MASTER

Accuracy Configurable ISP Accelerator for an Image-based Control System

Ravattu, Anoop Krishna

Award date:
2021

Department of Electrical Engineering

Electronic Systems Research Group

Accuracy Configurable ISP Accelerator for an Image-based Control System

Master Thesis Report

Anoop Krishna Ravattu

Student ID: 1336444

Supervisors:

Assistant Professor: Dr. Dip Goswami

Ph.D. Student: M.Sc. Sayandip De

Eindhoven, October 2021

Table of Contents

List of Figures

List of tables

Chapter 1 Introduction

1.1 Introduction

Ever increasing urge for the comfort and sophistication has always been the utmost priority in the evolving world of technology. The increase in number of automobile accidents due to human errors has only necessitated and reinforced the need for autonomous driving and the autonomous cars.

An autonomous car, in essence, is a vehicle customized to carry sensory units on the car that can make out its surroundings, navigate through traffic and also park without support from any human driver. Autonomous driving consists of an artificial intelligence system that can sense the surroundings, process the visual data so as to determine how to avoid collisions, operate car machinery like the steering and brake, and make use the Global Positioning System (GPS) to track the car's current location and destination.

Autonomous driving, no doubt, can positively revolutionize the transport industry. Automakers and tech companies are competing to be first with such technology. Complete autonomous vehicle still is a distant reality. However, quite a few car companies did use available technologies and a few of them stand out. The list includes Waymo, GM Cruise and Argo AI. Waymo, under Google's parent company Alphabet Inc., did impress with its autonomous vehicles driving over 5 million miles on public roads [20].

Cameras are the fundamental sensory units which are used to capture the real-world data that needs to be processed. These camera sensors need to be able to produce predictable and reliable images that can be used by computer vision. With the increase in the integration of the cameras in both mid and luxury vehicles, the camera capabilities allow Advanced Driver Assistance System (ADAS) to be employed for vision-based perception tasks, such as object recognition, mapping, and path planning, which are Image-Based Control (IBC) tasks that will enable a safer driving experience and bring us one step closer to a fully autonomous car.

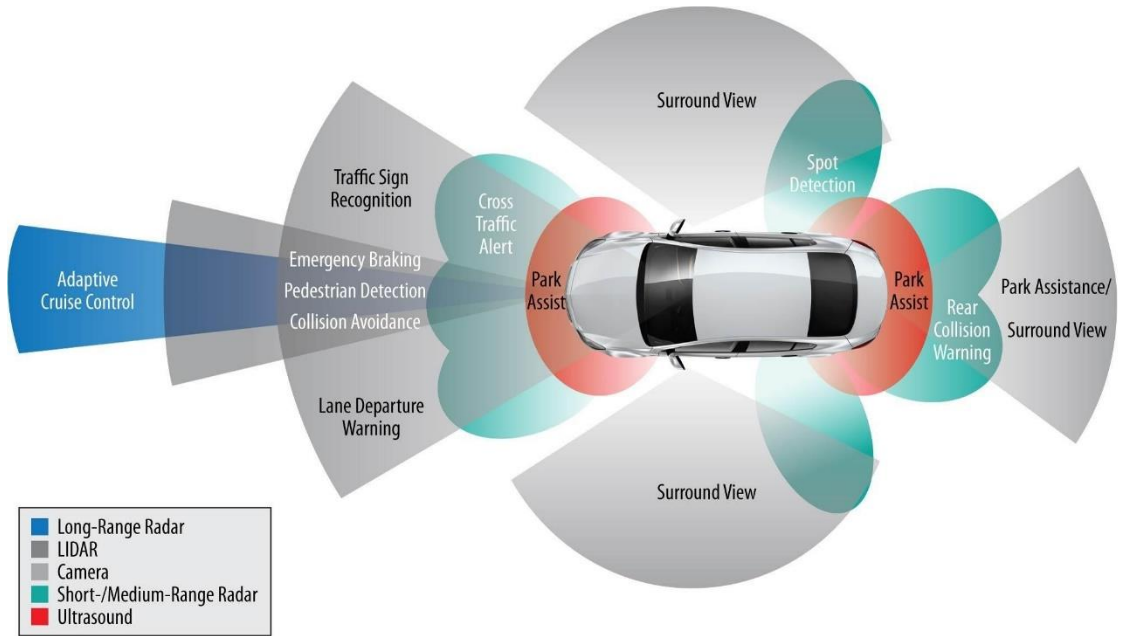


Figure 1. Advanced Driver Assistant System [17]

A traditional Image Signal Processor (ISP) is used to process the RAW image produced by the image sensor. The ISP converts the RAW image into a compressed image which can be later used by a computer vison application like lane detection. A typical ISP has five stages which include demosaicing, denoising, color transformation, gamut mapping, tone mapping and compression. Although, these stages are standard in the ISP, they require excessive computational intensity to produce a high-quality image. Since the computer vision applications do not need a high-quality image, the results can be approximated which decreases the computational intensity.

Approximate computing is a paradigm that will trade the accuracy for a better computation time and energy in error resilient applications by building the systems with inexact hardware or software components. A key challenge in the approximate computing is to identify the sections that can actually be approximated, without having the risk of crashing the application if a critical component is approximated. The focal point of this project is determining how to improve the latency of an ISP by using the concept of approximations.

Chapter 2 Background

2.1 Digital Image Processing

Image processing is a technique used to perform some operations on an image either to enhance it or extract useful information from it. Processing an image through an algorithm with the help of a computer is called digital image processing. For an image to be processed, it first needs to be captured by a camera sensor which can output digital values for a pixel.

All the images can be classified into two types: gray scale image and color image. A typical gray scale image has one channel which looks like a black and white image. Each pixel of this image is represented only by one value per pixel. On the other hand, to represent a color image, multiple channels are required. There are various color models like RGB (red, green, blue), CMYK (cyan, magenta, yellow, black) etc. to represent the color image. Among these formats, RGB is the most common type used to represent a color image with three values per pixel. As the very name suggest, RGB is made up of three different channels of red, green and blue. These colors are considered as the primary colors; as they can be added together to produce a broad array of colors. A more comprehensive way of adding colors to produce different colors is presented in Figure 2

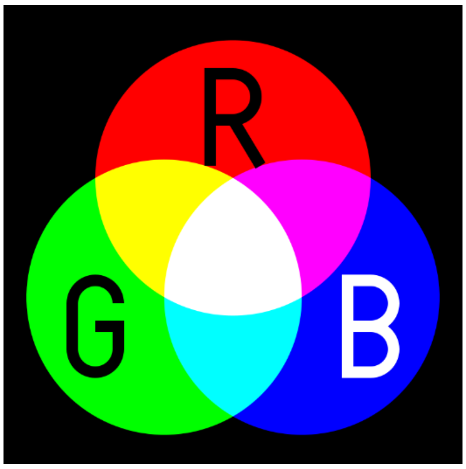


Figure 2. RGB color model [1]

It is clear from Figure 2 that adding of

- red and green produce yellow.
- red and blue produce pink.
- blue and green produce magenta.
- red, green and blue together produce white.

Thus, the combination of these three colors can produce a wide range of different colors which represents a color image. Typically each of the channel is represented in decimal numbers ranging from 0-255. For the values of R, G and B at 0 it produces the color of pure black, Similarly, for the values of R, G and B at 255 it produces pure white. Thus the total number of colors available for an image is 16,777,216 (256 x 256 x 256) when all the three channels are combined which covers a wide range of color spectrum.

2.2 Camera sensor

An image/camera sensor is a sensor which detects and conveys the information of the environment in the form of an image. The most common image sensors used in battery powered devices are CMOS type sensors as they are economical and have a low power consumption when compared to the CCD sensors.

A typical CMOS sensor consists of three components that are presented in Figure 3.

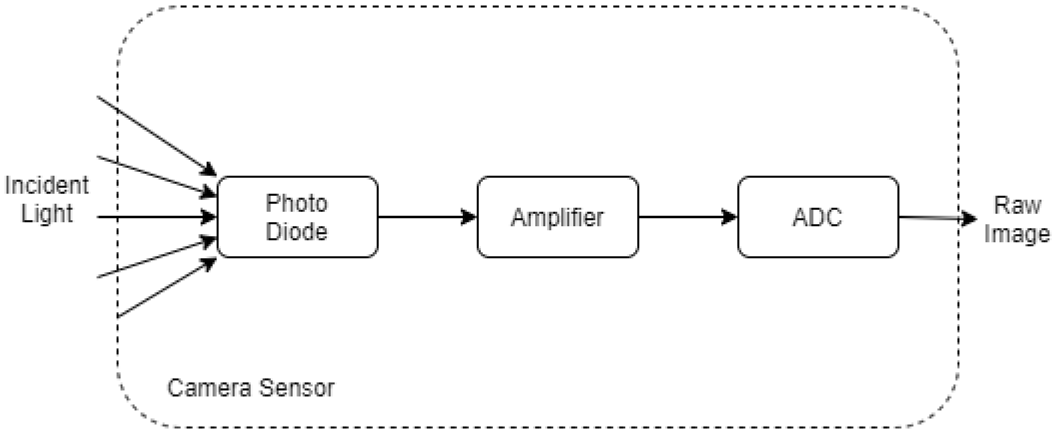


Figure 3. Structure of a Camera Sensor

The three components of a CMOS camera sensor are:

- Photo diode; used to capture the voltage of the light
- Amplifier; used to amplify the captured voltage
- An Analogue to Digital Convertor (ADC); which is used to convert the captured Analogue in to a digital pixel value

For a color camera sensor, a Color Filter Array (CFA) is placed on top of the sensor which allows only certain colors to pass through it. A CFA is a mosaic of tiny color filters placed over the pixel sensors. To capture the color information, the most common pattern used for a CFA is the Bayer pattern which is of size 2x2 and consists of two green color filters, one blue color filter and one red color filter. An example of CFA placed over pixel sensors is shown in Figure 4.

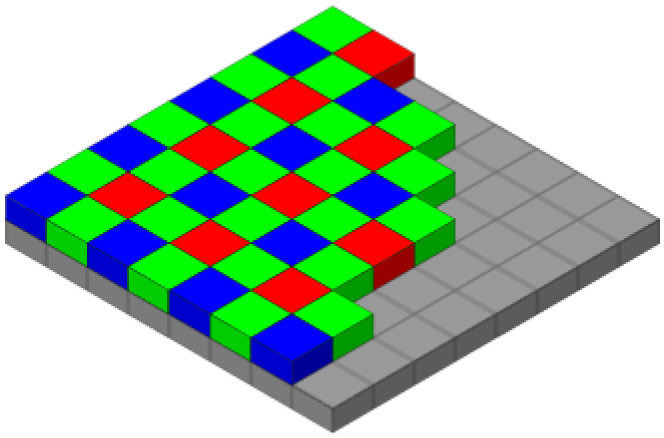


Figure 4. Color Filter Array placed over pixel sensors [2]

As the human eye is more sensitive to the green color, the green component filters present are twice as many as the red and blue components. The filters only allow their respective colors to pass through them giving the following resultant pattern (shown in Figure 5).

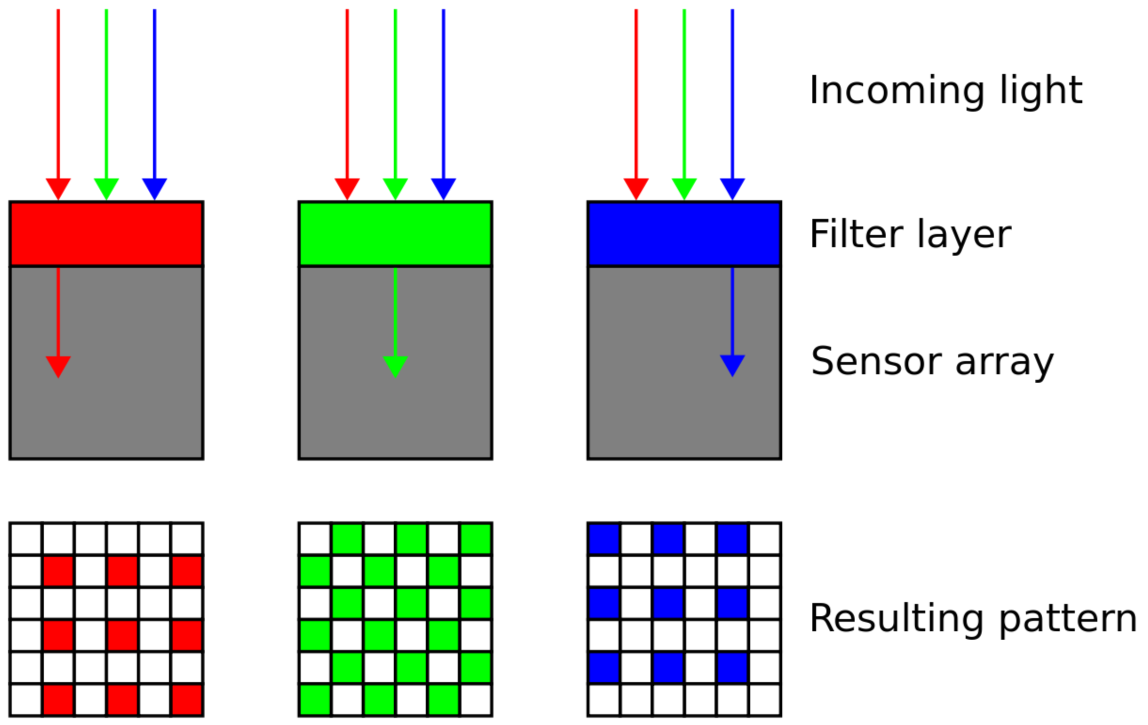


Figure 5. Raw image pattern [6]

All the three resulting patterns shown in the above figure make up the raw image which is given to an Image Signal Processor (ISP) for further processing.

2.3 Image Signal Processing Pipeline

When an image is captured by a camera sensor, it picks up noise from various sources which affect the quality of the captured image and this captured image is called raw image [15]. The raw image needs to be processed through a set of algorithms at each stage to remove these noises and make it visually more appealing to the end user. The combination of these stages which are used to produce the desired final image is called an image processing pipeline.

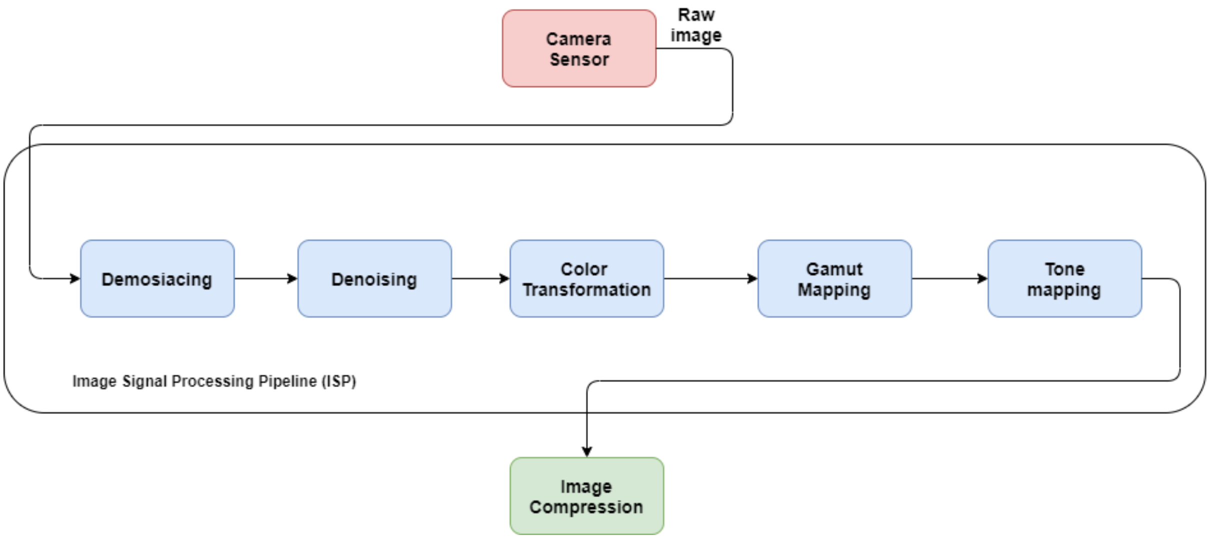


Figure 6. Image Signal Processing Pipeline (ISP)

An image processing pipeline will have multiple stages depending upon the application, the most common stages for a color image processing pipeline are shown in Figure 6 and a brief description of each of these stages are mentioned below:

2.3.1 Demosaicing

The first stage in most of the image processing pipelines is Demosaicing. This stage is used to re-construct a full colored image from the incomplete color samples created by the sensor. A single channeled grayscale raw image is converted into a three channel RGB image by using an interpolation algorithm. Figure 7 illustrates how the raw image looks like post demosaicing.

