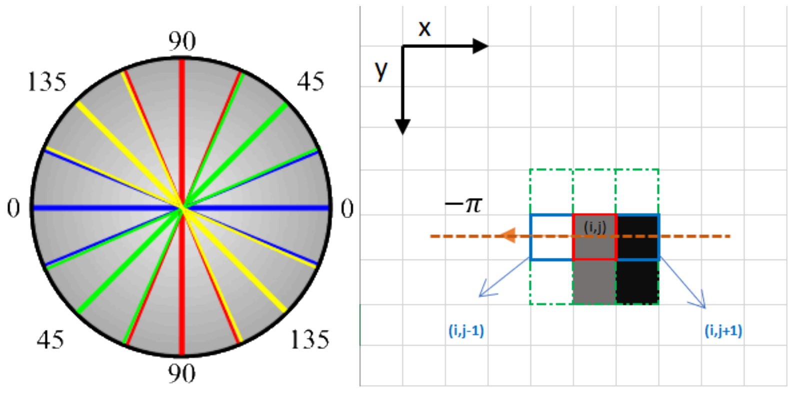
$$G_X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \; G_Y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$G = \sqrt{G_X^2 + G_Y^2} \; \theta = arctan(G_Y/G_X)$$

- Sobel Filter in x and y directions
- Filter outputs are processed to get the gradient magnitude and angle

Non-Maximum Suppression

The Gradient information will detect the same edge multiple times. Ideally we would want a single edge detected where the gradient of the edge has the maximum change. This is accomplished by using non-maximum suppression. This step removes superfluous edges and retains the gradient magnitude of the pixels where an edge is detected.



- The gradient information is used to find if the given pixel has the largest magnitude in the angle of the gradient
- If it is not the maximum, set the output pixel value to zero. Else set the output pixel value to the gradient magnitude

Double Threshold

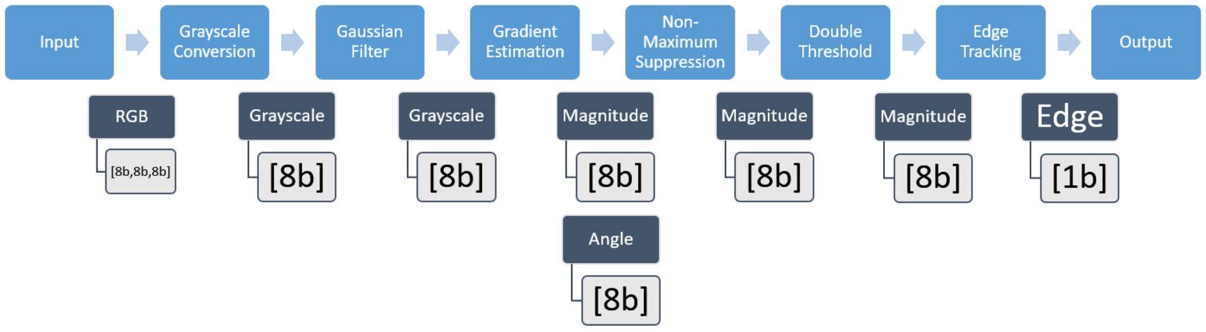
This step classifies pixels using two thresholds into strong, weak or irrelevant.

- All pixels whose gradient magnitude is greater than a set threshold are declared edges (strong)
- All pixels whose gradient magnitude is lesser than a set threshold are declared as not edges (irrelevant)
- Other pixels are not modified in this step (weak)

Edge Tracking by Hysteresis

- A pixel which was declared as a weak pixel in the previous step is declared an edge if any one of its neighboring pixels was declared as strong pixel in the previous step.
- If no neighboring pixel was declared as a strong pixel in the previous step, the pixel is declared as irrelevant.
- All strong pixels are declared as edges and irrelevant pixels are declared not edges.

Algorithm Flow



Optimization Objectives

We have chosen an image size of 640×465. For an $8b$ pixel the memory required will be $290kB$. If the algorithm is pipelined, it will require a large amount of memory. If the algorithm is not pipelined, it will have a very low throughput.

We decide to optimize for throughput, considering real time applications. Data will be streamed from one module to the next to resolve memory constrains. We will try to achieve as low latency as possible.

The main bottleneck is the kernel operations, as they need data from multiple rows. However, we note that all the major operations can be brought down to operations on a window buffer and line buffer. Hence, we need to implement the buffers optimally.

