

**FIELD-PROGRAMMABLE GATE
ARRAY (FPGA) PARTICLE FILTER
CRACK DETECTION ACCELERATION**

**ACCÉLÉRATION DE LA DÉTECTION
DE FISSURE VIA UN FILTRE
PARTICULAIRE IMPLÉMENTÉ DANS
UN CIRCUIT LOGIQUE
PROGRAMMABLE**

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Tim Chisholm, BSc
Captain

In Partial Fulfillment of the Requirements for the Degree of
Masters of Applied Science in Computer and Electrical Engineering

January, 2019

© This thesis may be used within the Department of National Defence
but copyright for open publication remains the property of the author.

Acknowledgements

First of all, I would like to acknowledge my academic supervisor, Dr. Sidney Givigi, who has supported me with guidance throughout my lengthy part-time MASc program at RMC. I would like to thank Dr. Come Rozon, who has provided excellent FPGA feedback. I would also like to thank Dr. Romulo Lins for providing his original design and assisting with calculations.

Last, but not least, I would like to acknowledge my wife, Christine Chisholm for her incredible support, especially in the last couple months of the programme attempting to balance the demands of both this research and our newborn daughter, Clara Violet Chisholm.

Abstract

Due to the related hazards, costly down-time, and detection inconsistencies associated with manual visual inspection for cracks in structures, there has been an emergence of proposed autonomous robots capable of conducting these inspections. UAVs have been especially prevalent due to the requirement for scanning structures which can be located in remote areas or pose significant hazards to personnel. Due to the resource limitations inherent to UAVs, the current solution when conducting crack detection is to transfer all applicable sensor data to a ground station where detection will occur at a later time, thereby preventing real-time decision based on the results. To allow on-board UAV detection, and therefore on-board decision making to occur, a crack detection particle filter has been optimised for parallel computation and implemented onto an FPGA. This research shows that an FPGA holds distinct trade-offs between computational speed, energy consumption, and physical footprint compared to that of traditional CPU designs, allowing for it to be an ideal system for UAV applications.

Résumé

En raison des risques associés, des temps d'arrêt coûteux et des incohérences de détection associées à l'inspection visuelle manuelle des fissures dans les structures, de nouveaux robots autonomes capables de mener ces inspections ont vu le jour. L'usage de drones est particulièrement répandu en raison de la nécessité d'inspecter des structures en zones isolées ou présentant des risques importants pour le personnel. En raison des limitations de ressources inhérentes aux drones, la solution actuelle lors de la détection de fissure consiste à transférer toutes les données de capteur applicables à la station au sol où la détection sera effectuée ultérieurement, empêchant ainsi une décision en temps réel basée sur les résultats. Pour permettre la détection embarquée sur un drone et donc la prise de décision embarquée, un filtre particulière pour la détection de fissure a été optimisé pour le traitement de l'information en parallèle et implémenté sur un circuit logique programmable (FPGA). Cette recherche montre que le FPGA fait des compromis distincts entre la vitesse de calcul, la consommation d'énergie et l'empreinte physique par rapport aux designs traditionnels des unités centrales, ce qui en fait un système idéal pour les applications sur les drones.

Contents

Acknowledgements	ii
Abstract	iii
Résumé	iv
List of Tables	viii
List of Figures	x
List of Code Blocks	xii
List of Acronyms	xiv
1 Introduction	1
1.1 Introduction	1
1.2 Literature Review	2
2 Problem and Thesis Statement	7
3 Technical Background	10
3.1 FPGA Technical Background	10
3.1.1 General	10
3.1.2 Resources	13
3.2 FPGA Selection	16
3.3 Design Tools	18
3.4 HLS Background	20
3.5 Video Format	21
3.6 Computer Vision	22
3.7 Crack Measurement Algorithm	25

4	Solution Design	27
4.1	Overall Design Strategy	27
4.2	Software Implementation	29
4.2.1	Image Acquisition	33
4.2.2	Sobel Edge Detection Filter	34
4.2.3	Particle Map Initiation	35
4.2.4	YUY2 to RGB Conversion	36
4.2.5	Particle Filter Calculations	37
4.2.6	Uncertainty Calculation	38
4.2.7	Likelihood Calculation	39
4.2.8	Re-sampling	40
4.2.9	Analysis and/or Data Storage	41
4.3	Hardware Implementation	42
4.3.1	General	42
4.3.2	Sections for Hardware Acceleration	42
4.3.3	Edge Detection Filter Acceleration	44
4.3.4	YUY2 to RGB Converter Acceleration	45
4.4	Hardware Issues/Potential of Remaining Components	46
4.5	Other Accelerations	48
5	Testing Methods	49
5.1	General	49
5.2	Detection and Measurement Accuracy	49
5.2.1	General	49
5.2.2	Camera vs SD Card	53
5.3	Computational Performance	54
5.4	Footprint	55
5.5	Energy Analysis	55
6	Results and Discussion	58
6.1	Detection Accuracy	58
6.2	Computational Performance	65
6.3	Footprint	70
6.4	Energy Analysis	71
6.5	Results Summary	76
7	Future Areas of Work	77
8	Conclusion	79

Bibliography	81
Appendices	87
A Zynq-7000 and Zynq-7000S SoCs	88
B Particle Filter Code	93

List of Tables

1.1	Various Image Processing Techniques	3
1.2	Comparison Summary of Various Design Platforms [16]–[18]	4
1.3	Computational Performance of GPU Parallel Particle Filter [11]	5
4.1	Computational Performance - Zynq-7030 (Software)	43
4.2	Computational Performance (Ranked) - Zynq-7030 (Software)	44
4.3	FPGA Resources Utilised - Sobel Edge Filter	45
4.4	FPGA Resources Utilised - YUY2 to RGB Conversion	45
4.5	FPGA Resources Utilised - Re-sample Section	47
5.1	Test Crack Measurements	52
6.1	Detection Accuracy - Zynq 7030 - 4,000 Particles	64
6.2	Detection Accuracy - Zynq 7030 - 8,000 Particles	64
6.3	Detection Accuracy - MATLAB - 4,000 Particles	64
6.4	Detection Accuracy - MATLAB - 8,000 Particles	64
6.5	Detection Accuracy - Zynq 7030 vs MATLAB	65
6.6	Computational Performance - Zynq 7030 (Software)	66
6.7	Computational Performance - Zynq 7030 (Hardware)	66
6.8	Computational Performance - Zynq 7030 (Software) vs Zynq 7030 (Hardware)	67
6.9	Computational Performance - Acer Netbook	68
6.10	Computational Performance - Acer Netbook vs Zynq 7030 (Hard- ware)	69
6.11	Computational Performance - EVGA S17	69
6.12	Computational Performance - EVGA S17 vs Zynq 7030 (Hardware)	70
6.13	Platform Footprints	71
6.14	Power Consumption - Zynq 7030 (Hardware)	72
6.15	Power Consumption - Acer Netbook	73
6.16	Power consumption - EVGA S17	74

6.17 Summary of Results 76

List of Figures

2.1	Scope of Solution Design	8
3.1	Internals of an FPGA (modified) [22]	11
3.2	FPGA Design Level [22](modified)	12
3.3	Various Possible LUT Configurations [26](modified)	13
3.4	Xilinx 7-Series CLB [28]	16
3.5	PicoZed Embedded Vision Kit	17
3.6	Vivado Block Diagram Example	19
3.7	Memory Allocation of YUY2 Format [33](modified)	22
3.8	Sobel Mask Matrices	23
3.9	Application of the Sobel Masks [34]	23
3.10	Hysteresis Thresholding [35](modified)	25
3.11	Crack Types [36], [37]	26
4.1	High Level Overview of Software Design	28
4.2	High Level Overview of Hardware Design	29
4.3	Command-line Interface for the Video Control Application	31
4.4	Design C/C++ Source Code and Header Files	32
4.5	Software Design Component Interaction	33
4.6	Hardware Block Diagram of Image Acquisition [38](modified)	34
4.7	Effects of Sobel Operator Adjustments	35
4.8	IO Diagram of the Likelihood Calculation	39
4.9	IO Diagram of the Re-Sample Calculation	40
4.10	Particle Spread with Different Re-sample Amounts	40
4.11	Baseline Test Image	43
5.1	Test Crack Sections	51
5.2	Results from Camera	53
5.3	Kill A Watt (R) P4400 Energy Meter	56

6.1	Edge Image from the Sobel Edge Detection	59
6.2	Edge Image from the Canny Edge Detection	60
6.3	Refined Edge Image from Sobel Edge Detection	61
6.4	Crack detection using the Sobel Edge Detection (FPGA Design) .	62
6.5	Crack Detection using the Canny Edge Detection (Original Design)	63
6.6	PL Power Estimation - Sobel Edge Detector	72
6.7	PL Power Estimation - YUY2 to RGB Converter	73
6.8	Energy Consumption Comparison (Lower is Better)	75
6.9	Energy Efficiency Comparison (Higher is Better)	75

List of Code Blocks

4.1	Particle Map Initialisation C/C++ Code	36
4.2	Simplified Code for YCbCr to RGB Conversion [39]	37
4.3	Uncertainty Calculation Code	38
4.4	Re-sample Function Arguments	47

List of Acronyms

APSoC	All-Programmable System on Chip
APU	Application Processing Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuits
BPP	Bit Per Pixel
BRAM	Block Random Access Memory
CDPF	Crack Detection Particle Filter
CDF	Crack Detection Function
CLB	Configurable Logic Block
CLI	Command-Line Interface
CPU	Central Processing Units
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
FPW	Frames Per Watt
GB	Gigabyte
GHz	Giga-Hertz
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HP	High Performance
HR	High Range
IDE	Integrated Development Environment
IO	Input / Output
IP	Intellectual Property
LiPo	Lithium Polymer
LUT	Look-up Table
LUTRAM	Look-up Table Random Access Memory
mAh	Milliamp Hour
MATLAB	Matrix Laboratory (Software)

MB	Megabyte
MM	Millimetre
MP	Megapixel
NDT	Non-Destructive Testing
OS	Operating System
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
PLC	Programmable Logic Cells
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
RNG	Random Number Generator
SD	Standard Deviation
SD Card	Secure Digital Card
SDSoc	Software Defined System-On-a-Chip
SoC	System on Chip
STRUM	Spatially Tuned Robust Multi-Feature
TDP	Thermal Design Power
UAV	Unmanned Aerial Vehicles
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Introduction

1.1 Introduction

There are many structures and components, such as buildings, bridges, gears, or propellers, that require their structural integrity to be monitored in order to detect any defect which can potentially cause failure and subsequent lost of life or profit. Depending on the material and the stress, defects can present themselves as cracks in the material. These cracks can be detected and monitored to ensure that components can be fixed or replaced before problems arise. Detecting cracks in these various structures and materials can be achieved by several methods. As it is generally desired not to damage the structure in the detection method, Non-Destructive Testing (NDT) will be the focus of discussion. This area of testing includes, but is not limited to image-based inspections, ultrasonic, magnetic-particle, liquid penetrant inspection, radiographic, eddy-current testing, and low coherence interferometry.

A large portion of crack detection is conducted through manual in-person visual testing in the form of a site inspection, but this is not the ideal solution. There are many reason why companies are seeking these inspections to become automated [1]. The first reason is a large variation between the inspector's experience and knowledge leading to the results being very subjective and significantly inaccurate, which can vary tremendously between inspectors [2]. The second reason is that these manual inspections can be very time consuming and cost the company a substantial amount of money especially if system down-time is required for the duration of the inspection. The third reason raised is that the locations requiring crack detection can be very impractical (or dangerous) for a human to inspect, like in the case where an inspector requires to wear fall restraint to inspect the structural members of bridges at high heights. [3]

In an effort to offset the disadvantages arising from human-based visual in-

spections there is an emergence of autonomous robots being proposed that are capable of automatically detecting cracks while being equipped with minimal resources such as a single optical camera as the sole crack detection sensor [4]–[6]. With image-based crack detection methods being proposed that are claiming accuracy rates over 95%, there is a strong argument to research automatic image-based crack detection systems as either a supplement or replacement for human inspections.

1.2 Literature Review

An image-based crack detection comparison essay that analysed 50 papers is summarized in Table 1.1 and reveals that there exists little commonality between the image-based crack detection methods that are currently being proposed and researched [7]. Literature has suggested that the vast emergence of different image based crack detection methods is due to the differences in several factors including the random shape and irregular size of cracks, and various noises such as irregularly illuminated conditions, shading, and blemishes in the acquired images [8]. Another major contributor to the differences in the design of crack detection techniques is variances in environment, including but not limited to moisture content, sunlight amount and intensity, or presence of fine debris such as sand, all which will affect the method that is the most beneficial [9]. Even with all these factors to consider, a large majority of these solutions are based on a desired specific trade-off between detection accuracy and computation speed.

Image Processing Technique	# of Papers
Morphological approach	5
Digital image correlation	4
Randomized hough transform	3
Ultrasonic pulse velocity technique	3
Wavelet transform	5
Median filtering	3
Gabor filtering	2
Otsu's method	3
Statistical approach	3
Threshold method	4
Supapixel algorithm	2
Data fusion filtering	3
Reconstruction technique	4
Photogrammatic technique	3
PA imaging	3
Percolation	3
Centroid method	2
Delaying and summing algorithm	2
GLCM texture analysis	3
Dijkstra's algorithm	1
Skeletonization techniques	2

Table 1.1: Various Image Processing Techniques
(adapted from [7])

Unfortunately, many of the potentially most accurate algorithms come at a huge computational cost [10]. Even running these algorithms on a performance computer can yield several seconds of runtime just to analyse a single frame. While this can suffice for running detection methods back in the office, it is not practical when implementing this on mobile platforms such as UAVs or wheeled robots which may require analysis in a timely manner. Fortunately, a few of these computational heavy crack detection algorithms can be drastically sped up by taking advantage of parallelism. When the code is optimised to run several hundred or thousand parallel operations, it can be placed into hardware specifically built to exploit this type of design framework. This can potentially yield much faster computational speeds than a CPU running the equivalent serial-based algorithm. [11], [12]

The run-time of the various techniques shown in Table 1.1 vary widely,

but they all show that crack detection systems require a substantial amount of computational power. This is shown in research such as [13] which used two different crack detection algorithms, a Sobel based and particle filter based, which run-time varied from 30 minutes to several days based on the ground station hardware. Another example of computational performance is taken from a UAV-based crack inspection system using a structured forest solution, which a ground station was able to compute in 4.4 seconds for a 7400 x 2580 pixel image [6]. A bridge crack detection robot was also designed based on the STRUM (spatially tuned robust multi-feature) classifier which acquired and analysed frames in around 2 minutes and 45 seconds [14].

Computer vision algorithms can be implemented into a wide range of systems. The four most common being Central Processing Units (CPU) [15], Graphics Processing Units (GPU), Field Programmable Gate Arrays (FPGA), and Application-Specific Integrated Circuits (ASIC) [16], each having a variety of reasons for being chosen, including performance, energy, footprint, cost, development complexity, etc. Table 1.2 shows a general summary of the trade-offs highlighting the relative pros and cons of each criteria.

Device	Computational Performance	Energy Consumption	Size	Cost	Design Complexity
CPU	Low	Low	Med	Low	Simple
GPU	High	Med	Large	Med	Simple
FPGA	High	High	Small	Med	Moderate
ASIC	High	High	Tiny	Low-High	Complex

Table 1.2: Comparison Summary of Various Design Platforms [16]–[18]

Table 1.2 generally holds true when the project is optimized for the specific strengths and weaknesses of the target platforms. If projects are not designed for their target platform, it is very likely that large performance penalties will be encountered, such as in the case if a serial program was written for an FPGA or a parallel program for a single processor CPU. There are also other device characteristics that must be considered above and beyond the considerations given in Table 1.2. For instance, while many of the platforms have the flexibility to be changed (or fixed) through the use of software updates, one of the major factors which cause developers to shy away from ASIC is that once manufactured they generally cannot be changed or updated.

Many image processing algorithms have seen speed increases around 15 to 30 times compared to a conventional CPU [19], [20] when optimized for parallel computation. One particular algorithm, particle filtering, has a strong potential for this type of optimisation. Particle filters are based on recursive Bayesian filter and calculate (re-sample, predict, update) the weight of each particle to determine density estimations [12]. There are several steps that are very computationally expensive, such as the calculation for sample weights. However, due to that calculation being independent of the weights of neighbouring particles, this has demonstrated tremendous performance improvements through parallelism. In addition, experiments with parallel particle filters have been conducted on GPUs (Table 1.3) and have resulted in remarkable speed-ups [11]. This is especially true for larger particle counts as serial computations start to become infeasible due to the required time.

Particle #	GPU time (s)	Serial Time (s)	Speedup (%)
200	0.046	0.827	1,797.8
400	0.062	1.104	1,780.5
600	0.068	1.739	2,557.4
800	0.083	2.246	2,706.0
1,000	0.093	3.525	3,790.3

Table 1.3: Computational Performance of GPU Parallel Particle Filter [11]

In 2016, Romulo Gonçalves Lins completed research at RMC in which he implemented a particle filter for the purposes of crack detection [21]. The CDPF he developed has shown to be accurate within 7.51% - 8.59% while retaining the possibility to increase the accuracy based on the particle count and image resolution. He chose a particle count of 4,000, which was a compromise between computational speed and accuracy. Even though this developed algorithm yielded an accuracy rate of approximately 92 percent on a personal computer, there are still potential improvements that can be made. In his paper, he proposed that “implementation in a Digital Signal Processor (DSP) or a Field-Programmable Gate Array (FPGA) is possible with many advantages over a software program operating on a conventional computer” [21].

To increase the accuracy rate and reduce all the runtime, footprint, and power requirements of this particle filter, this algorithm will be optimised then implemented onto an FPGA. In Chapter 4, this research will propose a breakdown on how each step of the original particle filter will be approached to

optimize for hardware acceleration.

2 Problem and Thesis Statement

While successfully implementing a solution for a ground based mobile robot, the original design's particle filter based crack detection solution is not fully optimised for the computational, footprint, or energy constraints inherited of those seen in resource limited applications such as airborne platforms (e.g., UAVs).

Using the original design's hardware resources which incorporated a PC Netbook, a UAV would not have the on-board resources to meet the large energy requirements of the Netbook to fully analyse a structure. Issue would also arise in regards to the handling of the much larger payload weight, especially the additional weight of a bigger energy source (e.i., larger battery) required to sustain a Netbook. The Netbook's computational performance limitations would also mean that the UAV requires to stay airborne longer to conduct the required crack detection, putting further strain on the limited on-board resources.

Computational performance poses a substantial issue with the original design's implementation onto resource-limited platforms. Standard serial implementations of image-based CDPF systems with large particle counts require long computational times, high-performance computers, bulky footprint, and consequential heavy power draw. Due to these requirements, the solution to operate directly on small resource limited platforms becomes infeasible. The current solution is to transfer all applicable sensor data (imagery) to a ground station where detection will occur at a later time, thereby taking up a larger storage or transmission requirements.

The goal of this research is to implement a functionally equivalent al-

gorithm onto an FPGA which allows for a reduction in weight and energy consumption, while retaining similar crack detection accuracy. In addition, this design is also expected to demonstrate an increase in computational performance shown in terms of a reduced run-time per analysed image frame. To allow for self-contained detection ability on this platform, the system will be optimized through the use of parallel computation and ported onto an FPGA.

Figure 2.1 shows the four key stages of crack detection systems, where this design is limited to the first two stages, meaning that the functionality of this designed crack detection system is limited to detecting cracks, and will not have the on-board ability to conduct analysis on those detected cracks.

