Accelerating compact digital image processing with real-time haze removal

ECE 8893 – Parallel Programming for FPGAs – Spring 2022
Georgia Institute of Technology

Final Report

Aloysius Leon Abreo
ECE Dept., College of Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
aabreo3@gatech.edu

Ankur Bindal
ECE Dept., College of Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
abindal8@gatech.edu

Anirudh Gorantla Nagaraja
ECE Dept., College of Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
anagaraja7@gatech.edu

I. Problem Description

Computer Vision applications have become ubiquitous in recent years. Image processing plays an import role in most vision applications, be it self-driving cars, AR or surveillance etc. There is also a need to run these applications on smart edge devices. Hence, we need fast and low power accelerators for these roles. FPGAs are particularly well suited for such roles because they can exploit the parallelism inherent in image processing tasks. Also, unlike GPUs they also have lower energy requirements. Using High Level Synthesis instead of directly writing RTL also allows us to prototype much faster. We are attempting to create an image pipeline and accelerate a subset of the tasks in the pipeline, namely Debayering, White Balancing and Haze Removal. The modular design of the pipeline enables us to add newer algorithms using HLS and integrate them easily with the existing data flow. This report discusses the implementation of this pipeline on Vitis HLS and details the results obtained from design space exploration.

II. Background

Image processing algorithms have been around for a long time. Consequently, a lot of research on the efficacy of these algorithms already exists in literature. As part of the image processing pipeline, we have selected a few algorithms for the pipeline that mimics a typical pipeline that would exist in a self-driving car or aerial imagery. These applications need a clear image for further processing (classification/line following/obstacle avoidance etc.) and thus having a clear image is essential. The algorithms we have selected takes a camera sensor input, makes it readable to the human eye, corrects the color of the image and removes any fog/haze in the image. A detailed explanation of all these algorithms is given below:

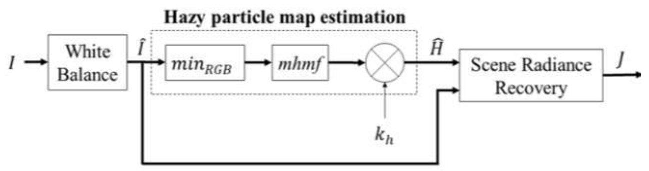


Figure 1: Image Processing Pipeline

i) Debayering

A typical image is composed of pixels. A pixel is a color representation of a point in the image an contains three components - R (red), G (green) and B (blue). Most edge applications in computer vision like drones, cars, traffic cams, etc. usually use camera sensors which do not capture all three of components of a pixel – R, G and B. Instead, to save costs, these cameras utilize a Bayer filter to capture and store only one of the R, G or B pixel value for each pixel. However, for any computer vision application, all three channel values are needed to recreate the scene color and accurately run other algorithms. A Bayer pattern image is shown in Figure 2 for visualization. In its current form, this image cannot be used as input data for any image processing algorithm. Since human eyes are more susceptible to green in the environment, green is captured more as compared to other colors in the Bayer filter pattern giving the image an overall green hue. Several interpolation techniques exist to effectuate debayering. These techniques calculate the missing R, G or B values at each pixel using the values captured at nearby pixels. This could just use values at distance 1 (nearest neighbor), an interpolation of values at distance 2 (bilinear interpolation) or distance 3 (bicubic or Lanczos interpolation).
In implementing the debayering scheme, we have used bilinear interpolation. However, it can also be extended to other interpolation methods. For each pixel, based on its row and column values, we can identify whether the Bayer pixel is Red, Green, or Blue and based on this information other color values are calculated. Based on these patterns, we identified 5 variants of 3x3 kernels that compute the average of required pixels to generate the missing value.