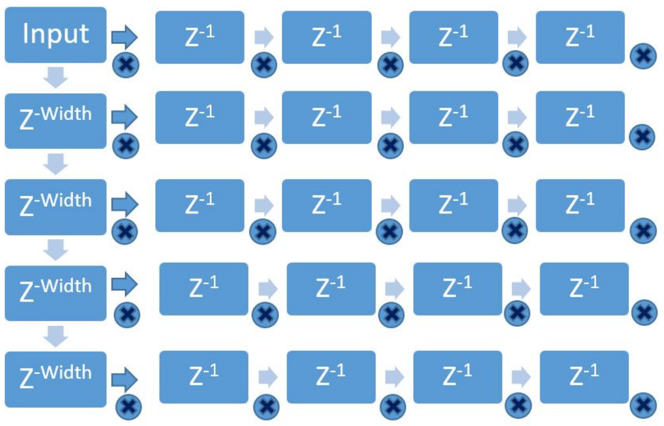
The initial idea was to do convolution block by block (to reduce latency). However, it will not have optimal memory access patterns, and hence a line buffer implementation was chosen

Optimized Algorithm

Convert to Grayscale

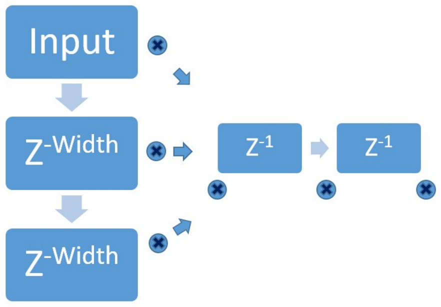
- Output Grayscale Pixel Value $= 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$
- Approximated Output Grayscale Pixel Value $\approx 0.25 \times R + 0.5 \times G + 0.125 \times B$
- Input is $32b$ data, having RGB data for $24b$ and $8b$ zero padding
- Implemented using shift operations
- Sufficient bits were given for addition and overflow
- Input/Output is 1 pixel every cycle

Gaussian Filter



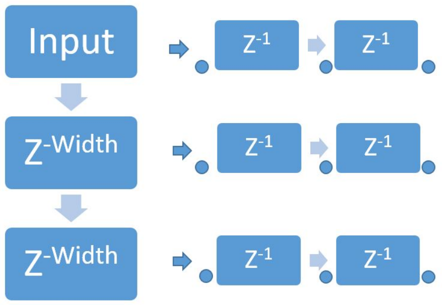
- OpenCV documentation had a non-separable filter
- Implemented using window buffers and line buffers
- Since we have a 5×5 kernel, we used 4 line buffers

Gradient Calculation



- This is a separable filter. If this is exploited in the implementation, it will result in 33% less multiplications.
- The gradient is implemented as $G = |G_X| + |G_Y|$ instead of $G = \sqrt{G_X^2 + G_Y^2}$
- We only need to know the angle of the gradient to a resolution of 8 directions
- Further, a 180 degree shift in the angle calculation does not change the non-maximum suppression output.
- Hence, we need only a $3b$ output for the gradient angle
- Instead of implementing $G_Y/G_X < Tan(\theta)$ and then comparing to get the angle, we implement $G_Y < Tan(\theta) \times G_X$
- Requires 2 line buffers

Non-Maximum Suppression

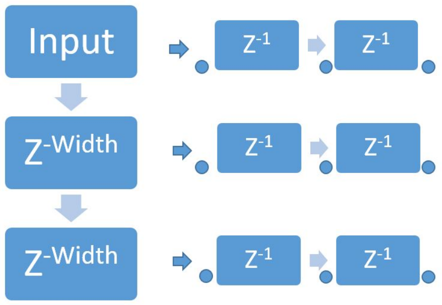


- Implemented as a operation using 3x3 window buffers
- Requires 2 line buffers for gradient magnitude and 1 line buffer for gradient angle.