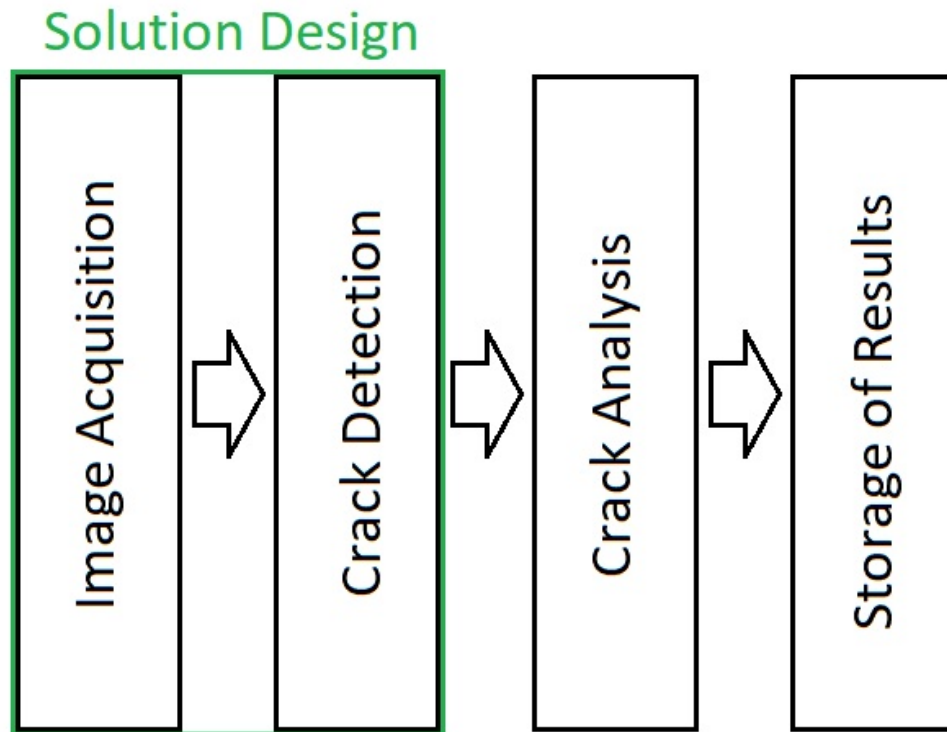


Figure 2.1: Scope of Solution Design

The hypothesis of this work is that a Parallel Crack Detection Particle Filter (CDPF) may be effectively implemented on a Field Programmable Gate Array (FPGA) for Unmanned Aerial Vehicles (UAV) applications allowing for

the on-board crack detection to further assess areas of interest on the structures being inspected. While UAV is the main target due to the resource limited nature, other applications include, but are not limited to, ground based vehicles, high volume crack analysis, inspections aided with augmented reality, and assisting manual on-site inspections. This research argues that an FPGA holds trade-offs between computational speed, energy consumption and physical footprint of an FPGA implementation compared to that of traditional CPU designs when implemented onto a UAV.

In regards to the literature reviewed, this research is unique as it is one of the only particle filter based crack detection systems that have been accelerated through the use of FPGA, allowing this research to more readily implemented for airborne applications. This platform will not require the downloading of structural imagery to a remote system for computational analysis, instead it will have the required resources on-board to conduct all related crack detection requirements, while only requiring to store or transmit the much smaller sized crack or particle data.

The research will be of great benefit to the continued automation and independence of robotic crack detection systems, as well as to provide a cost-effective, measurement consistent, and less dangerous solution than what would be seen in typical manual processes.

3 Technical Background

3.1 FPGA Technical Background

3.1.1 General

To better understand the components of this research, technical background in several areas will be first provided to establish a baseline knowledge. Due to the hardware platform used to accelerate this system, the first area of technical background to be discussed will be in regards to the hardware aspects and the design of FPGAs. After that, information will be provided in the field of computer vision due to the use of the image-based crack detection algorithm, which utilises the Sobel Edge Detector. Given the background of these two key subject areas, discussion of the more in-depth research can proceed.

FPGAs are a hardware solution that take the best aspects of ASICs and CPUs and combines them into a single package [22]. Like processors, but unlike ASICs, they can be ‘reprogrammed’ multiple times in order to meet a range of different applications. In most cases these components are also equipped with multiple sub-components that can be programmed into a design including logic blocks, DSPs, Block Random Access Memory (BRAM), and Input/Output (I/O) blocks, all of which can be used in a varying degree depending on the application. These components can be interconnected as required as shown in Figure 3.1.

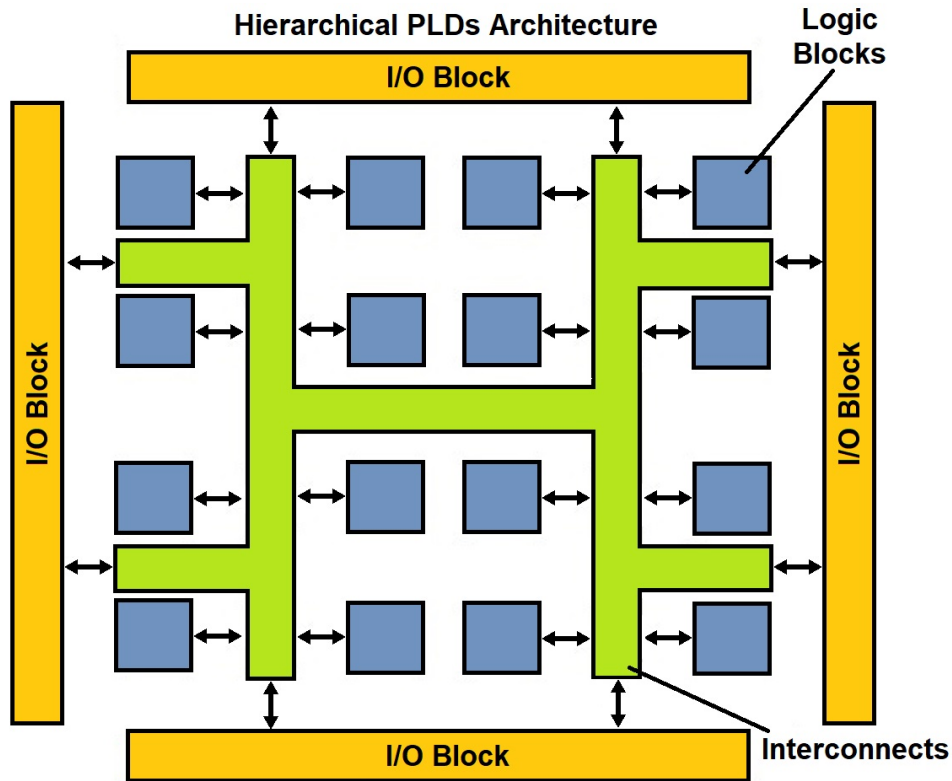


Figure 3.1: Internals of an FPGA (modified) [22]

Unlike processors, FPGAs run in truly parallel hardware, where thousands of independent computations can be completed simultaneously. For applications that can make use of this parallelism feature, designers can find large improvements in their runtime performance. GPUs also make use of parallelism, and can be faster in many cases, but it comes with an increase in energy consumption and physical footprint. FPGAs are also more deterministic than GPUs, and hence hold an advantage for real-time applications [23]. The main reason for choosing to conduct this research on an FPGA vice a GPU, was due to the lower power consumption and smaller footprint of an FPGA, making it more ideal for a resource limited platform, such as a UAVs.

One of the most difficult aspects of FPGA design compared to that of conventional software programming is the low level hardware design such as VHDL and Verilog which require knowledge on both a programming language and hardware aspects of the design to successfully produce a working system.

Unlike designs using higher level languages, the designs for FPGA are usually more complex and require much more time and effort to produce a functionally equivalent system [15]. Although this can be mitigated by design tools which can allow higher level hardware programming, such as Xilinx’s High Level Synthesis (HLS) design methodology [24], this still requires the designer to have an in-depth knowledge of the target hardware.

One of the core concepts for any FPGA designer to understand is that hardware is being designed, not software. Figure 3.2 shows the various processing levels of conventional CPU architectures compared to the much lower level hardware design. While software designs generally follow the higher “Application Software” level, FPGAs deal with the low level “Hardware” processing. FPGA designs have several advantages over software designs including higher result/run-time determinism and higher reliability. This comes at the cost of FPGA projects generally becoming more complex and having larger design times even when factoring design acceleration tools.

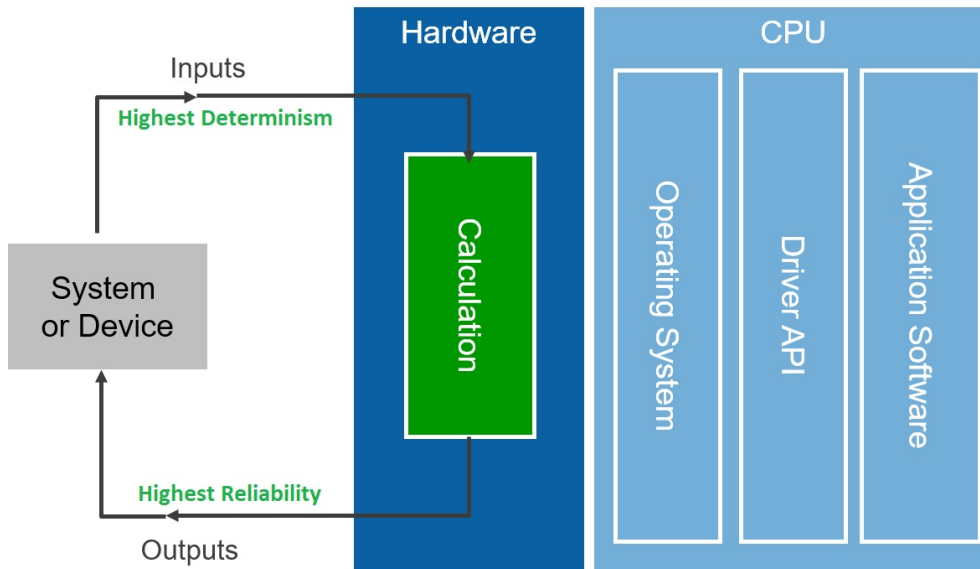


Figure 3.2: FPGA Design Level [22](modified)

3.1.2 Resources

As previously discussed, the Xilinx Zynq 7030 comes with several on-board resources for a developer to use in their design [25]. The Xilinx tool itself gives performance estimates including the resources used for each build which include Look-up Tables (LUT), Look-up Tables Random Access Memory (LUTRAM), Flip-Flops (FF), BRAM, DSPs, IO blocks, and BUFGs.

Each of the resources are limited in number and play a vital role in the build of a successful system. Each of the various FPGA components are further described as follows:

Lookup Tables (LUT): Along with FFs, LUTs are one of the most important components in the design of an FPGA. These are essentially programmed tables which give a pre-determined output based on an input. These allow for fast boolean logic decision making to occur in the device. Figure 3.3 provides examples of LUTs programmed as conventional combinatorial logic, such as ANDs, ORs, NANDs, XORs, etc., but it is important to know that LUTs are not restricted to just these and can be programmed to provide any custom requirement. In FPGAs, conventional combinatorial logic is in reality usually implemented on LUTs vice using the actual gates (AND/OR/NANDs/XORs/etc) to allow for greater flexibility when programming the FPGA board in terms of routing. The Zynq 7030 gives 78,600 LUTs resources to be used in a design [25].

AND			NAND			OR			NOR			XOR		
InA	InB	Out	InA	InB	Out	InA	InB	Out	InA	InB	Out	InA	InB	Out
0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
1	0	0	1	0	1	1	0	1	1	0	0	1	0	1
0	1	0	0	1	0	0	1	1	0	1	0	0	1	1
1	1	1	1	1	0	1	1	1	1	1	0	1	1	0

Figure 3.3: Various Possible LUT Configurations [26](modified)

Lookup Table Random Access Memory (LUTRAM): Like many of the resources listed here, LUTs can be used in a variety of ways which in this case includes the form of small 64-bit memory modules called LUTRAM. While LUTRAM is much smaller than BRAM, it is usually much faster. Only about 34% of the total 78,600 LUTs on the Zynq 7030 can be used as distributed memory in the form of LUTRAM, giving us a total available number

of 26,600. In cases of small memory requirements, it is generally advisable to consume one of the 26,600 LUTRAMs versus one of extensively more limited 265 BRAMs.

Flip-Flops(FF): As mentioned previously, FFs along with LUTs form the most important resources that make an FPGA operate. FFs allow the board to have extremely low-level bit-wise memory which stores states between clock cycles. Given those properties, FFs are widely implemented in the form of a register. With 157,200 FFs being provided, they form the vast majority of resources available on the Zynq 7030.

Block Random Access Memory (BRAM): Another type of memory available to designers of an FPGA. The Zynq 7030 has 265 BRAM blocks each having 36 Kilobit (Kb) capacity which equates to a total of 9.3 Megabit (Mb) available. These blocks are fairly configurable as FIFO, single/dual access and can also be adjusted in bit width and be chained together to form larger memory blocks.

Digital Signal Processor (DSP): One of the biggest disadvantages with FPGAs is the extensive amount of resources required to do floating point math operations, especially when considering multipliers of a large width (16/32/64-bit). In addition to the amount of resources being used, the size and complexity of these components lead to extensively higher run-times in comparison to even CPUs which generally have a floating point unit embedded. To give an idea of the scale of effort required to implement a multiplier on conventional FPGA resources, in order to multiply two 32-bit numbers together, you should expect to use over 2,000 components [22]. To address this major flaw, conventional FPGA manufacturers are embedding floating point DSPs directly on the FPGA boards to provide flexible design opportunities. In particular, our Zynq 7030 comes equipped with 400 DSP slices of the DSP48E1 variant. The DSP48E1 slice supports many independent functions including multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, widebus multiplexing, magnitude comparator, bitwise logic functions, pattern detect, and wide counter [27].

Input/Output (IO): These blocks, as shown in Figure 3.1, provide communications from the Programmable Logic section of the FPGA to other on-board resources such as the Application Processing Unit (APU) or RAM. The Zynq 7030 has 150 IO blocks for designers to use. With this design's large data arrays including image storage and likelihood vectors, this will be one of

the most demanded resources. When adding additional hardware accelerated modules to a design on the same FPGA, IO blocks become further utilised. We will see in later chapters how quickly these are consumed in comparison to other resources given the nature of the research.

BUFG: These components are used to provide a common low-latency clock values to the global level of the entire board. In our Zynq 7030 specifically, 32 of them can be employed on a single design. With the large size of the system being designed and the various different components being integrated, it is essential to ensure that all components are running on the same clock (when required) to ensure the correctness of the values.

As a final point, it is important to describe the layout of the FPGA architecture. These components are grouped together in slices and Configurable Logic Blocks (CLB). Specifically for the Xilinx Zynq 7030, a logic block consists of 2 logic slices, as shown in Figure 3.4. In turn, each slice consists of 4 LUTs and 8 FFs. The 1:2 LUT to FF ratio in each slice carries over to the high level totals, where 157,200 FFs is twice the amount of the 78,600 LUTs.

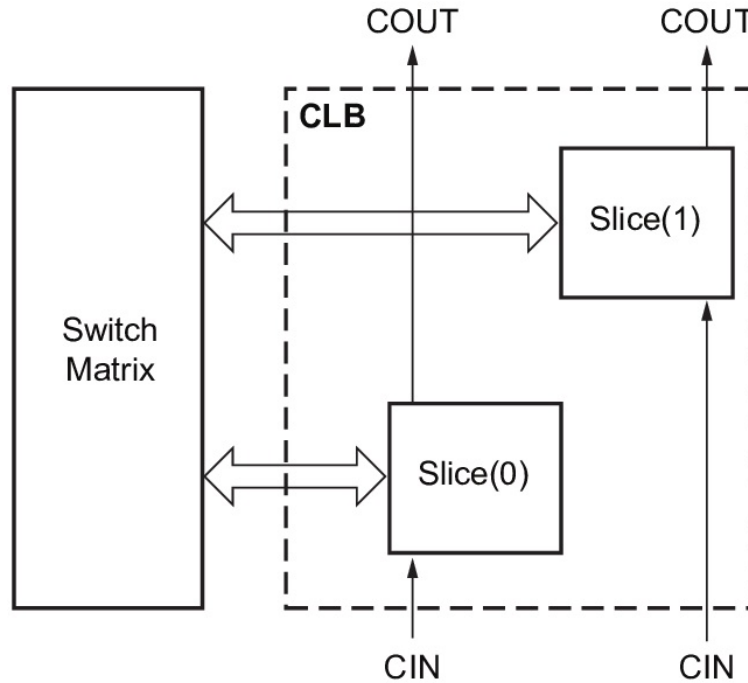


Figure 3.4: Xilinx 7-Series CLB [28]

3.2 FPGA Selection

Selection of the FPGA and associated peripherals required to support this design was the first step in the process, as future steps were dependant on the hardware, such as the coding language and design software. Due to the requirements associated with on-board UAV calculations, especially when considering heavy particle filter computations, this design required an embedded vision kit with a high-performance FPGA.

The hardware chosen for this design is the PicoZed Embedded Vision Kit (shown in Figure 3.5), which is available online for approximately \$1,500.00 USD. This kit was chosen for multiple reasons, the first being that the kit centres around the Xilinx Zynq 7030 All Programmable System on Chip (AP-SoC) which contains programmable logic equivalent to their Kintex-7 FPGA family. This contains 124,000 Programmable Logic Cells (PLC) as well as a dual-core ARM Cortex-A9 co-processor running at 1 GHz. The kit also contains all required components for vision-based projects ready in a single

package including the Python-1300-C camera module, capable of resolutions of 1280x1024 @ 210 frames per second. This camera is targeted towards machine vision and motion monitoring applications, making it an excellent choice in this design. Given the high FPS, it is expected to be able to obtain sharp images from a moving platform easily.

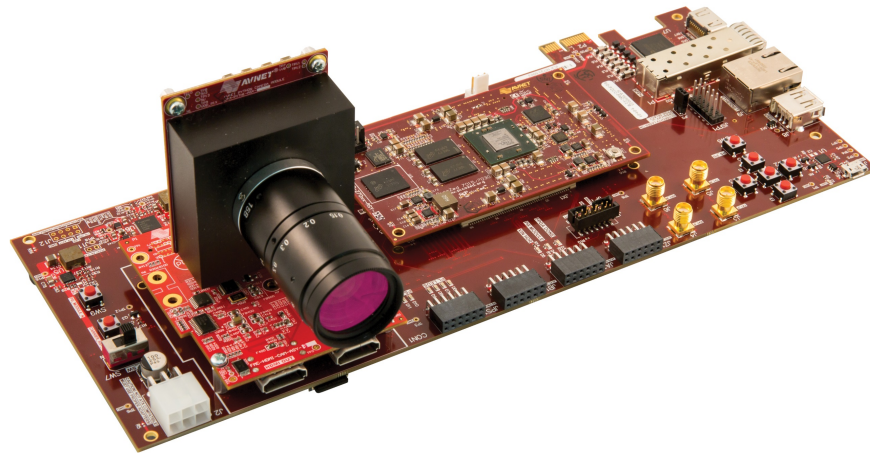


Figure 3.5: PicoZed Embedded Vision Kit

The technical background behind the multiple programmable logic resources were already covered in the previous section. The resource amounts that were discussed are determined by the model number and can be determined by Tables A.1 and A.2 (A.# Tables are attached as an appendix) and show that the Zynq 7030 falls into the upper-middle of the 7-Series product line. Compared to other models, the Zynq-7030 was an ideal choice due to the lower cost, but still having a 1 GHz Dual-core ARM Cortex APU and a relatively large amount of floating point DSPs and BRAM, all of which the lower models lack.

Although the details are covered in later chapters, the limiting factor of resources for a design of this type is the amount of IO blocks. The Zynq-7030 board being used in this design utilises an SBG485 package (refer to Table A.3 for more details) which contains 150 SelectIO pins (50 High Range (HR) I/O + 100 High Performance (HP) I/O). Much more flexibility could have been delivered if utilising a higher level Zynq component such as the Zynq-7100 which delivers 500 IO blocks. However, which the price of the Zynq-7100 SoC being \$3,324.69 on www.Digi-Key.com versus the \$331.50 cost of the Zynq-

7030 SoC, compromises were made.

3.3 Design Tools

The proposed design will be produced using Xilinx's Vivado Design Suite tools. One of the main factors of purchasing the PicoZed Embedded Vision Kit was the inclusion of licenses for these tools, which will allow easier and faster product design specifically for the target FPGA. The vision kit also came with a reference design which was compatible with Xilinx's 2016.2 series of design tools. Despite the availability of their 2018.2 version, certain Intellectual Property (IP) within the reference design could not be easily updated to be compatible with this newest version of the tool, due to some IP being discontinued.

