

Sub-Pixel Image Translation Estimation on a Nanosatellite Platform

by

Jürgen Lüdemann



*Thesis presented in partial fulfilment of the requirements for
the degree of Master of Engineering (Electronic) in the
Faculty of Engineering at Stellenbosch University*

Supervisors: Mr A. Barnard
Mr W. Smit

Co-supervisor: Dr D.F. Malan

April 2019

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date:April 2019.....

Copyright © 2019 Stellenbosch University
All rights reserved.

Abstract

Nanosatellites are limited in their physical size, which limits the physical size of payloads they can carry, thereby limiting the quality of images taken during CubeSat Earth Observation missions. Algorithms exist that combine partially overlapping images to produce better output image quality. These algorithms may either improve the signal-to-noise ratio via averaging, increase resolution via super-resolution or merely remove redundant information via mosaicing. Typically, they only function properly if the geometric transformations between consecutive images are known with high accuracy. They can either be applied terrestrially or on-board a satellite. Downloading large raw image data sets for terrestrial processing is impractical for a CubeSat mission, and therefore an on-board solution is desirable.

This thesis discusses the accurate determination of the transformation between consecutive images on-board, laying the foundation for efficient on-board de-noising, super-resolution and mosaicing. Two common methods used to determine translation – normalised cross correlation (NCC) and phase correlation – are investigated. From simulated results, NCC is shown to be the better candidate for our application. NCC achieves sub-pixel accuracy by making use of polynomial least squares regression. NCC is well suited for implementation on a satellite platform where images are captured in quick succession, resulting in partially overlapping images with little rotation between frames.

We compare two potential hardware platforms – the MicroZed 7020 and Jetson TK1 – and then describe how we implemented our proposed solution onto the former, using a hardware description language. Software simulation and firmware-implementation results, using simulated data, are compared and discussed. Subsequently, the MicroZed 7020's implemented design is characterised, compared and discussed in terms of algorithm and platform performance.

Opsomming

Nanosatelliete is beperk t.o.v hul grootte, gevolglik is die ‘loonvrag’ wat hulle kan dra ook beperk. Dit het ’n negatiewe uitwerking op die kwaliteit van die beelde, wat tydens waarnemings missies van CubeSats gemaak word. Algoritmes bestaan wat gedeeltelik oorvleuelende beelde kombineer om vir hoër kwaliteit ‘uitset’ beelde te sorg. Sulke algoritmes kan die sein-tot-ruis verhouding verbeter via ‘beeld-sommering’, die resolusie verhoog deur super-resolusie of oorbodige informasie via ‘mosaïekmetodes’ verminder. Sodanige algoritmes funksioneer slegs optimaal wanneer die geometriese transformasies tussen agtereenvolgende beelde tot ’n hoë vlak van akkuraatheid bepaal word. Die soort algoritmes kan aan boord, of op die grond toegepas word. Dit is onprakties om sulke groot ongeformateerde datastelle af te laai vir prosessering op die grond tydens die missie van ’n Nanosatelliet, dus geniet ’n aanboord oplossing voorkeur.

Hierdie tesis bespreek die akkurate bepaling van inter-beeld transformasies aan boord van ’n satelliet. Dit lê die fondament vir aanboord sein-suiwering, super-resolusie en mosaïek metodes. Twee algemene metodes – genormaliseerde kruiskorrelasie (GK) en fasekorrelasie – word ondersoek. Simulasieresultate dui aan dat GK meer doeltreffend is vir ons doeleindes. GK behaal sub-pixel akkuraatheid deur middel van polinomiese kleinste kwadraat regressie. GK is geskik vir ’n platform waar opeenvolgende deels-oorvleuelende beelde, met weglaatbaar klein relatiewe rotasie, intyds bewerk moet word.

Ons vergelyk twee potensieële hardeware platforms – die Jetson TK1 en die MicroZed 7020 – en implementeer die voorgestelde oplossing op die laasgenoemde, met die gebruik van hardewarebeskrywingskode. Die resultate van die sagtewaresimulasie word met die geïmplementeerde hardewareresultate vergelyk, met die gebruik van gesimuleerde data. Die MicroZed 7020 se geïmplementeerde ontwerp word ook uiteengesit, vergelyk en bespreek, in terme van die vermoë van die algoritme en hardewareplatform.

Acknowledgements

I would like to express my sincere gratitude to the following:

My Supervisors Mr Arno Barnard, Mr W. Smit, and Dr D. Francois Malan, for their guidance, training, optimism, encouragement and educational-mentoring which they have provided generously throughout this process. Further technical input from Dr L. Muller, Dr H.W. Jordaan, J.H. Wessel, G. Roux, J.J. Burns, AJ Merts and C.E. Roelofse.

Space Advisory Company for providing a bursary, an exciting topic, expertise and the suitable hardware. NVIDIA Corporation for sponsoring a Jetson TK1 development board for research purposes. Electrical & Electronic Engineering, Stellenbosch University, for use of their ESL facilities. Prof. W.H. Steyn for generously providing additional funding towards my attendance at various conferences.

Friends and family for support, specifically: C.E. Roelofse, the ESL community, and my parents for removing all possible obstacles. Lastly, Dr T.J. Keller's ministry, for showing me the beauty and dignity in all work.

‘For the law was given through Moses; grace and truth came through Jesus Christ.’ *John 1:17, ESV*

Dedication

To the forerunners of the faith

Contents

| | |
|---|-------------|
| Declaration | i |
| Abstract | ii |
| Opsomming | iii |
| Acknowledgements | iv |
| Dedication | v |
| Contents | vi |
| List of Figures | x |
| List of Tables | xiii |
| Nomenclature | xiv |
| Publication | xvi |
| 1 Introduction | 1 |
| 1.1 Project scope | 2 |
| 1.2 Overview | 4 |
| 2 Background | 6 |
| 2.1 Satellite environment | 6 |
| 2.1.1 Thermal | 6 |
| 2.1.2 Radiation event | 7 |
| 2.1.3 Relevant to next design phase | 8 |
| 2.2 Satellite systems | 8 |
| 2.3 Hardware technology | 9 |
| 2.3.1 Microprocessor unit | 9 |
| 2.3.2 Microcontroller | 9 |
| 2.3.3 GPGPU | 10 |
| 2.3.4 FPGA | 11 |
| 2.3.5 Discussion | 12 |

| | | |
|----------|--|-----------|
| 2.4 | Image registration methods | 12 |
| 2.4.1 | Feature-based methods | 12 |
| 2.4.2 | Area-based methods | 14 |
| 2.4.3 | Other methods | 19 |
| 2.4.4 | Current implementations | 20 |
| 2.4.5 | Discussion | 21 |
| 2.5 | Desirable SPITE application | 21 |
| 2.5.1 | Image stacking | 21 |
| 2.5.2 | Image noise | 23 |
| 2.6 | Summary | 24 |
| 3 | Algorithm Adaptation and Conceptual Comparison | 25 |
| 3.1 | Sub-pixel methods | 25 |
| 3.1.1 | Method 1: polynomial least squares regression | 25 |
| 3.1.2 | Method 2: Taylor series expansion | 27 |
| 3.2 | Further investigation and comparison | 28 |
| 3.2.1 | 2-Dimensional Fourier transform | 29 |
| 3.2.2 | Runtime and memory analysis | 29 |
| 3.3 | Summary | 34 |
| 4 | Algorithm Benchmarking | 36 |
| 4.1 | Preamble: generating test-images | 36 |
| 4.2 | Influence of SPITE accuracy on image stacking | 38 |
| 4.3 | Surface fit and registration method comparison | 42 |
| 4.4 | SPITE required window and template dimensions | 46 |
| 4.5 | SPITE range, offset and error | 49 |
| 4.6 | SPITE mis-registration investigation: Monte Carlo simulation | 55 |
| 4.7 | SPITE accuracy in noisy conditions | 57 |
| 4.8 | SPITE accuracy in rotations | 59 |
| 4.9 | SPITE accuracy in noise and rotations | 61 |
| 4.10 | Discussion | 63 |
| 5 | Hardware Investigation | 65 |
| 5.1 | MicroZed 7020 | 65 |
| 5.1.1 | Architecture | 65 |
| 5.1.2 | Benchmark: 2D-FFT | 67 |
| 5.1.3 | Radiation mitigation | 68 |
| 5.2 | Jetson Tegra K1 | 69 |
| 5.2.1 | Architecture | 69 |
| 5.2.2 | Benchmark: 2D-FFT | 71 |
| 5.2.3 | Radiation mitigation | 77 |
| 5.3 | Discussion | 78 |
| 6 | Hardware Implementation | 79 |

| | | |
|----------|---|------------|
| 6.1 | Methodology | 80 |
| 6.2 | Hardware architecture | 81 |
| 6.2.1 | System-level interfaces | 82 |
| 6.2.2 | Data-Level | 84 |
| 6.3 | Detailed math-blocks | 86 |
| 6.3.1 | Streaming math-blocks | 86 |
| 6.3.2 | $M_{29,30,31}$: sum area table | 88 |
| 6.3.3 | Sub-window loader | 89 |
| 6.3.4 | M_{16} : zero pad | 90 |
| 6.3.5 | $M_{22,23,27}$: 2D-FFT | 91 |
| 6.3.6 | $M_{24,25,26}$: conjugate and multiply | 92 |
| 6.3.7 | M_{28B} : data shift | 93 |
| 6.3.8 | M_{39} : max 3x3 | 95 |
| 6.3.9 | M_{40} : surface fit | 95 |
| 6.4 | Hardware accuracy comparison | 96 |
| 6.5 | Design for resource scalability | 98 |
| 6.5.1 | Further Optimisation | 99 |
| 6.6 | Runtime: individual math-blocks | 100 |
| 6.6.1 | Possible optimisations | 102 |
| 6.7 | Increasing robustness | 104 |
| 6.7.1 | Proposal 1 | 105 |
| 6.7.2 | Proposal 2 | 106 |
| 6.7.3 | Proposal 3 | 107 |
| 6.8 | Discussion | 108 |
| 7 | Experimentation and Results | 109 |
| 7.1 | Accuracy | 109 |
| 7.2 | Runtime analysis and results | 112 |
| 7.2.1 | Final implementation: frequency and resources | 112 |
| 7.2.2 | Runtime analysis | 113 |
| 7.2.3 | Runtime comparison vs MCU based system | 115 |
| 7.3 | Power | 118 |
| 7.4 | Summary | 122 |
| 8 | Conclusions and Recommendations | 123 |
| 8.1 | Overview | 123 |
| 8.1.1 | Contributions | 124 |
| 8.1.2 | Limitations | 124 |
| 8.2 | Future work | 125 |
| 8.2.1 | Hardware | 125 |
| 8.2.2 | Software | 126 |
| 8.3 | Conclusion | 127 |
| | Appendices | 128 |