Double Threshold

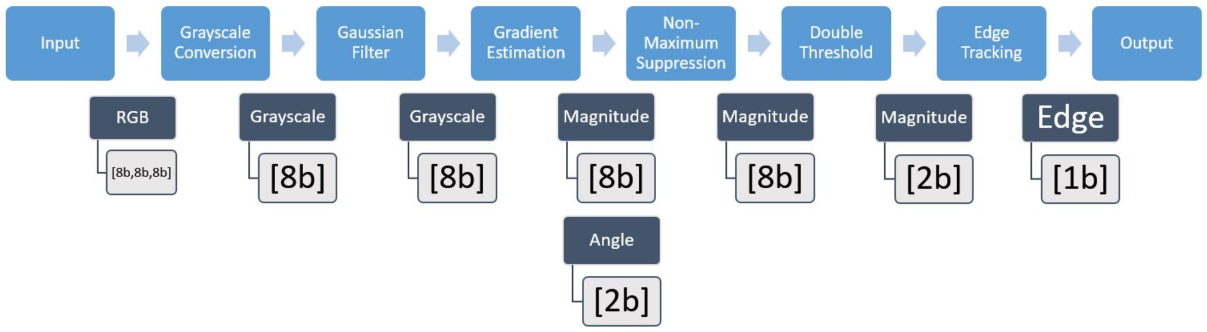
- Pixel by pixel operation
- Data is streamed from the input, processed and immediately streamed to the output

Edge Tracking by Hysteresis

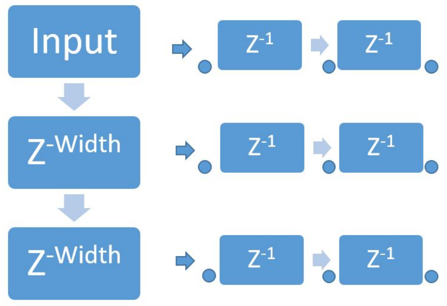


- Implemented as a operation using 3x3 window buffers
- Requires 2 line buffers

Optimized Algorithm Flow



Hardware Implementation | Line and Window Buffer



We have identified that all operations can be mapped into this computational unit consisting of line buffers and window buffers

- Input: Streaming Data
- Output: 3×3 matrix

This has been implemented in Block RAM

Initial design had an II issue. Reading the input, and then reading & writing on the same element in the line buffer could not be done in a single cycle. This was temporarily resolved by using 2 sets of line buffers which were used alternatively in each row.

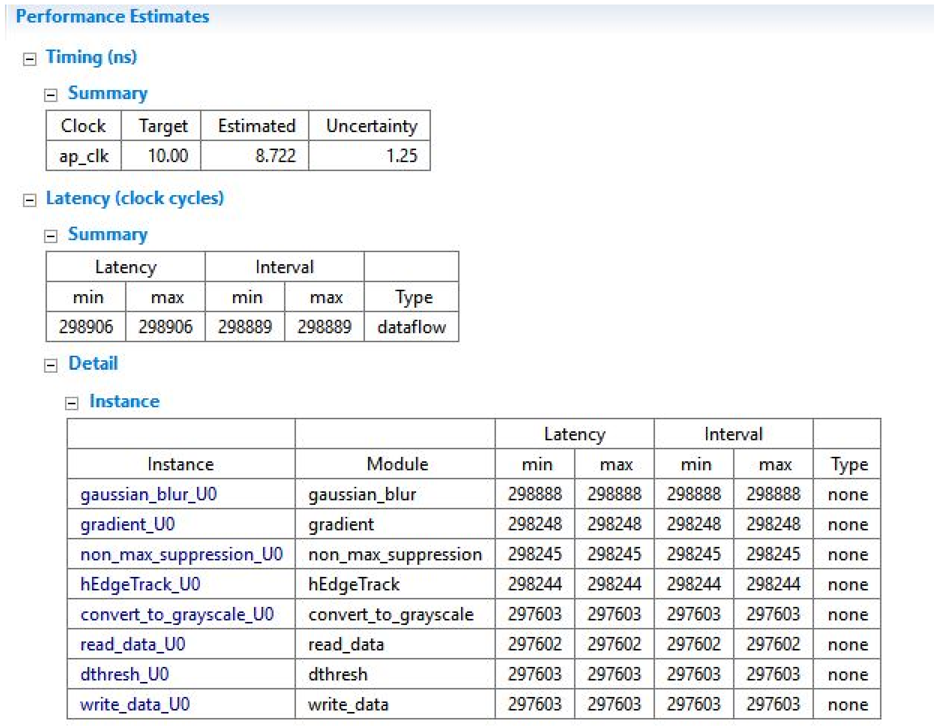
Issue was resolved by noting that the write back need not happen in the same cycle. The written back data will be accessed only after one row of data was processed. Hence, an inter loop dependency was made with sufficient cycles to get $II = 1$