Vivado contains many features to accelerate the design processing including the employment of an easy to use GUI interface, FPGA focused pre-built C/C++ libraries and Intellectual Property (IP) Blocks, automated integration between the components using visual block diagrams (Figure 3.6), visual interactive designs, debugging tools, simulation and synthesis options, and HLS programming.

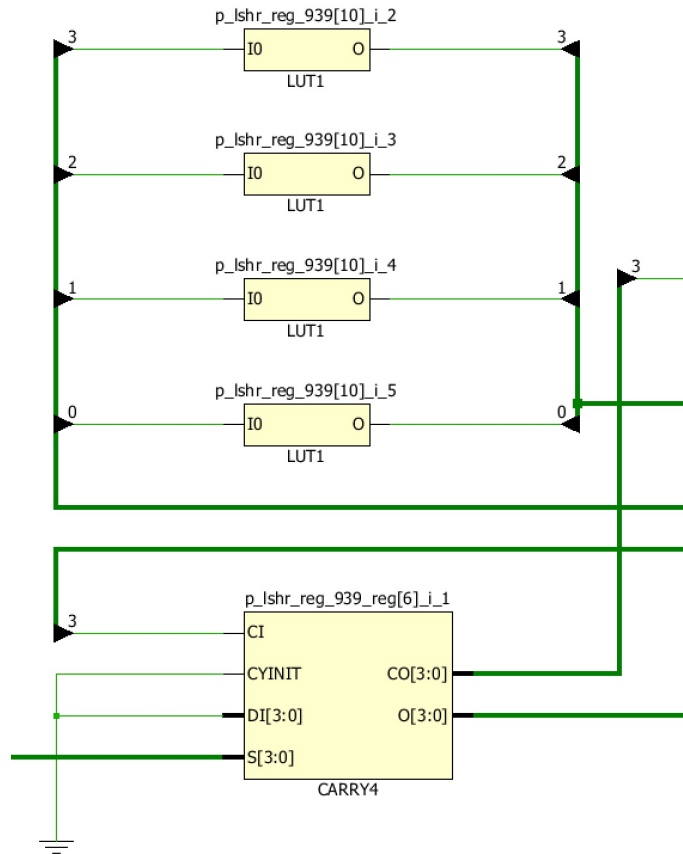


Figure 3.6: Vivado Block Diagram Example

The suite includes three main tools to utilize during a design; Vivado [29], Vivado HLS [30], and Software Defined System-On-a-Chip (SDSoC) [31]. SDSoC provides a tool which allows programming conventional C/C++ programs for use on the APU using a similar and easy to use Eclipse IDE. It also allows writing of HLS C/C++ in the same design which SDSoC can later automatically generate into hardware components for use on the FPGA, but which hardware details are generally invisible to SDSoC. Vivado HLS is the background tool (and can be used stand-alone) that SDSoC relies on in order to convert the user defined HLS C/C++ code into fully functioning hardware components. Users can enter the Vivado HLS tool to have greater control of the HLS C/C++ to VHDL/Verilog conversion, to the point of alternating the VHDL/Verilog code for any user requirement. The third tool, Vivado,

is also an excellent tool that is aimed at the overall hardware programming of the programmed components with the rest of the FPGA. While Vivado HLS allows building of individual hardware components, Vivado allows for the overall use of those hardware components, from logic routing, external clocks/resources, external component memory, data movement, etc. Vivado also allows for in-depth analysis of the hardware side, including power consumption and resources utilization that will be made use of in the research and discussed in future chapters.

This proposed system was mostly designed using the SDSoc tool as it allowed greater control of the entire design from the hardware accelerated components to the baseline linux operating system running on the ARM Cortex APU. Both Vivado and Vivado HLS were used in minor roles for additional information, small optimizations and some lower level tweaks [24].

3.4 HLS Background

Xilinx's SDSoc development environment employs a tremendously powerful HLS C/C++. This is a key method in order to design hardware functions at a much higher level than writing Verilog or VHDL from scratch. This language is part of Xilinx's High-Level Productivity Design Methodology which experts have shown to reduce project development time by 4X to 10X depending on several factors including, but not limited to the ability to leverage on pre-existing components such as IPs, projects compatibility for automated synthesis, and project complexity [24].

HLS C/C++ is mostly based on conventional C/C++ with a few hardware orientated requirements embedded within, including the use of PRAGMAS and ensuring that the code is hardware compatible/optimised. To ensure the code is written to optimise hardware, designers need to ensure many factors are considered, including using the properly sized data type (ex. Hardware can support off byte types, such as 4 or 10 bit vice the normal 8, 16, 32, 64 data types), specific hardware libraries, and allowing computational heavy algorithms to be conducted in independent series of loops. HLS C/C++ also includes limitations that prevent synthesis from occurring. These limitations include no operating system calls such as file reading/writing, the algorithm cannot contain unbounded or unambiguous loops, and the entire functionality of the code must be contained within the component [30].

Designers can place hardware instructions into their code which can guide various hardware aspects using SDSoc and HLS PRAGMAs. When using SDSoc PRAGMAs a designer can control the overall function and communications of the hardware component such as data movement from the software side to the hardware side, limit utilisation of board resources, and force certain memory types and communication buses to be used for the function. You can also choose from HLS PRAGMAs which are designed to control the internal aspects and optimizations of the hardware function, including controlling loop parallelism, inducing simultaneous execution operations, linking dependencies, setting required latencies, etc.

Once hardware orientated C/C++ is written with the desired PRAGMAs inserted, the SDSoc design suite is able to convert the written high level code into the hardware languages VHDL or Verilog. The designer is also required to provide the specific FPGA's board files which allows this hardware translation to be specifically compatible with the target board type. Without this board file, it would not be possible to produce an implementable design from the synthesised project.

3.5 Video Format

One of the greatest disadvantages that was discovered in the baseline reference design was the video format used. The original system was designed around a 24-Bit Per Pixel (BPP) RGB format, which was relatively easy to conduct computer vision operations with. This reference design on the other hand employs a YUY2 format which is in the family of YUV 4:2:2 formats, whereas each of the U and V values are only sampled every second pixel [32]. This compressed video format had an effective rate of 16 BPP, which taken with the reduced sampling rate meant the design already started off with a lower quality image than the original MATLAB RGB design. The issues that arose and the applied solutions for this non-ideal colourspace will be discussed in later chapters.

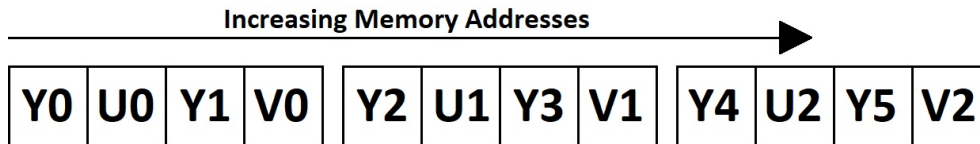


Figure 3.7: Memory Allocation of YUY2 Format [33](modified)

Given the provided colour-space details, and the resolution specifications of the embedded vision kits camera, the size of the image arrays that are used in this design can be determined. With the reference design, imaging from the camera is streamed to an array from the 1280x1024 pixel resolution camera. This stream contains a large amount of stride which equates to 2048, making the effective storage size of the image array 2048x1024 which results in a total of 2,097,152 pixels. With an effective depth of 16 BPP, this means that each image array consisting of a single frame takes up 4 Megabytes of storage space.

3.6 Computer Vision

Both the proposed and original crack detection solutions incorporate a substantial amount of computer vision to ensure accurate detection of the cracks. Specifically, the main element of these designs are an edge detector. While the original uses a Canny Edge detection filter, the research of this paper uses a Sobel Edge detection filter.

The Sobel edge detection filter is less complex than the original MATLAB design which was based on a canny filter. The first step of conducting the Sobel Edge detection on an image is usually to convert the RGB image to a grey scale image. However, with the design using a YUY2 format, the Y value already provides the intensity values, so no greyscale conversion is actually required. The next step is to apply the Sobel masks (shown in figure 3.8) in both the vertical and horizontal directions [34]. The vertical Sobel mask will detect vertical orientated edges and result in a picture as shown in figure 3.9 (b). Likewise, Figure 3.9 (a) is then passed with the horizontal sobel mask and shows the horizontal edges are shown in Figure 3.9 (c). The two edge detection orientation results are then combined using Equation 3.1. This results in the final produce of a grey-scale edge map.

-1	0	1
-2	0	2
-1	0	1

(a) Vertical Sobel Mask

-1	-2	-1
0	0	0
1	2	1

(b) Horizontal Sobel Mask

Figure 3.8: Sobel Mask Matrices



Figure 3.9: Application of the Sobel Masks [34]

$$E = \sqrt{E_x + E_y} \quad (3.1)$$

Where:

- E : Gradient Magnitude
- E_x : Convolution of Vertical Sobel Mask and Image
- E_y : Convolution of Horizontal Sobel Mask and Image

Although Equation 3.1 is ideal, it is slow to compute on hardware, so an alternative approximation is shown in Equation 3.2, and it is the one used by the proposed system's Sobel edge detection.

$$E = |E_x| + |E_y| \quad (3.2)$$

The original design's edge detection process on the other hand used a Canny Filter which was a much more involved and complex algorithm. The Canny Filter is a multi-step process which generally includes the following (or form of) [35]:

1. Remove the noise with a Gaussian filter
2. Find edge gradient and direction
3. Non-maximum suppression
4. Hysteresis thresholding

The first step of the Canny Filter is to smooth the image with a Gaussian filter to reduce noise. Several Sobel-like operators are then applied to the smoothed image which results in the edge gradients and directions of potential edges. The results are then scanned to ensure that potential candidates form local maximums, and if not, they are suppressed. The next process of hysteresis thresholding is based off gradient values minVal and maxVal . In this process, any candidates with gradient above the maxVal are considered edges, while any candidates with values below minVal are suppressed. For those candidates between minVal and maxVal , they are determined to be edges if they form a continuous connection of candidates that are above minVal to a candidate that has been determined an edge by being above maxVal . This process is visually shown in Figure 3.10. Candidate edge C can be confirmed an edge because it is connected to edge A, which is confirmed an edge by it being greater than maxVal . Candidate edge B does not connect to any candidate above maxVal , so it is not considered a confirmed edge.

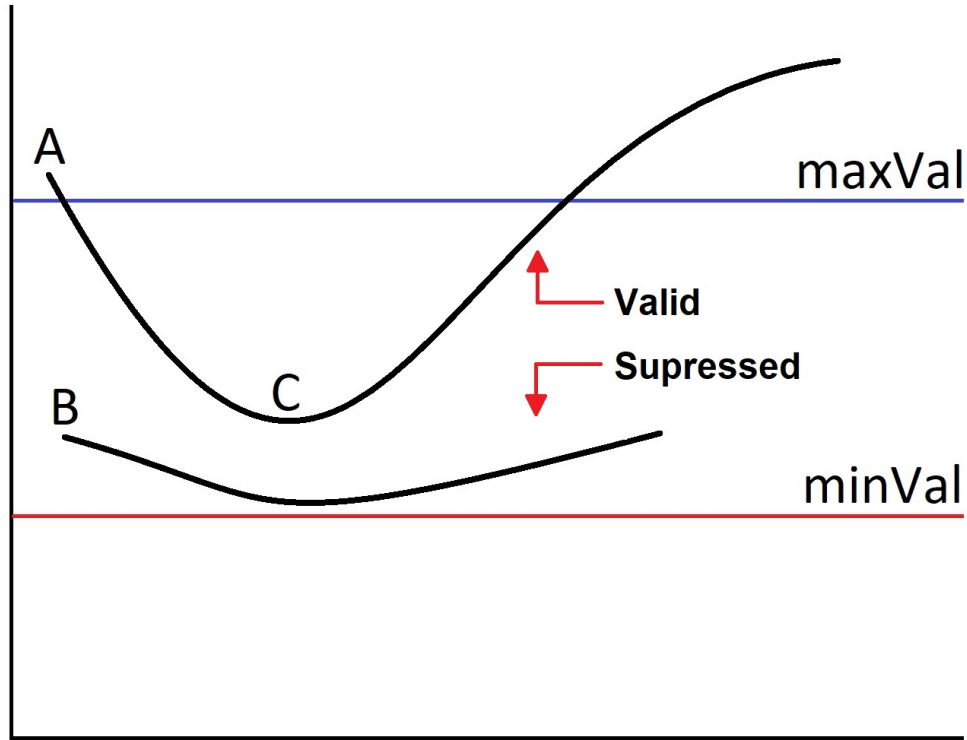


Figure 3.10: Hysteresis Thresholding [35](modified)

3.7 Crack Measurement Algorithm

The crack measurement algorithm is designed to estimate both the width and the length of detected cracks to give the user the ability to store this data later to ensure that crack propagation can be monitored [21]. Although this algorithm will not be incorporated into the design, it will still be used to evaluate the crack data resulting from the design.

Each of the cracks are classified based off their formation characteristics. Figure 3.11 shows the three different types that are used in our research and by the program. The first two types are classified as either Horizontal/Vertical or diagonal. It can be imagined that measuring these types of cracks are relatively straightforward and they are certainly possible with available software. As is indicated by the original work, these two classifications should cover approximately 94% of all cases [21].

Difficulties arise when the third type of classification is approached, the complex crack. With this type of crack, there are multiple branch cracks spreading in all directions, and even connecting back to itself several times as shown in Figure 3.11 (3) . This crack type is infeasible to measure as the length of the crack is not in a single continuous run.



(a) Vertical/Horizontal Crack

(b) Diagonal Crack

(c) Complex Crack

Figure 3.11: Crack Types [36], [37]

4 Solution Design

4.1 Overall Design Strategy

To ensure that the design could be built in manageable steps, this design was first fully implemented as a pure software solution. This means that first the entire design was built as C/C++ source code which ran through PetaLinux on the APU side. This method was chosen for several reasons, the first being that with early development there are many bugs and errors which result in numerous attempts at compiling the design. The difference between compiling for hardware and software is that a software design can compile in a few minutes while a hardware design can take over an hour to compile. The testing and configuration of the program was much quicker when the compile time is shorter as was the case in software. The second reason is that in using HLS C/C++, the hardware portion is based mostly from the software portion, so the effort is generally applicable when optimising the code for hardware. The third reason is to provide baseline of function run-times to calculate speed-ups. This is useful comparing the speeds, but also, it ensure things have actually moved to hardware. If developed onto hardware right away, there is potential that components unknowingly did not transfer to hardware, but having run-time references showing a large acceleration is a quick indicator that it had successfully moved over to hardware.

The interaction of the various components for the pure software solution is shown in Figure 4.1, where all crack detection elements are shown to be processed through Linux on the ARM Cortex-A9 APU. As shown, the hardware programmable logic is not used in this first solution.

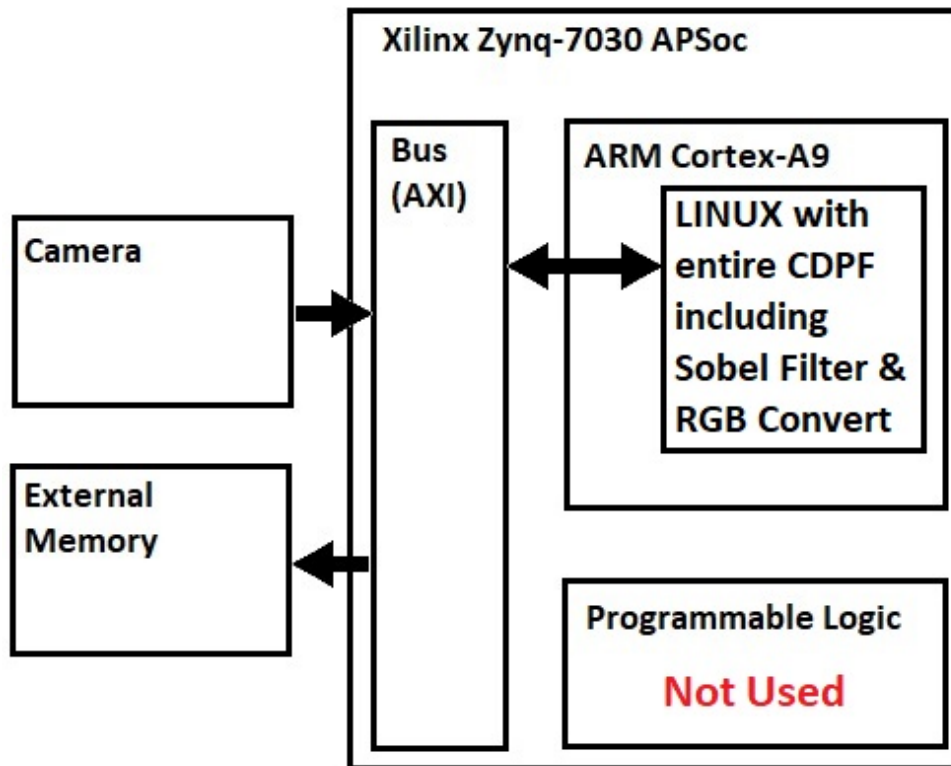


Figure 4.1: High Level Overview of Software Design

Once the software solution is working, certain components will be moved over to the FPGA to produce the final hardware accelerated design as shown in Figure 4.2. Section 4.3 discusses the steps that were taken to move the desired portions of the system over to hardware to produce the final product.

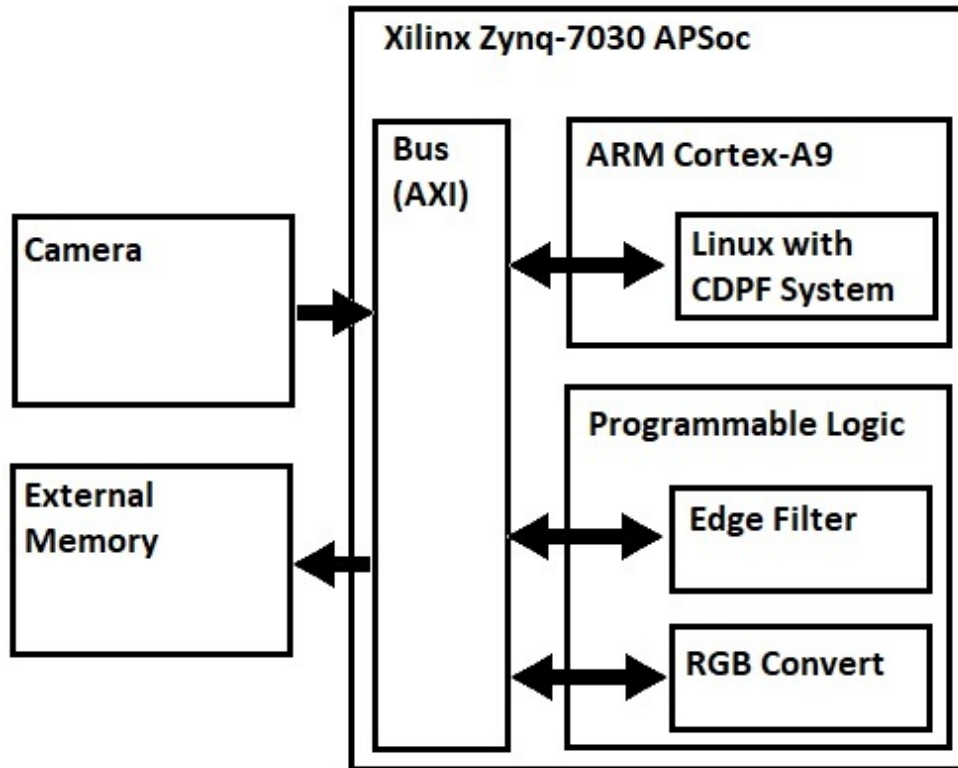


Figure 4.2: High Level Overview of Hardware Design

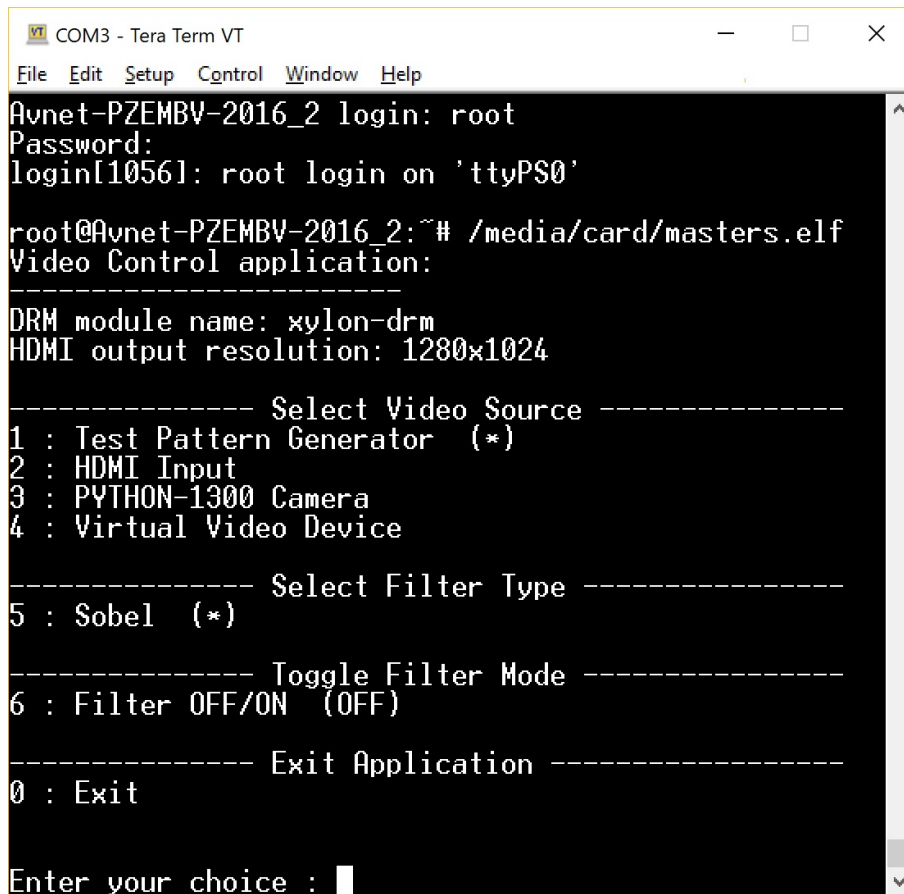
4.2 Software Implementation

To implement the redesigned algorithm, the code was manually converted to C/C++ from the original MATLAB based design. Although MATLAB offers plug-ins to automatically convert MATLAB code to C/C++, it was assumed that manually conversion would present a clearer and more optimised code. This held true for a couple of instances, for example, the elimination of the gray filter as it was already incorporated into the Sobel edge detection filter and the elimination of many of the video access functions which were replaced by stream reading processes.