| | | |
|----------|--|------------|
| A | Section 1, 2 | 129 |
| | A.1 Super-resolution | 129 |
| | A.2 Dynamic voltage and frequency scaling | 130 |
| | A.3 Fourier-rotation determination resolution simulation | 131 |
| | A.4 Image stacking | 134 |
| B | Section 3, 4, 5 | 136 |
| | B.1 Least squares regression | 136 |
| | B.2 Linear sub-pixel interpolation | 136 |
| | B.3 SNR post image stacking | 139 |
| | B.4 Surface fit error | 140 |
| | B.5 Noisy sectioned images | 141 |
| | B.6 Counters and base 2^n | 144 |
| | B.7 Benchmark implementation | 145 |
| | B.7.1 Instantaneous power: | 145 |
| | B.7.2 C-serial, C-parallel | 145 |
| | B.7.3 CUDA | 146 |
| C | Section 6, 7, 8 | 148 |
| | C.1 Resources | 148 |
| | C.2 FSM-Design methodology | 148 |
| | C.3 FPGA development steps and processes | 152 |
| | C.4 2D-FFT block diagram | 154 |
| | C.5 Polynomial fit C-code | 155 |
| | C.6 C-equivalent math-blocks | 157 |
| | C.6.1 M_{28B} | 157 |
| | C.6.2 $M_{24.25.26}$ | 158 |
| | C.6.3 $M_{32.33.34.35.36}$ | 158 |
| | C.6.4 $M_{22.23.27}$ | 159 |
| | C.7 Timing C-code | 160 |
| | C.8 FSM-streaming ability | 161 |
| | C.8.1 Standard streaming operation | 161 |
| | C.8.2 Versatility: float-fix-float | 165 |
| | C.8.3 Versatility: lower throughput | 167 |
| | C.9 Theoretical case study: SPITE energy | 169 |
| | References | 171 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Geometry of the problem. | 3 |
| 1.2 | The work required to be completed by the hardware implementation. Source image [1] | 4 |
| 2.1 | Example of window and template for NCC and phase correlation. . | 16 |
| 2.2 | Image stacking explanation | 22 |
| 3.1 | Typical NCC integer grid. | 26 |
| 3.2 | Example of window and template for NCC and phase correlation. . | 28 |
| 3.3 | Phase correlation mathematical flow | 30 |
| 3.4 | Simplified NCC mathematical flow | 32 |
| 3.5 | Memory usage and runtimes of algorithms over various sizes M . . . | 34 |
| 4.1 | Image A with the varying SNR's. | 40 |
| 4.2 | Image stacking SNR gain v.s. mis-registration | 41 |
| 4.3 | Noisy image: before and after image stacking mitigates noise. . . . | 42 |
| 4.4 | NCC vs phase correlation: vary sub-pixel methods 1. | 44 |
| 4.5 | NCC vs phase correlation: vary sub-pixel methods 2. | 46 |
| 4.6 | Test data, Image A: Engineering faculty | 48 |
| 4.7 | Test data, Image B: Simonswyk | 48 |
| 4.8 | Test data, Image C: Agricultural land | 48 |
| 4.9 | Test data, Image D: Bushes | 48 |
| 4.10 | NCC: determine appropriate window and template size. | 49 |
| 4.11 | Effective NCC SPITE range. | 50 |
| 4.12 | NCC: error over wide range of translations. | 51 |
| 4.13 | Area in which SPITE consistently fails. | 52 |
| 4.14 | Analysis: interesting error behaviour. | 53 |
| 4.15 | Analysis: interesting error behaviour, sub-pixel method the culprit 1. | 54 |
| 4.16 | Analysis: interesting error behaviour, sub-pixel method the culprit 2. | 54 |
| 4.17 | Histogram: SPITE Absolute error spread, Monte Carlo. | 56 |
| 4.18 | SPITE error Histogram. | 58 |
| 4.19 | Window and template, in presence of rotations. | 59 |
| 4.20 | Histogram of NCC Poly 3×3 accuracy in presence of various rotations. | 61 |

| | | |
|------|--|-----|
| 4.21 | Histogram of NCC Poly 3x3 accuracy in presence of noise and rotations. | 62 |
| 5.1 | Zynq 7000 SoC architecture [2]. | 66 |
| 5.2 | Zynq 7000 clocking logic. | 67 |
| 5.3 | Architecture of TK1 SoC [3]. | 70 |
| 5.4 | Power system of Jetson TK1 [3]. | 71 |
| 5.5 | Python script for 2D-FFT benchmarking the Jetson TK1. | 72 |
| 5.6 | Power domain measurements for 2048 × 2048 2D-FFT CUDA HF. | 74 |
| 5.7 | Performance 2D-FFT Jetson TK1, runtime and average power. | 75 |
| 5.8 | Performance 2D-FFT Jetson TK1, runtime and average power. | 76 |
| 6.1 | NCC SPITE low level math block layout. | 81 |
| 6.2 | SPITE hardware system-level architecture | 83 |
| 6.3 | Generic streaming math-block. | 86 |
| 6.4 | Architecture: sum area table math-block ($M_{29,30,31}$) | 88 |
| 6.5 | Architecture: sub-window loader | 89 |
| 6.6 | Architecture: zero-pad math-block (M_{16}) | 91 |
| 6.7 | Architecture: 2D-FFT math-block ($M_{22,23,27}$) | 92 |
| 6.8 | Architecture: math-block ($M_{24,25,26}$) | 93 |
| 6.9 | High throughput complex multiplier. | 93 |
| 6.10 | Diagrammatic representation M_{28B} reordering. | 94 |
| 6.11 | Architecture: M_{28B} | 94 |
| 6.12 | Architecture: M_{39} | 95 |
| 6.13 | 64 × 64 window and 32 × 32 template calibration test data-set. | 97 |
| 6.14 | Propogated error in M_{28B} , M_{37} and M_{39} due to 32bit float resolution. | 98 |
| 6.15 | Typical BRAM reading FSM, throughput of 1/3. | 103 |
| 6.16 | Proposal 1, radiation mitigation. | 106 |
| 6.17 | Proposal 2, memory radiation mitigation. | 106 |
| 7.1 | Theoretcial software vs. hardware error, Image B. | 110 |
| 7.2 | Distribution: theoretcial software vs. hardware error. | 111 |
| A.1 | Pixels effectively mapped from Cartesian plot to polar plot. | 131 |
| A.2 | Accuracy of rotation estimate vs amount of pixels mapped. | 132 |
| A.3 | Example: mapping rotation translation. | 133 |
| B.1 | Linear interpolation for sub-pixel shifts. | 137 |
| B.2 | Sub-pixel translation un-interpolated. | 138 |
| B.3 | Sub-pixel translation interpolated. | 138 |
| B.4 | Mis-registration error, same as Figure , just a larger range to show th | 139 |
| B.5 | Surface fit error, image A 1. | 140 |
| B.6 | Surface fit error, image A 2. | 141 |
| B.7 | Surface fit error, image A 3. | 141 |
| B.8 | Image A with the varying SNR's. | 142 |

| | | |
|------|--|-----|
| B.9 | Image B with the varying SNR's. | 143 |
| B.10 | Image C with the varying SNR's. | 143 |
| B.11 | Image D with the varying SNR's. | 144 |
| C.1 | Various BRAM resource requirements for different size SPITE. | 148 |
| C.2 | Flow diagram of a simple 'count up to n' FSM. | 149 |
| C.3 | 2D-FFT block diagram | 154 |
| C.4 | Standard FSM-Streaming operation via wave diagram. | 163 |
| C.5 | A typical one Math-Unit math-block. | 164 |
| C.6 | FSM-Streaming state transition diagram. | 165 |
| C.7 | Example: ease of math-block alteration 1b. | 166 |
| C.8 | Example: ease of math-block alteration 1b. | 167 |
| C.9 | Example: ease of math-block alteration 2a. | 168 |
| C.10 | Example: ease of math-block alteration 2b. | 169 |

List of Tables

| | | |
|-----|--|-----|
| 4.1 | SPITE: Absolute error spread, Monte Carlo. | 56 |
| 4.2 | SPITE error characteristic: average and percentage of failure. | 58 |
| 4.3 | SPITE error characteristic, in presence of rotations. | 60 |
| 4.4 | Histogram of NCC Poly 3x3 accuracy in presence of noise and rotations. | 63 |
| 5.1 | 2D-FFT MicroZed 7020 benchmark results. | 68 |
| 5.2 | Summary: runtime and energy for 2D-FFT on Jetson TK1. | 77 |
| 6.1 | Percentage error: 32bit float instead of 64bit double. | 97 |
| 6.2 | Memory required for various sizes of M | 99 |
| 6.3 | Runtime of various important math-blocks. | 101 |
| 6.4 | Theoretical comparison, radiation mitigation proposals. | 107 |
| 7.1 | First order timing and power analysis. | 113 |
| 7.2 | Final resource usage: MicroZed 7020 | 113 |
| 7.3 | MicroZed 7020: full system runtime. | 114 |
| 7.4 | Runtime comparison: FPGA vs. C-equivalent math-blocks. | 116 |
| 7.5 | MicroZed power measurements for various tests. Results show measured (ammeter) total power, the derived power from the total power, and the method by which it is derived. The total power values are averages over five runs. | 120 |
| C.1 | Theoretical case study: SPITE energy performance. | 170 |