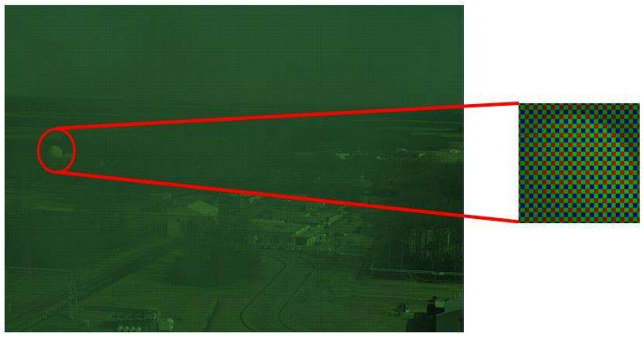


Figure 2: A Bayer filter image

These kernels are – *horizontal* (average horizontal neighbors), *vertical* (average vertical neighbors), *cross* (average horizontal and vertical values), *diag* (average values across both diagonals) and *static* (use pixel's own value). Based on pixel's row and column number, 3 of these 5 filters are then used to generate the complete 24-bit RGB pixel. An output of the debayering stage is shown in Figure 3.

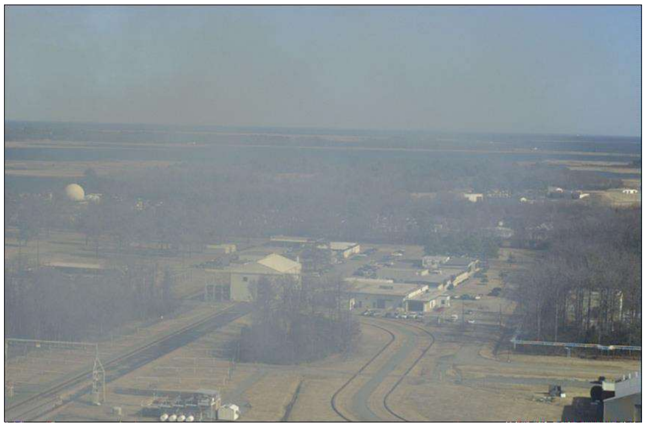


Figure 3: A Debayered Image

ii) White Balancing

As the name suggests, white balancing is the correction of color temperature in an image. The atmospheric light in images might have a certain tone to it. For example, a picture depicting sunset may have orange atmospheric light. Without white balancing, colors may appear unnatural in images, like skin tones. It is also necessary to white balance an image before haze removal techniques can be applied.

To perform white balancing, we have employed the "Simplest Color Balance" technique described in [1]. White balancing techniques normally take the most and least intense pixels in an image and saturate them. However, these pixels may simply be noise or the product of a poor camera sensor. To prevent this, we first make a histogram of the number of pixels of each intensity value. Based on the distribution of this histogram we saturate a percentage of pixels instead of a fixed number. After saturating these extremities, we apply an affine transformation to the remaining pixels to stretch them out to the actual intensity range (8 bit in this case). Finally, we clip any anomalous values (e.g., intensity greater than 255). We can observe the output of our white balancing implementation in Figure 4.

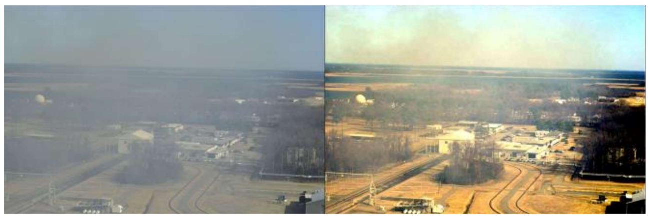
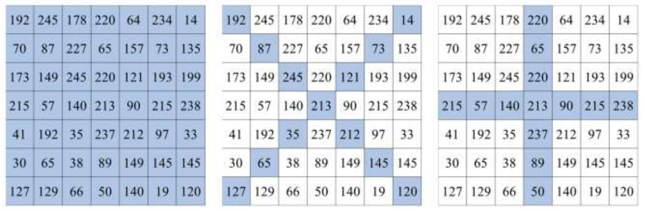