Specifically built for the embedded vision kit that was purchased, AVNet also provided reference designs to serve as starting point for this design. That reference design came with a pre-configured version of Petalinux which in-

cluded a workable command-line interface (CLI), drivers for all components including the Python-1300-C camera, and C/C++ platform/library. The included Petalinux was designed to be operated on the APU side using the ARM Cortex processor. This OS was built for the Zynq boards and has the ability to communicate to the developer's hardware accelerated components on the FPGA side.

As indicated, this design operates from a software application through an ARM Cortex compatible OS called Petalinux. Developers are given access to Petalinux while it is running on the development board through the use of a USB UART port. While connecting a PC to this port using a conventional micro-USB cable, a virtual terminal is able to be established in order to receive OS information as well as input user commands into the CLI, as is shown in Figure 4.3 using the video control application.



```
COM3 - Tera Term VT
File Edit Setup Control Window Help
Avnet-PZEMBV-2016_2 login: root
Password:
login[1056]: root login on 'ttyPS0'

root@Avnet-PZEMBV-2016_2:~# /media/card/masters.elf
Video Control application:
-----
DRM module name: xylon-drm
HDMI output resolution: 1280x1024

----- Select Video Source -----
1 : Test Pattern Generator  (*)
2 : HDMI Input
3 : PYTHON-1300 Camera
4 : Virtual Video Device

----- Select Filter Type -----
5 : Sobel  (*)

----- Toggle Filter Mode -----
6 : Filter OFF/ON  (OFF)

----- Exit Application -----
0 : Exit

Enter your choice : █
```

Figure 4.3: Command-line Interface for the Video Control Application

The software CDPF portion of the design was mainly developed within two C/C++ source files with their associated header files. These files are shown marked with a red box in Figure 4.4 along with the rest of the files that made up the design. The files that were not marked on Figure 4.4 came with the reference design and are an essential part of the overall build that control aspects such as video or CLI.

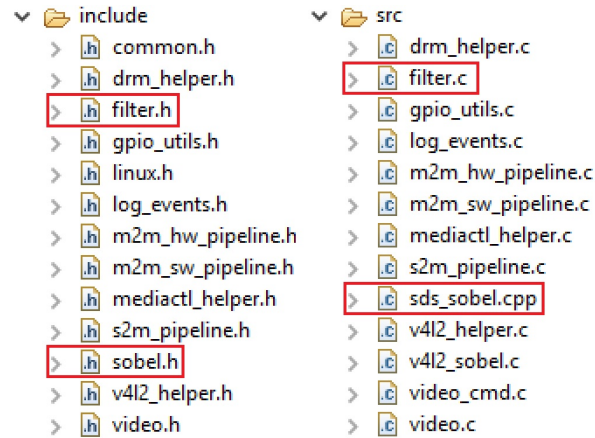


Figure 4.4: Design C/C++ Source Code and Header Files

The research consisted of several user designed components that work together with the rest of the system to allow for the overall operation of the design. The main sections of the design are listed as follows:

- Image Acquisition
- Sobel Edge Detection Filter
- Particle Map Initiation
- YUY2 to RGB Conversion
- Uncertainty Calculation
- Likelihood Calculation
- Resampling
- Analysis and/or Data Saving

Most of these sections have been based on the original MATLAB design and hence hold similar, if not exactly, the same role. With MATLAB coding running on a Windows PC being different than the C/C++ coding on a Petalinux device, there were several challenges that had to be overcome. Figure 4.5 shows the interaction and flow of these sections in the overall design. Each of the individual sections will be discussed in detail.

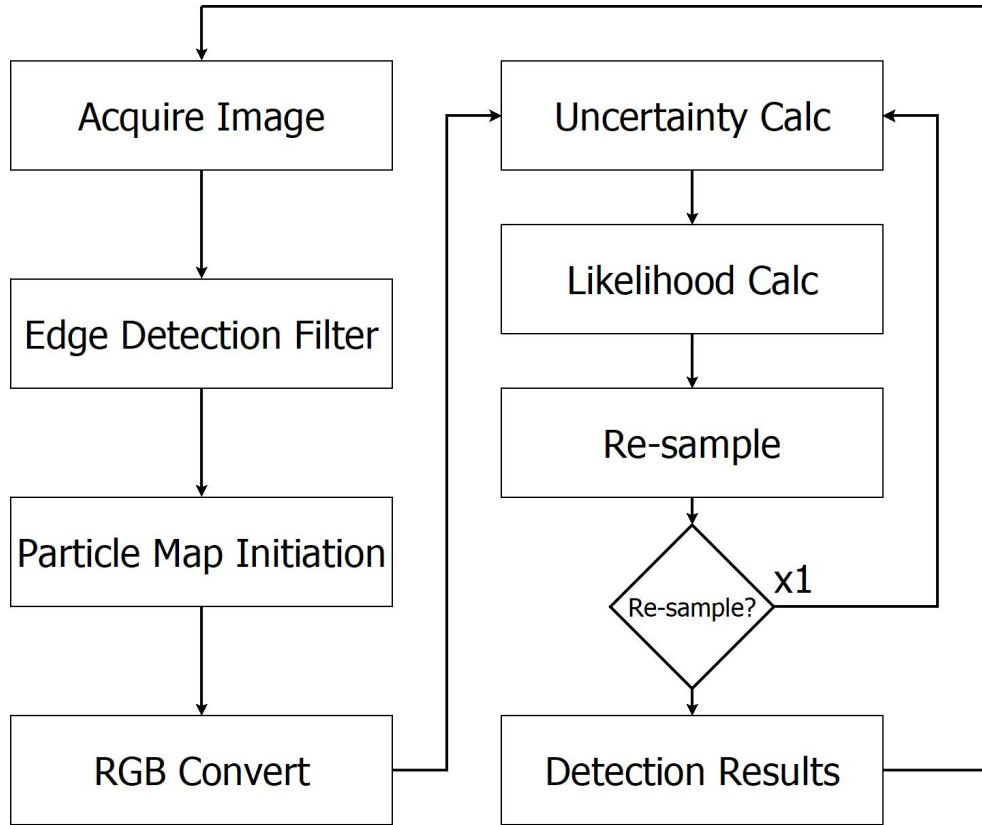


Figure 4.5: Software Design Component Interaction

4.2.1 Image Acquisition

The image used for detection is introduced to the system by either the on-board camera or reading a BMP file from the system's SD Card. For the camera, the reference design gave the framework to simply extract the image from a video stream, as shown in Figure 4.6. It also enabled the overlay of the crack associated particles back into the video stream once the crack has been detected. The obtained image is stored as local data array in the YUY2 data format to allow for local manipulation. Given the resolution of the camera (1280x1024) and the BPP (24-Bit), these data arrays would take up at least 3.9 MBs each.

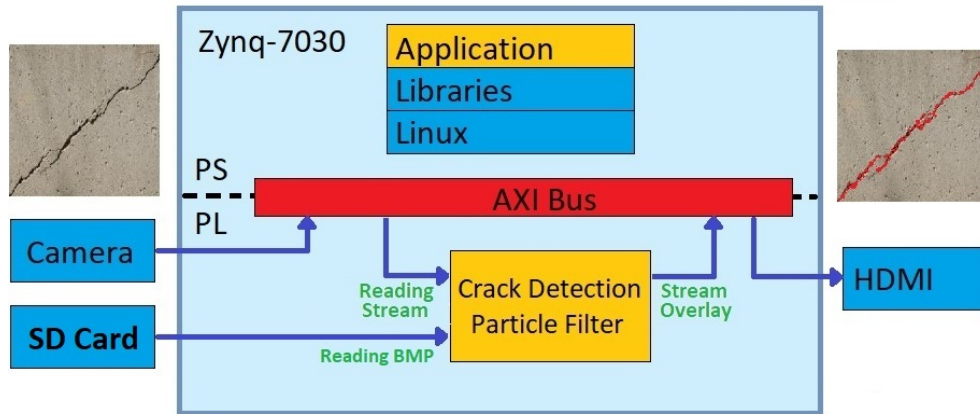


Figure 4.6: Hardware Block Diagram of Image Acquisition [38](modified)

For the image acquisition from an SD card, the CDPF reads the SD card directly as required in the form of a 24-bit RGB BMP file as shown in Figure 4.6. The video stream still exists, but is ignored as an input. Instead, the video stream is used only as an output to overlay both the crack associated particles and the entire image from the SD Card. The obtained image is still stored in a local data array in the format YUY2, but undergoes several extra steps including BMP image data extraction and YUY2 conversion, at which point it is used in the identical manner as the image originating from the camera.

4.2.2 Sobel Edge Detection Filter

The Sobel edge filter is a high speed edge detection algorithm that displays good detection results. This particular filter for the design accepts a YUY2 image array as an input, while providing the results as a greyscale intensity Edge Map for an output.

The original and this design's edge detection algorithms shared little in common for parameters. While the Canny filter allowed for adjustment of minval & maxval thresholds and the sigma value, the Sobel filter generally only allowed for the adjustment of the Sobel operator. The effects of adjusting the Sobel operator are shown in figure 4.7. As inferred from the images, increasing the Sobel operator increased the amount of potential edges detected, but also increased the noise and the amount of potentially incorrect edges.

The most generally used Sobel operator is the [1 2 1] map and will be used throughout the design as it was shown to produce good particle spread results.

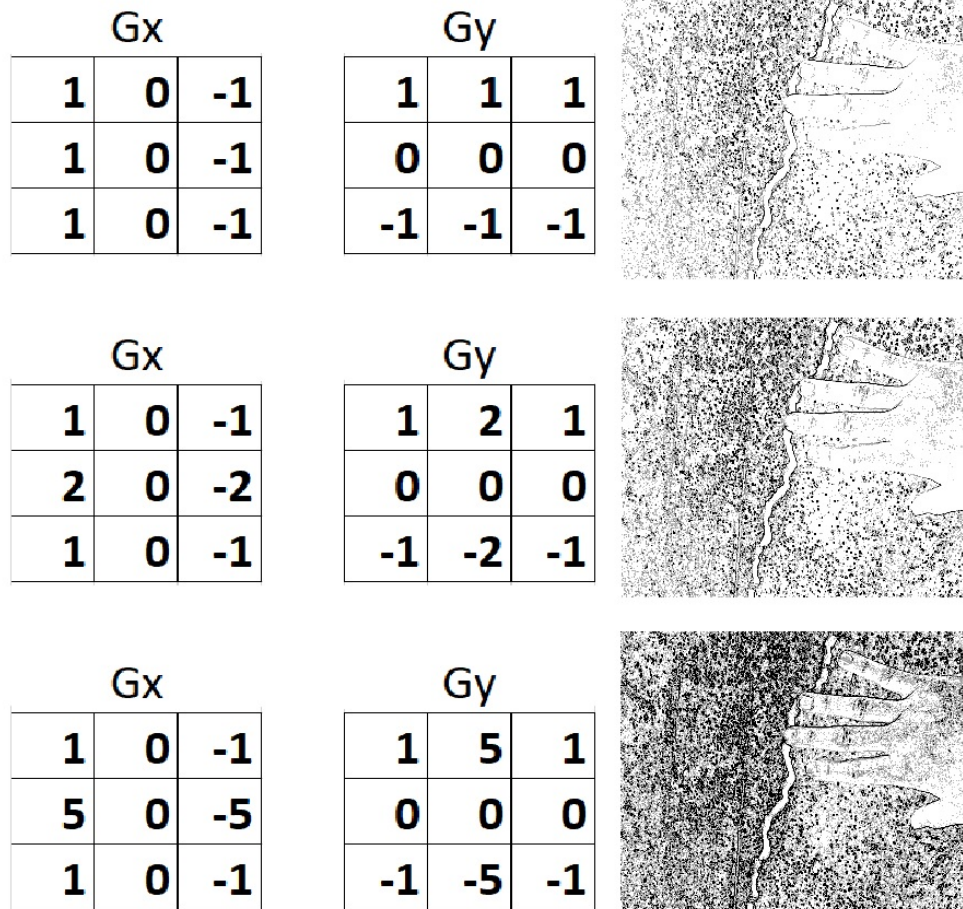


Figure 4.7: Effects of Sobel Operator Adjustments

4.2.3 Particle Map Initiation

The purpose of this section is to initiate the position and velocity matrix for the particles. The particles for this design are small points (the size a pixel), that are able to be moved throughout the image whose location is used to both store the current most likely locations of cracks, and also to allow for further probability estimations at those locations. In this design, 4,000 par-

ticles are uniformly distributed randomly throughout the image as a starting point while particle motion is initialised to 0, as is shown in Code Block 4.1. 4,000 Particles were chosen as an ideal compromise between performance and accuracy, as was also decided in the original design [21]. A two-dimensional array, shown in Equation 4.1 is used throughout the entire design and contains all data of the particles and therefore stores the location of the detected cracks.

$$particles = \begin{bmatrix} Y_{pos}^{particle \#1} & Y_{pos}^{particle \#2} & \dots & Y_{pos}^{particle \#4000} \\ X_{pos}^{particle \#1} & X_{pos}^{particle \#2} & \dots & X_{pos}^{particle \#4000} \\ Y_{vec}^{particle \#1} & Y_{vec}^{particle \#2} & \dots & Y_{vec}^{particle \#4000} \\ X_{vec}^{particle \#1} & X_{vec}^{particle \#2} & \dots & X_{vec}^{particle \#4000} \end{bmatrix} \quad (4.1)$$

Where:

- Y_{pos} : position of particle in Y direction (in Euclidean space)
- X_{vec} : velocity of particle in X direction
- Y_{pos} : position of particle in Y direction
- X_{vec} : velocity of particle in X direction

```

1 for (x = 0; x < NUM_PARTICLES; x++) {
2     particles[x][0] = rand() % height;
3     particles[x][1] = rand() % width;
4     particles[x][2] = 0;
5     particles[x][3] = 0;
6 }

```

Code Block 4.1: Particle Map Initialisation C/C++ Code

4.2.4 YUY2 to RGB Conversion

When extracting an image from the on-board camera, it is received in YUY2 format as per the configuration of the reference design. The image was decided to be converted from YUY2 to RGB for the following reasons:

1. Solution is based on the RGB colour-space, keeping the YUY2 colour-space would mean requiring different solution;

2. Original design was based on RGB colour-space, keeping with the same colour-space allows for an easier functionally equivalent comparison;
3. RGB colour-space is easier to analyse individual pixels, as each pixel has it's own colour data, unlike the shared blue-difference (Cb) and red-difference (Cr) chroma component values of the YUY2 colour-space pixels; and
4. RGB is easier to save in BMP file format.

The simplified code used to convert the YUY2 image into RGB is given in Code Block 4.2. The actual full code includes a loop so that a single call of the converter function converts an entire image frame, and therefore requires an unsigned char image array as the input, while producing an unsigned char image array as an output.

```
1 image.r = Y + (1.402525 * Cr);
2 image.g = Y - (0.343730 * Cb) - (0.714401 * Cr);
3 image.b = Y + (1.769905 * Cb) + (0.000013 * Cr);
```

Code Block 4.2: Simplified Code for YCbCr to RGB Conversion [39]

4.2.5 Particle Filter Calculations

The particle filter itself is comprised of three main calculations: the uncertainty Calculation, the likelihood calculation, and the re-sampling. the uncertainty calculation is designed to add noise to each of the particles' position and movement aspects. these particles with their noise are then passed through the likelihood calculation which uses both the edge map results from the Sobel filter and user defined target colour values to assign weights to each particle which follow a Gaussian distribution. These weights represent the likelihood of a particle being contained in a crack. The re-sample phase then assigns new particles at the locations of the old particles which were assigned high weights in the likelihood calculation. For more algorithmic design details, the exact C/C++ code used to implement the particle filter is shown in Appendix B.

4.2.6 Uncertainty Calculation

In this step of the process, randomness is added to both the position and velocity of each particle guided by two parameters, standard deviation of the position ($Xstd_{pos}$) and standard deviation of the velocity ($Xstd_{vec}$). While these parameters were also used in the original MATLAB based design, this design uses unique values to reflect the camera/image resolution, camera properties, and differences in the systems including colour-space converters. The specific parameters for the design are as follows:

$$Xstd_{pos} = 10$$

$$Xstd_{vec} = 50$$

The Uncertainty Calculation C/C++ code is shown in Code Block 4.3. As discussed, the first two elements of each $particles[x]$ array represent the x and y position, while the second two elements represent the x and y velocity. The randomness is calculated by multiplying the associated SD value (position or velocity) and a normally distributed random number, which is then added to each particle. As C/C++ does not natively support the generation of normally distributed random numbers, a custom function $randn(Mean, SD)$ was created to meet this requirement [40].

```

1  for (x = 0; x < NUM_PARTICLES; x++) {
2      particles[x][0] = particles[x][0] + (Xstd_pos * randn(0,1));
3      particles[x][1] = particles[x][1] + (Xstd_pos * randn(0,1));
4      particles[x][2] = particles[x][2] + (Xstd_vec * randn(0,1));
5      particles[x][3] = particles[x][3] + (Xstd_vec * randn(0,1));
6  }
```

Code Block 4.3: Uncertainty Calculation Code

With each particle having four individually tracked properties and each property requiring a distinct normally distributed random number, 4,000 particles result in the generation of 12,000 normally distributed random numbers for each Uncertainty Calculation. While this section contains only a small amount of code, the generation of 12,000 random numbers per Uncertainty Calculation is the bulk of computation in this step.

4.2.7 Likelihood Calculation

This step is used to determine the weight of each particle for the re-sampling phase. This weight is a combination of several factors including the correspondence to the target colour, proximity to a confirmed edge, and being within the actual image. The output of this step is an array of likelihoods associated with each particle. The Input/Output diagram is shown in Figure 4.8.