Nomenclature

Acronyms

| | |
|-------|--|
| ABFT | algorithm-based fault tolerance |
| BJT | Bipolar Junction Transistor |
| BRAM | Block Random Access Memory |
| CCD | Charge-Coupled Device |
| CUDA | Compute Unified Device Architecture |
| CP | Control Point |
| CPU | Central Processing Unit |
| DD | Displacement Damage |
| DN | Digital Number |
| DSF | Direct Solar Flux |
| DSP48 | 48bit Digital signal processing math-unit provided by Xilinx |
| DVFS | Dynamic Voltage and Frequency Scaling |
| FFT | Fast Fourier Transform |
| FIFO | First In First Out |
| FPGA | Field Programmable Gate Array |
| FPU | Floating Point Unit |
| FWFT | First Word Fall Through |
| GBSE | Gradient Based Shift Estimation |
| GPU | Graphics Processor Unit |
| GPGPU | General Purpose Graphics Processor Unit |
| GSD | Ground Sampling Distance |
| GT | Ground Truth |
| HDL | Hardware Description Language |
| HPC | High-performance four core cluster (HPC) |
| ILA | Integrated Logic Analyser |
| JTAG | Joint Test Action Group |
| LEO | Low Earth Orbit |
| LPC | Low Power Core |
| LUT | Look Up Table |
| MBU | Multi-bit Upset |
| MCU | Microcontroller Unit |
| MPU | Microprocessor Unit |
| NOP | No-operation |

| | |
|----------|--|
| OBC | On-board Computer |
| PAR | Place and Route |
| PT | Persistent Thread |
| Poly 3x3 | Polynomial least squares regression, using the maximum and the 8 points surrounding it. |
| RISC | Reduced Instruction Set Computer |
| RTL | Register-Transfer Level |
| RTU | Run Time Unit |
| SAT | Sum Area Table |
| SECDED | Single Error Correction Double Error Detection |
| SEE | Single Event Effects |
| SEL | Single Event Latch-up |
| SET | Single Event Transient |
| SEU | Single Event Upset |
| SIMD | Single Instruction Multiple Data |
| SoC | System on Chip |
| SPITE | Sub-Pixel Image Translation Estimation |
| SRIR | Super-Resolution Image Reconstruction |
| SSDA | Sequential Similarity Detection Algorithm |
| TDE | Total Dose Effects |
| TMR | Triple Modular Redundancy |
| UART | Universal Asynchronous Receiver-Transmitter |
| VIO | Virtual Input/Output |
| WNS | Worst Negative Slack |

Publication

Sections of the work presented in this thesis appear in the following congress paper [4]:

Lüdemann, J., Barnard, A., Malan, D.F., “Sub-pixel Image Registration on an Embedded Satellite Platform”, *69th International Astronautical Congress*, Bremen, Germany, 1-5 October 2018.

Chapter 1

Introduction

In the last decade [5], the space-engineering community has seen an exponential growth in the development, launch and operation of nanosatellites. A significant factor in this increase has been the acceptance of the CubeSat standard, which specifies 10cm×10cm×10cm satellite modules. This standard has inherent challenges, namely the size. A minimalist satellite would require an on-board computer for general house-keeping, an attitude determination and control system (ADCS) for pointing and manoeuvring, a transmitter receiver to send data and receive telecommands and lastly a power-system consisting of a battery, solar panels and control logic. Only then is the platform fully capable to accommodate a significant payload. Despite this volume constraint, two unit (2U) CubeSats with capable payloads have shown immense success [6]. An added engineering challenge is the harsh space environment in which these CubeSats operate.

CubeSat matrix imagers, achieve frame rates up to 5 frames per second [7], where consecutive images will be partially overlapping. Although such modules fit within CubeSat sizes, it is not easy to fully utilise the increased capture rate. The power requirements of high-data downlink transmitters, coupled with limited downlink opportunities pose a bottleneck. The alternative, is to implement *on-board* image processing. This would ideally distil large data sets into *useful* data only, thereby reducing the strain placed on downlink requirements.

Algorithms that perform desirable information reduction exist. They take *multiple* input images to produce one *output* image. Image averaging reduces overall noise in overlapping regions [8], mosaicing stitches together neighbouring images [9], discarding overlapping regions, and super-resolution recovers ‘lost’ resolution caused by aliasing [10]. One common denominator of all these methods is *accurate* image registration.

Therefore, we aim to accomplish accurate *image registration*, on a nanosatellite-compatible dedicated *hardware platform*, while also aiming for an optimal solution in terms of speed and efficiency.

Geometry of problem

Assume a CubeSat that is 3-axis stabilised as in Figure 1.1a. It captures two consecutive images at any pitch (rotation around Satellite_{*y*}), roll (rotation around Satellite_{*x*}) and yaw (rotation around Satellite_{*z*}) angles. Forward motion compensation - where the satellite is rotated to reduce image blur during imaging - is not performed between two consecutive captured frames.

The pitch, roll and yaw *rates* are all controlled to zero whilst capturing images, however jitter is present. Therefore, there is a small pitch and roll rate, resulting in unknown sub-pixel translations in both Image_{*x*} and Image_{*y*} directions in Figure 1.1b. We assume the image distortion caused by pitch and roll jitter is negligible. A small yaw rate results in small random rotations in the final image plane, θ .

Lastly, the satellites orbital motion – the main contributor to inter-frame translation – causes unknown *integer* and *sub-integer* pixel translations in both Image_{*x*} and Image_{*y*} directions.

Note how no scaling, shear, or warping occurs between the two consecutive images. For the remainder of this document, when discussing any shift, error or translation, we refer to the image axes, not the satellite body axes.

After subtracting the satellite’s known orbital translation, we are left with an unknown rigid image transformation (translation and possibly rotation) which we wish to determine to sub-pixel accuracy. For simplicity this offset will be expressed in the image coordinate frame.

SPITE: accurate image registration for this context

Image registration is the process of ‘aligning two or more images from the same scene’ [11]. In order to re-align two consecutive images, the transformation between them needs to be determined. Figure 1.1 makes the case that *only* rigid body transformations are present between consecutive images, resulting in translations and rotations in the image plane.

To determine this translation in the presence of small θ rotations, we define the following term: Sub-pixel Image Translation Estimation (SPITE). Note how this requires sub-pixel accuracy, provides a two dimensional answer, and excludes the rotation estimate.

The effects due to re-aligning images after image registration is not investigated.

1.1 Project scope

Determine SPITE, the red vector in Figure 1.1b, which contains x and y shift between two consecutive images, where the second image will

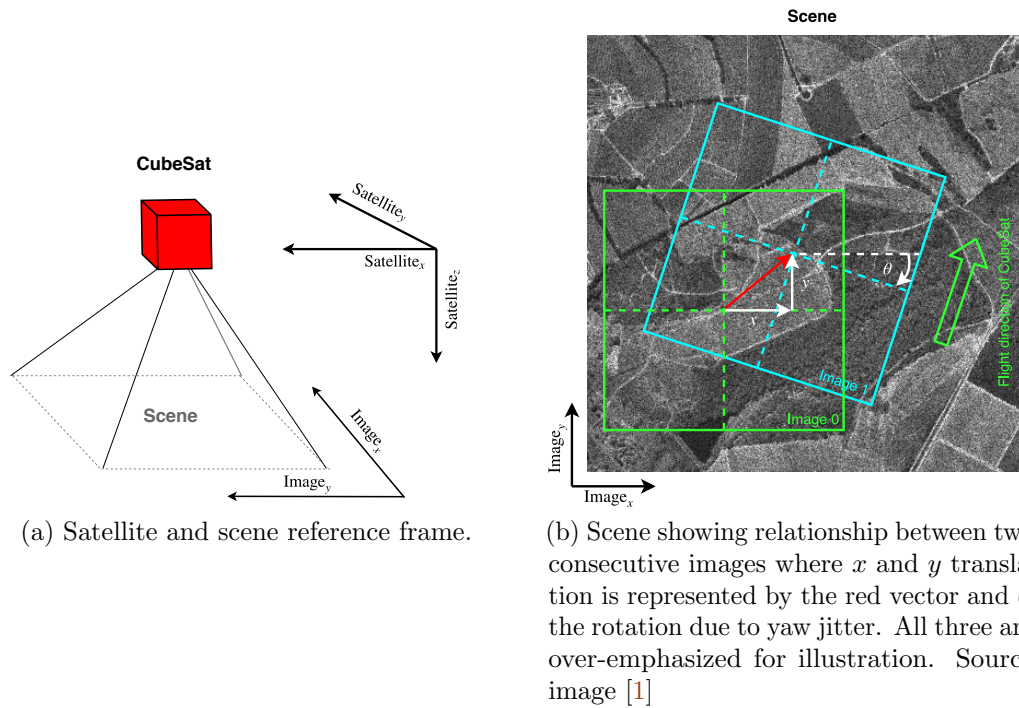


Figure 1.1: Geometry of the problem.

1. have a random shift offset in the x and y -directions due to jitter and orbital motion.
2. have sufficient overlap with the previous image
3. have a small rotation (yaw $< 1^\circ$)

These requirements imply a rigid mapping of consecutive images. No shear, warp or scaling needs to be estimated.

Required Outcomes: Determine an algorithm which can perform SPITE. Aim to perform SPITE to a error metric of ≤ 0.1 pixel, a starting point which we consider a good accuracy. Implement this on a specific hardware platform (Figure 1.2). Determine the performance metrics thereof. Comment on the projected ability in a CubeSat environment.

Additional Desirable Outcomes: As far as possible, the hardware implementation will aim to be easily scalable, portable, modular, fast, low-power consumption, reliable, and completely autonomous¹, such that the platform and implementation is a suitable CubeSat prototype. Research the possibility of including rotation estimation into SPITE.

¹The device should not require continuous human intervention to function properly.

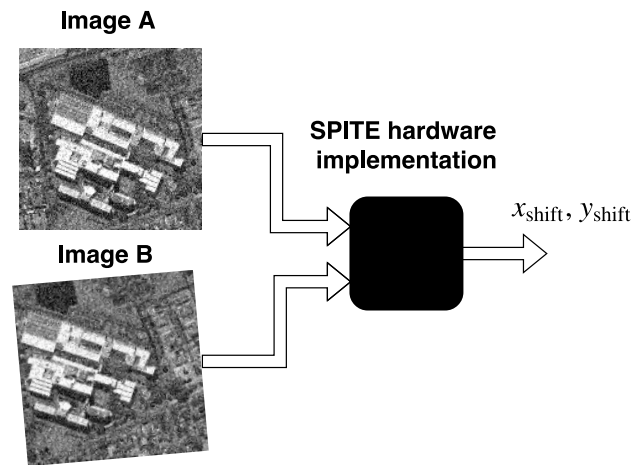


Figure 1.2: The work required to be completed by the hardware implementation. Source image [1]

1.2 Overview

Section 1 provides the overview and geometry of this thesis’s topic ‘Sub-Pixel Image Translation Estimation on a Nanosatellite Platform’, along with its scope.