Hardware Implementation | Top Level Scheduling

Dataflow optimization was used for implementing streaming functionality. In such an implementation, the II is the maximum II of each module. We note that the modules with the line buffers require additional cycles to load the line buffer. An overlap based scheduling scheme would allow for maximum throughput at $1 pixel/cycle$ for the overall algorithm We have implemented a block schedule scheme, and so the effective throughput is slightly less than $1 pixel/cycle$

Performance | Latency and II

- Gaussian Filter is the main bottleneck since it is a 5×5 kernel
- Estimated II : $640 \times 465 + 4 \times 640 \approx 300000$
- Estimated Latency : $640 \times 465 + 4 \times 640 + 2 \times 640 \times 3 \approx 300000$
- Pixels in error : 5407 | 1.8% Error Rate



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.722	1.25

Latency (clock cycles)

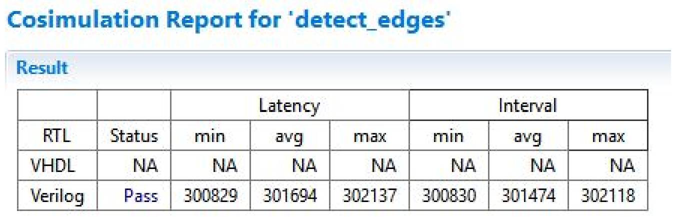
Summary

Latency		Interval		
min	max	min	max	Type
298906	298906	298889	298889	dataflow

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
gaussian_blur_U0	gaussian_blur	298888	298888	298888	298888	none
gradient_U0	gradient	298248	298248	298248	298248	none
non_max_suppression_U0	non_max_suppression	298245	298245	298245	298245	none
hEdgeTrack_U0	hEdgeTrack	298244	298244	298244	298244	none
convert_to_grayscale_U0	convert_to_grayscale	297603	297603	297603	297603	none
read_data_U0	read_data	297602	297602	297602	297602	none
dthresh_U0	dthresh	297603	297603	297603	297603	none
write_data_U0	write_data	297603	297603	297603	297603	none



Cosimulation Report for 'detect_edges'

Result

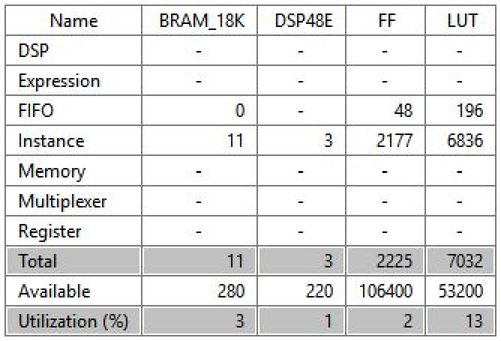
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	300829	301694	302137	300830	301474	302118

Performance | Hardware Utilization

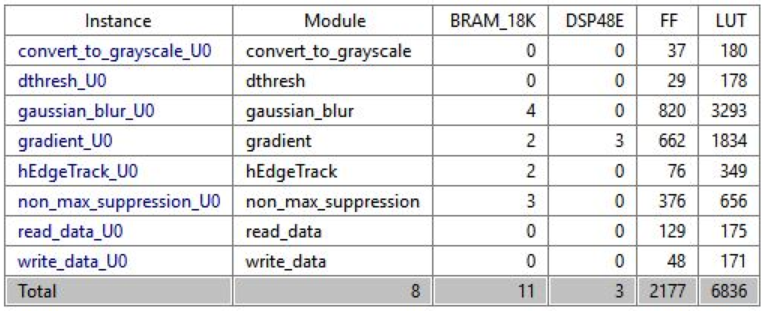
All the filter coefficients were small integers. Hence, filter multiplications were implemented as LUT in hardware To find the gradient angle, G_X and G_Y had to be multiplied by a large integer. There were 3 comparisons to be made to decide the gradient angle. Since this had to be calculated every cycle, 3 DSP slices (multiplication operation) were used for this.

Since all line buffers were large (length 640), they were implemented as block RAM.

The Gaussian filter had a 5×5 kernel, and so required 4 block RAMs. The other operations (Gradient estimation, Non-maximum suppression, edge tracking) had a 3x3 window buffer size and hence required 2 block RAMs each. Non-maximum suppression has an extra line buffer for storing the gradient angle, and so it uses another block RAM.