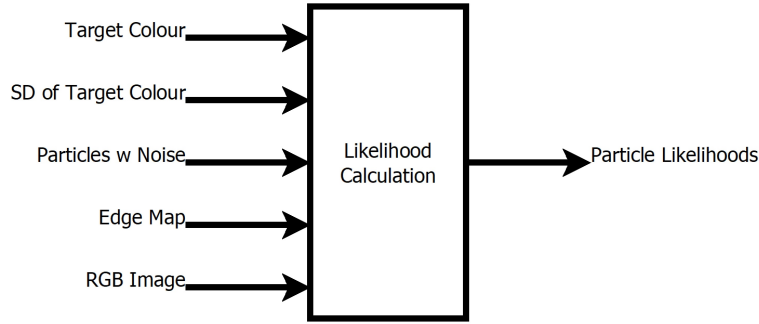


Figure 4.8: IO Diagram of the Likelihood Calculation

The likelihood calculation is based on Equation 4.2 from the original research [21]. It computes the particle weight of each particle based on a Gaussian distribution.

$$W_i = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\|h(rgb) - h(P_{xy})\|^2}{2\sigma^2}\right) \quad (4.2)$$

Where:

- W_i : weight assigned to each particle i
- σ : colour spread around crack, referred to as `Xstd_rgb` in this research. This design used a value of 5
- $h(rgb)$: colour spectrum histogram of the target colour of `RGB[5,5,5]`
- $h(P_{xy})$: histogram of colour intensity with pixels at column x and row y

4.2.8 Re-sampling

In this section of the algorithm, the system uses the likelihood array along with the particle positions to draw new particles towards the old particles with high weights. The Input/Output diagram for this calculation is shown in Figure 4.9. The re-sampling section occurs twice for each frame, along with additional Uncertainty and Likelihood Calculations, as it provides the best particle spread. If only a single re-sampling is done, the particles are too sparsely spread. If the re-sampling is done too many time, the particles start to condense away from the extremities of the crack and into the centre. These results are shown in Figure 4.10, where the red dots represent particle locations.

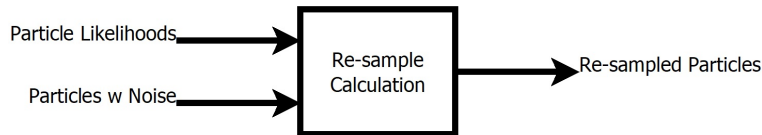


Figure 4.9: IO Diagram of the Re-Sample Calculation

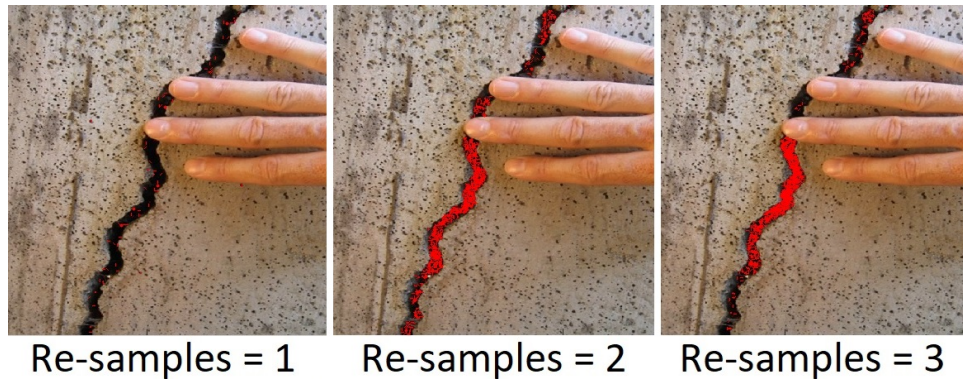


Figure 4.10: Particle Spread with Different Re-sample Amounts

The first step in the design's re-sampling stage is the weights are normalized as per Equation 4.3 [21].

$$\sum_{n=1}^N \pi^{(n)} = 1 \quad (4.3)$$

Where:

- N : number of particles
- π : particle sampling weights (Gaussian Distribution)

The particles probabilities are then calculated into a cumulative distribution function (CDF), where each element of the CDF is given by Equation 4.4 [41].

$$CDF_j = \sum_{i=1}^j \pi_i \quad (4.4)$$

For each old particle, given a random number r ($0 < r < 1$), the old particle is redistributed to the location of particle j , such that Equation 4.5 holds true [41].

$$CDF_{j-1} \leq r < CDF_j \quad (4.5)$$

4.2.9 Analysis and/or Data Storage

This section quickly touches on the data available after conducting crack detection. Although not extensive, provided below is a list of various data that can be extracted for future analysis, if required.

- camera image
- image with the particle map superimposed over
- particle locations
- particle likelihood data
- edge image
- run-time lengths of program and components

In this design, all the above data can be saved using an SD card installed in the embedded vision kit, where 273 frames can be saved per GB of SD Card space. In addition, all imagery data can also be displayed through the HDMI out port, while text-based data can be displayed on a virtual console through the UART port.

4.3 Hardware Implementation

4.3.1 General

With the design having been successfully implemented as a pure software version, the design was then altered to be implemented onto hardware. Segments of the algorithm were assessed for those that can benefit from parallel processing and programming onto the FPGA. Figure 4.2 provided a block diagram overview of the interactions between the various aspects of the board including the APU and FPGA. With components being moved to hardware, Linux continues to be the main processing point, and when required, Linux will send data over to the FPGA to be processed.

4.3.2 Sections for Hardware Acceleration

To determine which components to accelerate through hardware, the various software components first require to be ranked on their run-time compared to the total run-time of the entire design. Focus will be put on accelerating those components which take up the bulk of the design's run-time. To accomplish this, each software component must be tested on speed, in which the testing process will be explained in this section.

The software version tests of the crack detection algorithm running on the Zynq 7030 were conducted on Figure 4.11. This test image was used for this baseline run-time experiment, and will be used throughout the research to compare to the hardware accelerated run-times. For these baseline, timers were set-up throughout the code and run-times were captured in Table 6.6. For the timers, the `clock()` function was employed and relied on a comparison to the `CLOCKS_PER_SEC` definition to obtain accurate timings for the target APU. This baseline, and future run-time experiments, are based of the computational time required to analyse a single frame, and reflect the average of at least 10 frames timed. Sections such as Uncertainty Calc, Likelihood

Calc, and Re-sampling are actually conducted twice, as discussed in Subsection 4.2.8 for a single frame. As such, the time given for each of these sections are the total time accumulated for the entire frame.



Figure 4.11: Baseline Test Image

Section	Avg Time (s)
Edge Filter	0.369641
Particle Map Init	0.002001
RGB Convert	0.048694
Uncertainty Calc	0.011638
Likelihood Calc	0.002361
Resample	0.005665

Table 4.1: Computational Performance - Zynq-7030 (Software)

With the data compiled in Table 4.1 from the design running solely on

the ARM processor, the data was ranked based off the average combined time that each section took to run. This new ranking is shown in Table 4.2 and provided two algorithms that consume the bulk of the run-time. Factoring only the edge filter and the RGB conversion equates to 95.08% of the entire run-time of the design and hence will be the focus of the acceleration process.

Ranking	Section	Proportion of Total Time (%)
1	Edge Filter	84.01
2	RGB Convert	11.07
3	Uncertainty Calc	2.65
4	Resample	1.29
5	Likelihood Calc	0.54
6	Particle Map Initiation	0.45

Table 4.2: Computational Performance (Ranked) - Zynq-7030 (Software)

4.3.3 Edge Detection Filter Acceleration

Table 4.2 showed the edge detector taking up 84.1% of the total run-time. As discussed, the Sobel Edge detector is an excellent choice for an accelerated design as it provides good detection rates while superior computational performance. The Sobel filter has several characteristics that make it ideal for acceleration through parallelism. The entire function is contained in a double loop, meaning that given the non-recursive structure of the filter, multiple edge detection windows can be applied to the image simultaneously. Fortunately, the AVNET Embedded Vision Kit Reference Design [38] already provides a hardware optimized Sobel Filter to meet our exact requirements.

The Zynq-7030 contains enough resources to implement this algorithm with plenty to spare. The detailed post-implementation utilization report is shown in Table 4.3. As indicated in previous discussions, the IO blocks are by far the highest utilised resource, which is due to having both an input and output of the unsigned short size, which is relatively large data size at 16 bits each.

Resource	Total	Utilised	% Utilised
Look-up Tables	78600	261	0.33
LUTRAM	26600	2	0.01
FF	157200	325	0.21
BRAM	265	1.5	0.57
DSP	400	0	0.00
IO	150	34	22.67
BUFG	32	1	3.13

Table 4.3: FPGA Resources Utilised - Sobel Edge Filter

4.3.4 YUY2 to RGB Converter Acceleration

Due to the colour-space requirements indicated in subsection 4.2.8, the YUY2 to RGB Converter was required for the current research and was chosen to be accelerated due to consuming the second largest amount of total run-time. While not as high as the Sobel edge detector, the converter is still a substantial amount of the total run-time as 11.07%. In order to implement the converter into hardware, the resources shown in Table 4.4 were used.

Resource	Total	Utilised	% Utilised
Look-up Tables	78600	1689	2.15
LUTRAM	26600	36	0.14
FF	157200	2070	1.32
DSP	400	14	3.50
IO	150	26	17.33
BUFG	32	1	3.13

Table 4.4: FPGA Resources Utilised - YUY2 to RGB Conversion

Similar to the accelerated Sobel edge detection algorithm, this hardware accelerated YUY2 to RGB converter utilised a small amount of the Zynq 7030's resources. Similarities continue with the highest total percentage of a component utilised being 17.33% of the 150 IO Blocks which equate to 26 IO Blocks. 14 DSP blocks are consumed with this module, which is not surprising considering the component mostly involves floating point math calculations which the floating point math DSPs are certainly optimised to conduct.

4.4 Hardware Issues/Potential of Remaining Components

While conducting research to determine the potential of the other sections of the design being accelerated, a few issues arose that prevented acceleration without more research. These non-accelerated sections are covered in detail as follows.

Uncertainty Calculation: As described previously, this function makes use of 12,000 normally distributed random numbers in order to add uncertainty to the particles. This is the bulk of the computational effort used by this component. The issue is that HLS is unable to make certain library calls including the *rand()* function. It becomes increasingly more difficult to make a C/C++ normally distributed random number without using the *rand()* function as a base. Further issues arose when implementing a fully standalone Normally Distributed Random Number Generator (RNG) as HLS also does not allow unbounded loops which many RNGs rely on. Additionally, Xilinx does not provide a hardware random number generator, but has provided background info through several papers [42]. Although this issue could have been eventually overcome, the small 2.65% component run-time compared to the rest of the design would not equate to substantial acceleration.

Re-sample: The Re-sample section was actually able to be fully converted into hardware but was not feasible to implement into the design due to the amount of resources it consumed. As shown in Table 4.5, this component used too many IO blocks. While the board has enough IO blocks for this component alone to be implemented onto hardware, it does not have enough to also implement the YUY2 to RGB converter and the Sobel Edge Filter, meaning that one of the other hardware accelerated functions would have to be dropped, With the re-sampling section only taking up a small fraction of the total run-time, it did not make sense to implement onto hardware.

4.4. Hardware Issues/Potential of Remaining Components

Resource	Total	Utilised	% Utilised
Look-up Tables	78600	3591	4.57%
LUTRAM	26600	286	1.08%
FF	157200	4184	2.66%
DSP	400	28	7.00%
IO	150	108	72.00%
BUFG	32	1	3.13%

Table 4.5: FPGA Resources Utilised - Re-sample Section

The reason this component took up so much IO blocks was due to the size of the arguments, as shown in Code Block 4.4. The re-sample component required an input of a 32-bit (float data type) likelihood array and a 32-bit (float data type) array of random numbers while outputting an updated 16-bit (int data array) re-sample histogram. With the amount and size of the arguments required to make this component work, it is not feasible for hardware implementation.

```
1     int Resample(float *likelihood, float *random_numbers)
2     {
3         //Re-sample Code Here
4     }
```

Code Block 4.4: Re-sample Function Arguments

Likelihood Calculation: While the likelihood function has little preventing it from being hardware accelerated, due to the large size of the code and the small 0.54% total run-time amount, it was not worthwhile to accelerate.

Particle Map Initiation: Similar to the uncertainty calculation, this component also required the use of random numbers. The RNG issue coupled with the tiny 0.45% of the total run-time of the component, it was not worthwhile to accelerate.

As a result of these issues, the only two components to be moved to hardware will be the Sobel Edge Detector and the RGB converter. Given that those two components took up the vast majority of the total runtime, moving other components over would have little effect on computational performance.

4.5 Other Accelerations

Besides the use of hardware to accelerate the design, there were also a few software optimizations that resulted in substantial performance improvements. The most beneficial included a speed-up for the re-sample section through the use of a binary search method to replace the MATLAB *hist()* function used to create the histogram plot. Compared to a simple linear search, this specifically designed binary search provides an approximately 67% faster re-sample running on software alone. Another benefit of this binary search, is that as the particle count is known to be 4000, the worst-case time for the search can be calculated as $\text{Log}_2 4000 = 11.97$. If the re-sample component would have been feasible to implement onto hardware, this known fixed loop limit allowed ideal HW conversion, as it meets the requirement of having a known loop limit at compile time. This binary search loop limit must be manually adjusted when substantially changing the particle count.

5 Testing Methods

5.1 General

In order to confirm the results of this research, the proposed approach is tested in four distinct areas: detection and measurement accuracy, computational performance, physical footprint, and energy efficiencies. The first area, detection and measurement accuracy, is required to ensure a functional system meeting at minimum the detection accuracy given by the original design. The remaining three areas are highly desired properties for the system to possess, especially for implementation on resource limited platforms, such as UAVs.

To ensure consistency throughout the experiments, all accuracy and runtime analysis was conducted on Figure 4.11, the same as which was discussed in Subsection 4.3.2. This image demonstrates good crack characteristics (colour, visibility, and spread) and contains a distracting object in places (the hand) for better testing of adaptability. Furthermore, this picture will be set at the resolution of 1280 by 1024 pixels for all experiments, whether through the FPGA or MATLAB.

5.2 Detection and Measurement Accuracy

5.2.1 General

One of the key tests in this research was ensuring the accuracy of this research's design was comparable to the original design. Porting the original MATLAB version to the hardware version resulted in many of the components being redesigned and no longer operating the exact same way as before. The list of major redesign work that was conducted and therefore may alter the detection and measurement accuracy is listed as follows:

- Edge detector was changed from a Canny based edge detector to a Sobel edge detector for greater hardware efficiency and increased speed.
- A normal distributed RNG was created as C/C++ does not have a standard library for this function unlike MATLAB.
- Image processing involves both YUY2 and RGB vice just RGB used by the original design. Conversion from these colour-spaces will affect the image quality and therefore accuracy. This was due to the reference design set-up.
- Differences in the properties of the different cameras used in both solutions.
- Many of the parameters were adjusted due to the reasons listed above, or simply to obtain increased performance, and as such, will affect the detection rate.

While these changes were required to allow the system to operate on the new target platform, this resulted in different accuracy calculations compared to the original design. The accuracy goal for this research's design will be to achieve a comparable accuracy of the original design. Detection accuracy will be measured using the same method proposed in the original design [21]. In this, the particle data will be used in order to produce the measurements of the crack in terms of width measured in mm. Width will be the primary metric to determine the level of accuracy between the solutions, as the error rates become more substantial when comparing the relatively small width compared to the length. The calculations to determine the width of the crack are not part of the board's design, as the design scope is limited to crack detection, and as such, these calculations were accomplished separately.

The test compared only two platforms, the EVGA S17 and the Zynq-7030 (Hardware). The Acer Netbook does not need to be tested as it is being manually inserted the same image and also running the same MATLAB code, it is assumed to produce the same accuracy rate as the EVGA S17. The Zynq-7030 (Software) is also not tested as it should also have the same accuracy rate as the Zynq-7030 (Hardware), but also does not need to be tested as it was not indented as the final product.

Each test variety was ran 10 times each for the following conditions:

- Zynq-7030 (Hardware) - 4000 particles
- Zynq-7030 (Hardware) - 8000 particles

- EVGA MATLAB - 4000 particles
- EVGA MATLAB - 8000 particles

The test compared the detected cross-sectional thickness of 3 user chosen sections of the crack, as shown in Figure 5.1. These sections are identified by the y coordinate value that the cross-section runs along, section 1 is located along $y = 110$, section 2 is along $y = 495$, and section 3 is along $y = 790$. As this crack is considered to be classified as vertical, Equation 5.1 was used to determine the crack width at those locations, just as proposed in the original research [21]. Equation 5.1 works by comparing the two most distant particles that are along the cross-section of the crack to determine the width in pixels. Equation 5.1 is valid only for vertical cracks, where different equations are required to measure cracks of horizontal or diagonal orientation as detailed in [21].

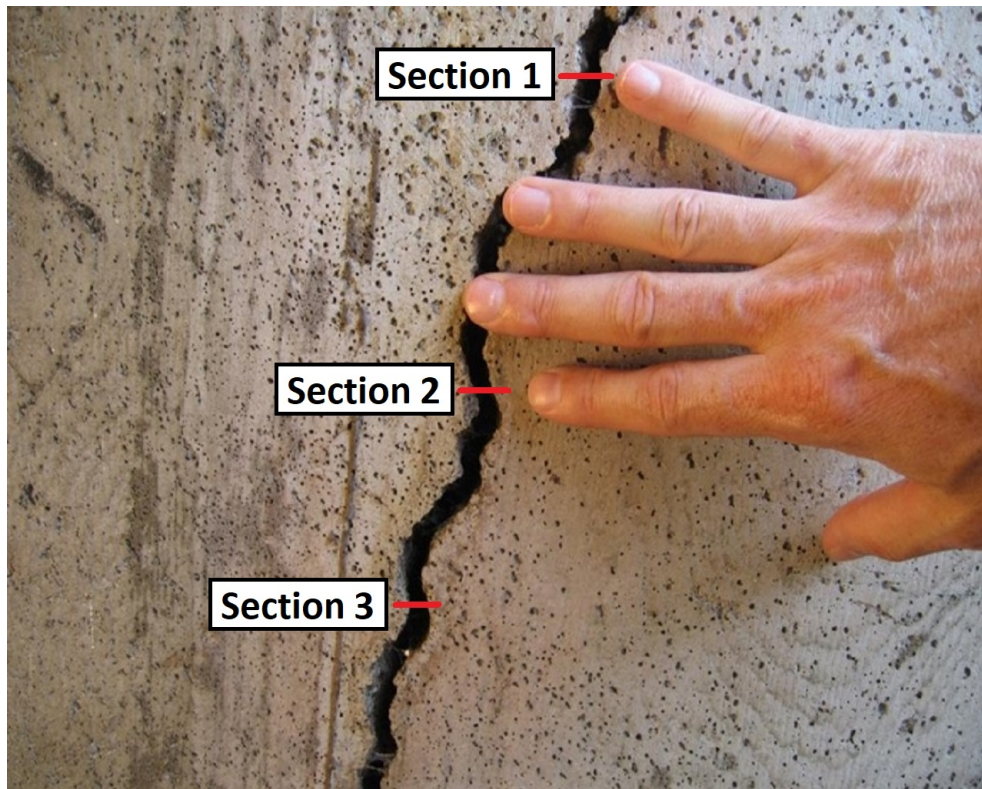


Figure 5.1: Test Crack Sections

$$w_p = x_2 - x_1 \tag{5.1}$$

Where:

- w_p : crack width in pixels
- x_2/x_1 : horizontal locations of the extremity particles within the target sub-blocks.

The true width of the crack cross-sections were measured in pixels and shown in Table 5.1. A camera calibration parameter of 1.19375 was then calculated in accordance with Reference [43] and then factored into the thickness measurements to determine the thickness in millimetres, which is cross-checked using a vernier calliper. As shown in the table, each chosen cross-section of the crack measures between 7.16-9.55mm. These are the measurements that will be used in comparing the two solutions in Chapter 6.