Section 2 provides the background of the satellite environment, satellite systems, and current hardware technologies which SPITE may be performed on. Lastly we discuss various mathematical image registration methods which may perform SPITE, along with an end-point application.

Section 3 further investigates promising image registration methods in more detail, and how they may obtain sub-pixel results. Lastly first order runtime and resource requirements for promising methods are tabulated.

Section 4 compares promising image registration methods with one-another in simulation. Thereafter we determine the chosen algorithm’s image size, and further characterise the algorithm in terms of range, type of error, and performance under simulated real-life scenarios where noise and rotations are present.

Section 5 compares two specific hardware platforms. Special emphasis is placed on architecture, benchmark results and possible radiation mitigation options.

Section 6 discusses the implementation for the chosen hardware platform, in all its various aspects.

Section 7 discusses tests and results of accuracy, runtime and power for the chosen hardware platforms implementation.

Section 8 concludes the project with a final discussion, and points at future hardware and software work-items which may/will improve the current SPITE implementation.

NOTE: when referring to Sections which have not yet been covered chronologically within the text, there is no need to read that Section to understand the current one, it is there for completeness. Subsequent Sections will refer back in order to tie loose ends if necessary.

Chapter 2

Background

The SPITE hardware prototype will need to perform in a space environment, therefore satellite environments and systems are discussed. Various suitable hardware categories, as well as potential image registration methods are investigated. Lastly an end-point application for SPITE – image stacking – is discussed.

2.1 Satellite environment

The satellite environment, namely outer space, is by no means moderate in comparison to the Earth’s environment. Unique factors which need to be taken into account when designing a payload suitable for a satellite are discussed.

2.1.1 Thermal

Thermal awareness is critical to satellite performance, especially when temperatures ranges are exaggerated due to eclipse and sunlit parts of the orbit. There are three ways in which thermal energy is transferred: *conduction*, *radiation* and *convection*.

The sources of thermal energy acting on a satellite in orbit are: direct solar flux (DSF) (1.367 W/m^2)¹, solar radiation reflected off the Earth’s surface *Albedo* ($\pm 30\%$ of DSF), Earth Infrared ($237 \pm 21 \text{ W/m}^2$) [12]. Internal sources also generate heat e.g. friction and electrical activity.

The satellite’s main method of cooling is via radiation, as the vacuum-like surrounding is not conducive for convection, and any conduction will only move heat from one part of the satellite to another.

Various satellite subsystems such as power systems [13] and imagers [14] are temperature sensitive, therefore, to increase lifetime and efficiency of these subsystems, any prototype needs to be *aware* of its thermal contribution. For a

¹This number varies with the Earth’s distance from the sun due to changes in season as well as the Solar cycle.

motionless device such as a processor, this is measured in energy consumption per unit of time (Watts), as this directly corresponds to generated thermal energy.

This thesis proposes an *algorithmic* prototype which is run on a chosen hardware platform. Only operational power requirements are determined to establish its thermal contribution. A full thermal investigation is out of scope.

2.1.2 Radiation event

Radiation events for this discussion can be considered as nuclear interactions mostly in the form of collisions, where the following high energy particles may be expelled: atoms, protons, neutrons, electrons, positrons and photons [15]. Radiation levels, and consequently the adverse effects on electronics in space are much greater than experienced terrestrially due to the Earth's magnetic field acting as a shield. For a good introduction of the sources, fluctuations of intensity with regards to orbit-parameters, and time of year see [16], however a sufficient discussion of the *effects* on sequential logic circuits are provided. The experienced effects are broken up into two main categories: cumulative and singular effects.

Only singular effects are investigated further, since we desire to see from an *algorithmic* perspective what may be altered to reduce malfunctioning, a component lifetime comparison and selection is outside the scope of this study.

Single Event Effects (SEE), the 'Unwanted or erroneous response from an electronic device triggered by the passage of a high energy particle through the active region of that device.' [15], can be subdivided into the following categories [17][18][19]:

1. Single Event Transient (SET) occurs when a transient voltage is induced by the charge alteration at a specific point in a logic circuit due to an ionised particle strike. This will propagate through the whole combinational logic.
2. Single Event Upset (SEU) occurs when a SET is latched ie. a sequential logic element like a flip-flop stores this corrupted transient result for future use.² Another cause may be direct disturbance of a memory circuit's state as the deposited charge from an ionising particle strike causes one of the transistors to switch as if it received a valid input.
3. Multi-bit Upset (MBU), similar to SEU, but when logic is packed in a tighter format, a single particle track may cause multiple upsets as opposed to only one.

²In colloquial language this may be referred to an unintentional bit-flip or a bit-upset

4. Single Event Latch-up (SEL) occurs when the device enters an abnormal high-current state. If the device is not permanently damaged due to this current, power cycling may restore the device to normal operation.

The only effects considered for mitigation are SEU and MBU as they can be mitigated from a low-level algorithmic-functional implementation approach, which is part of the project scope. Mitigation abilities of selected hardware is discussed in Sections 5.1.3, 5.2.3 and further possible approaches proposed in Section 6.7. The remaining effects may be mitigated from a higher-system level, e.g. power-cycles etc. and these are out of scope of this study.

2.1.3 Relevant to next design phase

The following phenomena, unique challenges to the space environment, are mentioned for completeness. They are however only relevant to subsequent design phases, and are out of scope for this thesis. Orbital debris [20], considerable vibration during launch phase [21], outgassing due to vacuum [22], oxidisation and erosion due to atomic oxygen most significant at LEO, structural changes to polymers due to ultraviolet radiation and extreme thermal changes due to eclipse/sunlit part of orbit [23]. Physical-layout, material and component selection along with risk-evaluations, all form part of good design methodology.

2.2 Satellite systems

Complete satellite systems are complex, balancing the needs of various sub-systems in a small form factor is a challenge. A system engineer therefore needs to know how a payload will interact with the current integrated payloads on a certain satellite. Payload metrics are therefore crucial, not only as a performance metric, but from a *system impact* assessment and mission analysis. The hardware metrics required in the scope (Section 1.1) are now qualified:

Power: power requirement will influence the thermal design (Section 2.1.1), the power requirement is also important for the power budget. Therefore an average power and time required to finish one full calculation is crucial.

Operating frequency: the processing platform's clock frequency directly influences the time per calculation and power requirements. See Appendix A.2 to see the relation between frequency, voltage and power. For an FPGA design, a certain operating frequency may be chosen to ease the task of integration with other sub-modules, thus max frequency will not necessarily equate operating frequency. Furthermore, depending on the final layout and operating frequency, since a physical wire with an oscillating voltage will induce electromagnetic waves (Maxwell's equations [24]), these may interfere with neighbouring subsystems, depending on their requirements.

Scalable, portable and well-defined: These properties aid easy integration where various use-cases are plausible, and is generally a good design goal for re-usable IP.

2.3 Hardware technology

Current classes of hardware technologies which may be used to solve the problem of SPITE on-board a satellite are: Microcontroller Units (MCUs), General Purpose Graphic Processing Units (GPGPUs) and Field Programmable Gate Arrays (FPGAs), these are briefly discussed.

To ground the discussion, we first discuss the smallest, most significant processing unit, the Microprocessor Unit (MPU).

2.3.1 Microprocessor unit

When discussing the MPU, we are only interested in the physical processing system, thus no peripherals or memory form part of a MPU. MPUs can be classified and compared in the following manner [25]:

1. *Bus-width:* generally 2^n bit, usually varying between 8, 16, 32 or 64bits. Larger widths enable higher data transfer (between external memory and internal registers), a larger addressing range and a richer instruction set per clock cycle.
2. *Underlying hardware:* different MPUs will have different hardware units, examples include integrated dedicated Floating Point Units (FPUs), and advanced instruction scheduling logic such as in superscalar MPUs.
3. *Clock:* the maximum frequency at which the MPU may operate directly influences throughput and instantaneous power.

2.3.2 Microcontroller

Microcontroller Units (MCUs) are a combination of an MPU with memory, I/O and other dedicated peripherals such as timers [25] on a single chip. Therefore, when investigating MCUs, the underlying MPU is crucial. MCUs come in all ranges and sizes. They execute a serial C-application, which is compiled for the specific MCU.

MCUs fulfil a whole host of applications, they are commonly used in low-power situations, or where near real-time control is required. Due to their requirement for low-power and lean functionality, they are not commonly used for large data computations. Operations span from kitchen appliance control and interfacing to time-critical control systems. The required development time for a bare-bones MCU is medium in comparison to GPGPU's and FPGA's,

as it is an embedded C-application which is created, this is if a certain layout with all the additional required peripherals is already finalised.

Besides comparing underlying MPU features and characteristics, MCUs can be classified and compared in the following manner.

1. *Memory*: the type and size of on-chip memory will influence functionality.
2. *Available peripherals*: communication such as UART and built-in timers increase the potential ability of the end-point application. Debugging logic increases rapid development. Built in clock logic which has various frequency options further increases functionality.

A ‘System on Chip’ (SoC) integrates various higher level required modules for a full system to run – hence *System* – on a single chip. These may include co-processors such as Graphics Processing Units (GPUs) and Wi-Fi modules amongst others.

This reduces time to market and increases the ability significantly. Once again the SoC’s performance is largely a function of the underlying MCU.

Since multiple MCUs have been combined to create multicore architectures, comparing the number of MCU cores, and how they interact, is another factor to consider when investigating SoCs.

To keep in step with common terminology, a highly capable MCU which is usually found on a SoC will be referred to as a Central Processing Unit (CPU).

2.3.3 GPGPU

When an advanced MCU is attached to a Graphics Processing Unit (GPU), we end up with what is commonly referred to as a General Purpose GPU (this assumes that the GPU is able to do other calculations besides *only* driving a screen output). These are usually SoCs. They are often found in modern-day mobile-phones and tablets, to handle real-time image processing. The complex high-level features that these hybrids permit, are easily exploited when running an operating system on it. Serial as well as parallel applications may be developed for such a system. The CPU is no less than a very capable MCU as discussed in Section 2.3.2, often consisting of multiple cores with some variation of a 32bit ARM architecture. Therefore a GPGPU is better suited for high performance calculations, in specific algorithms with inherent parallelism.