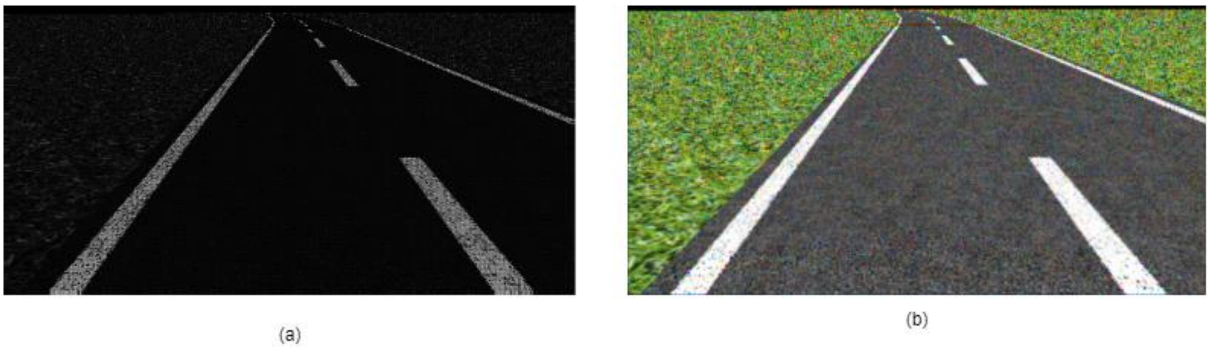


Figure 7. a) Raw image b) Demosaiced raw image

2.3.2 Denoising

The second stage of the pipeline is called denoising. When an image is captured, various noises are associated with it. The most common types of noises associated with digital images are gaussian noise and, salt and pepper noise. The goal of the denoising stage is to remove these unwanted noises and recover the original image. Figure 8 illustrates a clear difference between noisy image and denoised image.

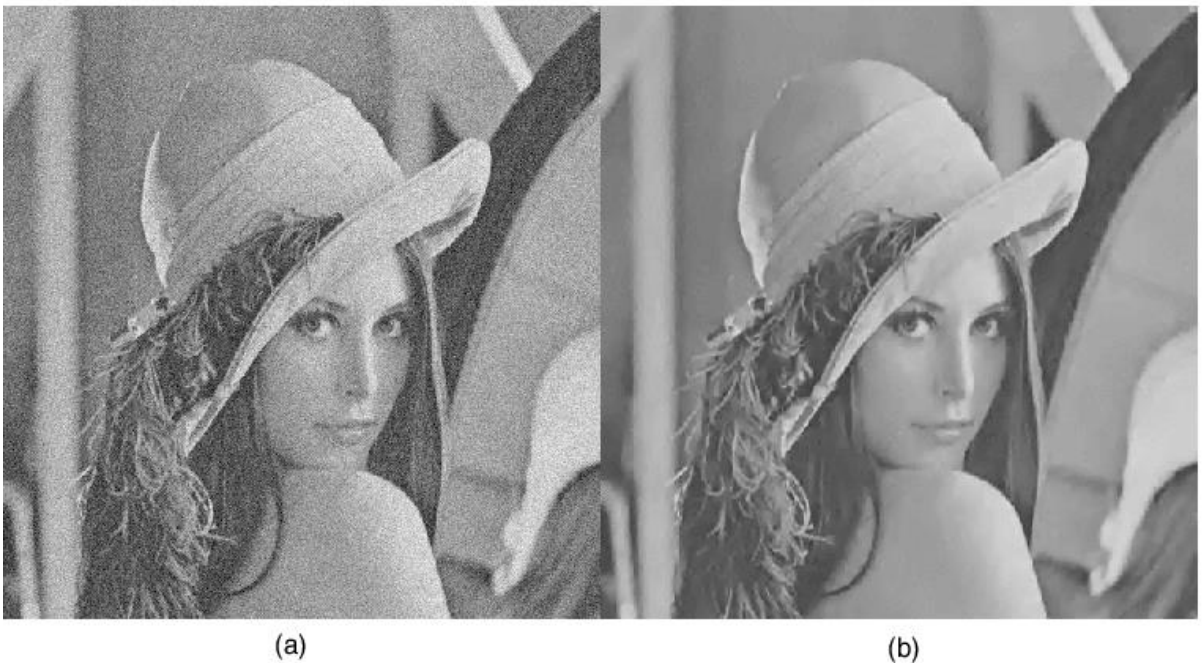


Figure 8. a) Noisy image, b) Denoised image [22]

Various denoising algorithms like local means, nonlocal means and median filtering are used to remove these noises. A brief explanation on each of the denoising methods is given below:

Local Mean Filtering: Local mean filtering is a simple smoothing filter which changes the pixel value that are unrepresentative of their surroundings [21]. The technique employed in mean filtering is taking the average of surrounding pixels including itself and replacing it with the obtained value. Depending on the number of surrounding pixels considered the smoothness of the image will be changed.

Non-Local Mean filtering: Unlike the mean filter, the Non-Local Mean (NLM) pixel takes the average of the all pixels in the image based on how alike they are with the target pixel and assigns the weight for all the pixels to be averaged [21].

Median filter: Unlike the local mean filter, the median filter sorts all the surrounding pixel in an order and finds the median of it and replaces with the target pixel. Median filtering is useful for removing the random intensity spike and helps in preserving the edges.

2.3.3 Color Transformation and White balancing

In this stage, color balancing of each pixel is done which is the global adjustment of the intensities of all the colors in a pixel which is done in two steps. First step is the color transformation, which reduces the intensity of green color to match with that of red and blue. The second step is a white balancing transformation, which reduces the temperature of each pixel to match that of the lightning in the scene. These two transformations can be applied by multiplying each pixel with a matrix of predefined weights. These weights are generally decided by the camera manufacturer. Figure 9 illustrates how the intensities of the picture are adjusted post color transformation and white balancing.

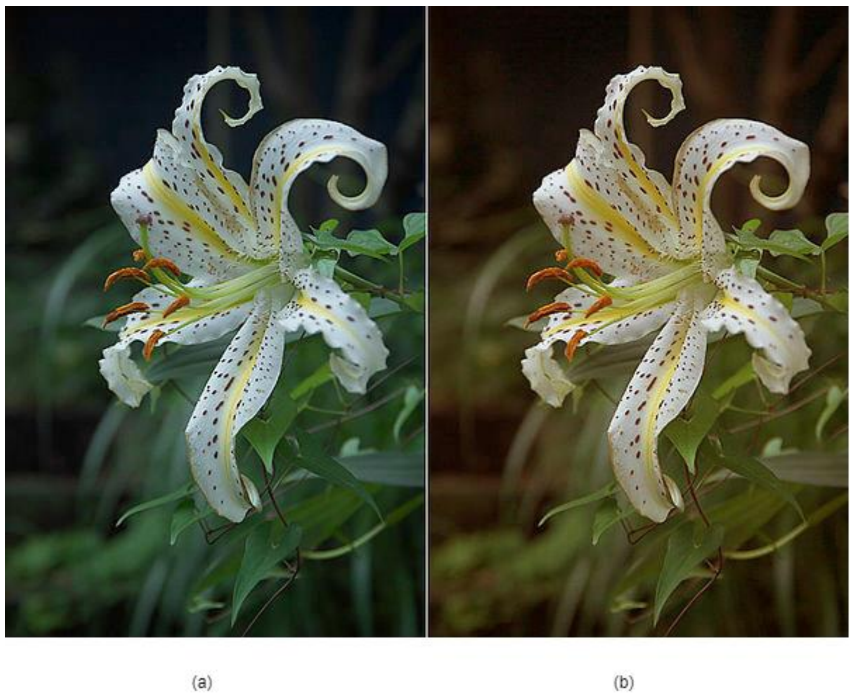


Figure 9. a) Normal image, b) Post Color Transformation [23]

2.3.4 Gamut Mapping

Gamut also known as the color space is a subset of colors that can be supported by various physical devices like printers and monitors. There are several industrial standards like Adobe RGB, sRGB, CMYK etc. for the color spaces. Among these, most of the monitors use the sRGB while the printers use the

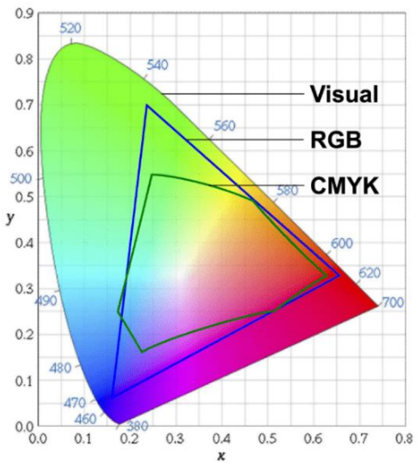


Figure 10. color model of visual spectrum, RGB and CMYK [7]

CMYK color space. It can be seen from Figure 10 that the color space of the human eye is the biggest of all. It is not possible to replicate the exact color seen by a human eye to a monitor which has a smaller sRGB gamut. The gamut of CMYK is much smaller when compare to sRGB as not all the color intensities displayed on a monitor can be printed. To map the colors from a bigger color space (sRGB) to a smaller (CMYK) a gamut mapping algorithm is required.

2.3.5 Tone Mapping

Tone mapping is a technique which is used to compress the image's dynamic range and apply some aesthetic effects. LCD Monitors and projectors all have a limited dynamic range that is not adequate to reproduce the full range of light intensities present in natural scene. Tone mapping addresses this by making dark areas brighter, at the same time it will not overexpose the brighter areas. Gamma compression is one of the most common algorithms used in tone mapping. Figure 11 illustrates how the brightness of darker areas are improved by using tone mapping.

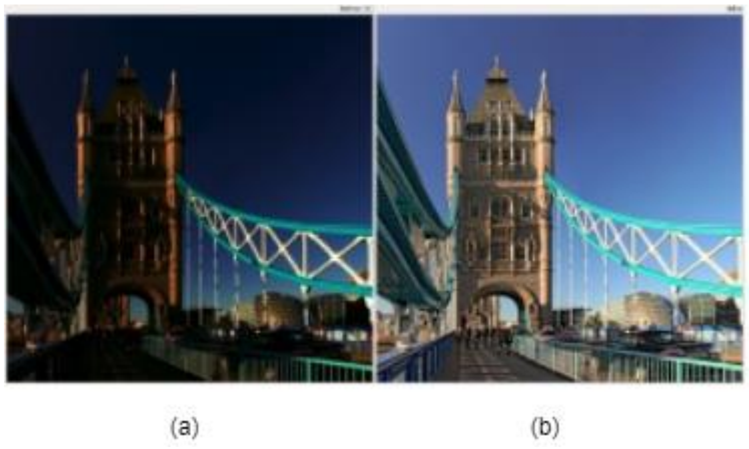


Figure 11. a) Before Tone Map, b) After Tone Map [24]

2.3.6 Image Compression

After processing an image, a compression algorithm is applied before storing the image to reduce the storage costs and communication costs. In essence, an image compression technique involves removing or grouping together certain parts of image that look alike to reduce the size of the image. Image compression techniques can be classified into two, they are lossy compression and lossless compression techniques. As the name suggests, a lossless compression technique maintains the same quality of the image as before it was compressed. Some of the image formats that apply lossless compression are PNG, RAW and BMP. The compression techniques performed on these formats are reversible and can be reverted back to the original image [19]. On the other hand, lossy compression discards some parts of the image and reduces the size of the image. However this does not mean that compressed image will look bad. JPG also known as JPEG is the most commonly used lossy compression technique [19].

The main goal of this project is to approximate the ISP by skipping the above-mentioned stages. Since the image compression is a mandatory stage to save communication and storage costs, it cannot be skipped. Instead, the output from the skipped ISP stages is given to a dedicated image compression block.

2.4 Image Based Control Systems

IBC loop systems are closed loop control systems which are used typically for applications that use visual feedback for motion control. Many modern-day applications like Lane Keep Assist System (LKAS) and Advance Driver Assistance System (ADAS) use the concept of IBC to inform the driver when the car deviates from the lane. The three tasks that make up a typical IBC system are:

Sensing task; to process the raw image provided by the camera.

Computation task; to implement the control algorithm so as to make a decision based on the processed image.

Actuation task; to send the signal that actuates the respective motor based on the decision from the computation task.

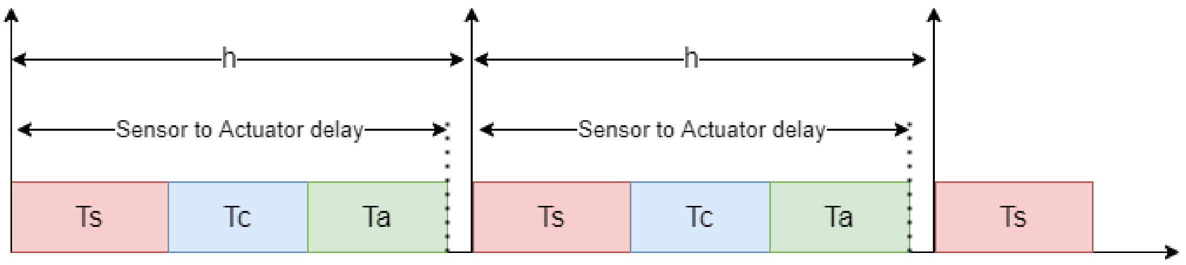


Figure 12. Sensor to Actuation delay

As shown in Figure 12, these three tasks are always performed sequentially and periodically with a sampling period of h. The total time taken to actuate a task after sensing is called sensor to actuator delay (τ). The sampling period is always greater than the sensor to actuator delay to make sure that that all the three tasks have been done before doing the next sensing task.

2.4.1 Lane Keep Assist System (LKAS)

Lane Keep Assist System (LKAS) is a type of IBC which is a driver assistance system that uses computer vision algorithms to notify the driver if he/she departs from the lane. A camera present in the car captures the image and processes these images to identify the lane markers present on the road. Typically, an edge detection algorithm like Sobel filter is used to perform feature extraction task. Once the lane markers were identified on the road, the disposition of the car from the middle of the lane to the lane markers are identified and if the deviation is too much, the driver of the car will be immediately be notified regarding the lane departure. Prior literature showed that approximation has benefits in the closed loop system. In most of the closed loop systems ISP acts as a bottleneck. So a runtime reconfigurable variable accuracy ISP accelerator was designed to overcome the bottleneck. This project focuses on implementing an ISP accelerator which can be used to study the benefit of approximations using a LKAS system.