Crack Location	Thickness (mm)
Section 1 (Y110)	9.55
Section 2 (Y495)	7.16
Section 3 (Y790)	7.16

Table 5.1: Test Crack Measurements

The results will be in the form of the following two metrics:

- Avg Uncertainty (mm): Detected width difference from true measurement in mm.
- Avg Error (%): Error percentage comparing true crack width to detected crack width.

The test will be considered successful if the accelerated hardware design produces measurement data that is at least functionally equivalent to the measurements of the original MATLAB design in terms of cross sectional thickness.

5.2.2 Camera vs SD Card

In order to compare the results of the proposed design to that of the original, it was required to ensure that identical imagery was being fed into both solutions. This makes the camera solution not feasible for direct comparison. To elevate this issue, both solutions were fed the same image through manual implementation of a BMP file. For the FPGA, the image file was loaded through the SD Card which was previously shown in Figure 4.6.

While the camera itself was not used in this accuracy test, the full design is still fully functional. Figure 5.2 shows the FPGA using the camera in a test environment. These results come from the FPGA design fully implemented and obtaining actual imagery from it's on-board camera. The results show excellent particle spread within all five test cracks.

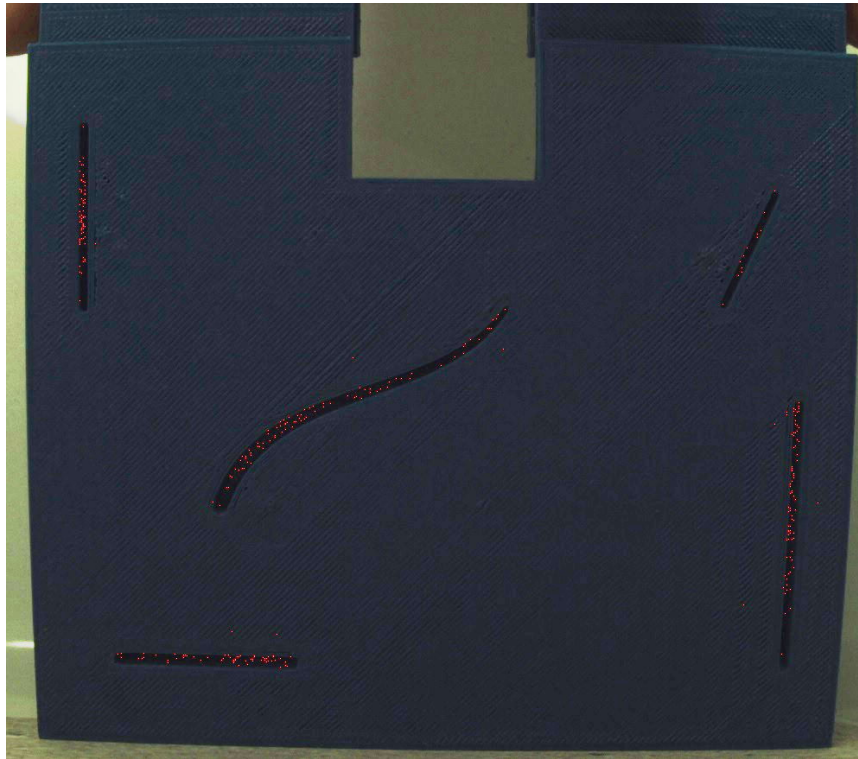


Figure 5.2: Results from Camera

While the camera is an important component in the design, it is an accept-

able testing alternative to allow manual feeding of a BMP file. It shows the flexibility of the design with crack detection being conducted from multiple video sources and not locked onto a single camera type.

5.3 Computational Performance

To meet the computational performance objectives of this research, an increase in crack detection speed must be observed compared to the original design. The original design was programmed through MATLAB on a conventional Acer Netbook, which was installed on a ground mobile robot. Usually both platforms would have cameras of different resolutions, but to ensure testing consistency, images of the same resolution were manually fed into both systems as opposed to obtaining the image from their cameras. This allowed the performance metric of runtime alone to have been used as a comparison, vice having to resort to a resolution-neutral metric such as MegaPixels computed per second.

Computational performance of the crack detection particle filter will be compared between the original MATLAB implementation, the embedded vision kit software version, and the full accelerated hardware implementation on the Xilinx Zynq Board. As the Netbook is meant for portability rather than power, a performance laptop is also included as a benchmark. The 4 test cases are as follows

- Acer Aspire ONE D250 - Atom N270 1.6 GHz - 1 GB DDR2 RAM - MATLAB
- EVGA S17 - Intel Core-i7 6820HK 2.70 GHz - 32 GB DDR4 RAM - MATLAB
- Xilinx Xynq 7030 - ARM Cortex-A9 1.0 GHz - 1 GB DDR3 RAM - Software
- Xilinx Xynq 7030 - ARM Cortex-A9 1.0 GHz - 1 GB DDR3 RAM - Hardware

The computational performance test will be passed if the Zynq-7030 (Hardware) obtains a lower runtime for the same image compared to that of the Acer Netbook computer employed for crack detection on a mobile robot.

5.4 Footprint

In many applications, such as UAV implementation, physical footprint is a key metric to show suitability. The following metrics are compared between the platforms:

- Weight (g)
- Dimensions (LxWxH) (mm)
- Volume (mm²)

Weight is a useful metric in many cases as a lower weight reduces the payload burden on mobile or aerial platforms and is generally preferred. The dimension and volume metrics are also beneficial aspects of a platform, where a smaller platform taking up less volume is generally more desirable, especially when with a restricted area where smaller dimensions are actually required.

While the dimension and volume will be roughly measured using a standard mm ruler, the weight will be assessed using a standard utility scale. The weight will not include items such as batteries or power adapters to ensure consistency of the measurements, but also to factor in the likely possibility that the platforms could be wired into the pre-exist mobile or aerial vehicles and run on their batteries without the need for internal batteries or 120V power adapters.

This test is considered successful if it can be shown that the embedded vision kit possesses beneficial footprint aspects compared to the other alternative platforms. Although no specific criteria are put in place for this test, it will be argued that one platform holds an ideal footprint over the other two.

5.5 Energy Analysis

Energy efficiency is tested by measuring the entire system's power consumption. In order to properly show improvement in power consumption over the original design, energy efficiency will be measured using the following two metrics:

- Total Power Consumption (Watts)

- Power Efficiency (Frames Per Watt)

The total power consumption is a useful metric as it allows the solutions to be compared on the energy requirements to simply employ the platform. Even though one option might be more power efficient, the overall energy requirements might make it infeasible for many implementations such as UAVs.

In order to measure the total power consumption in Watts, an energy meter shall be used to monitor the wattage consumed by the device at various operating times. The energy monitor chosen for this task is the P3 International Kill-A-Watt P4400 model as shown in Figure 5.3. This meter connects between the device desired to be measured and a standard 120V household electrical outlet. The meter is able to measure several useful measurements including voltage, current, power, line frequency, and power factor. For the purposes of this work, there is a focus on the power (watts) as the main metric being measured using this tool. The primary reason for this model being the chosen solution for this testing is the 0.2% accuracy on measuring this quantity. Other meters on the market have error margins as high as 3%, which is much higher than the Kill-A-Watt.



Figure 5.3: Kill A Watt (R) P4400 Energy Meter

The second metric of power efficiency is useful in comparing the power consumption based on performance. Frames Per Watt (FPW) is utilised for this purpose and is calculated using Equation 5.2. Given that all frames from

the various platforms are set at a 1240 by 1024 resolution, this metric will be consistent throughout the test.

$$\textit{Frames per Watt} = \frac{3600}{\textit{runtime(seconds per frame)} \cdot \textit{Watts}} \quad (5.2)$$

For this test, all batteries will be removed from the laptops, so that power from or to the battery will not alter the results. The laptop screens will also be turned off during this test, as they are usually not used during actual implementation. Most other non-required items will be turned off as well, like Bluetooth and WiFi. This test will be considered successful if the FPGA has the largest power efficiency measured in Frames per Watt compared to the other platforms.

6 Results and Discussion

6.1 Detection Accuracy

The accuracy section will be started with a brief look at the performance of the edge maps. The results from the Zynq-7030 (Hardware), which is Sobel filter based, is shown in Figure 6.1. The resulting edge map from the MATLAB design, which is Canny filter based is shown in Figure 6.2. The Sobel filter outputs a grey-scale edge map, while the Canny filter produces a black & white edge map.



Figure 6.1: Edge Image from the Sobel Edge Detection



Figure 6.2: Edge Image from the Canny Edge Detection

While the Sobel filter does output a grey-scale edge map, the Likelihood Calculation end up eliminating non-maximum values, which results in a large reduction of noise. This refined Figure is shown in 6.3, and has been inverted for easier visual comparison to the Canny filter. As expected, even after the refinement, the Canny filter still produces cleaner edges with less noise.



Figure 6.3: Refined Edge Image from Sobel Edge Detection

These edge filters are a key component to producing the final crack detection map for both the Zynq-7030 and the MATLAB designs. The crack detection results from the Zynq-7030 is shown in Figure 6.4, while the results from the original MATLAB design is shown in Figure 6.5. Both solutions show similar ability to fill the length of the crack, while differences are seen in the ability to fill the thickness of the crack. The MATLAB design (Canny filter) in Figure 6.5 shows reasonable spread across the width of the crack, but the Zynq-7030 (Sobel filter) shows an even better thickness spread. This is likely due to one (or both) of the following reasons:

- The parameters (colour target, colour spread, particle position SD, and particle velocity SD) of the original MATLAB design were kept at default values while the Zynq-7030 had to be re-tuned due to different edge filter, and hence was specifically tuned for optimal results on subject test image.

- The Canny Filter produced distinct edge boundaries which may limit particle movement, resulting in thinner particle spread. The Sobel filter produced more "fuzzy" boundaries, which may have allowed for greater tendency for spread creep, and hence making the detection thicker.

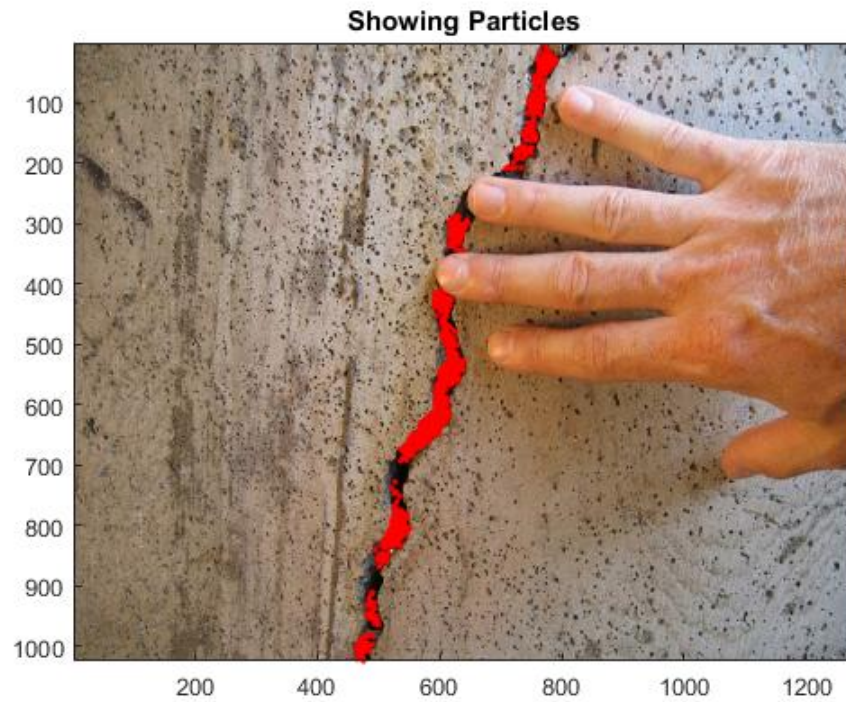


Figure 6.4: Crack detection using the Sobel Edge Detection (FPGA Design)

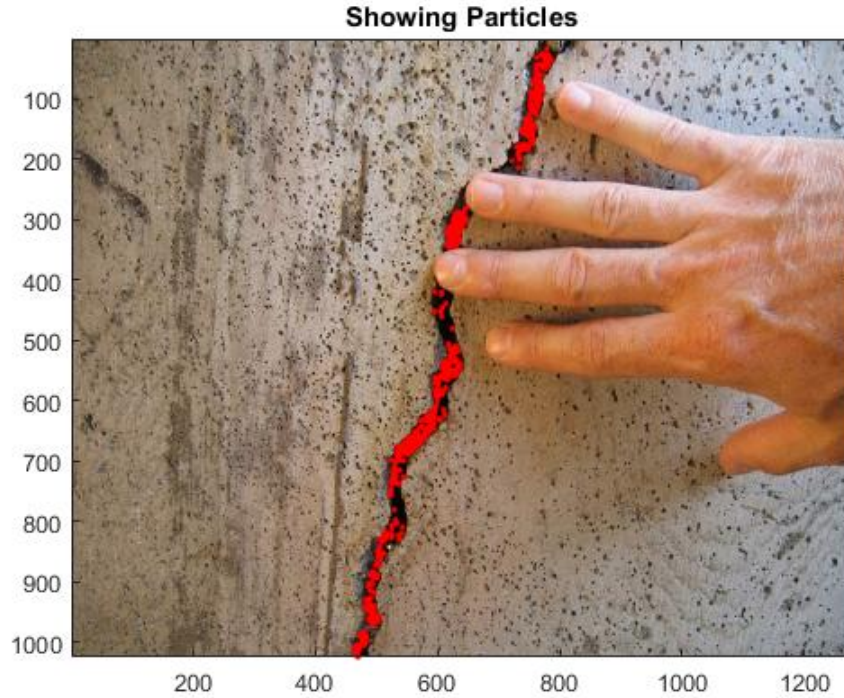


Figure 6.5: Crack Detection using the Canny Edge Detection (Original Design)

The accuracy results of the crack measurements for the MATLAB and Zynq-7030 solutions using 4,000 and 8,000 particles are shown in Tables 6.1, 6.2, 6.3, and 6.4. Both the average uncertainty and the average error are calculated based on the crack detection results against the measured cross-sectional widths given in Table 5.1. Both solutions show significant measurement uncertainty, but perform substantially better at the higher 8,000 particle count, which is expected.

6.1. Detection Accuracy

Crack Location	Avg Uncertainty (mm)	Avg Error (%)
Section 1 (Y110)	0.5458	13.75
Section 2 (Y495)	0.2804	20.00
Section 3 (Y790)	0.6209	16.67
Overall	0.4823	16.81

Table 6.1: Detection Accuracy - Zynq 7030 - 4,000 Particles

Crack Location	Avg Uncertainty (mm)	Avg Error (%)
Section 1 (Y110)	0.1252	6.25
Section 2 (Y495)	0.1051	11.67
Section 3 (Y790)	0.2203	11.67
Overall	0.1502	9.86

Table 6.2: Detection Accuracy - Zynq 7030 - 8,000 Particles

Crack Location	Avg Uncertainty (mm)	Avg Error (%)
Section 1 (Y110)	2.4284	31.25
Section 2 (Y495)	1.8076	31.67
Section 3 (Y790)	0.9463	21.67
Overall	1.7274	28.19

Table 6.3: Detection Accuracy - MATLAB - 4,000 Particles

Crack Location	Avg Uncertainty (mm)	Avg Error (%)
Section 1 (Y110)	0.2053	21.25
Section 2 (Y495)	1.9027	23.33
Section 3 (Y790)	0.1001	23.33
Overall	0.7360	22.64

Table 6.4: Detection Accuracy - MATLAB - 8,000 Particles

Table 6.5 shows the difference in accuracy rates of the two solutions. The Zynq-7030 had shown great results compared to the original MATLAB based solution. At 4,000 particles, the Zynq-7030 provides a 16.81% accuracy, which is 11.38% better than the MATLAB design. At 8,000 particles, the Zynq-7030 performs even better at a 9.86% accuracy, a 12.78% improvement over the orig-

inal design, for this test case.

Particle Count	Zynq-7030 Avg Error (%)	MATLAB Avg Error (%)	Δ Error (%)
4,000	16.81	28.19	+11.38
8,000	9.86	22.64	+12.78

Table 6.5: Detection Accuracy - Zynq 7030 vs MATLAB

Given the resulting accuracy rates given in Table 6.5, the detection accuracy of the Zynq-7030 is considered to successfully pass the test. While the Zynq-7030 has successfully passed for this particular test image at the determined parameters, it is likely that the MATLAB based design could out-perform in other sceneries based off different crack types, image/camera properties, or parameters. As such, while the Zynq-7030 drastically out-performed the MATLAB design in this scenario, the goal of this test is not to prove a sustainable improved accuracy rate of the accelerated solution over the original design, but rather it is functionally able to detect cracks comparable to the original design, which this example has suggested.

6.2 Computational Performance

The run-time performance of the Zynq-7030 (Software) is shown in Table 6.6. This is essentially the same data that is given in the hardware implementation chapter previously, but is reiterated to show the improvements that are being yielded converting both the Edge detector and RGB converter to hardware. This software version already shows a good total run-time at 0.44 seconds per frame.

6.2. Computational Performance

Section	Avg Run-time (s)	Std Dev (s)	Time (%)
Edge Filter	0.369641	0.000225	84.01%
Particle Map Init	0.002001	0.000010	0.45%
RGB Convert	0.048694	0.001640	11.07%
Uncertainty Calc #1	0.005820	0.000032	1.32%
Likelihood Calc #1	0.001313	0.000029	0.30%
Resample #1	0.002785	0.000039	0.63%
Uncertainty Calc #2	0.005818	0.000022	1.32%
Likelihood Calc #2	0.001047	0.000051	0.24%
Resample #2	0.002881	0.000040	0.65%
Total	0.440000	-	100.00%
FPS	2.27		

Table 6.6: Computational Performance - Zynq 7030 (Software)

Table 6.7 shows the run-time of both the edge detector and the RGB converter after being converted to hardware and showed an incredible speed-up. While previously consuming 95.08% of the total time, these two components are able to drastically improve the run-time of the entire program. With the program previously running at 2.27 FPS utilising only software components, with the accelerated hardware components, the program is now able to run at 15.00 FPS.

Section	Avg Run-time (s)	Std Dev (s)	Time (%)
Edge Filter	0.030747	0.000043	46.10%
Particle Map Init	0.002004	0.000015	3.00%
RGB Convert	0.014184	0.000016	21.27%
Uncertainty Calc #1	0.005823	0.000029	8.73%
Likelihood Calc #1	0.001316	0.000034	1.97%
Resample #1	0.002818	0.000033	4.23%
Uncertainty Calc #2	0.005822	0.000034	8.73%
Likelihood Calc #2	0.001034	0.000028	1.55%
Resample #2	0.002950	0.000041	4.42%
Total	0.066698	-	100.00%
FPS	15.00		

Table 6.7: Computational Performance - Zynq 7030 (Hardware)

Table 6.8 shows the speed-up percentages of each of the accelerated com-

ponents. The edge detector increased by a factor of 1210.19%. As that previously made for 84.01% of the entire program run-time when on software, this huge speed-up benefits the entire program nearly as much. With the RGB converter also being sped-up by 343.31%, it can be confirmed that both of these component have been successfully accelerated using hardware.

Section	Software Run-time (s)	Hardware Run-time (s)	Speed-up (%)
Edge Filter	0.3696411	0.0307473	1202.19%
Particle Map Init	0.0020008	0.0020036	0%
RGB Convert	0.0486935	0.0141835	343.31%
Uncertainty Calc	0.0058202	0.0058232	0%
Likelihood Calc	0.0013134	0.001316	0%
Resample	0.0027846	0.0028183	0%
Uncertainty Calc	0.005818	0.005822	0%
Likelihood Calc	0.0010473	0.0010342	0%
Resample	0.0028808	0.00295	0%
Total	0.4399997	0.0666981	659.69%

Table 6.8: Computational Performance - Zynq 7030 (Software) vs Zynq 7030 (Hardware)

With the hardware accelerated design being confirmed, the MATLAB versions will be analysed next to gather total run-times. The first run-time shown in Table 6.9 represents the results from running the original design on the Acer Netbook. As expected, the edge filter stands out as one of the most computationally heavy items in the list, especially considering it is Canny filter based.