A thorough comparison between different GPGPU hybrids will also include a comparison of the underlying CPU architecture – an investigation similar to that of Section 2.3.2. On a high-level, number of CPU and GPU cores, maximum clock frequency and type of underlying GPU hardware is important for the type of application. Development, as well as the optimal performance of an application of such a design, will also depend on what type of libraries are available for the system.

2.3.4 FPGA

Filed Programmable Gate Arrays (FPGAs) consist of customisable logic gates and semi-customisable hard-coded highly-optimised logic blocks such as high-throughput DSP-math blocks and high-density memory sections. These are generally referred to as resources on the FPGA fabric.

Instead of creating a serial C-application, the underlying logic is mapped with the available resources on the fabric. Consider as an example the addition of two numbers. In C, the addition of two 32bit integers: `'z = x + y'`. This code is decomposed into various lines of assembly code according to the underlying architecture, which then performs this addition, precisely how this occurs is not necessary for the developer to know.

FPGAs make use of Hardware Description Language (HDL). The developer thus *describes* the type of hardware he desires – at a high or low-level of abstraction – which the synthesis and implementation tools converts to appropriate paths between the required hardware blocks. E.g., the aforementioned integer addition. Very-high-speed-integrated-circuits Hardware Description Language (VHDL) – which is used in this project – could instantiate a 32bit adder with two inputs. Both 32bit values 'x' and 'y', which are stored in registers are mapped via two 32bit buses to these inputs. The adder's output will then yield the addition of these two integers (in bus form). The distinction is thus: C-application makes use of pre-existing underlying hardware/architecture along which it moves data to and from pre-existing hardware blocks (where all this is abstracted from the developer), where HDL-synthesis and implementation will consider all underlying resources, in order to optimally map the precise functionality required using its available resources. Thus the downside to such configuration is that the developer needs to be aware of the underlying hardware to create the most optimal design. The FPGA design methodology allows for the designer to decide what degree of parallelism the underlying functionality will have. Consequently, development time is generally higher than both MCU and GPGPU designs but when done well, generally yields the most optimal algorithm decomposition.

The following needs to be considered when comparing different FPGAs: the type and quality of resources available on the fabric. FPGAs generally come with a host of hard-coded capability such as dedicated math units, easily accessible high speed local memory called Block Random Access Memory (BRAM). Recent FPGAs even have fully fledged MCUs, enabling both simple bare-metal C-applications as well as operating systems to be hosted. The type and availability of clocks and clock-management, as well as the speedgrade of the chip will influence the speed v.s. power flexibility. Other peripherals may also be hard-coded as in the case of SoCs.

2.3.5 Discussion

Image operations are generally easily mapped to parallel computational structures because the same operation is often applied at a per-pixel, per row/column or per $n \times n$ grid basis. SPITE has inherent parallelism since it is an image processing algorithm. Therefore, standalone MCU's are negated due to their inability to tackle large parallel problems efficiently.

SoCs with powerful ARM MCUs are not negated in case of small data-sizes. SoCs such as GPGPUs are perfectly suited for high-level quick development of image processing algorithms although the design may lack in low-level reconfigurability, presenting itself vulnerable to the space-based environment.

An FPGA which also supports a high-degree of parallelism, along with its configurability, also shows promise for determining SPITE. Such an FPGA in conjunction with a MCU (or CPU) – where high-level features may be added due to its easier coding style – show specific promise. Furthermore, both GPGPUs and FPGAs are able to adapt for power, frequency, portability and scalability. An in-depth comparison between two specific development boards: Jetson TK1 (GPGPU) and MicroZed 7020 (FPGA + CPU) is provided in Section 5.

2.4 Image registration methods

This section provides a discussion of various image processing techniques which may be implemented to determine transformations between two images. Given the nature of the problem (Section 1.1) and the possibility of being implemented on an FPGA, the *validity* and *suitability* of these methods will be discussed.

In this discussion *Transformation* refers to any combination of: a shift, rotation, change in scale, and any shear or warp resulting in a possibly distorted image. A *translation* refers to a (x, y) shift only, and is thus a subset of a transformation. Note in Section 1, there is no shear, warp, or scaling involved in the modelled geometry – only rigid translations need to be determined. *Image Registration* is the commonly used term for determining the transformation. In each of the following descriptions, the initial image is represented by I_0 . I_1 is a transformed and mostly overlapping version of I_0 .

Image registration methods can be sub-divided into *feature-based* and *area-based* methods [26].

2.4.1 Feature-based methods

A geometrically intuitive process of image registration is a feature detection method. Given sufficient corresponding sets of matched features, various types of transformations may be determined. Where larger sets of corresponding features are available for two images, estimating more complex transformations

becomes viable. A general step-by-step overview of feature-based methods is as follows [26][27]:

1. *Feature Detection*: I_0 and I_1 are convolved with a mask to determine the location of points of interest, referred to in literature as Control Points (CPs). The type of mask will determine the type of features that are highlighted. These may be corners, edges, or specific regions.
2. *Feature Descriptor*: The identified features are assigned a descriptor. The ability to differentiate properly between the various features is dependant on the uniqueness, stability, independence and invariance of the descriptor [27].
3. *Feature Matching*: The various features' descriptors in I_0 are compared with those of I_1 , generally a sum of squared differences type method is used to determine the similarity between the potential feature matches.
4. *Transformation Estimation*: Given the nature of the distribution of the matched features, and foreknowledge of the appropriate type of the model which best describes the expected transformation, the transformations parameters are estimated.

Although this concept is intuitive, these various steps allow for multiple points of failure namely *localisation*, *matching* and *alignment* errors [26].

Localisation error is the error induced due to the inaccuracy of the CP's coordinates. These discrepancies are as a result of the type of mask used to convolve during feature detection and the calculation of the centroid of the CP. Methods which do not calculate the CPs to sub-pixel accuracy, lose critical information in this first stage, feature detection.

Matching errors occur when corresponding CPs are incorrectly matched. This will either be as a result of an image with highly repetitive content or as a result of bad choice of descriptors. Zitová and Flusser [26] claim that a trade-off may be necessary between the descriptors'

- *invariance*, descriptions of the same feature across I_0 and I_1 need to be the same
- *uniqueness*, unique descriptions for different features
- *stability*, the feature descriptor for I_0 and its corresponding features descriptor in I_1 , should still be similar *if* there is a slight unknown deformation between the two
- *independence*, given a vector feature description, the elements thereof should be functionally independent

Alignment error is the difference between the chosen transformation model and the actual geometric transformation. This occurs when an inaccurate transformation model is assumed, or when the parameters for the specific model are inaccurately determined. When I_0 is mapped to I_1 according to the relevant transformation with its calculated parameters, interpolation will be required between discrete values, this will also yield a certain error. Note that both localisation and matching errors will aggravate the observed alignment error.

Validity: Using a common feature-based method such as Scale Invariant Feature Transform (SIFT) or Speeded-Up Robust Features (SURF³) will solve the problem of determining translation. Variations of both SIFT and SURF are able to estimate translation to sub-pixel accuracy [28][29][30].

Suitability: Consider the constraints of the problem mentioned in the scope (Section 1.1). Benefits such as scale, shear, warp and rotation estimation, are an added benefit, but not required. Although efficient FPGA implementations of such are possible on an FPGA [31], the scope is better described as a *feature-tracking* endeavour [32].

A quick study of [31] will show the non-triviality of such an FPGA implementation. Given that more is gained from a feature-based implementation than is required, area-based methods are investigated in hope of a simpler yet effective, robust and accurate method to achieve SPITE.

2.4.2 Area-based methods

Area-based methods are best suited in cases where the main transformation is translation. In the presence of more complex transformations, these methods' efficacy deteriorate [26].

There are three main categories in area-based methods [26]. *Correlation*-like methods, *Fourier* methods and *Mutual information* methods. Correlation and Fourier methods are investigated in more detail pertaining to SPITE. Mutual Information methods are ignored as their main purpose is multi-modal matching; matching between images of similar scene but using different sensor types as is prevalent in the medical field.

Two specific methods, phase correlation and Normalised Cross Correlation (NCC) are introduced, further investigation will be conducted on their ability in Section 3.2.

³Both SIFT and SURF are proprietary algorithms, where a license is required for industrial use.

2.4.2.1 Phase correlation

Phase correlation – a Fourier method – relies on the fact that a spatial shift will be evident in the *phase* of the Fourier transform thereof. This method is robust in the presence of narrow-band noise, geometric distortions and changes in illumination between subsequent images [33][34].

Consider two images, $f(x, y)$ and $g(x, y)$, where $g(x, y)$ is a copy of $f(x, y)$, translated by a cyclic shift (x_0, y_0) in the x and y direction [32]. The two dimensional Fourier transforms are given by:

$$F(\xi, \eta) = \mathcal{F}\{f(x, y)\} \text{ and } G(\xi, \eta) = \mathcal{F}\{g(x, y)\} \quad (2.1)$$

making use of Fourier's *time-shifting* property⁴

$$\begin{aligned} G(\xi, \eta) &= \mathcal{F}\{g(x, y)\} \\ &= \mathcal{F}\{f(x - x_0, y - y_0)\} \\ &= F(\xi, \eta)e^{-j2\pi(\xi x_0 + \eta y_0)} \end{aligned} \quad (2.2)$$

Recall from signal theory that the shift does not influence the amplitude of the Fourier transform, only the phase spectrum is influenced, therefore the value of the shift is contained in the *phase* spectrum. If this phase-relationship is to be retrieved, the cross-power spectrum of $f(x, y)$ and $g(x, y)$ is employed:

$$\frac{F(\xi, \eta)G^*(\xi, \eta)}{|F(\xi, \eta)G^*(\xi, \eta)|} = e^{-j2\pi(\xi x_0 + \eta y_0)} \quad (2.3)$$

where $G^*(\xi, \eta)$ refers to the complex conjugate of $G(\xi, \eta)$. Making use of the inverse Fourier transform and the *duality* property,

$$\mathcal{F}^{-1}\{e^{-j2\pi(\xi x_0 + \eta y_0)}\} = \delta(x - x_0, y - y_0) = c_{phase_norm}(x, y) \quad (2.4)$$

a Dirac-delta with the peak located at the shifted value is retrieved [33].

Validity This method is able to determine SPITE given the maximum of $c_{phase_norm}(x, y)$ with its surrounding elements is found and used with a sub-pixel method [34]. Furthermore, [36] mentions that when the translation is a spatial shift (not cyclic spatial), the phase correlation surface will be “approximately zero everywhere” except at the corresponding translation. This is important since the image translation is not cyclic, however the peak is still prominent. This allows us to use this in practical image translation scenarios.