Figure 4: White balancing using "Simplest Color Balancing"

iii) Dehazing

The next component of the image pipeline involves the removal of fog/haze from an image (dehazing). This is essential for clear imagery in poor weather conditions, especially for self-driving and aerial imagery applications. The method of dehazing described in the paper [2] involves calculating median values to generate a Hazy Particle Map for the image. This Hazy Particle Map is subtracted from the actual image to get an image without haze. Computing the Hazy Particle Map involves calculating the median values for a given window of pixel values. The paper proposes a Hybrid Median Filter (mHMF) which is used to calculate the median values of the image data. For a given channel, a 7x7 tile is used to calculate the median in a square window, diagonal window, and cross window as shown in Figure 4. The final median value is calculated from these three intermediate values.



192	245	178	220	64	234	14
70	87	227	65	157	73	135
173	149	245	220	121	193	199
215	57	140	213	90	215	238
41	192	35	237	212	97	33
30	65	38	89	149	145	145
127	129	66	50	140	19	120

192	245	178	220	64	234	14
70	87	227	65	157	73	135
173	149	245	220	121	193	199
215	57	140	213	90	215	238
41	192	35	237	212	97	33
30	65	38	89	149	145	145
127	129	66	50	140	19	120

192	245	178	220	64	234	14
70	87	227	65	157	73	135
173	149	245	220	121	193	199
215	57	140	213	90	215	238
41	192	35	237	212	97	33
30	65	38	89	149	145	145
127	129	66	50	140	19	120

Figure 5: Square window, diagonal window, cross window

Calculating a median for a given set of values also involves some sorting. To accelerate this, the paper describes an algorithm based on Batcher's parallel sort algorithm which is implemented in our FPGA synthesized design.

The input to a Hazy Particle Map estimation is a white-balanced image. The image data is read and the minRGB() value for each pixel is calculated and stored in a channel. This channel is passed through the mHMF to compute a median. The final calculated median value is multiplied with a tunable weight factor 'k_h' (between 0 and 1) depending on the haze in the image. $\widehat{H}(x,y)$ is the estimated particle map and $\hat{I}(x,y)$ is the white balanced input image.

$$\widehat{H}(x,y) = k_h * mhmf(\, min_{RGB}(\, \hat{I}(x,y)\,)\,)$$

The calculated channel with median values is read back into an image which represents the Hazy Particle Map shown in Figure 6.

This particle map can be 'subtracted' from the original, white-balanced image, and the tone recovery can be used to restore the image. Subtraction of the Hazy Particle Map from the actual white balanced image is done using the following expression:

$$\hat{J}(x,y) = \frac{\hat{I}(x,y) - \hat{H}(x,y)}{1 - \hat{H}(x,y)}$$

$\hat{J}(x,y)$ is the recovered scene radiance after Haze Removal. A detailed explanation of the computation of the equations is given in reference 3.