2.5 Related Work

This project is mainly focused on implementing an accuracy configurable ISP on a Field Programmable Gate Array (FPGA) with an algorithmic approximation for a better QoC of the IBC system. Although there are some end-to-end case studies available on using the approximation for a closed loop system on software, these case studies on the hardware for a closed loop system are very limited. This project provides an implementation that can be used for a case study of applying three different approximations at a subsystem level and evaluating the performance implications in an IBC with a Hardware-in-Loop (HiL) setup.

2.5.1 Applications of Image based control (IBC)

The compute intensive part of image processing stage of IBC results in a large sampling period because of the design time choice. This has a negative impact on the QoC of the underlying system.

Instead of designing the controller for a Worst Case (WC) scenario, [5] proposes an IBC in which the sampling period can be dynamically reconfigured based on the workload variations using a scenario aware approach. Due to this sampling period of the controller adapts continuously based on the actual case scenario. Thereby resulting in the decrease of average sampling period with respect to the WC sampling period. However, this approach considers the ISP as a black box and assumes that there is a constant delay in the image preprocessing block.

2.5.2 Applications of Approximate Computing

Approximate Computing also known as inexact computing is a computing paradigm that will trade the computation time and energy for the accuracy of result in error resilient applications. Software techniques like bit precision scaling, subsampling, skipping the tasks and hardware techniques like decreasing the refresh rate, using the inexact hardware, clock over gating can be used to speed up the execution times of the task and at the same time reduce the overall energy consumption [6] [7] [8].

The three important approximations techniques that are addresses in this thesis are coarse-grain, fine-grain and subsampling approximations. As the name suggests, in the coarse-grained approximation some levels of data processing are skipped from processing. For example, if an image processing pipeline contains 10 stages, some of these stages can be skipped so that there is an improvement in energy consumption and latency. On the other hand, fine-grained approximation doesn't skip any stages but it reduces the precision of data that will be processed through the mentioned stages. For example, instead of using a 32 bit-length variable to hold the pixel value, a 24 bit-length variable can be used which reduces the number of clock cycles that is needed to perform any arithmetic operations. In subsampling approximation, the number of pixels that needs to be processed will be reduced by skipping some rows and columns of pixels in the image which results in better energy consumption. All these approximations result in less clock cycles to process at the cost of precision.

Since [5] considers the image processing block as a black box with a constant delay, [9] proposes an IBC system which extends the work of the [5] by taking the performance implication of the image processing into consideration. ISP is the main bottleneck in sensing task as the individual stages of ISP are computationally intensive. [9] proposed an approximation technique where some of the stages of ISP can be skipped during runtime using a scenario aware approach. By approximating ISP during the run time, workload of sensing task can be reduced which resulted in decrease of sampling period. A Software-in-Loop (SiL) simulator was implemented in [11] to show that QoC of the IBC improved because of approximated ISP. However only coarse-grained approximation with stage skipping has been considered. A fine-grained approximation with a variable data size could have improved the QoC further.

[10] proposes an end-to-end case study of approximate inference system for a deep neural network (DNN) based smart camera system. The impact of approximation and its implication on the performance of whole system was studied. This whole system includes four different sublevel approximations: approximate sensor, memory, computation and communication.

This project extends the work of [9] by implementing the ISP on FPGA and studying the performance implications of approximations. In addition to the stage skipping (coarse-grain approximation), this project enables the fine-grain and subsample approximation.

Chapter 3 Problem Statement

3.1 Motivation

As explained in Section 2.4, a typical IBC system includes sensing, computation and actuation tasks. It can be seen from Figure 13 that sensing task is comprised of processing raw images through an ISP and image processing block for lane detection. Due to the heavy workload of ISP when processing the images through the traditional ISP stages, the sensing task acts as a major bottleneck. Since the sampling period of the system should always be greater than the sensor to actuator delay, the increased time of sensing task has a direct impact on the sampling period which results in processing of lower number of frames per second (FPS). The low FPS rate results in the degradation of QoC. The possible performance gains by approximating the ISP can have a significant impact on sensing task which in turn reduces the sampling period and have a positive effect on the QoC.

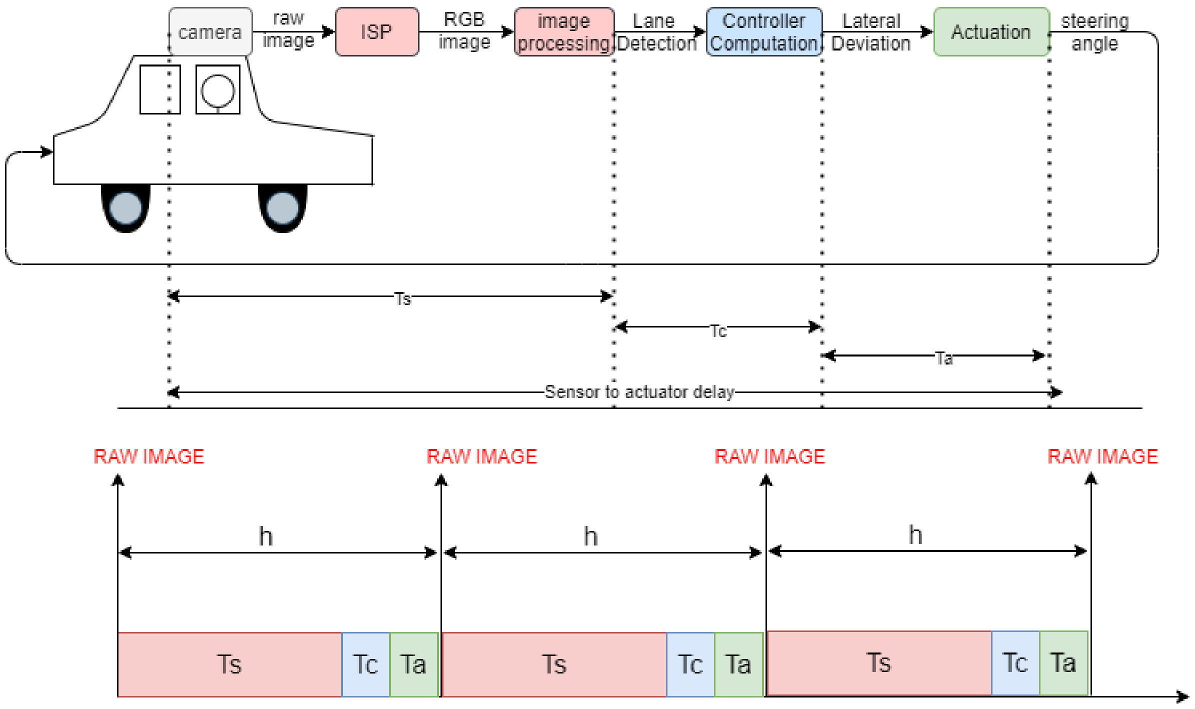


Figure 13. IBC of an LKAS

Since the main bottleneck is from the traditional ISP, this project focuses on implementing various algorithmic approximations on the imaging pipeline of an ISP.

3.2 Research Questions

This project addresses the following research questions:

- How to generate Hardware IP blocks for the ISP functionality in Vivado HLS tool?
- How the approximation techniques like coarse-grain, fine-grain and subsampling can be implemented on the ISP with runtime reconfigurability?

Chapter 4 Tool

4.1 Vivado HLS Tool

This project uses Vivado HLS provided by XILINX to implement the ISP functionality on the hardware. Vivado HLS can be used to generate the Verilog or VHDL code using high level languages like C, C++ and system C. Once the RTL code is synthesized from the high-level language, the functionality of the RTL is verified with respect to the algorithmic description. After the verification of RTL functionality, Intellectual Property (IP) cores are generated which has the desired functionality of the ISP. These IP logic blocks are then ported on to a field programmable logic array (FPGA).

The main advantages of using a Vivado HLS Tool over a traditional Hardware Description Language (HDL's) are:

1. Can develop the algorithm at C/C++ level which abstracts the implementation details at the Register Transfer Logic (RTL).
2. Can validate the functional correctness of the design logic much faster than a traditional HDL.
3. Can use certain directives or optimizations at C level to control the synthesis of the desired hardware implementation.
4. Can create multiple hardware implementations from the same C source code by changing the directives which helps in design space exploration.

In essence, it improves the productivity of the hardware designers as they can work at a higher level of abstraction and at the same improves system performance for software designers as they can target the FPGAs to accelerate the compute intensive algorithms.

4.2 Synthesizing the C/C++ to RTL

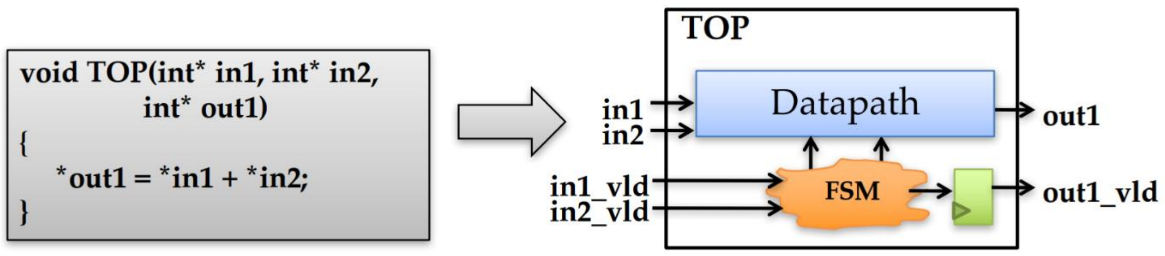
Vivado HLS doesn't allow a typical C implementation to be synthesized into RTL code. For a C implementation to be synthesized, it should not have the following:

1. Recursive functions: Recursive functions cannot be synthesized.
2. Dynamic memory allocation: Since the hardware implementation must specify the requirement before synthesizing, memory can neither be allocated nor released with functions like *malloc()* and *free()*.
3. System calls: system calls like *printf()*, *scanf()* cannot be synthesized as these function calls depends on the operating system (OS) in which the compiler is running.
4. Pointer casting: Although Vivado HLS allows pointer arrays, it does not allow the array of these pointers to point to additional pointers.

When the Vivado HLS friendly C implementation is given as input to the tool, The C code is synthesized as follows.

4.2.1 Functional Arguments

When the top-level function is synthesized in Vivado HLS, the functional arguments are synthesized into RTL ports. Vivado HLS allows to specify the type of I/O protocol for the functional arguments and this process is called interface synthesis [21]. These I/O protocols are used to sequence the data in and out of the block after the block has started processing the data. It can be observed that how the functional arguments (in1, in2, out 1) in C++ are converted into ports from Figure 14. In addition to the I/O some additional control ports (in1_vld, in2_vld, out_vld) will be added to the design. Figure 14 illustrates how the top-level function defined in C/C++ is synthesized into ports at RTL level.

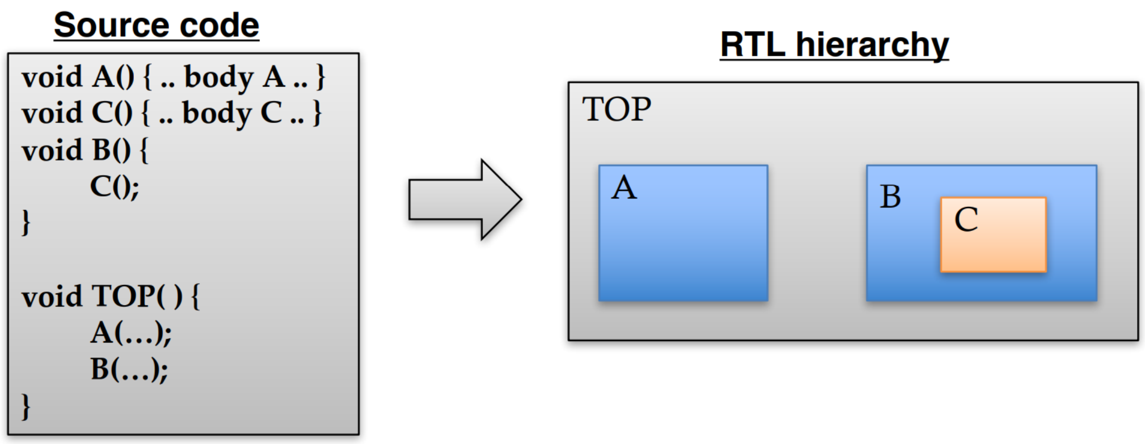


```
void TOP(int* in1, int* in2,
          int* out1)
{
   *out1 = *in1 + *in2;
}
```

Figure 14. Function arguments synthesized into I/O ports [25]

4.2.2 Functions and Function Hierarchy

All the C functions are synthesized into RTL modules. If there is a hierarchy of sub-functions the final RTL implementation follows this hierarchy unless the sub-functions are *inlined* to dissolve the hierarchy. A typical C/C++ function hierarchy synthesized into RTL modules is shown in the below Figure 15.



Source code

```
void A() { .. body A .. }
void C() { .. body C .. }
void B() {
        C();
}

void TOP( ) {
        A(...);
        B(...);
}
```

Figure 15. Function Hierarchy at RTL level [25]

4.2.3 Arrays

By default, an array in the C/C++ code is typically implemented as the memory blocks in the RTL. These memory blocks are usually Random Access Memories (RAM's). However if these arrays are implemented in the top-level function arguments, they will be implemented as the ports to access a BRAM outside the design. Figure 16 illustrates on how an array in C/C++ is synthesized into memory blocks.

Certain directives can be used to partition the given array and map into multiple RAM's or multiple arrays can be merged into one RAM. Depending on the application, an array can also be completely partitioned into individual elements and mapped to registers.

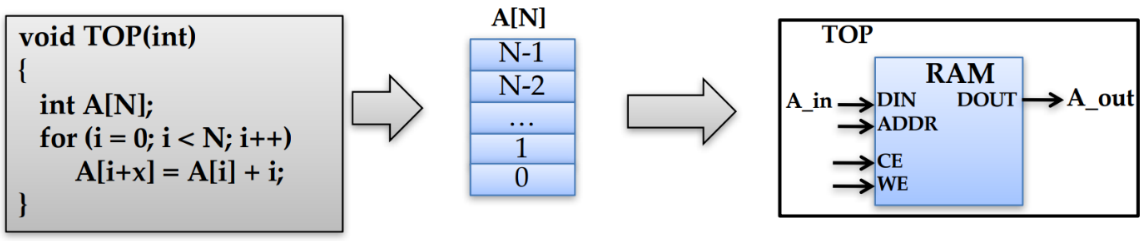

```
void TOP(int)
{
  int A[N];
  for (i = 0; i < N; i++)
     A[i+x] = A[i] + i;
}
```

Figure 16. Arrays synthesized as RAM's [25]

4.2.4 Loops

By default, loops in a C function are kept rolled which means that the synthesis creates the logic only for one iteration. This means the RTL design executes this logic for each iteration of the loop sequence. Having only one instance of the loop body results in all the loop iterations performed sequentially which can cause a substantial delay in the execution of the function. Some directives like loop unrolling which creates multiple instances or loop pipelining which allows the logic to have an initiation interval of II =1 can be used for better latency. More details on how the loops can be optimized using these techniques in Section 5.2.

4.3 Performance Metrics

Once the design is synthesized to see if the design requirements are met various performance metrics can be analyzed from the report generated by Vivado HLS. The following performance metrics can be analyzed from the synthesized report.

1. Latency: The number of clock cycles required for a function to generate all the output values.
2. Initiation interval (II): Number of clock cycles before a function or a block can accept a new set of inputs.
3. Area: The total amount of resources that are needed to implement the given function. This is usually made up of DSP48s, Registers, Look up tables (LUTs) and BRAMs. It gives a utilization % of how many resources are available on the FPGA board and how many of these resources has been utilized.

Chapter 5 Design Flow

5.1 Design Flow

This specific project uses the Halide implementation provided by Buckler in [11] to implement the C++ ISP. The design flow of this project can be viewed from Figure 17. The functionality of the ISP specified by the Halide pipeline has been converted to C++ and is given as an input to the Vivado HLS tool to synthesize the RTL. The test bench in C++ is created by using the data from the output images of the Halide pipeline.