Suitability: Consider Eqs (2.3) and (2.4), there are two Fourier transforms, a point-wise complex multiply, a point-wise division and one inverse Fourier transform. This method is suitable as it is simple enough from a data control point of view to implement relatively rapidly onto an FPGA.

⁴All Fourier properties mentioned adhere to the convention provided by Lathi and Ding [35].

2.4.2.2 Normalised cross correlation

Cross correlation is the second area-based method that is investigated in depth. It is more susceptible to noise [26] and rotations [37] than phase correlation. Furthermore it is susceptible to scale and perspective changes as well as being computationally more expensive [38] than phase correlation. Since standard cross-correlation is also susceptible to changes in brightness between consecutive images, it is imperative that it is normalised properly [39].

Consider Figure 2.1 for clarification of the subsequent discussion – two partially overlapping images, with a certain window and template size.

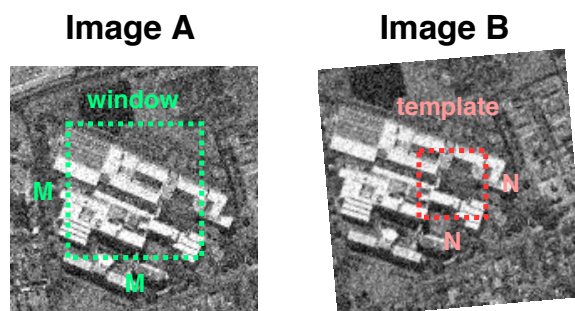


Figure 2.1: Window and template of the original image (left) and its translated companion (right), note how the template has to be fully contained within the window, and how Image B has a slight rotation present. Source image [1].

Normalised Cross Correlation (NCC) yields the following cross correlation coefficient matrix [38]:

$$c_{NCC_norm}(x, y) = \frac{\sum_{x,y}[f(x, y) - \bar{f}_{u,v}][t(x - u, y - v) - \bar{t}]}{\sqrt{\sum_{x,y}[f(x, y) - \bar{f}_{u,v}]^2 \sum_{x,y}[t(x - u, y - v) - \bar{t}]^2}} \quad (2.5)$$

where t is the moving template, \bar{t} is its mean, $f(x, y)$ is the part of the initial image which falls under the moving template at coordinate offset (u, v) , and $\bar{f}_{u,v}$ is its mean. Consider the numerator of Eq. (2.5), if

$$f'(x, y) = f(x, y) - \bar{f}_{u,v} \quad (2.6)$$

and

$$t'(x, y) = t(x, y) - \bar{t} \quad (2.7)$$

then

$$c_{NCC_norm}(x, y)_{\text{Numerator}} = \sum_{x,y} f'(x, y)t'(x - u, y - u) \quad (2.8)$$

this may be computed by Fourier transforming it:

$$c_{NCC_norm}(x, y)_{\text{Numerator}} = \mathcal{F}^{-1} \left\{ \mathcal{F} \{ f'(x, y) \} \times \mathcal{F}^* \{ t'(x, y) \} \right\} \quad (2.9)$$

where $\mathcal{F}^* \{ t'(x, y) \} = \mathcal{F} \{ t'(-x, -y) \}$ from the *time reversal* property, which states that $\mathcal{F}^* \{ a(x) \} = \mathcal{F} \{ a(-x) \}$. This holds for two dimensions as well.

The problem with this approach is that $\bar{f}_{u,v}$ is the mean *under the moving template* and needs to update for every set of (u, v) combinations, implying 2×2 D-FFT's and 1×2 D-IFFT will have to be calculated for *every* point in the full (u, v) range. If however Eq. (2.5)'s numerator is expanded using the definitions of $f'(x, y)$ and $t'(x, y)$, the equation is simplified in the following neat manner [38]:

$$\begin{aligned} c_{NCC_norm}(x, y)_{\text{Numerator}} &= \sum_{x,y} [f(x, y)t(x - u, y - v) - \bar{t}f(x, y) \\ &\quad - \bar{f}_{u,v}t(x - u, y - v) + \bar{f}_{u,v}\bar{t}] \\ &= \sum_{x,y} f(x, y)t'(x - u, y - v) - \bar{f}_{u,v} \sum_{x,y} t'(x - u, y - v) \end{aligned} \quad (2.10)$$

$t'(x, y)$ has a mean of zero, since by definition it has already been removed: $t'(x, y) = t(x, y) - \bar{t}$, therefore $\sum_{x,y} t'(x - u, y - v) = 0$, then Eq. (2.10) becomes

$$c_{NCC_norm}(x, y)_{\text{Numerator}} = \sum_{x,y} f(x, y)t'(x - u, y - v) \quad (2.11)$$

Note how this removes the dependency on $\bar{f}_{u,v}$ in the numerator (Eq. (2.9)). Now the Fourier method may be applied to calculate the numerator, and is only required once.

Consider the denominator in Eq. (2.5): $\sum_{x,y} [t(x - u, y - v) - \bar{t}]^2$ it only needs to be computed once. The iterative part of the denominators calculation is $\sum_{x,y} [f(x, y) - \bar{f}_{u,v}]^2$. Re-computing the 'under the template mean' for each combination of (u, v) is computationally taxing, Lewis [38] reduces this by using a Sum area table (SAT)⁵ optimisation, discussed in detail in Section 3.2.2.2.

The final form of NCC, as proposed by Lewis [38] is:

$$c_{NCC_norm}(x, y) = \frac{\mathcal{F}^{-1} \left\{ \mathcal{F} \{ f(x, y) \} \times \mathcal{F}^* \{ t(x, y) - \bar{t} \} \right\}}{\sqrt{\sum (t(x, y) - \bar{t})^2} \sqrt{\sum (f(x, y) - \bar{f}_{u,v})^2}} \quad (2.12)$$

Validity NCC is able to determine SPITE given the maximum of $c_{NCC_norm}(x, y)$ with its surrounding elements is found and is used for a sub-pixel method [39].

⁵Often referred to in literature as an integral image

Suitability Once again the control of this problem is simpler than its feature-based counterparts. It is somewhat more complex than phase correlation, since more effort is put into determining the normalisation factor for each element in $c_{NCC_norm}(x, y)$, this is investigated in depth in Section 3.2

2.4.2.3 Fourier based rotation determination

Section 1.1 mentions that rotation estimation is desirable, if it is easy to integrate with a SPITE operation. *De Castro and Morandi* [33] propose a Fourier method similar to phase correlation. It makes use of the following two facts:

1. the rotation of an image in the spatial domain (around any centre), yields a subsequent equivalent angular rotation of the magnitude spectrum of the image's Fourier transform
2. the magnitude spectrum of the Fourier transform of the image is invariant to translation in the spatial domain

Therefore, the rotation can be determined, the image re-rotated, after which the translation may be determined [37][32]. But how precisely is the perceived rotation in the magnitude spectrum determined? By taking the polar transform of the magnitude spectrum⁶, it maps the rotational movement to 1-dimensional translation. Thereafter simple cross correlation may be used to determine the rotation θ_{rot}

$$\theta_{rot} = \mathcal{P}\langle |\mathcal{F}\{f(x, y)\}| \rangle \star \mathcal{P}\langle |\mathcal{F}\{g(x, y)\}| \rangle \quad (2.13)$$

where \star refers to cross correlation, $|\dots|$ refers to the magnitude of, $g(x, y)$ is the rotated and translated version of $f(x, y)$ and $\mathcal{P}\langle \dots \rangle$ is the polar transform:

$$\begin{aligned} r &= \sqrt{(x - x_c)^2 + (y - y_c)^2} \\ \alpha &= \tan^{-1}\left(\frac{y - y_c}{x - x_c}\right) \end{aligned} \quad (2.14)$$

where (x_c, y_c) is the centre coordinate.

Now, after θ_{rot} is determined, $g(x, y)$ must be re-rotated by $-\theta_{rot}$. Now the translation may be determined using either of the aforementioned area-based methods. Note how *this* rotation is determined using similar operations (FFT) as for both NCC and phase correlation, making this a viable option to integrate with SPITE. Similarly, $\mathcal{P}\langle \dots \rangle \star \mathcal{P}\langle \dots \rangle$ from Eq. (2.13) can be determined using 2D-FFTs, since cross-correlation easily maps to Fourier domain calculations, due to the *Convolution Theorem* [35].

A large part of this method uses 2D-FFT calculations. Since these math blocks will already be required for the translation estimation (either NCC

⁶The log-polar transform may be used to determine rotation as well as scale [36], this is sometimes referred to as the Fourier-Mellin transform

or phase correlation), the re-use of such blocks makes Fourier based rotation determination attractive. Note, if hardware blocks need to be re-used, the transform sizes will need to be the same.

Therefore, to determine the feasibility of a potentially lean rotation estimation using pre-existing hardware (2D-FFT), we investigate this in simulation. A simulation showing rotation determination accuracy v.s. input image dimension is required. Input image dimension will determine the transform size. Since rotation determination accuracy is a function of the resulting polar image, a simulation which brings these concepts together is shown in Appendix A.3. This is to perform a quick feasibility study to determine whether such an algorithm implementation is sustainable. The simulation discusses the relationship between the resolution of the resulting *polar* image and its corresponding Fourier magnitude image via a simulation. This simulation has multiple values of polar *resolution* v.s. Fourier magnitude image *resolution*. The outcome is that the resolution required to *accurately* determine a rotation, requires a too-large polar resolution. For instance, to approach 0.1° rotation estimation error, using a 511×511^7 original image, a 512×512 polar-transformed image is required (see Figure A.2). To store both image's polar-transforms, along with the original image to re-rotate, requires

$$\begin{aligned} \text{Memory} &= 2 \times (\text{Mem}_{\text{polar}} + \text{Mem}_{\text{original}}) \times \text{Bytes per float} \\ &= 2 \times (512^2 + 512^2) \times 4 \\ &= 8\text{MB} \end{aligned} \tag{2.15}$$

as a quick comparison, the MicroZed 7020 has a total of $\approx 0.6\text{MB}$ BRAM. This exceeds the FPGA's BRAM resources by far. Thus this is infeasible for a lean, portable, FPGA implementation.