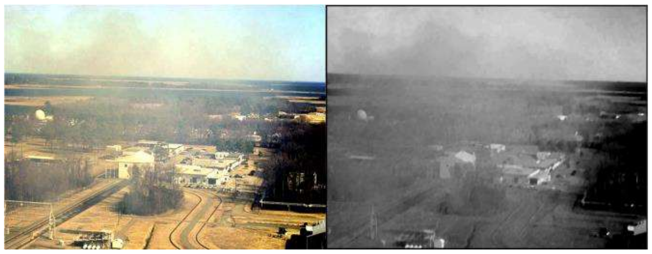


Figure 6: White balanced image & Estimated Hazy Particle Map

III. HLS Implementation

We used HLS (High Level Synthesis) to implement the pipeline on an FPGA which involves using C++ to realize the image processing algorithms and synthesize them into a Hardware Description Language (Verilog) IPs that can be executed on hardware. A baseline C-model was written to integrate the pipeline elements together and verify expected functionality. Since HLS is eventually synthesized to fixed size hardware elements (eg. registers, muxes, DSPs etc.), any dynamic allocation of memory at runtime or use of object-oriented design (classes) is not possible. This was major challenge with implementing the given algorithms. We started with implementations that used traditional functions (std::sort, std::vector etc.) from the STL (Standard Template Library) and modified the code to make it synthesizable using Vitis HLS. Results from the C-model implementations matched the expected results from our reference papers and that provided a stable starting point for further work.
As part of Design Space Exploration (detailed in section IV) tasks we looked at various approaches to optimize the algorithm implementations to integrate them within the pipeline, support modular addition and decrease utilization of used resources on the board. This section describes the implementation of the image processing pipeline modules in HLS.

i) Debayering

As discussed in section II, debayering consists of finding the missing pixel values for R, G and B from the neighboring pixel values that were captured by the bayer filtered camera. This employs use of different kernels to do an interpolation of neighboring values and the operation is very similar to any kernel applied in Convolution Neural Networks (CNNs). However, unlike a CNN kernel, the Debayer kernels applied on each value are not the same and are dependent on the row and column index of the pixel. As discussed in section II, we identified that based on which value (R/G/B) is missing, one of five different kinds of kernels is used for debayering. The kernels are as shown in Figure 7 with weights scaled to avoid floating/fixed point calculations errors.
This kernel implementation is a challenge in HLS since standard optimizations that are usually applied to CNN kernels cannot be extended to debayer kernels due to presence of multiple conditional statements and variation in kernels based on the condition result. On further design exploration, we identified that while CNN kernels are large and each weight is of the same size as a pixel/channel value, 3x3 debayer kernels are much simpler and the corresponding weight values are small.

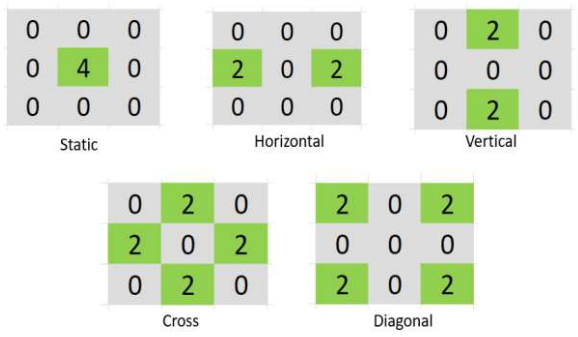


Figure 7: Debayer Kernels

We used this observation to optimize the debayer module code and remove redundant conditional statements. Each channel only uses 4 out of the 5 kernels described above and each kernel requires a maximum of 3 bits. So, we combined the kernels and create one unified kernel for each channel (R, G, B). Each unified kernel element was now sized at 16 bits with each nibble representing value from the corresponding debayer kernel used for that color channel. First, the index of a pixel is computed and then used to index into the correct nibble for the corresponding debayer kernel while the other values masked for correctness. Using this approach, we were able to reduce the number of instances from 5 kernels to 3 kernels which can be applied in tandem to generate all 3 channels in a single loop. Since the size of kernels was increased from 3 bits to 16 bits, we did see some increase in BRAM utilization. However, as the size and dimension of our kernels is very small, the increased cost was offset by the substantial improvement in overall latency as well as improvement in other resource utilization (DSPs) since the logic for debayering is now simplified with this optimization.

ii) White Balancing

We implemented the naïve version of white balancing in C++ to check whether the "Simple" algorithm described in section II works as expected. We fine-tuned our parameters and decided to use a pixel count percentage of 5% to decide our Vmin and Vmax boundaries. As can been seen from Fig. 8 this bound is needed to stretch the histogram to provide a more even distribution of pixels for all channels across all intensities.