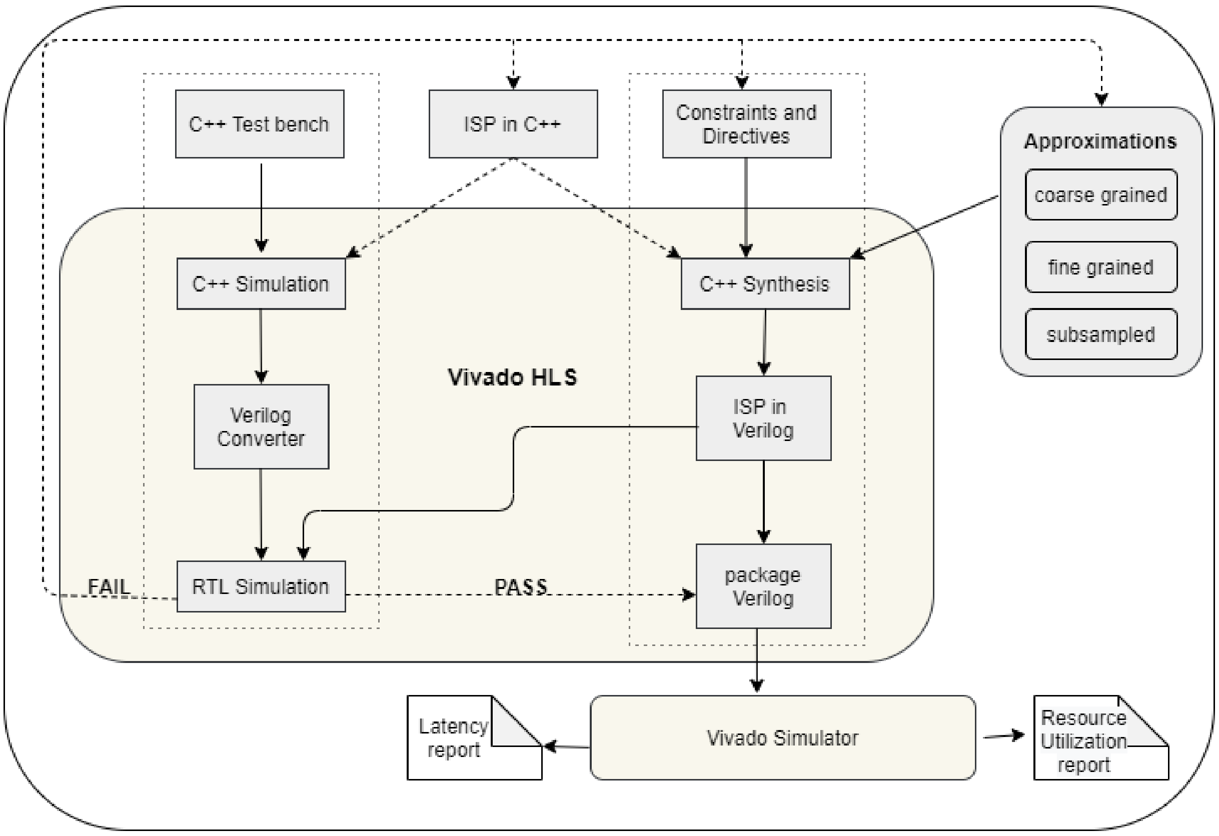


Figure 17. Design Flow

The first step in the design flow is to validate the C++ functionality. This is done to check, if the functionality specified by C++ is same as the functionality specified by Halide. This is done using the ”simulation” option provided by the tool.

After passing the C++ simulation, the next step is to synthesize the design to generate the RTL code. Apart from the ISP functionality of C++, some other directives/pragmas have to be mentioned that indicates what type of interface is required or how should the hardware be designed.

The third step in the design flow is to co-simulate the design. As the name suggests, two types of simulation happen at this stage. One is C++ simulation and the other is RTL simulation that is used to check if the functionality of RTL code is similar to that of C++ code. The C++ simulation at this step is used to generate the test vectors for the RTL simulation. If the co-simulation fails, it implies that the

RTL functionality is not similar to that C++ functionality. If the co-simulation passes, the RTL functionality has the desired behavior and can be preceded to the next step.

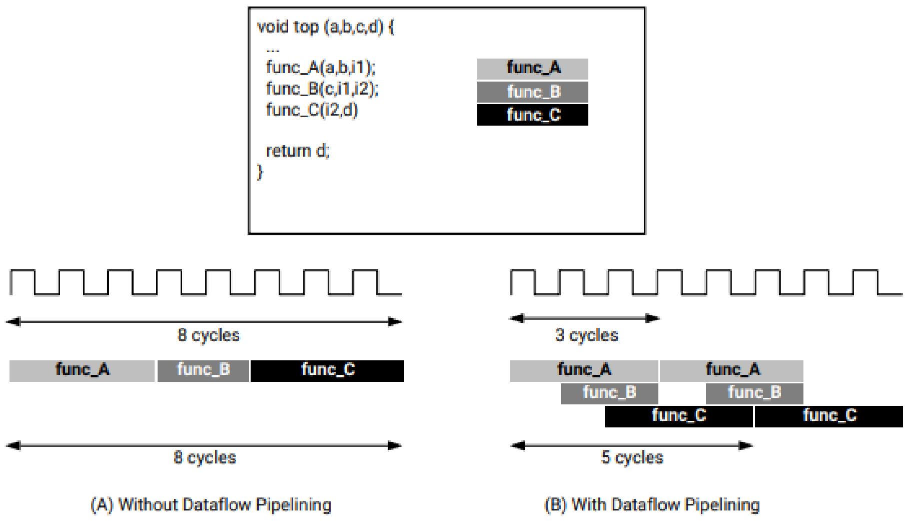
After the co-simulation, the next step is to export the generated RTL code to be used in Vivado for implementation on hardware. In this step, pin planning, placing and routing can be done before implementing on the hardware. The latency, utilization reports from this can be used to get an accurate number of cycles and the resources required to implement the ISP on an FPGA.

5.2 Optimization Techniques

To Improve the performance of the generated RTL in terms of latency, throughput or Area various directives can be added to the C code in the HLS tool which can have a direct impact on the RTL. In this section a brief explanation of the most commonly use optimization techniques and how they improve the performance is explained.