2.4.3 Other methods

In order to place NCC and phase correlation in their broader context, we briefly mention other common methods and novel approaches. This is by no means an exhaustive list. Drawbacks of each, and why it disqualifies itself from SPITE for this context is mentioned. This is not to say that these reasons invalidate such an approach, rather that at first glance they seem less appealing to investigate further.

Sequential similarity detection algorithm or SSDA, states that the full accuracy is only required near the maximum of the cross-correlation coefficient. Thus in-situ correlation sums are calculated at random locations, this then estimates where the maximum cross correlation is. SSDA however can not guarantee that

⁷This image is 511×511 such that there is a central pixel, this simplifies the geometry, but could be altered. Such an FPGA implementation would still use the same amount of memory that a 512×512 implementation.

the maximum cross correlation coefficient is found [38]. Reliability is important, as SPITE needs to run *autonomously*, this requires the best answer without fail.

Gradient Descent Search assumes that the translation between frames is small. Even if our described context fits this description, the inherently serial nature disqualifies any potential parallelism which may be gained from a custom hardware implementation [38].

Hierarchical motion estimation makes use of multi-resolution image pyramids. This method first searches for a significant match in a representative lower resolution image. Since it is calculating at a lower resolution, this involves fewer steps. This then ‘seeds’ where the next iteration of the search will occur. The disadvantage to this method is that if not enough low frequency information is present in the higher resolution image, the down-sampled representative image will lose high frequency content. This once again reduces the reliability on the best match [32].

Image Pre-interpolation Debella-Gillo and Käab [39] interpolate the original image to a higher resolution with bicubic interpolation *before* applying the NCC feature tracking method (as in Section 2.4.2.2). This is based on the assumption that the original image’s content was band-limited and that the optical system sampled above Nyquist [34]. They then [39] compared image pre-interpolation to standard NCC performed on the original image. Sub-pixel accuracy was determined by using the correlation surface’s areas of interest (maximum) to inform the following three sub-pixel methods: bicubic interpolation, parabola fitting and Gaussian fitting. They found all three of the sub-pixel methods to be inferior to the *pre-interpolation* method. The memory required to accommodate an up-sampled image, will result in a bulky FPGA implementation, possibly requiring external memory interfacing, reducing portability. Furthermore, the image content and sensor assumption further reduces portability.

2.4.4 Current implementations

Currently there is significant research being conducted on image-registration *after* downlinking the data-set. Since these calculations are run on the ground, a lean, embedded implementation is not required for success. Therefore, the investigation is turned to *on-board* processing.

Rais [40] provides an extensive summary of possible algorithms to implement on-board a satellite, he gives the Gradient Based Shift Estimation (GBSE) method a positive review. GBSE relates the shift between two subsequent images by determining the temporal derivative, and then determining the image gradient which corresponds to the sub-pixel shift. GBSE uses Lucas-Kanade to solve an overdetermined system. Rais [40] claims when varying illumination is present between the images, two alternative methods should be used. One of these, phase correlation, has been discussed in Section 2.4.2.1. In this case histogram equalisation is required to make GBSE viable. Furthermore, GBSE

can only determine sub-pixel shifts, so another method needs to ‘seed’ this algorithm with the integer-pixel translation. This GBSE implementation was proposed to Centre National d’Études Spatiales, the French space agency, for a potential future mission, showing the algorithm’s maturity and quality of implementation.

The research which we found that is closest to what we propose [41] – multi-band, on-board image registration to sub-pixel accuracy – makes use of the attitude control system to generate grids. Adding dependency on another satellite subsystem, attitude control system, adds complexity. We aim for an easily portable solution, therefore this research is not investigated further. Their solution was developed and benchmarked for an FPGA. They also discuss the influence thereof on the image processing chain, both before and after data downlink, these are helpful metrics – some of which will be addressed in our research.

2.4.5 Discussion

Feature-based methods, due to their complexity, are not investigated further. Complexity is an issue since the algorithm may need to be implemented on an FPGA, where simplicity is key for rapid development. Area-based methods were also investigated. Phase correlation and NCC were discussed in some detail, they provide an integer-pixel translation estimation. Sub-pixel methods, required for both to yield sub-pixel results, will be discussed in Section 3.1. An area-based rotation determination was briefly examined, however an appropriate implementation would require too much memory to be considered ‘lean’. No processing on the image before applying either area-based method was discussed despite [26] mentioning how NCC may benefit from pre-processing techniques such as filters. Current methods implemented on-board satellites were briefly investigated, findings confirmed phase correlation as a viable option when under varying illumination.

We identify NCC and phase correlation as the most suitable candidates for implementation, these are developed further in the following chapters.

2.5 Desirable SPITE application

To validate the suitability of SPITE for an end-point application, the effect of its accuracy is determined on one of the simplest algorithms which is dependant on accurate registration: image stacking.

2.5.1 Image stacking

Image stacking, sometimes referred to as averaging or image/pixel averaging, is the process whereby sequential images which have overlapping scene content

are spatially realigned with one-another, so that a given pixel index across all the aligned images refer to the same spatial position. The average of each index across the various images is then stored in the averaged image, see Figure 2.2. This operation is performed to reduce the random noise present in the individual images⁸.

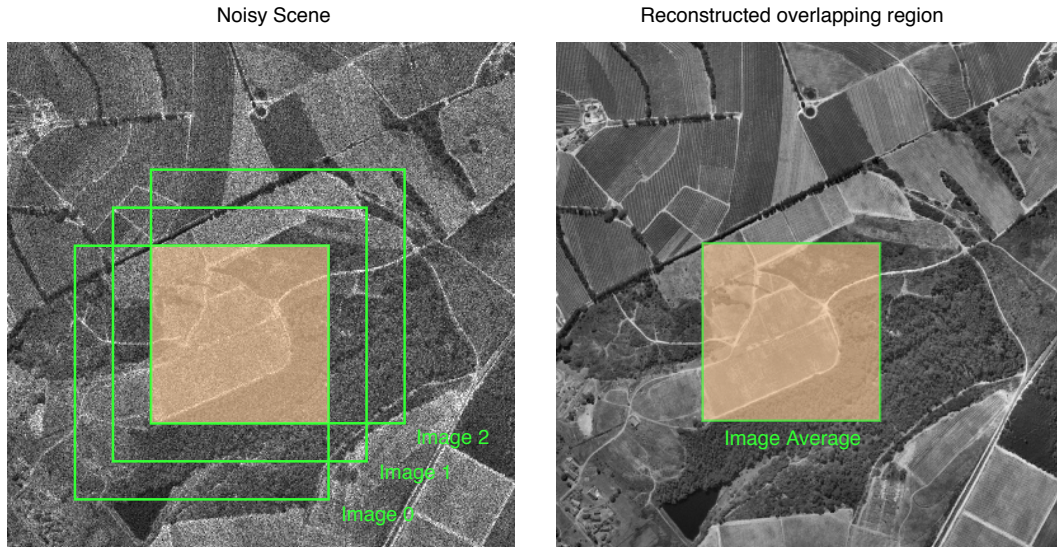


Figure 2.2: The process of image stacking, three partially overlapping images (left) are re-aligned such that the overlapping region may be averaged, resulting in the lower noise content averaged image (right). Source image [1]

The clarity of the resultant image is dependant on the accuracy to which the translations are determined and the number of images averaged. Inaccurate results may produce images which are worse off than the individual images. Errors which are biased in one direction will also reduce the quality of the resultant image, as the effect of this error will accumulate as the number of images averaged increases⁹.

Banks [8] states that when image pixel noise is spatially and temporally uncorrelated, the SNR obtained by stacking M images is higher than any of the individual input images by a factor of \sqrt{M} . See Appendix A.4 for the derivation.

⁸This method of noise removal at such an early stage of the image processing pipeline may be preferred to filtering, since filtering alters the image's frequency content.

⁹This is the case if: $T_{01} = \text{SPITE}(I_0, I_1)$, $T_{12} = \text{SPITE}(I_1, I_2)$, $T_{23} = \text{SPITE}(I_2, I_3)$, the translation between I_0 and I_3 will be $T_{01} + T_{12} + T_{23}$ the sum of all previous SPITE calculations. In the presence of a bias, the bias in a certain direction is added three times, accumulating in the bias direction.

2.5.2 Image noise

Image stacking seeks to mitigate noise present in an image. Therefore, a quick discussion regarding sources of noise, Signal to Noise Ratio (SNR) and how to create a specific noisy image is helpful.

Consider a CCD imaging system, according to [42], this system can be divided into three sub-systems mentioned in order of operation:

1. photons arrive at a certain photosite (a.k.a pixelsite / pixel pad). Electrons are freed proportionally to the number of photons arriving due to the photoelectric effect. The amount is determined by measuring the voltage.
2. the voltage undergoes a certain non-linear compression
3. the voltage, representative of the pixel value, is converted from a voltage to a digital value

For this discussion, various sources of noise will be divided into two categories: *Contingent* and *non-deterministic* noise. Contingent noise, despite having a non-deterministic nature, is classified by the author as *unique* to the satellite system. Thus such sources can usually be mitigated through pre/post processing or calibration. Sources include interference from other satellite sub-systems [43] as well as noise inherent to the imager such as *Pattern noise* [44]. These will be ignored as they vary across satellite systems.

Non-deterministic noise, which is generally trickier to mitigate, consists of *dark*, *read out*, and *Shot noise* [45] [46].

Dark noise: $N_{\text{Dark}} \propto \sqrt{I_{\text{Dark}} T_{\text{Integration}}}$. It is a function of dark current and integration time. For a given temperature, a certain rate of thermal electrons are generated inside the silicon of the CCD following a Poisson distribution. This temperature dependency is a result of lattice defects or metal impurities within the imager's silicon lattice [47]. Dark current describes this rate [46] and occurs in sub-system 1.

Read-out noise is the error introduced during the reading of the analogue pixel value. The main contributor thereto is the pre-amplifier's noise figure: sub-system 3. For this discussion, let all other electronic noise from the imaging system fall under this banner. Peebles [48] discusses how thermal noise approximates band-limited white noise, thus the electronic noise, which can be considered thermal noise, approximates band-limited white noise. Quantisation noise which occurs in sub-system 3 and falls under read-out noise, is equivalent to rounding off, Peebles [48] discusses that this 'rounding-off' action introduces errors which are uniformly distributed.