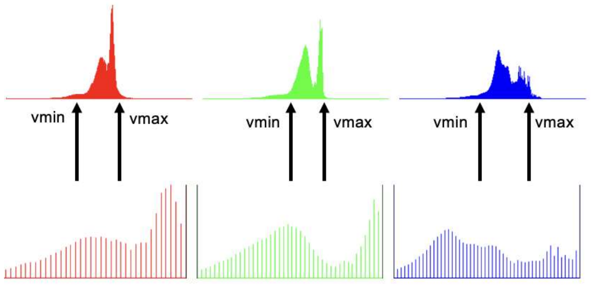


Figure 8: Histogram stretching in White Balancing

In the HLS implementation we collect global data in the de-bayering stage and calculate the vmin & vmax once the date for the entire image has been collected. This is explained in greater detail in section IV.2.

Once we have the vmin and vmax, we simply apply a linear transformation on all pixels in the tile. This step is unique in this regard. Unlike the other steps, it is a point operation and doesn't depending on the neighboring pixel values. This is also the reason this step has the lowest latency. We also decided to implement white balancing on the FPGA because we do not have to deal with the overhead of identifying the neighboring pixels before applying a transformation on them. We can simply make an in-place change on the pixel for all the streams independently.

iii) Dehazing

The dehazing step involves calculating the minimum RGB values for the given image. This minimum RGB channel is then passed through a mHMF – modified Hybrid Median Filter that computes the median values of the entire image. The median value is calculated using a 7x7 sliding window across the entire channel. For a given 7x7 window containing 49 elements, three median values are calculated (as described in Figure 5) for the different window elements. The median among these 3 values is computed and inserted for the corresponding pixel.

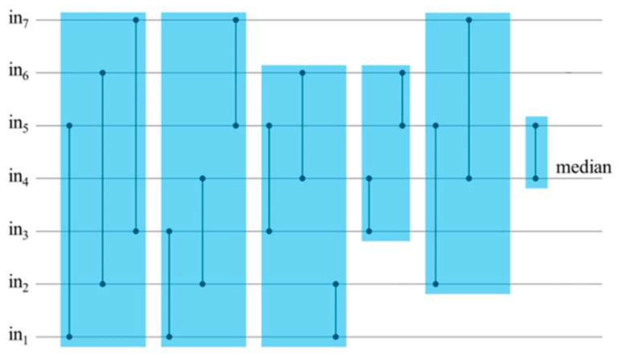


Figure 9: Median calculation based on Batcher's Sorting Network

Median calculation involves sorting values (49 elements in the square windows and 13 elements in the diagonal and square windows) to find the median value. The paper describes a sorting mechanism that is based on Batcher's parallel sorting network. The implementation involves using multiple "compare and swap" steps to compute the median as described in Figure 8. The compare and swap unit checks two inputs and sorts it in descending order. As show in in Figure 9, after 6 cycles, input 4 is selected as the median. The blue boxes marked in the figure operate in parallel and this improves the overall latency of the design.

iv) Denoising

The next theoretical step in the pipeline is radiance recovery[3] and was not part of the original implementation plan (due to limitations on our bandwidth and stringent evaluation timelines). However, we decided to implement a simpler Gaussian Smoothening operation to remove the noise from the recovered scene radiance from dehazing and reduce overly saturated values produced for improvements in the output image quality. Moreover, since the gaussian filter is fixed here, we directly calculated the output to keep the code minimal. Since this was not part of the initial plan, we did not run pipeline latency and utilization evaluations with denoising module. The results section details the denoised image results from the native C model implementation.

IV. Design Space Exploration

Our initial HLS implementation required multiple optimizations. Since our pipeline processes images, loading the complete image on to the FPGA BRAMs is not a viable solution. Consequently, we kept the image in the FPGA DRAM and accessed parts of it. This is a classic example of tiling a large input image to improve resource utilization. As part of design space exploration, we explored multiple tiling sizes to get the most efficient tile size for our design.