5.2.1 Pragma Dataflow

The dataflow optimization is an optimization technique which is used for task level pipelining. In the C code given to Vivado HLS let us consider there are three functions *func_A(), func_B() and func_A()* as shown in Figure 18. If the functions A, B and C takes 3,2 and 3 clock cycles to finish respectively and since these three functions are performed sequentially the latency is same as the throughput which is equal to 8 cycles. When the directive/pragma which specifies dataflow optimization is specified, Vivado HLS tries to find the data flow between these sequential tasks (functions or loops) and creates channels (like a FIFO) that allows the consumer loop or functions to start before the producer has completed.


```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Figure 18. Dataflow Pipeline [13]

5.2.2 Pragma Pipeline

As mentioned earlier, dataflow pipelining has a limitation of working at top level functions. For these top-level functions to have a better throughput and latency the subfunctions and the loops inside these functions should be optimized. Pipelining Is the most common technique used for optimizing the loops or subfunctions. A pipelined function or loop can process new inputs every X clock cycles where X is the initiation interval (II). A perfectly pipelined loop or function has N=1 which means it can accept input every clock cycle. Loop pipelining can be understood better from Figure 19.

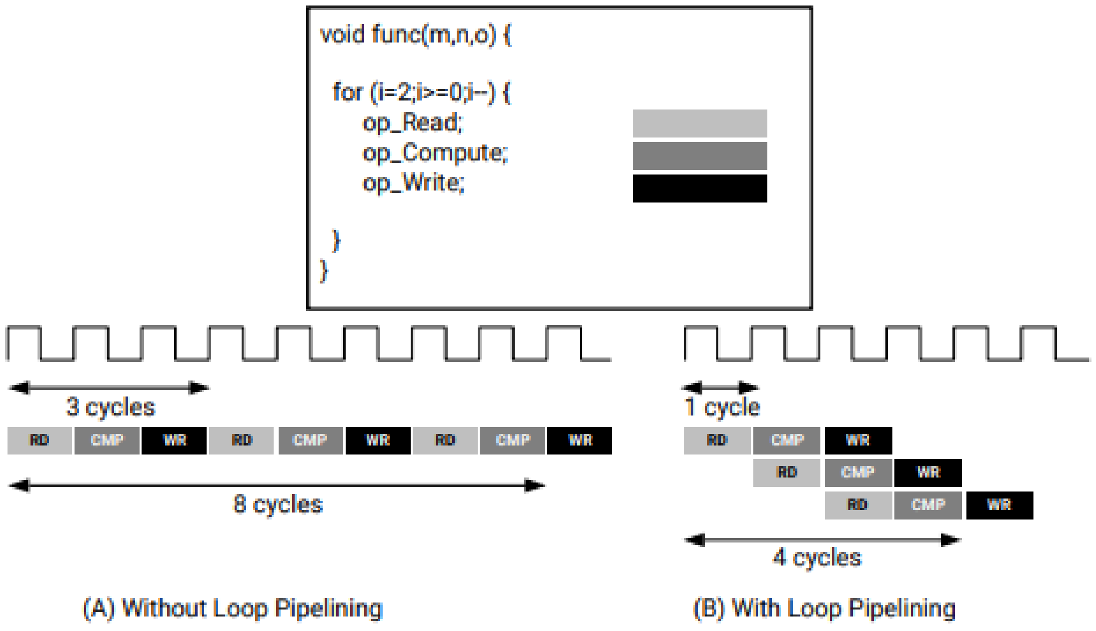


Figure 19. loop/Function pipeline [13]

Pipelining a loop or function allows the operations inside to be executed concurrently. In A it takes 3 cycles before a new read operation can be done which means the initiation interval (II) is 3 and needs 8 cycles to write the output. As the loop iteration count increases the latency of that loop increases and can get really higher to perform all these operations sequentially. Figure B shows using the loop pipelining concept where an initiation interval of II=1 can be achieved with a pipeline depth of 3.

5.2.3 Pragma Inline

Inlining a function is usually done when subfunction needs to be dissolved into the function that is calling the sub function. By dissolving a subfunction into the calling function, the hierarchy in the RTL implementation is completely eliminated. This allows operations within the function to be shared with the surrounding operations resulting in a more effective implementation. However, *inlinng* a function cannot be done if the said function is being called by more than one function.

5.2.4 Pragma Loop Unroll

As mentioned earlier, the loops in C/C++ are by default kept rolled when it is synthesized for RTL implementation which means the RTL logic is created only for one iteration of the loop. If the loop is not unrolled all the iterations in the loop will be carried out sequentially which results in high latencies for the loop. By using the *unroll* pragma more copies of the loop body can be produced and the loop iterations can be executed concurrently. Loop unrolling can be full unrolling or partial unrolling with a factor of N where N is the number of copies of the loop body that is needed in the RTL implementation for concurrent execution. This allows an increase in data access and a better throughput. If there is a nested loop and the outer loop has pipeline pragma specified, for a better initiation interval the inner loops will automatically be unrolled full factor (i.e. the number of loop iterations).

5.2.5 Pragma Loop flatten

Loop flatten is a technique that can be used to dissolve the nested loops into a single loop hierarchy. RTL implementation requires one clock cycle to move from an outer loop to an inner loop and one more clock cycle to move from an inner loop to outer loop. By flattening these nested loops, it allows them to be under a single loop and saves the clock cycles that needs to jump between the loops.

However, Loop flatten can only be implemented if:

1. There is no logic specified between the loop statements.
2. Outer loops should not have a body content.
3. Inner loops cannot have variable bounds.

5.2.6 Pragma loop trip count

In some cases, the number of iterations loops will not be constant. It rather depends on the value which is computed previously. When a loop with variable bounds is given to Vivado HLS to synthesize, the tool cannot give out the latency needed to compute the loop as there are no fixed bounds which specifies how many times the iteration will happen in RTL. To overcome this issue we can use pragma "Loop_Trip_count" which specifies what can be the maximum, minimum and the average number of times a loop will be executed. This allows the tool to analyze how loop latency contributes to the overall latency and performs appropriate optimizations. Using this pragma has no impact on synthesis, this is purely used for reporting purposes

5.2.7 Array optimizations

All the above-mentioned pragmas or the optimization techniques are used to have a better latency and throughput for the loops and functions. However there can be bottlenecks while trying to implement these techniques. One of the major bottle necks that does not allow for a better throughput are the arrays that are implemented inside these loops or functions. The arrays that are defined in the C in Vivado HLS are implemented as memories in the RTL implementation and is typically implemented as random access Memory (RAM) if it needs both read and write operations performed on it or a read only memory (ROM) if it is required to only read from the array. Some of the most frequent used Array optimization techniques are given in the next sections:

5.2.8 Array partition

When arrays defined in Vivado HLS are synthesized, they will be mapped into one big memory blocks with only two ports for reading and writing. There can be a bottle neck while doing a multiple read/write access because of the limited number of ports of the big memory. Array partition can be applied on this big array to partition the array into smaller arrays or individual elements. This optimization results in RTL with multiple smaller memories or multiple registers instead of one large memory. Vivado HLS provides various types of array/memory partition techniques. One of these techniques is array partition complete which is used to patriation the entire array into individual elements as shown in Figure 20. This improves the throughput of the design as the number of read and write ports are increased but increases the area as it needs more memory instances.

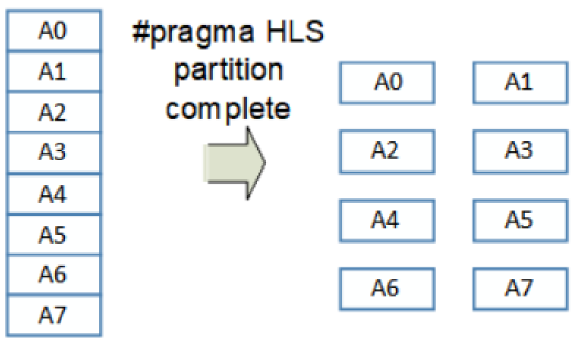


Figure 20. Pragma Array partition

5.2.9 Array Reshape

Array Reshape is a type of technique that can be used in combination with array partition where the parallelism of data access can be maintained in by concatenating the elements of smaller arrays. This is a vertical type of array mapping where the number of BRAM's are reduced and at the same time provide access to more data in one cycle. The Array Reshape optimization can be done in 3 modes as shown in Figure 21.

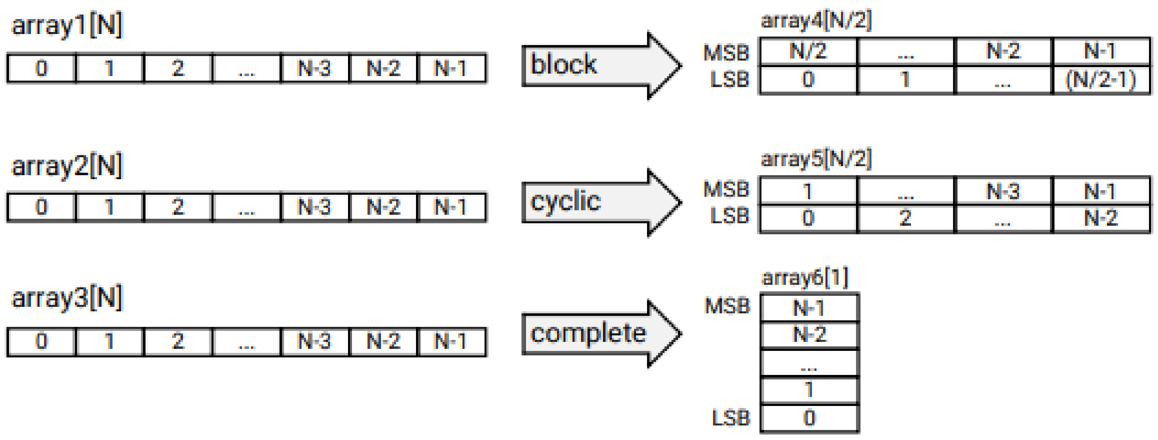


Figure 21. Array Reshape [13]

5.2.10 HLS Datatypes

All the native datatypes which are Supported by C are supported by Vivado HLS and can be synthesize to generate the RTL implementation. These include:

1. Signed integer types: signed char, short, int, long.
2. Unsigned integer types: unsigned char, unsigned int, unsigned long and unsigned short.
3. Floating point types: float, double, long double.
4. Bool type.

Float and double are the only datatypes provided by the native datatypes if high precision is needed. But sometimes this precision might not be enough or a datatype with more precision than an int and less precision than a float might be needed. To facilitate this Vivado HLS allows the use of arbitrary precision datatypes for both C and C++ where bit width of the datatype can be user defined.

In addition to the arbitrary precision types, Vivado HLS allows arbitrary precision fixed point types that allows fractional arithmetic to be easily handled.

Vivado HLS has a library for fixed point types called *"ap_fixed.h"* and can be used to define a fixed-point variable as:

ap_[u]fixed <W, I, Q, O> where

ap_ufixed: unsigned fixed-point type

ap_fixed: signed fixed point type

W: total word length

I: Integer word length

Q: Quantization mode

O: Overflow mode

Word length is the total number of bits the fixed-point datatype has and Integer word length is the number bits that is needed to hold the integer part before the decimal point as shown in Figure 22. Quantization modes and Overflow mode arguments are used to define how to store the data in case of an overflow or underflow.

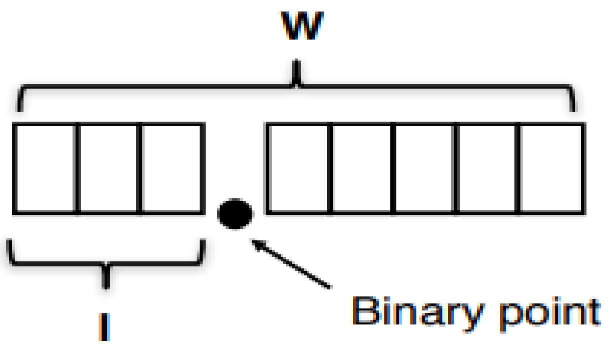


Figure 22. Fixed point data type integer and fractional part [25]

Chapter 6 Implementation and Experimentation

6.1 Baseline Implementation

In order to study the impact of algorithmic approximations in an ISP, a baseline implementation which includes all the stages of ISP has been developed. The functionality of ISP in [13] was developed in C++ and passed to Vivado HLS. Initially each stage was separately implemented and the IP cores (logic blocks which has the functionality of each stage) were developed to check if the functionality requirements were met. Once all the functionality requirements were met for each stage, all the stages were combined together to generate the complete ISP.

6.2 Architecture

The architecture of the proposed ISP that can be implemented on the hardware is given in Figure 23.

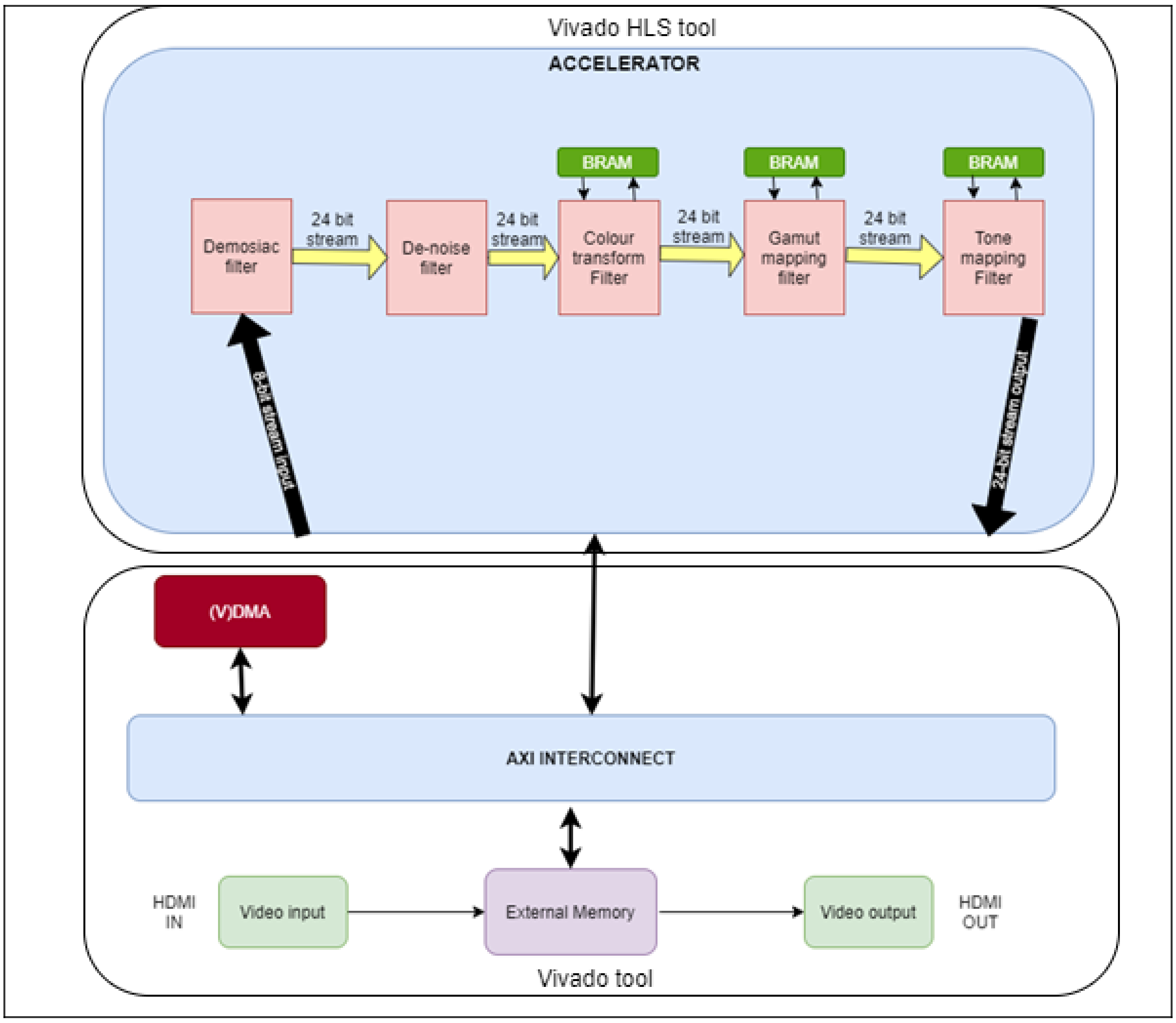


Figure 23. Architecture of ISP

The architecture of the baseline implementation consists of:

1. Video input, where the raw images of 8 bit per pixel are received from the camera sensor.

2. An AXI interconnect which is used to establish the bus connections between all the components on the board like external memory, VDMA.
3. Video dynamic memory access (VDMA) which is a memory map to stream interface and vice versa. This is used to send and receive the images as a stream of data.
4. An accelerator where all the stages are implemented.
5. Three Block Random Access Memory (BRAM’s) interfaces to store the weights used by color
6. transformation, gamut mapping and tone mapping stages.
7. Video output, where the processed images with 24 bits per pixel are sent to the image compression block.
8. An external memory for the raw images to be read and for the compressed images to be stored.

The main focus of this project will be the accelerator where all the ISP stages are implemented. As shown in Figure 23, the accelerator for the ISP can be implemented using the Vivado HLS tool and the IP thus generated can be used in the Vivado tool to make connections with the other blocks like VDMA and external memory.

The scope of this thesis is limited to Vivado HLS tool where end to end functionality of ISP is verified with respect to the RTL implementation, and obtaining the performance metrics like latency and area.

6.3 Design Choice and Initial Results

6.3.1 Design Choice

The floating-point multiplications and divisions are very expensive in terms of area and latency when implemented on hardware as one float multiplication requires 3 DSP’s. This results in an increase of both clock cycles and utilized hardware resources. Even though it is very expensive, the floating-point representation is implemented as the baseline implementation. When optimizing the ISP, the floating-point representation can be changed to fixed point representation, which will decrease both the latency and clock cycles.

6.3.2 Initial Profiling

An initial profiling was done on all the stages of ISP to get a rough estimate of the number of hardware resources needed to implement the ISP and the amount of clock cycles needed to process the image through each stage of ISP. The performance estimates of implementing the ISP on **Zynq UltraScale+ ZCU106 Evaluation platform** with a clock frequency of **100 MHz** are shown in Table 1.

ISP Stage	BRAM's	DSP's	Flip Flop's	LUT's	Latency (Clock cycles)	Latency (ms)
Demosaic (DM)	6	49	4468	9058	5437610	54
Denoise (DN)	4	0	1047	4142	19005954	190
Color-Transformation (CT)	0	24	3728	7765	4456961	44
Gamut Mapping (GM)	4	15	3887	5573	777126401	7777
Tone Mapping (TM)	6	15	2916	5932	436995331	4369
Total	20	103	20661	31314	1243019257	12430
Available Resources	624	1728	460800	230400	-	
Utilization (%)	~ 5	~8	~4	~13	-	

Table 1. initial Profiling

It can be seen from the above table that resource utilization is very low when compared to the resources on the FPGA. Also, for some of the stages clock cycles are too high. The goal of the next section is to reduce the number of clock cycles by increasing the resource utilization

6.4 Optimizing the accurate hardware

After generating the accurate hardware with the required functionality, the next step was to optimize it in an FPGA friendly manner. As it can be seen from the profiling section, the time taken to process one raw image through all the stages is too high. An ideal ISP should process images at a faster rate to meet the Frames Per Second (FPS) requirement of 20-30. Various optimizations like function pipelining, loop unrolling and array partitioning which are explained in Section 5.2 [13] can be done to decrease the latency by decreasing total number of clock cycles.

In this section each ISP stage will be synthesized and various type of RTL implementations will be generated for each stage with different set of pragmas and datatypes and a comparison can be made among these implementations. A detailed explanation will be done on the synthesis of one of the stages (denoising) of the ISP.

6.5 Optimizing the Denoising Stage

When an image is captured, there will some noise added to the image. There are various types of noises that can be present on the image like gaussian noise or salt and pepper noise. Various denoise filters can be used to remove these noises such as mean filter (local and non-local) or a median filter. As mentioned in Section 2.3.2, the main advantage of the median filter Is that it preserves the edges [26]. Since a Sobel type of filter is used in post processing for the lane extraction and preserving the edges in the image helps with lane detection. Hence, this project uses a median filter to denoise the image.

In the median filtering to calculate the output pixel the surrounding pixels of the input pixel are taken into consideration and the median of all these pixels is calculated and given as the value for output pixel.

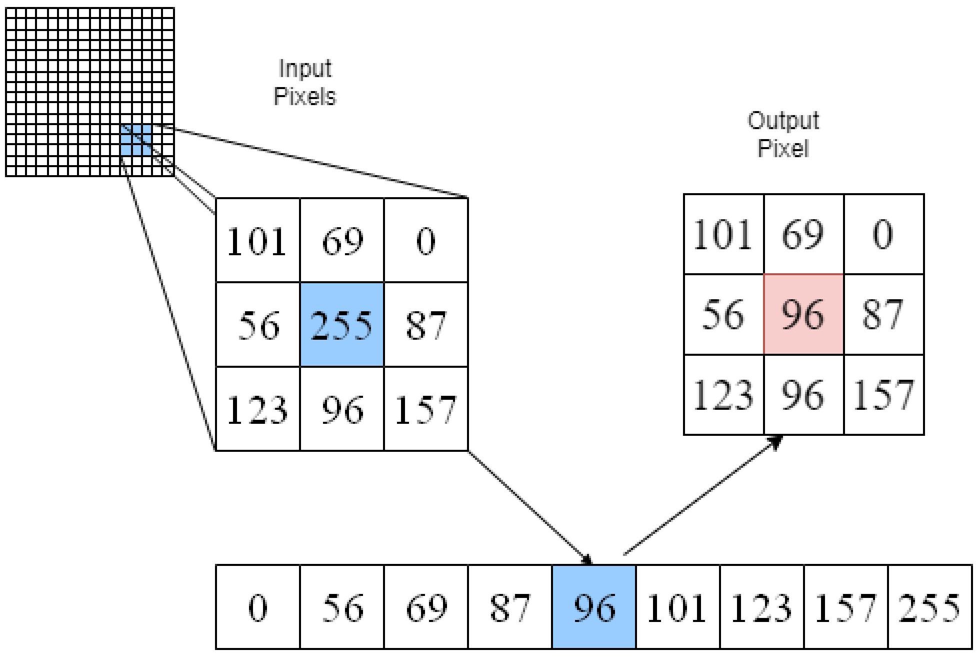


Figure 24. Median filtering for Denoising

6.5.1 Optimizing with Line and Window Buffers

It can be seen from the above Figure 24 that to compute one pixel, a window (3x3) of 9 pixels is needed and then only the output pixel can be computed by finding the median. This means that to compute one pixel 9 reads has to be done before any computation takes place which results in increased latency. To overcome this issue, the first optimization was the code to be restructured in such a way that the concepts of *line buffer* and *window buffer* can be used. A *line buffer* is a 2D array which is used to store several rows of input image. The number of rows that will be cached depends on the height of the kernel/window. On the other hand, *window buffer* which is of the same size of window is used to cache values in the current window and are used to access these values simultaneously in one clock cycle. The C++ code implementation for the use of line and window buffers is given below.