6.2. Computational Performance

Section	Avg Run-time (s)	Std Dev (s)	Time (%)
Edge Filter	1.288425	0.005714	67.36%
Particle Map Init	0.002863	0.000028	0.15%
RGB Convert	0	0	0%
Uncertainty Calc #1	0.004274	0.000528	0.22%
Likelihood Calc #1	0.281040	0.002697	14.69%
Resample #1	0.005967	0.000150	0.31%
Uncertainty Calc #2	0.003744	0.000140	0.20%
Likelihood Calc #2	0.320083	0.005541	16.73%
Resample #2	0.006316	0.000200	0.33%
Total	1.912711	-	100.00%
FPS	0.52		

Table 6.9: Computational Performance - Acer Netbook

The results of the Acer Netbook are relatively slow compared to even the Zynq-7030 (Software) version. This is most likely due to the following reasons:

1. Unlike the Zynq which runs as straight C/C++ source code on an extremely minimal Petalinux OS, the MATLAB experiments run on high level MATLAB code through the actual MATLAB program on Windows 7/10 OS. These programs are full of features and tie up many computer resources which result in a slow run time.
2. The Sobel filter is much simpler and faster than the Canny filter.
3. It is not exactly clear why the likelihood calculation is so much faster on the Zynq's APU then through MATLAB, but through analysis, it seems that the entire component is generally slower throughout its entire operation and not slowed down by a single function or operation in the component. If anything, this would be due to the component running on a low level C/C++ source code vice MATLAB.

Table 6.10 shows the speed-up of the accelerated design compared to that of the netbook. The accelerated designed scored an overall speed-up of 2867.71%, which is quite remarkable and meets the criteria for the computational test to be marked as successful.

6.2. Computational Performance

Section	Netbook Run-time (s)	Zynq 7030 (Hardware) Run-time (s)	Speed-up (%)
Edge Filter	1.2884253	0.0307473	4190.37%
Particle Map Init	0.0028634	0.0020036	142.91%
RGB Convert	0	0.0141835	0.00%
Uncertainty Calc	0.0042739	0.0058232	73.39%
Likelihood Calc	0.2810397	0.001316	21355.60%
Resample	0.0059673	0.0028183	211.73%
Uncertainty Calc	0.0037435	0.005822	64.30%
Likelihood Calc	0.3200827	0.0010342	30949.79%
Resample	0.0063155	0.00295	214.08%
Total	1.9127113	0.0666981	2867.71%

Table 6.10: Computational Performance - Acer Netbook vs Zynq 7030 (Hardware)

To further show the acceleration potential of the FPGA board acceleration, refer to the results of Table 6.11 to show the timings from the EVGA S17 Gaming Laptop. Similar to all other versions of crack detection systems analysed, again the edge filter remains at the top of the list for computational time. But the results from this computer are much faster than the Acer Netbook, which was expected.

Section	Avg Run-time (s)	Std Dev (s)	Time (%)
Edge Filter	0.085978	0.003390	73.25%
Particle Map Init	0.000300	0.000079	0.26%
RGB Convert	0	0	0%
Uncertainty Calc #1	0.000464	0.000055	0.40%
Likelihood Calc #1	0.013994	0.000507	11.92%
Resample #1	0.000562	0.000041	0.48%
Uncertainty Calc #2	0.000503	0.000070	0.43%
Likelihood Calc #2	0.014894	0.000910	12.69%
Resample #2	0.000684	0.000044	0.58%
Total	0.117379	-	100.00%
FPS	8.52		

Table 6.11: Computational Performance - EVGA S17

Although the EVGA S17 shows excellent improvement over both the Acer

Netbook and the Zynq-7030 (Software), it still runs slower than the hardware accelerated design of the Zynq-7030 (Hardware) as shown in Table 6.12. The hardware design is able to out perform the EVGA S17 by 175.99%. With the EVGA S17 being a high performance laptop, it is remarkable for the Zynq-7030 (Hardware) to out-perform it. This additional comparison shows that in these certain scenarios, an FPGA can be a better choice for computational performance compared to even the fastest PCs.

Section	EVGA Run-time (s)	Zynq 7030 (Hardware) Run-time (s)	Speed-up (%)
Edge Filter	0.085978	0.0307473	279.63%
Particle Map Init	0.0003003	0.0020036	14.99%
RGB Convert	0	0.0141835	0.00%
Uncertainty Calc	0.0004639	0.0058232	7.97%
Likelihood Calc	0.0139937	0.001316	1063.35%
Resample	0.0005622	0.0028183	19.95%
Uncertainty Calc	0.0005031	0.005822	8.64%
Likelihood Calc	0.0148942	0.0010342	1440.17%
Resample	0.0006838	0.00295	23.18%
Total	0.1173792	0.0666981	175.99%

Table 6.12: Computational Performance - EVGA S17 vs Zynq 7030 (Hardware)

6.3 Footprint

The results of measuring and weighing each platform is shown in Table 6.13. The embedded vision kit is the lightest platform and also takes up the least volume, both of which are ideal. Although, due to the irregular shape of the embedded vision kit, some dimensions end up being larger than both the netbook and the laptop, and therefore, not ideal in all footprint aspects compared to the other two.

Platform	Weight(g)	Dimensions (LxWxH) (mm)	Volume (cc)
FPGA	388	280 x 113 x 35	1,107
Netbook	906	278 x 184 x 26	1,329
EVGA S17	4,100	408 x 295.5 x 27.18	3,276

Table 6.13: Platform Footprints

The height of the devices measured in Table 6.13 were calculated with each platform having no camera (screen closed and FPGA on-board camera removed) due to the high likelihood of these platforms employing an external camera. This is true for the FPGA as well, as it has a PCIe based camera, which can be more optimally placed using a PCIe extension cable. Without this extension cable relocation, the camera attached to the FPGA kit measured at a height of 96mm.

In most sceneries, based on the weight and volume alone, the embedded vision kit is a more ideal choice for most aerial and ground base vehicles due to payload and space limitations. The embedded vision kit holds greater footprint aspects than the other two platforms and is considered to pass the footprint test.

6.4 Energy Analysis

The results of the FPGA energy analysis is shown in Table 6.14. While consuming only 7.6 watts while operating the crack detection system, the FPGA shows low power usage. Vivado also provides the breakdown of power consumption of each hardware component. While these values are not a direct measurement of the components actual power consumption, they still represent a reasonable estimation based on the implementable design, and are the most reasonable way to measure individual hardware components. The Crack Detection portion of the Programmable Logic (PL), consumed a total of 0.164 Watts based on the Vivado estimation, as shown in Figure 6.6. The YUY2 to RGB Converter as shown in Figure 6.7 was estimated at 0.222 Watts.

Condition	Avg Power (Watts)	SD (Watts)
FPGA idle	6.2	0.1
FPGA processing CDPF on HW Side	7.6	0.1

Table 6.14: Power Consumption - Zynq 7030 (Hardware)



Figure 6.6: PL Power Estimation - Sobel Edge Detector

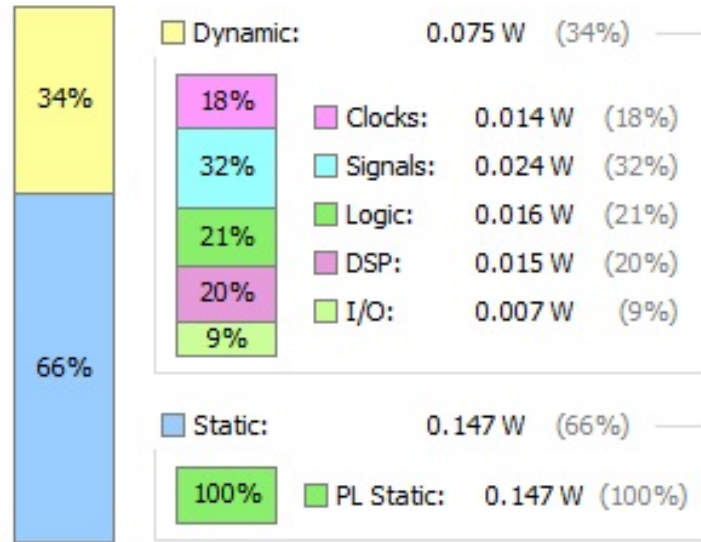


Figure 6.7: PL Power Estimation - YUY2 to RGB Converter

Table 6.15 shows the power consumption of the Acer D250 Netbook. At 12.6 Watts, it consumes more energy than the Zynq-7030 (Hardware). This result is expected due to the older technology of the Netbook coupled with the general purpose design that results in a trade-off in energy consumption. There is also extra hardware on the Netbook which consume energy including the video card, sounds card, wifi, and other components that while not actively used, can potentially cause a standby drain.

Condition	Avg Power (Watts)	SD (Watts)
Netbook Idle	8.2	0.1
Netbook processing MATLAB CDPF	12.6	0.6

Table 6.15: Power Consumption - Acer Netbook

The results of the EVGA S17 performance laptop are shown in table 6.16. This platform consumes far more energy than either the FPGA or the netbook, with it taking 52.3 Watts to process the MATLAB crack detection system. This is not surprising considering it is a gaming laptop with a Intel i7-6820HQ rating at a TDP of 45 Watts [44] and also containing a NVidia Geforce 980M chip also consuming a considerable amount of energy if at all utilised during

experiment.

Condition	Avg Power (Watts)	SD (Watts)
EVGA S17 Idle	30.8	0.1
EVGA S17 processing MATLAB CDPF	52.3	0.3

Table 6.16: Power consumption - EVGA S17

The resulting power consumptions are shown in Figure 6.8 with the FPGA being the lowest energy consuming platform. As mentioned in Chapter 5, this metric was obtained by measuring the overall energy consumption of the board using an energy meter. The most desired result of this metric is to consume the least amount of energy. The second metric, energy efficiency in Frames per Watt (FPW), was also calculated and shown in Figure 6.9. This number was calculated by using the total runtime determined in section 6.2 and comparing it to the total power consumption. This metric represents the platforms ability to detect cracks using the least amount of energy per frame. With the FPGA out-performing in both computational performance and energy consumption, this led to the difference in power efficiency becoming even greater than what was seen in the power consumption results. The FPGA was calculated running at 7105.26 Frames/Watt compared to the EVGA S17 at 586.46 Frames/Watt and the netbook at 149.38 Frames/Watt.

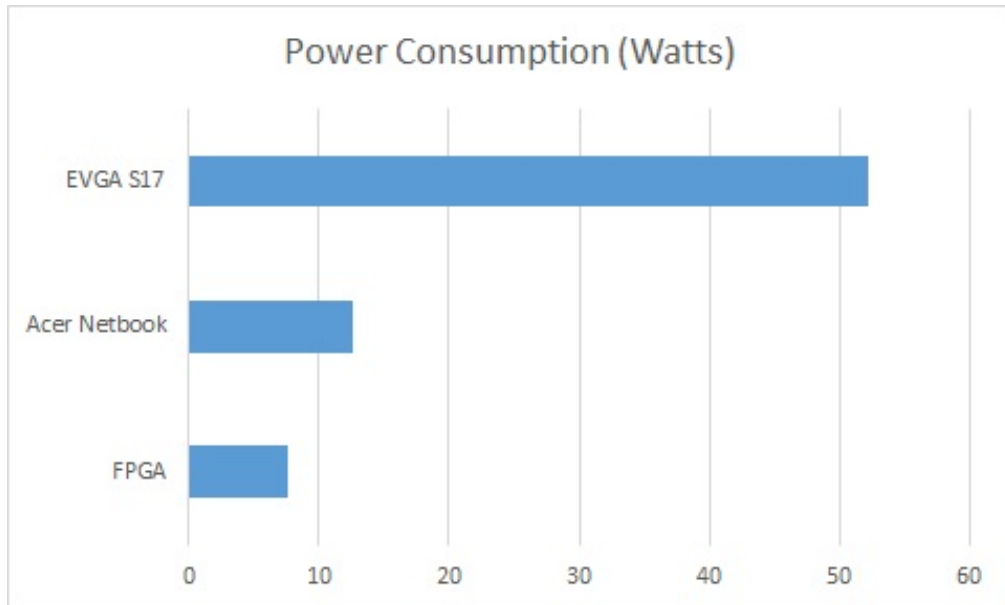


Figure 6.8: Energy Consumption Comparison (Lower is Better)

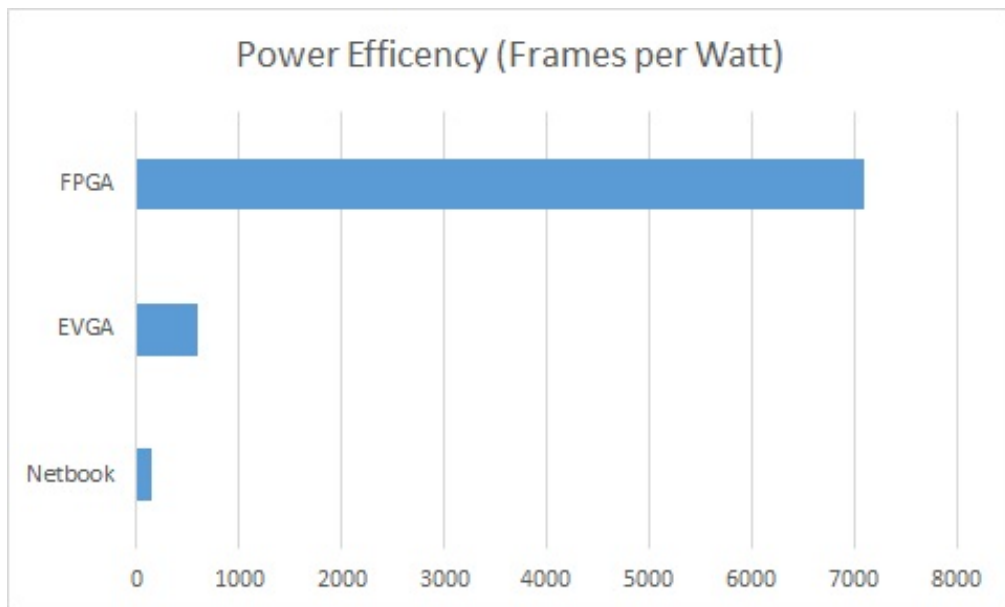


Figure 6.9: Energy Efficiency Comparison (Higher is Better)

These energy values show promising ability for this device to be implemented onto an actual UAV. Using the DJI Inspire 2 drone as a test example, it's 22.8v 6000 mAh LiPo battery would allow the FPGA to be operated for 12 hours and 50 minutes if the battery solely ran the FPGA [45]. But when factoring in the power consumption rate by flight (approx 254 Watts), the UAV equipped FPGA could fly up to 22 minutes, down from the UAV's specified 23 minute maximum flight time. As value is based on the UAV using it's 449g camera system, the 388g FPGA replacing this camera system is also achievable weight-wise, and will not consume additional energy due to any further weight load.

It has been shown that the hardware accelerated design displays superior energy consumption and efficiency characteristics compared to the other platforms. Given this result, the FPGA design is successful in regards to the energy analysis test.

6.5 Results Summary

Table 6.17 summarises the results from each area of testing. In each of these areas, the best score is underlined to allow the best performing platform to be emphasised. As shown in Table 6.17, the Zynq-7030 achieves the best scores in all areas except length and height.

Area	Zynq-7030 (Hardware)	Acer Netbook	EVGA S17
Error Rate (4000 Particles)	<u>16.81%</u>	28.19%	28.19%
Error Rate (8000 Particles)	<u>9.86%</u>	22.64%	22.64%
Computational Performance	<u>0.067s</u>	1.913s	0.117s
Weight	<u>388g</u>	906g	4,100g
Length	<u>280mm</u>	<u>278mm</u>	408mm
Width	<u>113mm</u>	184mm	296mm
Height	<u>35mm</u>	<u>26mm</u>	27mm
Volume	<u>1,107cc</u>	1,329cc	3,276cc
Power Consumption	<u>7.6 W</u>	12.6 W	52.3 W
Power Efficiency	<u>7105 FPW</u>	149 FPW	586 FPW

Table 6.17: Summary of Results

7 Future Areas of Work

This research's design provides a functional baseline work for a crack detection particle filter system. While it serves its purpose well, there are several areas that can be further developed to improve the capabilities of this system including further FPGA optimizations, colour-space standardization, increased accuracy, and further reduction in footprint.

In the areas of FPGA optimizations, more work could be conducted on the actual programmable logic side of the design. This research relied mostly on the automatic conversion between C/C++ to VHDL/Verilog and therefore there are likely many optimizations that could be further employed to produce additional speed-up of the design. With the design using only 10% of the board resources (besides IO blocks), additional work could also be applied to move other sections onto the PL side. Additional work for acceleration could also be potentially applied to further accelerate the existing components by using additional FPGA resources.

Another area of development that would further improve this research is the standardization of colour-space throughout the design. As the original design used RGB, but the reference design which our proposed solution was based on YUY2, there had been a computational loss on the conversion between these colour-spaces in order to follow the original design. This may occur several times depending on the requirement to load and save BMP images from the device. In addition to the computational (and therefore acceleration loss), there is also the loss in image quality from the original image that devolves the image upon every conversion that occurs. Further work could investigate the optimal colour-space to run a CDPF on and either change the underlying system of the device to operate using RGB, or convert all algorithms and image file loading/saving operations to YUY2 or any similar colour-spaces such as YCbCr.

While this work has shown that the crack detection solution proposed is functionally equivalent to the original design, the accuracy of the crack detection system can be further worked on to exceed both designs. Given the amount of board resources still available after implementing the accelerated design, additional work can be applied to research a more functional edge detection algorithm, which could utilise the parallel nature of the FPGA even further, leading to a more accurate and faster solution.

Another limitation of this design was the relatively low 1280x1024 resolution of the FPGA camera. A higher resolution camera would offer several advantages in this research including less computational overhead of calculations between frames, less images required to be taken, greater quality, and/pr the ability to increase the UAV range from subject.

This list touches on the many area of opportunities in the field of FPGA-based CDPF systems. With better FPGAs (and other hardware based solutions) becoming more affordable, better systems can be developed in the future that are potentially more accurate and faster, all while being a fraction of the size.

8 Conclusion

A feasible solution to conduct on-board crack detection for resource-limited platforms such as UAVs was developed in this research. This was accomplished by implementing a hardware design onto a Xilinx Zynq-7030 based embedded vision kit, which optimised the crack detection system into a parallel computational architecture allowing it to be greatly accelerated.

This FPGA research and design showed excellent results when compared to both a small portable Netbook and a performance gaming laptop. The FPGA design was shown to be a more desirable selection in terms of increased computational performance, greater overall energy efficiency, smaller footprint and having the same functional accuracy as the original MATLAB based crack detection system.

The results from Chapter 6 have concluded that this hardware accelerated design performs 1.76x to 28.7x faster than the original MATLAB based design running on a conventional PC while also consuming 1.66x to 6.88x less energy. These two metrics equate to the FPGA solution having an energy efficiency (Frames/Watt) 12.1x to 47.5x higher. With the drastically superior computational and energy performance, coupled with the low device footprint, this design is a comprehensible solution for many robotic platforms.

Compared to much of the literature discussed in Chapter 1, the run-time of 0.067 seconds per frame that this design achieved shows that this Sobel filter based crack detection technique has far greater computational performance than many other techniques.

This research has contributed to the field of hardware computer vision with the unique and substantial construction of a hardware version of a Sobel based particle filter crack detection system. In addition, the hardware YUY2 to RGB converter made to support this work is also a unique design.