We combined multiple functions together to create an image pipeline. Although this offered good latency since the complete image frame was copied to BRAM first, the utilization of BRAM resources was extremely high even for a relatively small image resolution of 512x384. We saw ~1000% BRAM Utilization when running these algorithms individually on complete frames. This limits the scalability of these algorithms on smaller FPGA hardware. We wanted to make this image processing pipeline compact so that it can be scaled and be image resolution agnostic. We looked at modifying the pipeline such that an input tile of the image can be loaded from DRAM, processed and the final output tile is stored back to DRAM. This requires some considerations as explained below:

1. Debayer and dehazing algorithms work on kernels and they need extra information to generate a smaller output tile. To incorporate for this, we fetch overlapping tiles from DRAM which loads the extra pixels needed on all image boundaries to produce a smaller output tile. Our current implementation requires a single pixel padding for debayering and 3 pixel padding for dehazing so

there is an overhead of 8 pixels which overlap while loading consecutive tiles from DRAM.

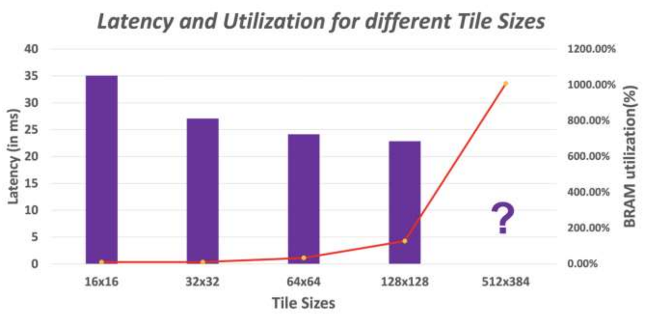


Figure 10: Latency vs Utilization for different input Tile sizes

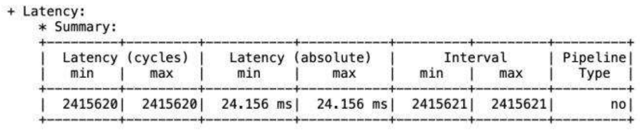
2. A second important consideration is that White balancing requires data from a global histogram data the input image. However, it is not possible to do this if input is fetched and processed in a tile-by-tile manner as it will only have local histogram data. To solve this, we assumed that the image histogram data will not change much from 1 frame to the next in a real time environment (stream of continuous incoming images) and thus we can use the histogram data calculated on the first frame and use it for white balancing the subsequent frame. This allows us to combine the histogram calculation with debayering operation so that white balancing is simple and can be applied to the incoming tile based on last frame's histogram results.

3. Lastly, choosing an optimal tile size is crucial to get a balance between performance and resource utilization. Higher tile size implies less overhead of overlap and faster processing but at the same time translates to high resource utilizations. Having smaller tile size of the other hand requires significantly less resources but tile overlap starts to dominate to effect latency. This tradeoff analysis between tile sizes, BRAM utilization and Latency is described in Figure 10. We conducted multiple experiments to find an optimal tile size that worked well for our input image size.

With these optimizations, we tried running different images and tile configurations using our HLS code and verified images obtained from the baseline C simulation.