```
for (int row = 0; row < Image_height row++) {

    for (int col = 0; col < Image_width; col++) {

        for(int i = 0; i < 3; i++) {

            window[i][0] = window[i][1];
            window[i][1] = window[i][2];
            }
        window[0][2] = (line_buffer[0][col]);
        window[1][2] = (line_buffer[0][col] = line_buffer[1][col]);
        window[2][2] = (line_buffer[1][col] = pixel_in[row][col]);
        if (row == 0 || col == 0 ||
        row == (MAX HEIGHT - 1) ||
        col == (MAX WIDTH - 1))
        {
        pixel_out[row][col].r = 0;
        pixel_out[row][col].g = 0;
        pixel_out[row][col].b = 0;
            }
        else
        {
            pixel_out[row][col] = median_filter(window);
        }
    }
}
```
Since the adjacent windows overlap more often while calculating the output of a pixel, it implies a high locality of reference. This means that the input pixels can be buffered locally and cached to be used multiple times rather than reading the pixel every time which is costly. By rewriting the C++ code to read each pixel exactly once and storing in the local memory. A pictorial representation of how the line buffer and the window buffer works on an image is shown in below Figure 24Figure 25. It can be seen how a portion of the image is stored locally into both the window and line buffers for each iteration of the loop.

In Figure 25, the pixels outlined in black are stored in the line buffer and the pixels outlined in red are stored in the window buffer. The input image is read into the line buffer pixel by pixel. Every time, when the loop is executed, the window is shifted and filled with one pixel from the input and two pixels from the line buffer. In addition to that, the input pixel is also shifted into the line buffer so that the process can be repeated on the next line.

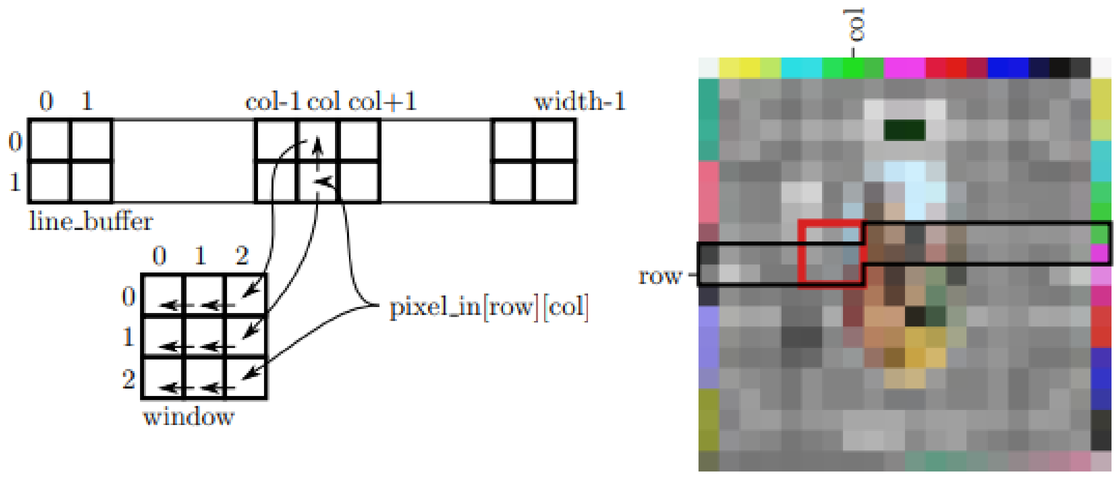


Figure 25. using Line and window buffer to process an image [27]

The performance metrics of denoising stage with the use of line and window buffer is given in the below Table 2

Performance Metrics	**Denoise with Line Buffer and window Buffer (Floating point/Naive)**
Clock frequency (MHz)	100
Clock Cycles	19005954
Latency (time in ms)	190
BRAM's	4
DSP's	0
LUT's	4142
Flipflops	1047

Table 2. Denoise Naive/Line Buffer Profiling

It can be observed from the above table that the DSP's used are 0. This is because median filter is a sorting filter which just compares the pixel values with the other pixel values and gives the output. However other stages will have some arithmetic calculations int the functionality where we can see the usage of DSP's.

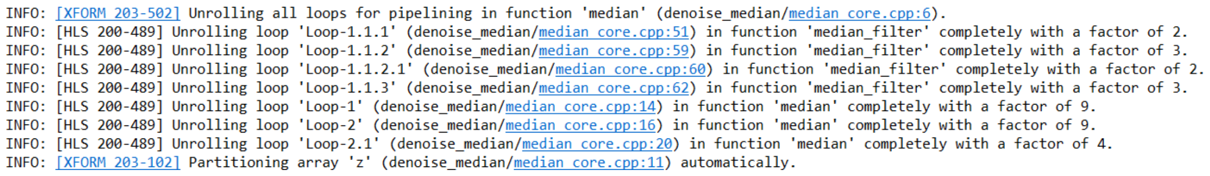
6.5.2 Optimizing for an Initiation Interval of II = 1

The next step was to pipeline the subfunction which calculates the median of the window to achieve an initiation interval of II =1. This can be given to the C++ code by simply specifying the pipeline pragma as shown below:

```
for (int row = 0; row < Image_height row++) {

    for (int col = 0; col < Image_width; col++) {
        #pragma HLS pipeline II=1
```

The benefit of using Vivado HLS is when specify a pipelining is that it automatically applies the optimizations of loop unrolling or flattening or some other optimizations mentioned in the previous sections to get the target initiation interval. All these optimizations that are performed can be seen from console window while the RTL is being synthesized as shown in Figure 26.



```
INFO: [XFORM 203-502] Unrolling all loops for pipelining in function 'median' (denoise_median/median_core.cpp:6).
INFO: [HLS 200-489] Unrolling loop 'Loop-1.1.1' (denoise_median/median_core.cpp:51) in function 'median_filter' completely with a factor of 2.
INFO: [HLS 200-489] Unrolling loop 'Loop-1.1.2' (denoise_median/median_core.cpp:59) in function 'median_filter' completely with a factor of 3.
INFO: [HLS 200-489] Unrolling loop 'Loop-1.1.2.1' (denoise_median/median_core.cpp:60) in function 'median_filter' completely with a factor of 2.
INFO: [HLS 200-489] Unrolling loop 'Loop-1.1.3' (denoise_median/median_core.cpp:62) in function 'median_filter' completely with a factor of 3.
INFO: [HLS 200-489] Unrolling loop 'Loop-1' (denoise_median/median_core.cpp:14) in function 'median' completely with a factor of 9.
INFO: [HLS 200-489] Unrolling loop 'Loop-2' (denoise_median/median_core.cpp:16) in function 'median' completely with a factor of 9.
INFO: [HLS 200-489] Unrolling loop 'Loop-2.1' (denoise_median/median_core.cpp:20) in function 'median' completely with a factor of 4.
INFO: [XFORM 203-102] Partitioning array 'z' (denoise_median/median_core.cpp:11) automatically.
```

Figure 26. Automatic Optimizations by Vivado HLS compiler

If the target initiation interval is not met it will give a warning on why target was not reached, what was the bottleneck and what can be done to achieve the target initiation interval as shown below.

```
WARNING: [SCHED 204-69] Unable to schedule 'store' operation
('line_buffer_addr_1_write_ln53', denoise_median/median_core.cpp:53) of
variable 'window[2]',
denoise_median/median_core.cpp:53 on array 'line_buffer' due to limited
memory ports.
Please consider using a memory core with more ports or partitioning the
array 'line_buffer'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 3,
Depth = 23.
INFO: [SCHED 204-11] Finished scheduling.
```

6.5.3 Optimizing by Array Partition

The initiation interval of II=1 is not achieved because there were more read accesses that was done on line buffer than it has ports in one clock cycle. Following the suggestions provided in the console, array partition optimization can be applied on the *line_buffer* so that it can have more read/write ports. This can be used by specifying the pragma as

```
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1
```

When the C++ code is synthesized to generate the RTL implementation with mentioned pragmas the synthesis results shows that pipelining with the initiation interval of II=1 is successful and the following message will be displayed on the console window:

```
INFO: [HLS 200-10] ----------------------------------------------------
----------
INFO: [HLS 200-42] -- Implementing module 'median_filter'
INFO: [HLS 200-10] ----------------------------------------------------
----------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'L2'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1,
Depth = 22.
INFO: [SCHED 204-11] Finished scheduling.
```

The performance metrics from the new synthesis report are given in the below Table 3.

Performance Metrics	**Naïve implementation**	**Floating point implementation with Optimization directives**
Clock frequency (MHz)	100	100
Clock Cycles	19005954	136962
Latency (time in ms)	190	1.37
BRAM's	4	2
DSP's	0	0
LUT's	4142	6996
Flipflops	1047	11448

Table 3. performance metrics of Naïve vs floating point with optimizations

6.5.4 Floating to Fixed point conversion

Since the denoise median filter has no arithmetic calculation, the datatype that can be used to store the data can be a *char* which has a size of 8 bits and range of (0-255) as the pixel values range from 0-255. However the arithmetic calculations in other stages of ISP are done with fractional precision where the pixel values are converted from (0-1) range by dividing each pixel with 256. Instead of adding an additional overhead with scaling and descaling which requires multiplication and divisions just for the denoise stage, it was found that using the same datatype gives better performance. Hence a fixed-point type with a bit size of 18 is used and the performance implications are shown in the below Table 4.

Performance Metrics	Naïve Implementation (Floating point)	Floating point + optimization directives	Fixed point + Optimization directives
Clock frequency (MHz)	100	100	100
Clock Cycles	19005954	136962	134658
Latency (time in ms)	190	1.37	1.34
BRAM's	4	2	2
DSP's	0	0	0
LUT's	4142	6996	1063
Flipflops	1047	11448	1164

Table 4. Performance metrics comparison for the fixed-point implementation

It can be observed from Table 4 that because of changing the datatype from 32-bit floating-point representation to 18-bit fixed point representation the number of LUT's and FF's came down by 6,000 and 10,000 approximately without having a negative impact on clock cycles. There is no significant improvement in the clock cycles or latency because the initiation interval of 1 is already achieved with floating point implementation.

6.5.5 Clock Frequency

The RTL implementations so far synthesized have the default clock period at 100 MHz which means each clock cycle takes (1/100Mhz) = 10 nanoseconds (ns). In timing analysis there are two types of slack (positive and negative) that are associated with clock frequencies. Positive slack means that the signal can traverse through the combinational logic from the start point to end point timing path in time and operate correctly. On the other hand Negative slack means the data signal is unable to traverse the combinational in time to ensure the correct operation.

Slack = Target time – Arrival time

It is important that the slack is always positive to ensure the correct operation of the circuit. For all the stages of ISP clock frequency was increased till the slack was positive and it was found that at a clock frequency of 250 MHz the slack remains positive and going beyond that is resulting in a negative slack for some of the stages. Hence the clock frequency for all the stages was set to 250 MHz as all these stages need the same clock and no stage should have a negative slack. By Changing the clock frequency to 250 MHz the time required for one clock cycle changed from 10 ns to 4 ns (1/250 MHz). Since each clock cycle takes less time, the overall latency decreased from 1.39ms to 0.539ms.

The comparison of all the above-mentioned implementations including the one with better clock frequency is given in below Table 5.

Performance Metrics	**Naïve Implementation (Floating point)**	**Floating point + optimization directives**	**Fixed point + Optimization directives**	**Fixed point + optimization directives + better clock**
Clock frequency (MHz)	100	100	100	250
Clock Cycles	19005954	136962	134658	134568
Latency (time in ms)	190	1.37	1.34	0.539
BRAM's	4	2	2	2
DSP's	0	0	0	0
LUT's	4142	6996	1063	1063
Flipflops	1047	11448	1164	1164

Table 5. Performance metric comparison of various denoise implementations

6.6 Optimized Results of ISP Stages

The similar design approaches are used for optimizing the other stages of Demosaicing, Color Transformation, Gamut Transform and Tone Mapping. For every stage after getting an accurate result the same flow of using the pragma for pipeline with initiation interval of II=1 is used and based on the issues that caused the II=1 to not be met, different optimizations mentioned in the Section 5.2 were used and at the end every stage had an initiation interval of II=1. The final clock cycles for each stage are similar as every stage had the function pipelining with II=1. However the area (BRAM's, LUT's, DSP's and FF's) were different for each stage as some of these stages had some complex arithmetic calculations. For the rest of the stages, the naïve implementation was converted into fixed point representation and optimization directives were applied with a higher clock frequency. The results of naïve implementation and the final optimized implementation for each of these ISP stages are tabulated in the following sub sections.

6.6.1 Demosaic Results

The results of demosaicing baseline vs optimized implementations are given in the below Table 6. Unlike denoise implementation, demosaicing implementation needs DSP's as it requires some arithmetic computations like multiplications and additions.

Performance metrics	**Naïve implementation**	**Optimized implementation**
Clock Frequency (MHz)	100	250
Clock Cycles	4750585	135740
Latency	47.5	0.543
BRAM's	6	2
DSP's	22	11
LUT's	4252	3526
FF's	2658	2562

Table 6. Performance comparison of naive vs optimized implementation for Demosaic stage

For Demosaicing, the latency came down from 47 milli seconds to 0.543 milliseconds after optimizations which achieved a speedup of 82x.

6.6.2 Color Transformation Results

The results of color transformation baseline (naive) vs optimized (better clock) implementations are given in below Table 7.

Performance metrics	Naïve implementation	Optimized implementation
Clock Frequency (MHz)	100	250
Clock Cycles	4456961	135740
Latency	44.5	0.553
BRAM's	0	2
DSP's	24	9
LUT's	7765	6751
FF's	3728	8576

Table 7. Performance comparison of naive vs optimized implementation for Color Transformation stage

For color transformation, the latency came down from 44.57ms to 0.55ms after optimization and a speedup of 80x was achieved.

6.6.3 Gamut Mapping result

The results of color transformation baseline (naive) vs optimized (better clock) implementations are given in below Table 8.

Performance metrics	Naïve implementation	Optimized implementation
Clock Frequency (MHz)	100	250
Clock Cycles	777126423	393354
Latency (ms)	7771	1.573
BRAM's	4	4
DSP's	15	62
LUT's	8183	62480
FF's	4904	92106

Table 8. Performance comparison of naive vs optimized implementation for Gamut Mapping stage

For gamut mapping, the latency decreased from 7.7 seconds to 1.5 milli seconds after optimization which can be considered as huge speedup. In gamut mapping each pixel is processed through an inner most loop with a bound of 3708, before optimization this loop was executed in sequential manner which led to a huge increase in latency.

6.6.4 Tone Mapping Results

The results of Tone Mapping baseline (naive) vs optimized (better clock) implementations are given in below Table 9.

Performance metrics	Naïve implementation	Optimized implementation
Clock Frequency (MHz)	100	250
Clock Cycles	436995331	131743
Latency (ms)	4370	0.527
BRAM's	6	4
DSP's	33	15
LUT's	5932	60994
FF's	2916	181869

Table 9. Performance comparison of naive vs optimized implementation for Tone Mapping stage

Similarly in tone mapping, the latency speedup is also in range of 1000's because of more inner loops compared to demosaic, denoise and color transformation. After optimization, the Tone mapping takes 0.527ms. since all the inner loops that were being executed in sequential manner were actually comparing the pixel values with predefined weight, to achieve an initiation interval of 1 complete array partition is done which resulted in the elements of arrays being synthesized as individual registers. This explains the high increase in the LUT's and FF's and decrease of BRAM's.

The speedups achieved from a naïve implementation to optimized implementation is very high because in the naïve implementation the functionality is executed sequentially. By using of the pragmas and optimization directives more instance of the logic can be created which allows the concurrent execution of the function logic. For gamut mapping and tone mapping, it can be seen that the speedup achieved is in terms of 1000's, this is because of these two stages having inner loops with higher trip count compared to the other stages.

6.7 Approximations

As mentioned in Section 3.1, IBC can be benefit from approximating the ISP. The total latency of the ISP is 3.753ms (0.543 (DM) + 0.539 (DN) + 0.553 (CT) + 1.573 (GM) + 0.527 (TM)). This latency can be further reduced with various approximations like coarse-grain, fine-grain and subsampling are used to approximate the ISP. Each of these approximations are explained in detail in the following subsections

6.7.1 Coarse-grain approximation

The concept of coarse-grained approximation with respect to this project is to skip some stages of ISP during the runtime. To implement this functionality, the architecture of the ISP will have some minor modifications.

The new additions to the architecture are:

1. Configuration bits, which are used to decide which stages can be skipped at the run time.
2. A pass-through block for each stage which is used to skip stages during run time.

As the name suggests, the pass-through block allows image data to pass through it. It takes input from one of the stages and gives it to the next stage without modifying data. A pass-through block can be a FIFO with a depth of 1.

The architecture of ISP with stage skipping is explained in Figure 27. It can be seen from the figure that there is no pass-through block for demosaicing stage. Literature survey showed that skipping demosaicing stage cause the LKAS to fail resulting in the vehicular crash [9]. Since the QoC of LKAS will be highly sensitive to the demosaicing stage, this stage can never be skipped.

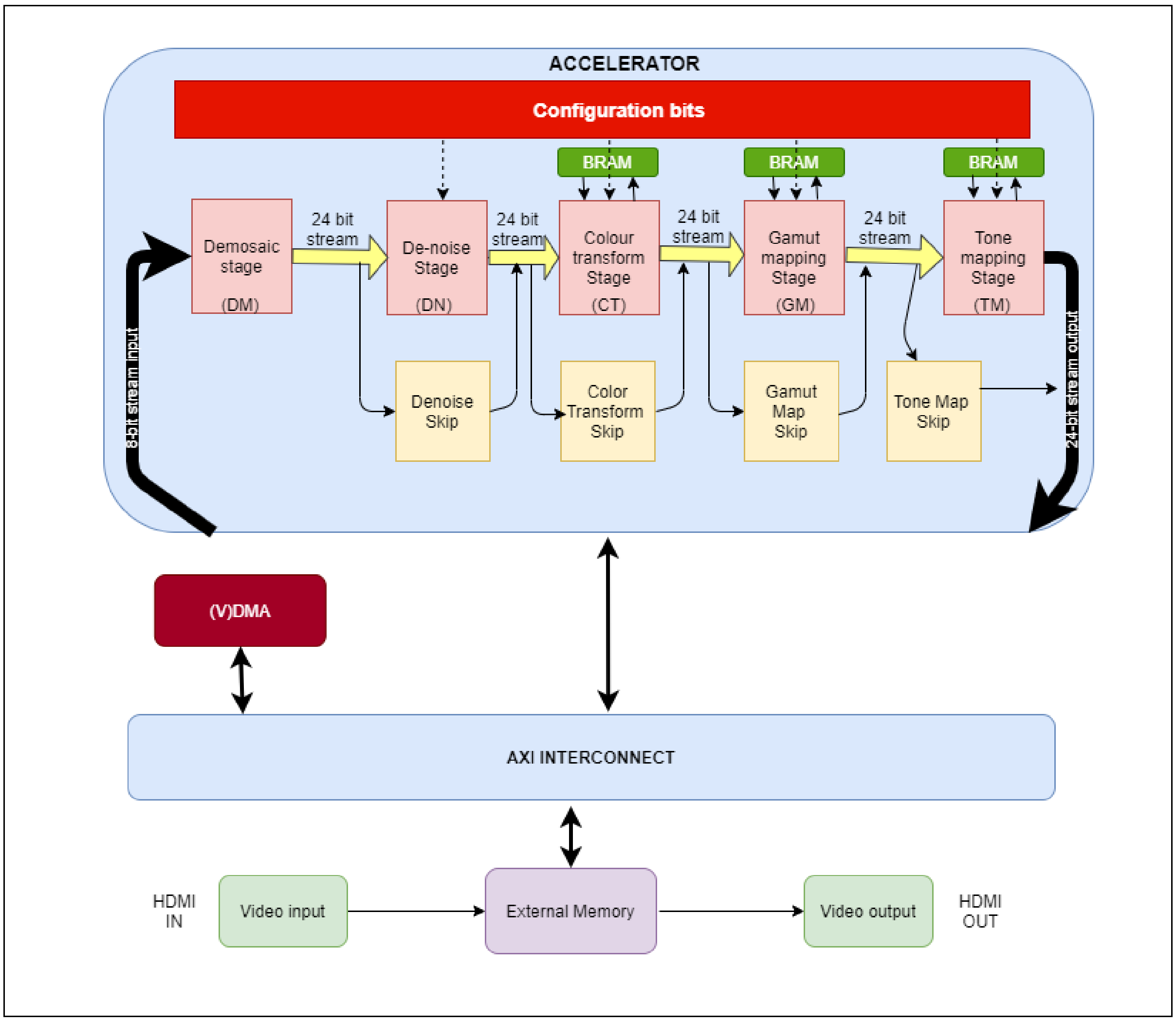


Figure 27. Coarse-grain Architecture

Since there are only four stages that can be skipped, a total of 16 different configurations will be implemented in the coarse-grained approximation as presented in Table 10.

Version	ISP Stages	Explanation	Configuration
V0	DM, DN, CT, GM, TM	No stages skipped	1111
V1	DM, DN, CT, GM	Skip tone mapping	1110
V2	DM, DN, CT, TM	Skip gamut mapping	1101
V3	DM, DN, CT	Skip gamut and tone mapping	1100
V4	DM, DN, GM, TM	Skip Color transformation	1011
V5	DM, DN, GM	Skip color transformation and tone mapping	1010
V6	DM, DN, TM	Skip color transformation and gamut mapping	1001
V7	DM, DN	Keep Demosaicing and denoising	1000
V8	DM, CT, GM, TM	Skip Denoising	0111
V9	DM, CT, GM	Skip denoising and tone mapping	0110
V10	DM, CT, TM	Skip denoising and gamut mapping	0101
V11	DM, CT	Keep Demosaicing and color transformation	0100
V12	DM, GM, TM	Skip denoising and color transformation	0011
V13	DM, GM	Keep Demosaicing and gamut mapping	0010
V14	DM, TM	Keep Demosaicing and tone mapping	0001
V15	DM	Keep only Demosaicing	0000

Table 10. Different run time configurations for coarse grained approximation

The hardware implementation was implemented in such a way that these versions can be dynamically reconfigured during the run time. All the stages of ISP in coarse-grain work sequentially, i.e. when the raw image is given to the ISP, the latency for each configuration will essentially be sum of the time taken for each stage in that configuration. The images produced from each version mentioned in the table are given below in Figure 28.

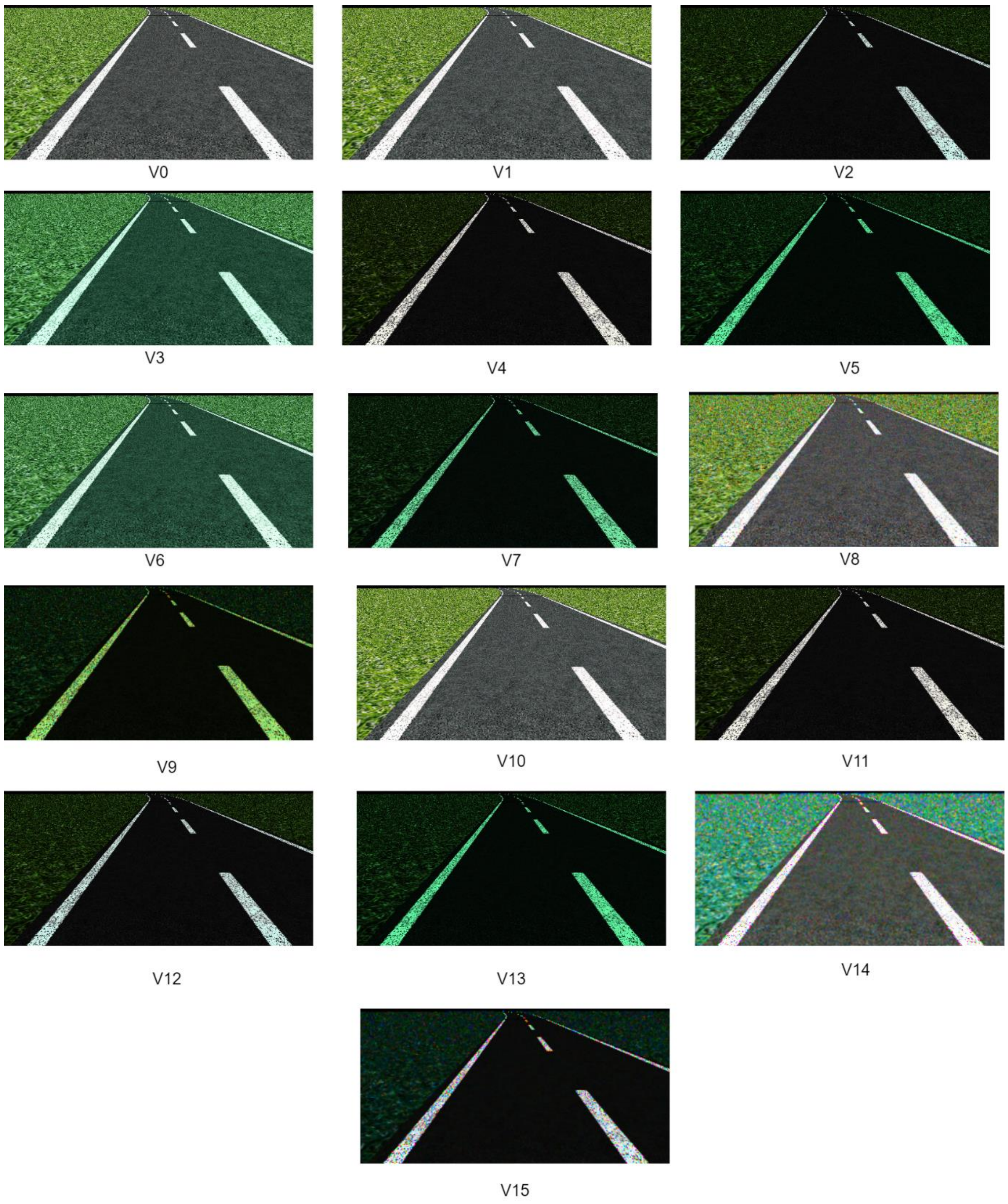


Figure 28. Output images of various versions from the coarse-grain pipeline

The latency for each of these reconfigurable ISP versions is calculated by adding the latency of individual stages. For example, for V12 the total latency is 2.643ms (0.543(DM) + 1.573 (GM) + 0.527 (TM)). The total latency for each of these versions are given in the below Figure 29.

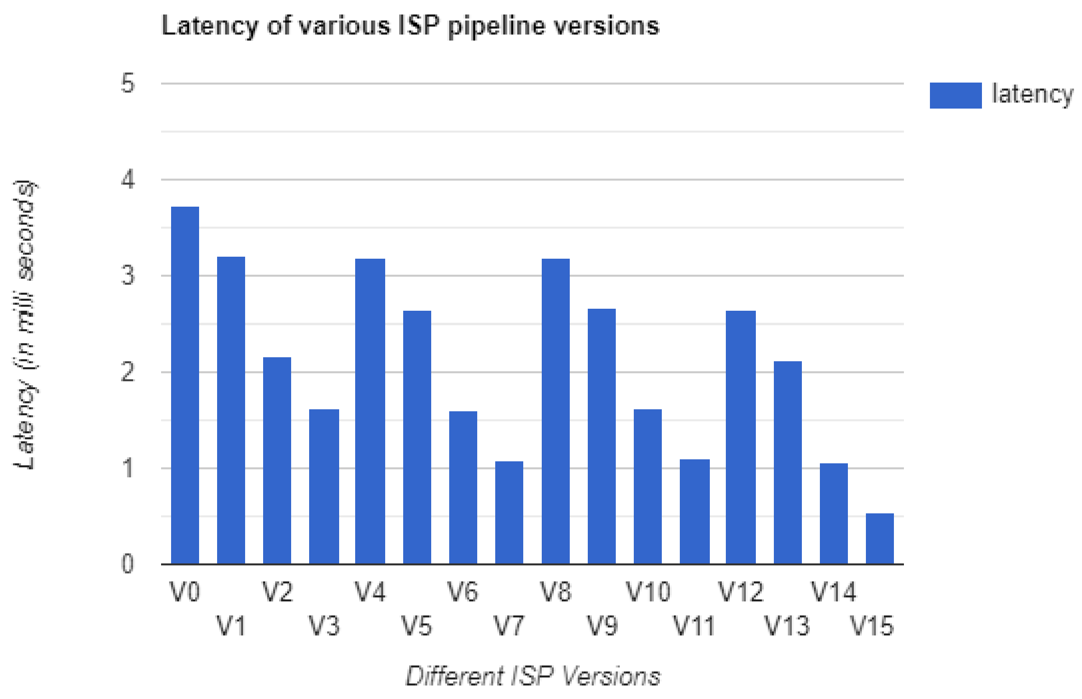


Figure 29. latency of the different ISP pipelines with coarse-grain approximation