Bibliography

- [1] G. P. Bu, S. Chanda, H. Guan, J. Jo, M. Blumenstein, and Y. C. Loo, “Crack Detection using a Texture Analysis-based Technique for Visual Bridge Inspection,” p. 8, 2015.
- [2] A. Coppe, R. T. Haftka, Nam-Ho Kim, and C. Bes, “A statistical model for estimating probability of crack detection,” in *2008 International Conference on Prognostics and Health Management*, Denver, CO, USA: IEEE, Oct. 2008, pp. 1–5, ISBN: 978-1-4244-1935-7. DOI: 10.1109/PHM.2008.4711418. [Online]. Available: <http://ieeexplore.ieee.org/document/4711418/> (visited on 11/14/2018).
- [3] Alberta, *Bridge Inspection & Maintenance (BIM) Manual v3.1*, Mar. 2018. [Online]. Available: <http://www.transportation.alberta.ca/Content/docType30/Production/BIMMnlv3.1.pdf> (visited on 11/14/2018).
- [4] R. G. Lins, S. N. Givigi, A. D. M. Freitas, and A. Beaulieu, “Autonomous Robot System for Inspection of Defects in Civil Infrastructures,” *IEEE Systems Journal*, vol. 12, no. 2, pp. 1414–1422, Jun. 2018, ISSN: 1932-8184. DOI: 10.1109/JSYST.2016.2611244.
- [5] J. P. Z. D. Paz, E. C. Castañeda, X. Y. S. Castro, and S. M. R. Jiménez, “Crack detection by a climbing robot using image analysis,” in *CONIELECOMP 2013, 23rd International Conference on Electronics, Communications and Computing*, Mar. 2013, pp. 87–91. DOI: 10.1109/CONIELECOMP.2013.6525765.
- [6] H. Yu, W. Yang, H. Zhang, and W. He, “A UAV-based crack inspection system for concrete bridge monitoring,” in *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Jul. 2017, pp. 3305–3308. DOI: 10.1109/IGARSS.2017.8127704.

-
- [7] A. Mohan and S. Poobal, "Crack detection using image processing: A critical review and analysis," *Alexandria Engineering Journal*, Feb. 2017, ISSN: 11100168. DOI: 10.1016/j.aej.2017.01.020. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S1110016817300236> (visited on 07/06/2018).
- [8] P. Wang and H. Huang, "Comparison analysis on present image-based crack detection methods in concrete structures," in *2010 3rd International Congress on Image and Signal Processing*, vol. 5, Oct. 2010, pp. 2530–2533. DOI: 10.1109/CISP.2010.5647496.
- [9] Z. Qu, F.-R. Ju, Y. Guo, L. Bai, and K. Chen, "Concrete surface crack detection with the improved pre-extraction and the second percolation processing methods," *PLoS ONE*, vol. 13, no. 7, Jul. 26, 2018, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0201109. pmid: 30048514. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6062087/> (visited on 11/14/2018).
- [10] V. Baltazart, P. Nicolle, and L. Yang, "Ongoing tests and improvements of the MPS algorithm for the automatic crack detection within grey level pavement images," in *2017 25th European Signal Processing Conference (EUSIPCO)*, Aug. 2017, pp. 2016–2020. DOI: 10.23919/EUSIPCO.2017.8081563.
- [11] K.-Y. Liu, L. Tang, S.-Q. Li, L. Wang, and W. Liu, "Parallel particle filter algorithm in face tracking," in *2009 IEEE International Conference on Multimedia and Expo*, Jun. 2009, pp. 1817–1820. DOI: 10.1109/ICME.2009.5202876.
- [12] S. Liu, G. Mingas, and C. S. Bouganis, "Parallel resampling for particle filters on FPGAs," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec. 2014, pp. 191–198. DOI: 10.1109/FPT.2014.7082775.
- [13] F. C. Pereira and C. E. Pereira, "Embedded Image Processing Systems for Automatic Recognition of Cracks using UAVs," *IFAC-PapersOnLine*, vol. 48, no. 10, pp. 16–21, 2015, ISSN: 24058963. DOI: 10.1016/j.ifacol.2015.08.101. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S2405896315009684> (visited on 01/14/2019).
- [14] P. Prasanna, K. J. Dana, N. Gucunski, B. B. Basily, H. M. La, R. S. Lim, and H. Parvardeh, "Automated Crack Detection on Concrete Bridges," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2, pp. 591–599, Apr. 2016, ISSN: 1545-5955, 1558-3783. DOI: 10.

- 1109/TASE.2014.2354314. [Online]. Available: <http://ieeexplore.ieee.org/document/6917066/> (visited on 01/15/2019).
- [15] M. S. Chelva and S. V. Halse, "A Performance Study of GPU, FPGA, DSP, and Multicore Processors For Embedded Vision Systems," *ITSI Transactions on Electrical and Electronics Engineering (ITSI-TEEE)*, vol. 3, no. 5, p. 6, 2015.
- [16] AnySilicon, "FPGA vs ASIC, What to Choose?," Jan. 30, 2016. [Online]. Available: <https://anysilicon.com/fpga-vs-asic-choose/> (visited on 07/06/2018).
- [17] X. Tian and K. Benkrid, "Mersenne Twister Random Number Generation on FPGA, CPU and GPU," *IEEE*, Jul. 2009, pp. 460–464, ISBN: 978-0-7695-3714-6. DOI: 10.1109/AHS.2009.11. [Online]. Available: <http://ieeexplore.ieee.org/document/5325420/> (visited on 07/06/2018).
- [18] Xiaoyin Ma, J. R. Borbon, W. Najjar, and A. K. Roy-Chowdhury, "Optimizing hardware design for Human Action Recognition," *IEEE*, Aug. 2016, pp. 1–11, ISBN: 978-2-8399-1844-2. DOI: 10.1109/FPL.2016.7577311. [Online]. Available: <http://ieeexplore.ieee.org/document/7577311/> (visited on 07/06/2018).
- [19] A. Mohanty, N. Suda, M. Kim, S. Vrudhula, J. s Seo, and Y. Cao, "High-performance face detection with CPU-FPGA acceleration," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 117–120. DOI: 10.1109/ISCAS.2016.7527184.
- [20] G. van der Wal, D. Zhang, I. Kandaswamy, J. Marakowitz, K. Kaighn, J. Zhang, and S. Chai, "FPGA acceleration for feature based processing applications," in *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jun. 2015, pp. 42–47. DOI: 10.1109/CVPRW.2015.7301365.
- [21] R. G. Lins and S. N. Givigi, "Automatic Crack Detection and Measurement Based on Image Analysis," *IEEE Transactions on Instrumentation and Measurement*, vol. 65, no. 3, pp. 583–590, Mar. 2016, ISSN: 0018-9456, 1557-9662. DOI: 10.1109/TIM.2015.2509278. [Online]. Available: <http://ieeexplore.ieee.org/document/7377063/> (visited on 06/14/2018).
- [22] National-Instruments, "FPGA Fundamentals," May 3, 2012. [Online]. Available: <http://www.ni.com/white-paper/6983/en/> (visited on 06/22/2018).

-
- [23] C. Adams, "FPGA or GPU? - The evolution continues," in *Military Embedded Systems*, Sep. 16, 2014. [Online]. Available: <http://mil-embedded.com/articles/fpga-gpu-evolution-continues/> (visited on 11/14/2018).
- [24] Xilinx, "UltraFast High Level Productivity Design Methodology Guide - UG1197 (v2018.2)," Jun. 6, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf.
- [25] —, "Zynq-7000 SoC Data Sheet: Overview - DS190 (v1.11.1)," Jul. 2, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
- [26] D. Benson, "Getting Started with FPGAs: Lookup Tables and Flip-Flops," Jun. 9, 2017. [Online]. Available: <https://www.allaboutcircuits.com/technical-articles/getting-started-with-fpgas-look-up-tables-and-flip-flops/> (visited on 10/09/2018).
- [27] Xilinx, "7 Series DSP48E1 Slice User Guide - UG479 (v1.10)," Mar. 27, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.
- [28] F. Hussein, "Hexarray: A Novel Self-Reconfigurable Hardware System," Mar. 2017. DOI: 10.13140/RG.2.2.25809.02406.
- [29] Xilinx, "Vivado Design Suite User Guide: Getting Started - UG910 (v2018.1)," Apr. 4, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug910-vivado-getting-started.pdf.
- [30] —, "Vivado Design Suite User Guide: High-Level Synthesis - UG902 (v2016.2)," Jun. 8, 2016. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf.
- [31] —, "SDSoC Environment User Guide - UG1027 (v2017.4)," Jan. 26, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1027-sdsoc-user-guide.pdf (visited on 11/14/2018).
- [32] Fourcc, "YUY2 yuv pixel format." [Online]. Available: <https://www.fourcc.org/pixel-format/yuv-yuy2/> (visited on 08/05/2018).

-
- [33] S. Estrop and G. Sullivan, “Recommended 8-Bit YUV Formats for Video Rendering,” Microsoft, Nov. 2008. [Online]. Available: <https://docs.microsoft.com/en-us/windows/desktop/medfound/recommended-8-bit-yuv-formats-for-video-rendering> (visited on 11/27/2018).
- [34] TutorialsPoint.com, “Sobel Operator.” [Online]. Available: https://www.tutorialspoint.com/dip/sobel_operator.htm (visited on 08/06/2018).
- [35] OpenCV, “Canny Edge Detection,” Dec. 22, 2017. [Online]. Available: https://docs.opencv.org/3.4.0/da/d22/tutorial_py_canny.html (visited on 09/08/2018).
- [36] CivilDigital.com, “Diagonal Cracks in Foundation and Walls,” Jul. 8, 2014. [Online]. Available: <https://civildigital.com/diagonal-cracks-foundation-walls/> (visited on 09/09/2018).
- [37] Andersal, “Strata Managers Guide to Building Defects.” [Online]. Available: <https://www.andersal.com.au/services/concrete-cancer-spalling-repair/guide-to-building-defects/> (visited on 09/09/2018).
- [38] AVNet, “PicoZed Embedded Vision Kit PYTHON-1300-C SDSoC Platform Version 2016_2,” Jan. 3, 2017. [Online]. Available: <http://picozed.org/support/design/14961/111>.
- [39] Microsoft, “Color Conversion (YCbCr to RGB).” [Online]. Available: <https://msdn.microsoft.com/en-us/library/ff635267.aspx> (visited on 10/08/2018).
- [40] S. K. Park and K. W. Miller, “Random Number Generators: Good Ones Are Hard to Find,” *Commun. ACM*, vol. 31, no. 10, pp. 1192–1201, Oct. 1988, ISSN: 0001-0782. DOI: 10.1145/63039.63042.
- [41] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005, ISBN: 0-262-20162-3.
- [42] C. Baetoni, “High Speed True Random Number Generators in Xilinx FPGAs,” p. 11,
- [43] R. G. Lins, S. N. Givigi, and P. R. G. Kurka, “Vision-Based Measurement for Localization of Objects in 3-D for Robotic Applications,” *IEEE Transactions on Instrumentation and Measurement*, vol. 64, no. 11, pp. 2950–2958, Nov. 2015, ISSN: 0018-9456. DOI: 10.1109/TIM.2015.2440556.

- [44] Intel, “Intel® Core™ i7-6820HQ Processor Specifications,” 2015. [Online]. Available: <https://ark.intel.com/products/88970/Intel-Core-i7-6820HQ-Processor-8M-Cache-up-to-3-60-GHz-> (visited on 11/27/2018).
- [45] DJI, “Inspire 2 – Specs, Videos, Downloads and FAQ.” [Online]. Available: <https://www.dji.com/ca/inspire-2/specs> (visited on 01/16/2019).

Appendices

A Zynq-7000 and Zynq-7000S SoCs

Feature Summary [25]

Table A.1: Zynq-7000 and Zynq-7000S SoCs

Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processor Core	Single-core ARM Cortex-A9 MPCore™ with CoreSight™			Dual-core ARM Cortex-A9 MPCore™ with CoreSight™						
Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor									
Maximum Frequency	667 MHz (-1); 766 MHz (-2)			667 MHz (-1); 766 MHz (-2); 866 MHz (-3)			667 MHz (-1); 800 MHz (-2); 1 GHz (-3)		667 MHz (-1) 800 MHz (-2)	
L1 Cache	32 KB Instruction, 32 KB data per processor									
L2 Cache	512 KB									
On-Chip Memory	256 KB									
External Memory Support ⁽¹⁾	DDR3, DDR3L, DDR2, LPDDR2									
External Static Memory Support ⁽¹⁾	2x Quad-SPI, NAND, NOR									
DMA Channels	8 (4 dedicated to Programmable Logic)									
Peripherals ⁽¹⁾	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
Peripherals w/ built-in DMA ⁽¹⁾	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
Security ⁽²⁾	RSA Authentication, and AES and SHA 256-bit Decryption and Authentication for Secure Boot									
Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master 2x AXI 32-bit Slave 4x AXI 64-bit/32-bit Memory AXI 64-bit ACP 16 Interrupts									

Table A.2: Zynq-7000 and Zynq-7000S SoCs (Cont'd)

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100	
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100	
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix@-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex@-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K	
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400	
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800	
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)	
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020	
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs	
	PCI Express (Root Complex or Endpoint) ⁽³⁾		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8	
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs										
	Security ⁽²⁾	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication										

Notes:

1. Restrictions apply for CLG225 package. Refer to the UG585, Zynq-7000 SoC Technical Reference Manual (TRM) for details.
2. Security is shared by the Processing System and the Programmable Logic.
3. Refer to PG054, 7 Series FPGAs Integrated Block for PCI Express for PCI Express support in specific devices.

Table A.3: Device-Package Combinations: Maximum I/Os and GTP and GTX Transceivers

Package ⁽¹⁾	CLG225			CLG400			CLG484			CLG485 ⁽²⁾				SBG485 ⁽²⁾				
Size	13 x 13 mm			17 x 17 mm			19 x 19 mm			19 x 19 mm				19 x 19 mm				
Ball Pitch	0.8 mm			0.8 mm			0.8 mm			0.8 mm				0.8 mm				
Transceiver Speed (max)										6.25 Gb/s				6.6 Gb/s				
Device	PS I/O ⁽³⁾			SelectIO			PS I/O ⁽³⁾			SelectIO			PS I/O ⁽³⁾			GTX		
	HR ⁽⁴⁾	HP ⁽⁵⁾		HR ⁽⁴⁾	HP ⁽⁵⁾		HR ⁽⁴⁾	HP ⁽⁵⁾		HR ⁽⁴⁾	HP ⁽⁵⁾		HR ⁽⁴⁾	HP ⁽⁵⁾		HR ⁽⁴⁾	HP ⁽⁵⁾	
XC7Z007S	84	54	–	128	100	–												
XC7Z012S										128	4	150	–					
XC7Z014S				128	125	–	128	200	–									
XC7Z010	84	54	–	128	100	–												
XC7Z015										128	4	150	–					
XC7Z020				128	125	–	128	200	–									
XC7Z030														128	4	50	100	
XC7Z035																		
XC7Z045																		
XC7Z100																		

Notes:

1. All packages listed are Pb-free (FBG and FFG with exemption 15). Some packages are available with a Pb option.
2. The Z-7012S and Z-7015 devices in the CLG485 package and the Z-7030 device in the SBG485 package are pin-to-pin compatible.
3. PS I/O count does not include dedicated DDR calibration pins.
4. HR = High Range I/O with support for I/O voltage from 1.2V to 3.3V.
5. HP = High Performance I/O with support for I/O voltage from 1.2V to 1.8V.

Table A.4: Device-Package Combinations: Maximum I/Os and GTP and GTX Transceivers (Cont'd)

Package ⁽¹⁾	FBG484				FBG676				FFG676				FFG900				FFG1156				
Size	23 x 23 mm				27 x 27 mm				27 x 27 mm				31 x 31 mm				35 x 35 mm				
Ball Pitch	1.0 mm				1.0 mm				1.0 mm				1.0 mm				1.0 mm				
Transceiver Speed (max)	6.6 Gb/s				6.6 Gb/s				12.5 Gb/s				12.5 Gb/s				10.3 Gb/s				
Device	PS I/O ⁽²⁾		GTX		PS I/O ⁽²⁾		GTX		PS I/O ⁽²⁾		GTX		PS I/O ⁽²⁾		GTX		PS I/O ⁽²⁾		GTX		
	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	HR ⁽³⁾	HP ⁽⁴⁾	
XC7Z007S																					
XC7Z012S																					
XC7Z014S																					
XC7Z010																					
XC7Z015																					
XC7Z020																					
XC7Z030	128	4	100	63	128	4	100	150	128	4	100	150									
XC7Z035					128	8	100	150	128	8	100	150	128	16	212	150					
XC7Z045					128	8	100	150	128	8	100	150	128	16	212	150					
XC7Z100													128	16	212	150	128	16	250	150	

Notes:

1. All packages listed are Pb-free (FBG and FFG with exemption 15). Some packages are available with a Pb option.
2. PS I/O count does not include dedicated DDR calibration pins.
3. HR = High Range I/O with support for I/O voltage from 1.2V to 3.3V.
4. HP = High Performance I/O with support for I/O voltage from 1.2V to 1.8V.

B Particle Filter Code

```
1
2 // Start of Uncertainty Calculation
3
4 for (x = 0; x < NUM_PARTICLES; x++) {
5     particles[x][0] = randn(particles[x][0],(float) Xstd_pos);
6     particles[x][1] = randn(particles[x][1],(float) Xstd_pos);
7     particles[x][2] = randn(particles[x][2], (float) Xstd_vec);
8     particles[x][3] = randn(particles[x][3], (float) Xstd_vec);
9 }
10
11 // Start of Likelihood Calculation
12
13 // Xstd_rgb is the SD of Target Colour
14 Inacc_A = -log(sqrt(2 * M_PI) * (float) Xstd_rgb);
15 Inacc_B = -0.5 / ((float) (Xstd_rgb ^ 2));
16
17 for (x = 0; x < NUM_PARTICLES; x++) {
18     m = particles[x][0];
19     n = particles[x][1];
20     i = (m >= 1 && m <= height);
21     j = (n >= 1 && n <= width);
22
23     if (i && j) {
24         if ((char)(img_edg[((m-1)*stride)+(n-1)])==255) {
25             F = 2;
26         }
27         else {
28             F = 1;
29         }

```

```

1     D[0] = img_rgb2[(m*3*1280)+(n*3)] - Xrgb_trgt[0];
2     D[1] = img_rgb2[(m*3*1280)+(n*3)+1] - Xrgb_trgt[1];
3     D[2] = img_rgb2[(m*3*1280)+(n*3)+2] - Xrgb_trgt[2];
4
5     //Create Tendency of particle to target color
6      //(Lower value corresponds to greater tendency)
7     float D2 = pow(D[0],2) + pow(D[1],2) + pow(D[2],2);
8     //Determine Possible Range for the Target Colour
9     likelihood[x] = Inacc_A + Inacc_B * D2 / (float)F;
10  }
11  else {
12      //Delete particles outside image area
13      likelihood[x] = -INFINITY;
14  }
15  }
16
17  // Start of Resample section
18
19  // Generating Random Numbers
20  for (x = 0; x < NUM_PARTICLES; x++) {
21      T[x] = ((float)rand_int()/32767.0);
22  }
23
24  for (x = 0; x < NUM_PARTICLES; x++) {
25      local_likelihood[x] = resample_likelihood[x];
26      if (local_likelihood[x] > max_likelihood) {
27          max_likelihood = local_likelihood[x];
28      }
29  }
30
31  for (x = 0; x < NUM_PARTICLES; x++) {
32      local_likelihood[x] = exp(local_likelihood[x] - max_likelihood);
33  }
34
35  for (x = 0; x < NUM_PARTICLES; x++) {
36      sum_likelihood = sum_likelihood + local_likelihood[x];
37  }

```

```

1  for (x = 0; x < NUM_PARTICLES; x++) {
2    local_likelihood[x] = local_likelihood[x] / sum_likelihood;
3  }
4
5  // Cumulative summation
6  for (x = 1; x < NUM_PARTICLES; x++) {
7    local_likelihood[x] = local_likelihood[x]+local_likelihood[x-1];
8  }
9
10 for (x=0;x<NUM_PARTICLES;x++){
11   int first = 0;
12   int middle = NUM_PARTICLES/2;
13   int last = NUM_PARTICLES-1;
14   float T_local = T[x];
15   for(y=0;y<HISTO_LOOP;y++){
16     if (local_likelihood[middle] <= T_local){
17       first = middle +1;
18     }
19     else{
20       last = middle - 1;
21     }
22     middle = (first + last)/2;
23   }
24   resample_histo[x] = first+1;
25 }
26
27 for (x = 0; x < NUM_PARTICLES; x++) {
28   for (y = 0; y < 4; y++) {
29     temp_particles[x][y] = particles[x][y];
30   }
31 }
32
33 for (x = 0; x < NUM_PARTICLES; x++) {
34   for (y = 0; y < 4; y++) {
35     particles[x][y] = temp_particles[histo_copy[x] - 1][y];
36   }
37 }

```