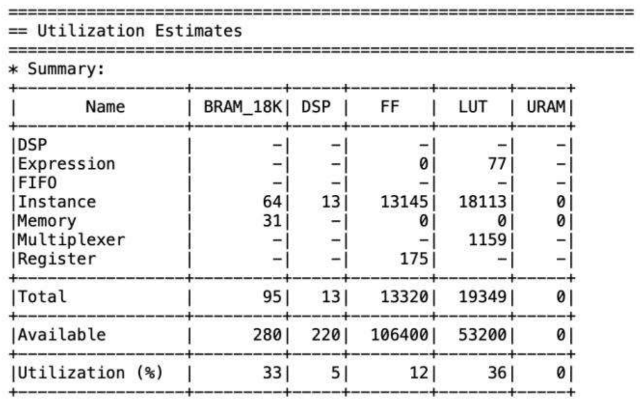
IV. Results and Evaluations

We implemented our design in Vitis HLS and measured utilization and latency numbers for a PYNQ-Z2 FPGA board. Our image pipeline implementation achieved a real-time latency of 24.1 milliseconds which translates to ~41 frames/sec for a 512x384 image resolution with a 72x72 tile size (including padding for pipeline modules).



```
+ Latency:
    * Summary:
    +---------+---------+-----------+-----------+---------+---------+----------+
    |  Latency (cycles) |  Latency (absolute)   |      Interval     | Pipeline|
    |   min   |   max   |    min    |    max    |   min   |   max   |   Type  |
    +---------+---------+-----------+-----------+---------+---------+----------+
    |  2415620|  2415620| 24.156 ms | 24.156 ms|  2415621|  2415621|       no|
    +---------+---------+-----------+-----------+---------+---------+----------+
```

Figure 11: Latency results for the image processing pipeline



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|     77|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       64|   13|   13145|  18113|    0|
|Memory           |       31|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   1159|    -|
|Register         |        -|    -|     175|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |       95|   13|   13320|  19349|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       33|    5|      12|     36|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Figure 12: Utilization estimates for the image processing pipelines

The corresponding CPU implementation (C simulation) takes 2.1 seconds. This is a **~80x speedup** of the pipeline on the FPGA as compared to a C implementation. The Vitis HLS results for latency and utilization are marked in Figures 11 and 12. Intermediate images were written out from the image pipeline and the results for the processed images can be observed in Figure 13.

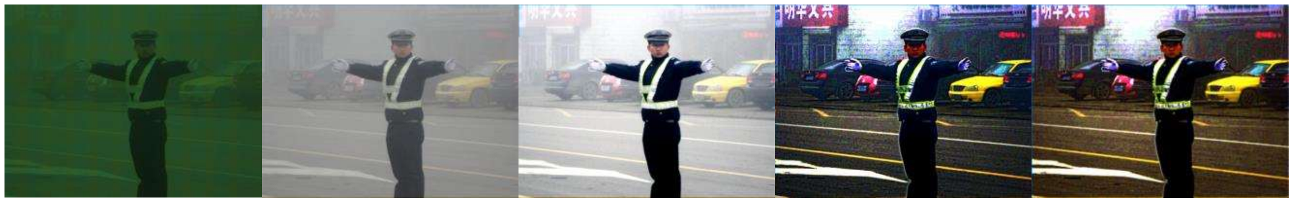


Figure 13: From L to R – Camera Sensor output, Debayered Image, White balanced image, Dehazed and recovered scene radiance, Gaussian smoothened image

V. Conclusion

A fast and efficient image processing pipeline has been implemented on a PYNQ-Z2 FPGA board. Accelerating image pipelines have a wide application in various image processing schemes that demand real time image analysis with a low power and hardware cost overhead. This project successfully demonstrates the feasibility of a modular image processing pipeline and highlights the potentials of accelerating these algorithms on FPGA hardware.

VI. REFERENCES

[1] N. Limare, "IPOL Algorithm: Simplest Color Balance," http://www.ipol.im/pub/algo/lmps_simplest_color_balance/

[2] Ngo, D.; Lee, G.-D.; Kang, B. A 4K-Capable FPGA Implementation of Single Image Haze Removal Using Hazy Particle Maps. Appl. Sci. 2019, 9, 3443. https://doi.org/10.3390/app9173443

[3] Geun-Jun KIM, Seungmin LEE, Bongsoon KANG, Single Image Haze Removal Using Hazy Particle Maps, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2018, Volume E101.A, Issue 11, Pages 1999-2002, Released on J-STAGE November 01, 2018, Online ISSN 1745-1337, Print ISSN 0916-8508, https://doi.org/10.1587/transfun.E101.A.1999, https://www.jstage.jst.go.jp/article/transfun/E101.A/11/E101.A_1999/_article/-char/en