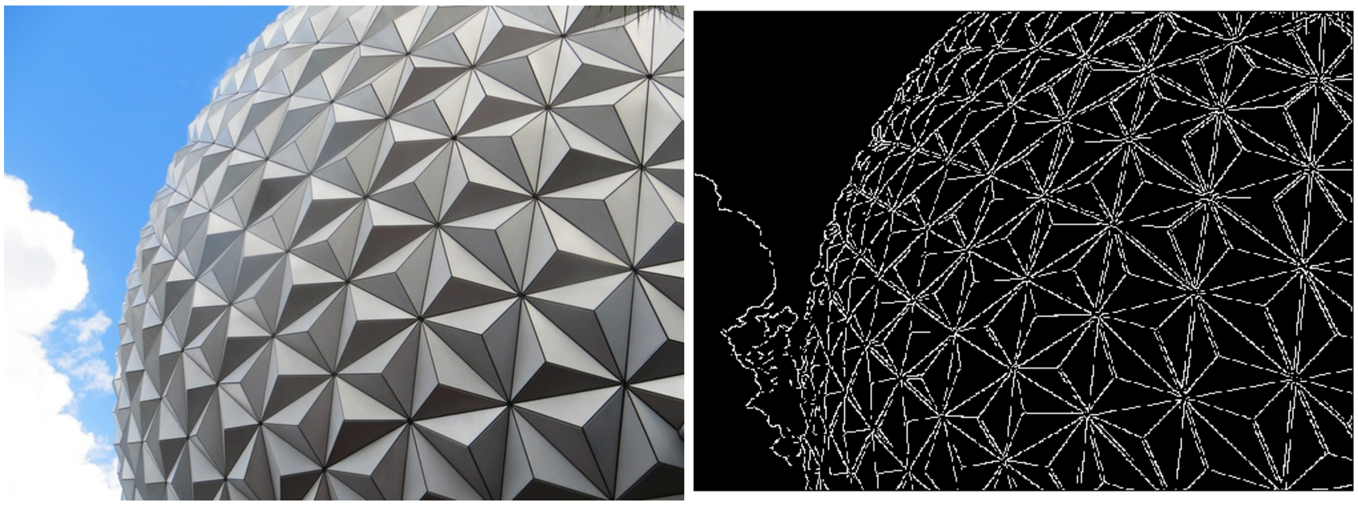


Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	48	196
Instance	11	3	2177	6836
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	11	3	2225	7032
Available	280	220	106400	53200
Utilization (%)	3	1	2	13



Instance	Module	BRAM_18K	DSP48E	FF	LUT
convert_to_grayscale_U0	convert_to_grayscale	0	0	37	180
dthresh_U0	dthresh	0	0	29	178
gaussian_blur_U0	gaussian_blur	4	0	820	3293
gradient_U0	gradient	2	3	662	1834
hEdgeTrack_U0	hEdgeTrack	2	0	76	349
non_max_suppression_U0	non_max_suppression	3	0	376	656
read_data_U0	read_data	0	0	129	175
write_data_U0	write_data	0	0	48	171
Total	8	11	3	2177	6836

Tested Image



Possible Further Work

- Save the image as its transpose, so that the line buffer size and II is reduced
- Implement the separable convolution part
- Integrate with HLS stream library code (Possibly more optimal hardware for the line buffer)
- Compare with HLS CV routines
  - Since we have optimally chosen the pixel width in every stage, and we have optimized the way the gradient angle is calculated, our implementation may be more efficient than the generalized HLS library routines
- Optimize with error rate and find optimal pixel width
- We have formulated the algorithm to use one basic computational unit. If our optimization objective was hardware utilization, we could try and implement resource sharing using this computational unit.

References

- Canny Edge Detector, OpenCV 2.4.13.7 documentation [Link]
- Canny Edge Detection Step by Step in Python Computer Vision, Towards Data Science [Link]
- Canny Tutorial , Noah Kuntz , Drexel University [Link]
- Separable Convolution, Steve on Image Processing and MATLAB [Link]
- Vivado Design Suite User Guide UG902 [Link]
- Vivado HLS Optimization Methodology Guide UG1270 [Link]
- Vivado Design Suite Tutorial UG871 [Link]