The coarse-grain approximation done is similar to that of the software-in-Loop setup in [9]. Since the hardware implementation done is not in a closed loop system and it was not possible to evaluate the QoC. However, it was made sure that the images produced from the Halide pipeline in [9] which satisfied the QoC is identical to the images produced by the hardware. Hence it can be said that doing this coarse-grain approximation for the ISP does not have a negative impact on QoC when used in LKAS.

6.7.2 Fine-grain Approximation:

As mentioned in the design choice, the floating-point representation is very expensive both in terms of latency and resources to be implemented on an FPGA. One of the optimizations to address this issue is to use **fixed point** representation.

Unlike a single precision floating point representation which takes 32 bits (1 for sign, 8 for exponent and 23 for fractional part) to store the data, in fixed point representation, we can configure the number of bits that can be used represent the data in a finite range. However, if the bit width is too small to hold the data (which is not in finite range), it might cause data overflow/underflow issues. Various techniques like rounding to the nearest value in the finite range can be used to handle these mechanisms.

The final stage of optimizations for all the stages allowed the fixed-point datatype with a bit width of 18. The goal of this approximation is to reduce bit width of each pixel in image further more for an improved latency. The RTL implementation is generated in such a way that the bit-width of datatypes can be reconfigured dynamically at the run time. The results for two different fine-grain approximation configurations with a bit width of 16 bits and 14 bits have been tested on one of the stages (Color Transformation). The performance comparison between these two fine-grain approximations and the accurate implementation are given in the below Table 11.

Performance metrics	Accurate Implementation (18 bit)	Fine-grain Implementation (16 bit)	Fine-grain Implementation (14 bit)
Clock Frequency (MHz)	250	250	250
Latency (ms)	0.553	0.546	0.530
BRAM's	2	2	2
DSP's	9	9	9
LUT's	6751	6216	5199
FF's	8576	7872	6502

Table 11. comparison between fine-grain implementations

It can be seen from the results that the latency decreased slightly and there is a reduction in the LUT's and FF's. However since this project does not use a closed loop system to evaluate the QoC, it cannot be said whether the fine-grain approximation has a negative impact or positive impact. Hence the hardware generated can be used in future in closed loop system as a means to evaluate the fine-grained approximation.

6.7.3 Subsampling Approximation

Most modern camera sensors record an image of resolution much higher than required for a computer vision application like object detection. By reducing the resolution of the image i.e. the number of rows and columns in an image, the latency can be considerably reduced.

Subsampling is a technique used to reduce the size of an image by selecting a subset of the image. Nearest neighbor sub sampling is one of the techniques that can be used where we can skip alternate row and column pixels when reading the image to reduce the size of the image. Figure 30 illustrates how the nearest neighbor subsampling technique discards the alternate row and columns are done.

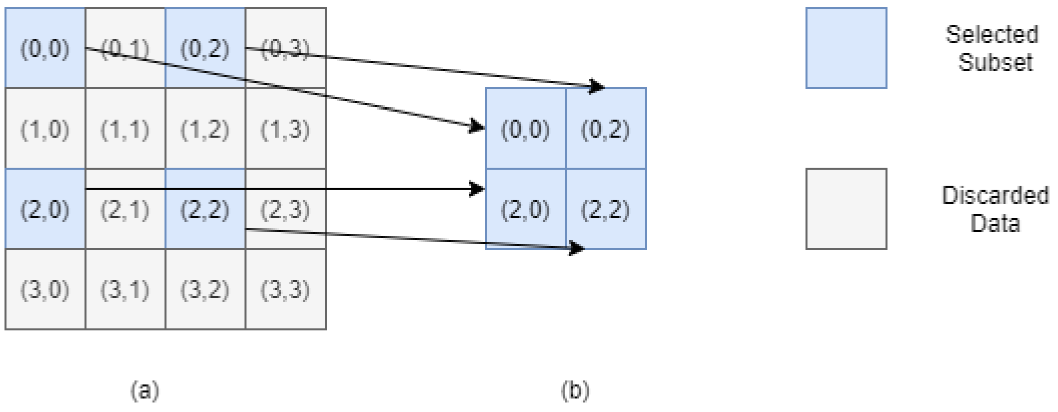


Figure 30. (a) original dataset, (b) selected subset

By using the subsampler technique mentioned above, the image with dimensions of 256 x 512 come down to a size of 128 x 256 as shown in the below Figure 31.

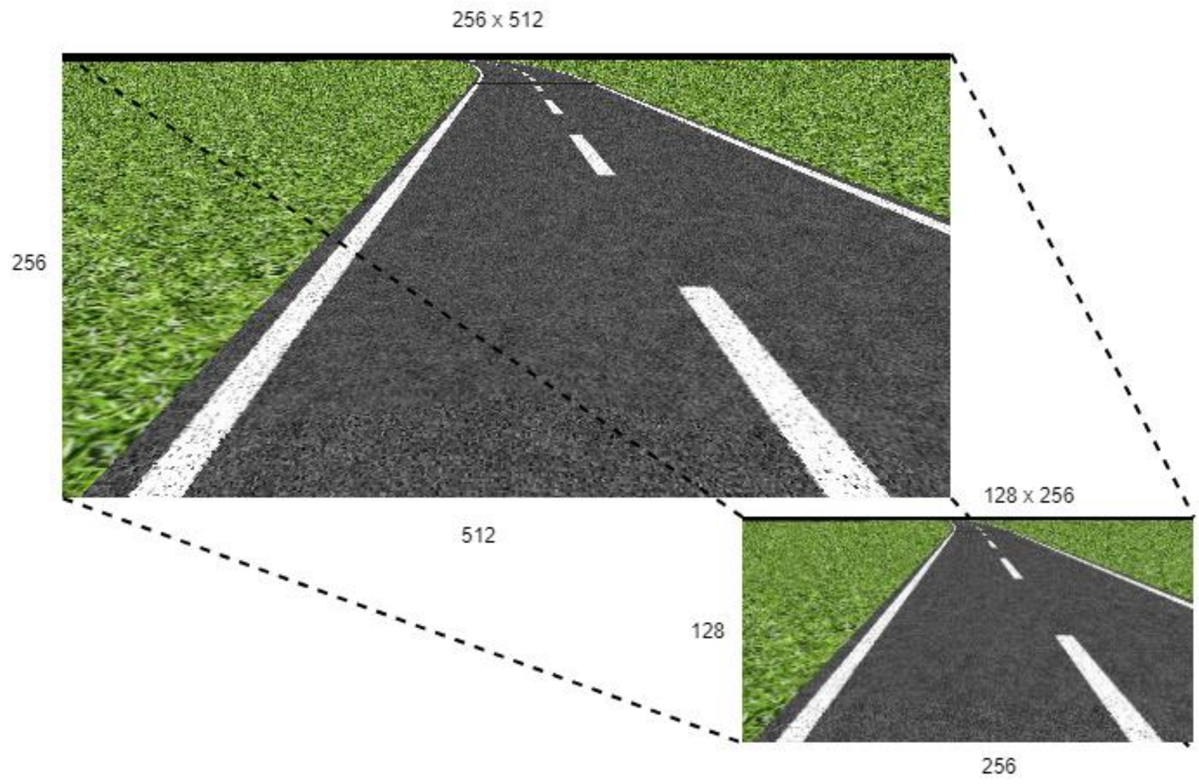


Figure 31. performing Subsampling on an image

Figure 31 illustrates how subsampling of the image is done with a factor of 2. Theoretically, using a subsampler with a factor of 2 reduced the size of the image by 75% which leads to an increase in processing speed of the image up to 300%. The subsampling approximation is applied on one of the stages (Color Transformation) and performance comparison between an accurate implementation and subsampled approximate implementation is given in below

Performance metrics	Accurate Implementation	Sub sampled approximate Implementation
Clock Frequency (MHz)	250	250
Clock Cycles	131146	32842
Latency (ms)	0.553	0.139
BRAM's	2	2
DSP's	9	9
LUT's	6751	6749
FF's	8576	8574

Table 12. performance comparison of accurate implementation vs subsampled approximation

It can be seen from the results that the latency is decreased from 0.553ms to 0.139ms, so a speedup of 3x is achieved by using the subsampling approximation. The sub sampler stage will be implemented in a way that it can be either enabled or disabled during the run time similar to that of all the other stages of ISP in the coarse-grain approximation. The architecture with subsampling approximation included is given in Figure 32.

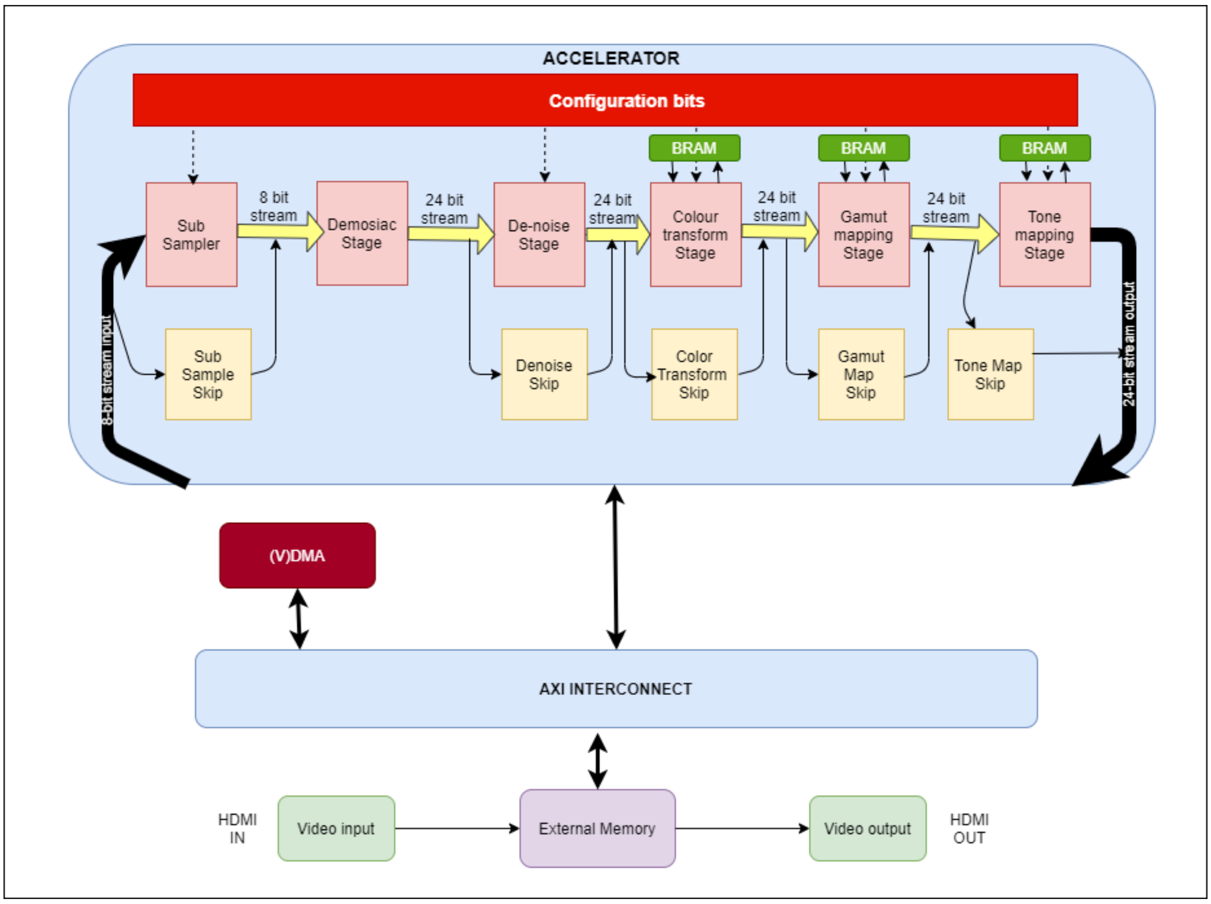


Figure 32. Subsampler Architecture

As mentioned earlier in the fine-grained approximation, since the Hardware in Loop system to evaluate the closed loop system is out of this project's scope, the QoC cannot be studied for the subsampler approximation. However, the hardware is generated in such a way that subsampler approximation can enabled or disable at runtime. The generated IP can be used in future to study the impact of QoC if a Hardware in Loop System is implemented.

Chapter 7 Conclusions

7.1 Summary

This work focused on implementing ISP accelerator on an FPGA with run time reconfigurability for a better QoC of LKAS system. Firstly, the ISP was written from scratch in C++ implementation. Then this C++ implementation was modified in such a way that it is acceptable by Vivado HLS to synthesize the RTL. Then the generated RTL was analyzed and optimized to have a better latency and area with the use of multiple coding styles and optimization directives. This work focused mainly on improving the latency of ISP and chose the biggest FPGA board available by tool as it has a greater number of resources and all the optimizations that were done are focused on improving the latency to generate the accurate hardware. Once the accurate hardware was generated the architecture was modified to include an option of approximating this hardware. Multiple approximation techniques were employed on this accurate hardware with an option of runtime reconfigurability.

7.2 Future Works

There are several directions in which further research can be carried based on this work. The first one is to implement the FPGA based reconfigurable ISP created in this project in a closed loop system and test the fine-grain approximation and subsampling approximation against the QoC of the LKAS. If these approximations do not have a negative impact on QoC, both of these approximations can be used in parallel with the coarse-grain approximation.

The scope of this project was limited to Vivado HLS tool was limited to performing the approximation techniques through C++. Hence, only software type of approximations was implemented. However using these algorithmic approximations IP (intellectual property) cores were generated. These IP cores can be used further in Vivado tool to perform more hardware type of approximations like decreasing the DRAM refresh rate for a better energy efficiency.

As mentioned in the summary, this project’s main focus was on improving the latency which came at the cost of increased area (resources). One more direction in which research can be made is putting a constraint on the area for an increase in latency and studying the various approximations in the closed loop system and its performance implication on the QoC.

As the scope of this project is limited to Vivado HLS tool, further research can be continued using the IP generated by this tool and connect with the other FPGA parts like external memory, VDMA in Vivado tool and generate the bitstream to be implemented on the FPGA.

Chapter 8 Bibliography

[1] “File:AdditiveColor.svg,” Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/File:AdditiveColor.svg. [Accessed: 16-Oct-2020].

[2] “File:Bayer pattern on sensor.svg,” Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/File:Bayer_pattern_on_sensor.svg. [Accessed: 16-Oct-2020].

[3] “Bayer filter,” Wikipedia, 08-Sep-2020. [Online]. Available: https://en.wikipedia.org/wiki/Bayer_filter. [Accessed: 16-Oct-2020].

[4] T. Kinden, John, K. M. O. 23, and K. Muessig, “Color Management 101 - Color Spaces,” Alder Color Solutions, 17-Jan-2019. [Online]. Available: https://aldertech.com/color-101-color-spaces/. [Accessed: 16-Oct-2020].

[5] S. De, S. Mohamed, D. Goswami and H. Corporaal, "Approximation-Aware Design of an Image-Based Control System," in IEEE Access, vol. 8, pp. 174568-174586, 2020, doi: 10.1109/ACCESS.2020.3023047.

[6] L. Anghel, M. Benabdenbi, A. Bosio, and E. I. Vatajelu, “Test and reliability in approximate computing,” 2017 International Mixed Signals Testing Workshop (IMSTW), 2017.

[7] S. Mittal, “A Survey of Techniques for Approximate Computing,” ACM Computing Surveys, vol. 48, no. 4, pp. 1–33, 2016.

[8] A. Agrawal, Z. Sura, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, and

V. Srinivasan, “Approximate computing: Challenges and opportunities,” 2016 IEEE International Conference on Rebooting Computing (ICRC), 2016.

[9] S. De, S. Mohamed, K. Bimpisidis, D. Goswami, T. Basten and H. Corporaal, "Approximation Trade Offs in an Image-Based Control System," 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2020, pp. 1680-1685, doi: 10.23919/DATE48585.2020.9116552.

[10] S. K. Ghosh, A. Raha, and V. Raghunathan, “Approximate inference systems (AxIS),” Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, 2020.

[11] M. Buckler, S. Jayasuriya, and A. Sampson, “Reconfiguring the Imaging Pipeline for Computer Vision,” 2017 IEEE International Conference on Computer Vision (ICCV), 2017.

[12] J. Ragan-Kelley, A. Adams, and D. Sharlet, “An introduction to halide,” ACM SIGGRAPH 2015 Courses on - SIGGRAPH '15, 2015.

[13] Xilinx.com, “Vivado HLS Optimization Methodology Guide,” 20-Dec-2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf. [Accessed: 18-Oct-2020].

[14] D. Bagni, “Median Filter and Sorting Network for Video Processing with Vivado HLS,” tudelft. [Online]. Available: https://cas.tudelft.nl/Education/courses/et4351/Median.pdf. [Accessed: 18-Oct-2020].

[15] "Camera Tuning: Understanding the Image Signal Processor and ISP Tuning - PathPartnerTech", *PathPartnerTech*, 2021. [Online]. Available:

https://www.pathpartnertech.com/camera-tuning-understanding-the-image-signal-processor-and-isp-tuning/. [Accessed: 06- Oct- 2021].

[16] *Tspace.library.utoronto.ca*, 2021. [Online]. Available: https://tspace.library.utoronto.ca/bitstream/1807/89598/3/DiCecco_Roberto_201806_MAS_thesis.pdf. [Accessed: 06- Oct- 2021].

[17] "Advanced Driver Assistance Systems (ADAS) - Rijschool Mirrer Purmerend", *Rijschool Mirrer Purmerend*, 2021. [Online]. Available: https://mirrer.nl/advanced-driver-assistance-systems-adas/. [Accessed: 06- Oct- 2021].

[18] "https://ars.els-cdn.com/content/image/1-s2.0-S0305748812000266-gr1.jpg", *Seenthis.net*, 2021. [Online]. Available: https://seenthis.net/sites/1699683. [Accessed: 06- Oct- 2021].

[19] "What Is Image Compression? - KeyCDN Support", *KeyCDN*, 2021. [Online]. Available: https://www.keycdn.com/support/what-is-image-compression. [Accessed: 06- Oct- 2021].

[20] M. Bayern, "The top 3 companies in autonomous vehicles and self-driving cars | ZDNet", *ZDNet*, 2021. [Online]. Available: https://www.zdnet.com/article/the-top-3-companies-in-autonomous-vehicles-and-self-driving-cars/. [Accessed: 06- Oct- 2021].

[21] *Xilinx.com*, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf. [Accessed: 06- Oct- 2021].

[22] researchgate.net, 2021. [Online]. Available: https://www.researchgate.net/publication/268190682_Image_denoising_using_balanced_multiwavelets_and_scale_mixtures_of_Gaussians. [Accessed: 06- Oct- 2021].

[23] "Talk:Color balance - Wikipedia", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Talk:Color_balance. [Accessed: 06- Oct- 2021].

[24] "Tone Mapping - RawPedia", *Rawpedia.rawtherapee.com*, 2021. [Online]. Available: https://rawpedia.rawtherapee.com/Tone_Mapping. [Accessed: 06- Oct- 2021].

[25] *Csl.cornell.edu*, 2021. [Online]. Available: https://www.csl.cornell.edu/courses/ece5775/pdf/lecture02.pdf. [Accessed: 06- Oct- 2021].

[26] 2021. [Online]. Available: https://www.theobjects.com/dragonfly/dfhelp/3-5/Content/05_Image%20Processing/Smoothing%20Filters.htm. [Accessed: 10- Oct- 2021].

[27] R. Kastner, J. Matai and S. Neuendorffer, "Parallel Programming for FPGAs", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1805.03648v1. [Accessed: 10- Oct- 2021].

[28] Xilinx.com. 2021. [online] Available at: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf> [Accessed 19 October 2021].