Photon noise, also known as Shot noise or photon-Shot noise¹⁰, is due to the rate at which the photons arrive at the CCD pad. This variation is dependant on the number of incident photons. Photon noise is proportional to the incoming photons such that $N_{\text{Photon noise}} \propto \sqrt{\text{Photons}}$. This deviation follows the Poisson distribution [46]. The significance thereof is that for well-lit images, the number of incident photons will be significantly large. When this is the case, the Poisson distribution tends towards a Gaussian distribution [44].

All the non-deterministic noise contributors except photon noise, despite being random in nature, are still *system dependant*. Therefore, we only model the effect of photon noise in order to keep results as generic as possible.

When discussing an image's SNR, the definition found in Banks [8] is adopted:

$$\text{SNR} = \frac{S}{\sigma} \quad (2.16)$$

where S is the intensity of the pixel and σ is the noise component, the standard deviation of S due to noise.

2.6 Summary

Satellite environments were investigated, clearly radiation mitigation techniques are required, and power runtime statistics need to be determined.

Two specific types of hardware platforms show promise, namely GPGPU SoCs, and FPGAs, these seem to meet important design requirements which are desirable for integration with nanosatellite systems.

NCC and phase correlation were found to be suitable candidates for image registration for the context. The benefit of a possible end-point application – image stacking – is presented, namely a \sqrt{M} gain in SNR, where M images are re-aligned and averaged.

¹⁰Sometimes in literature spelt 'Shott' instead of 'Shot', name derived from Walter H. Shottky, the physicist who discovered the Shottky phenomena [49].

Chapter 3

Algorithm Adaptation and Conceptual Comparison

NCC and phase correlation are the two algorithms which are investigated further. Both provide a 2-dimensional function evaluated on a regular integer grid as an answer, as shown in Eqs (2.4) and (2.12). This output will be referred to as the correlation coefficient $c_{norm}(x, y)$. Therefore, some method of interpolation is required to retrieve a sub-pixel answer. Two methods are presented, polynomial least squares regression and a Taylor series expansion. Lastly, a runtime and memory requirement investigation and comparison is conducted between NCC and phase correlation.

3.1 Sub-pixel methods

3.1.1 Method 1: polynomial least squares regression

This method models the area around the maximum of $c_{norm}(x, y)$ as a polynomial of the form:

$$z(x, y) = ax^2 + by^2 + cxy + dx + ey + f \quad (3.1)$$

Consider $z(x, y)$ as a height of the coefficient matrix at the coordinates (x, y) as shown in Figure 3.1. Eq. (3.1) can be represented in matrix form as

$$\mathbf{z} = A\mathbf{x} \quad (3.2)$$

where \mathbf{z} is a column vector containing the heights of various points, A is a matrix holding their coordinates, and \mathbf{x} is the unknown parameter column from Eq. (3.1).

$$\mathbf{z} = \begin{bmatrix} z(x_1, y_1) \\ z(x_2, y_2) \\ \vdots \\ z(x_n, y_n) \end{bmatrix}, A = \begin{bmatrix} x_1^2 & y_1^2 & x_1 y_1 & x_1 & y_1 & 1 \\ x_2^2 & y_2^2 & x_2 y_2 & x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^2 & y_n^2 & x_n y_n & x_n & y_n & 1 \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}$$

Note the difference between the column vector \mathbf{x} , and the various coordinates (x_n, y_n) .

The method of least squares is used to estimate the unknown parameters: a, b, c, d, e, f . The maximum of the correlation coefficient indicates the *integer* offset of the translation, therefore the maximum value of $c_{norm}(x, y)$ with its surrounding values are used as observations for \mathbf{z} . This is visualised in Figure 3.1:

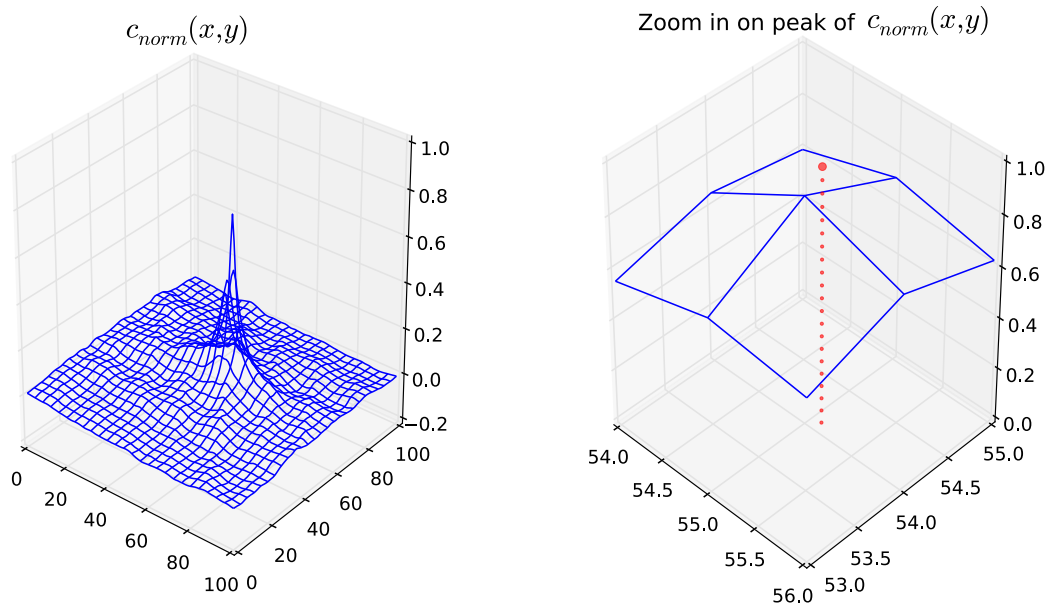


Figure 3.1: Left: $c_{norm}(x, y)$ clearly showing the integer-spaced output grid. Right: peak which is zoomed in, where $n = 9$. Note the red dot: the final sub-pixel translation determined by this polynomial least squares regression. The number of points selected may be altered, here it is 3×3 , but 5×5 and 7×7 are also investigated (in Section 4.3), this is what is referred to as ‘Poly 3×3 ’ or ‘Poly 5×5 ’ etc.

Note that n has to be 6 for a unique solution to be found, because of the 6 unknowns which are solved. A square is taken around the maximum, where $n = 9, 25$ or 49 , due to squares of dimensions $3 \times 3, 5 \times 5$, and 7×7 respectively.

Thus the system is overdetermined ($n > 6$) and mathematically no solution may exist. However, using least squares we get a suitable result. Having $n > 6$ also builds some robustness into the system, reducing the effect of zero mean Gaussian noise on the parameter's result.

Note however that a solution is not *guaranteed* from the data's side, e.g. when all the pixels are zero, \mathbf{x} is a zero column vector, therefore both denominators of Eqs (3.6) and (3.7) are zero, yielding no solution.

Least squares, a regression method which gives an estimate for the parameters in \mathbf{x} , yields the following:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{z} \quad (3.3)$$

for more detail on least squares regression see Appendix B.1. Once the parameters are approximated, the sub-pixel coordinate location of the maximum needs to be determined. This is achieved by taking the partial derivatives of Eq. (3.1) in both x and y and setting them equal to zero:

$$\begin{aligned} \frac{dz}{dx} &= 2ax + cy + d = 0 \\ x &= \frac{-d - cy}{2a} \end{aligned} \quad (3.4)$$

and

$$\begin{aligned} \frac{dz}{dy} &= cx + 2by + e = 0 \\ y &= \frac{-cx - e}{2b} \end{aligned} \quad (3.5)$$

after substituting Eq. (3.4) into Eq. (3.5) the coordinates to the turning point is

$$y_{tp} = \frac{cd - 2ae}{4ab - c^2} \quad (3.6)$$

and

$$x_{tp} = \frac{-d - cy_{tp}}{2a} \quad (3.7)$$

where (x_{tp}, y_{tp}) are the coordinates of the turning point, the sub-pixel maximum which corresponds to the sub-pixel translation.

3.1.2 Method 2: Taylor series expansion

This method – used in the astronomical image-registration module [50] – makes use of a second order Taylor series expansion to estimate the peak. Once again the maximum and the surrounding values of $c_{norm}(x, y)$ are used to fill a 3×3 matrix: B . This makes it convenient to compare directly to a polynomial least squares regression which in one case (Poly 3×3) also makes use of the same 3×3 grid.

The discrete first and second order derivatives of B are: $B_x, B_y, B_{xx}, B_{yy}, B_{xy}$. The middle element of these matrices is used to form $f_x, f_y, f_{xx}, f_{yy}, f_{xy}$ after which the peak is determined to be [51]:

$$x_{\text{peak}} = \frac{f_{yy}f_x - f_yf_{xy}}{f_{xy}^2 - f_{xx}f_{yy}} \quad (3.8)$$

$$y_{\text{peak}} = \frac{f_{xx}f_y - f_xf_{xy}}{f_{xy}^2 - f_{xx}f_{yy}} \quad (3.9)$$

these peak values then need to be offset by the integer offset, the location of the B within $c_{\text{norm}}(x, y)$. B. Welsch [50] mentions that ‘this method may be subject to bias.’ A full range test is done on the candidate sub-pixel method, to determine any bias in Section 4.5.

3.2 Further investigation and comparison

In this section, the phase correlation and NCC algorithms are broken down to better grasp their inner workings, pitfalls pros and cons. This exercise is invaluable, as findings will yield fundamental insights into how to best approach an FPGA design in its various sections. In this, and all subsequent discussions the *window* will refer to an $(M \times M)$ section of a larger image, and the *template* will refer to a smaller $(N \times N)$ sub-part of the translated image, where $N = M/2$ (see Figure 3.2). A section of the image is used in order to make the calculation less dependant on the total image, and more dependant on the local image data. Naturally, the size of the window and template influences runtime and memory requirement. Such a comparison is made between NCC and phase correlation implementation in Section 3.2.2.

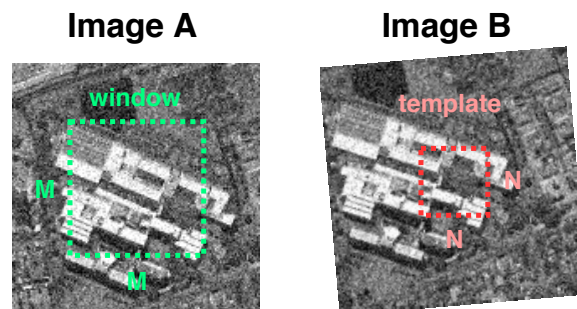


Figure 3.2: Window and template of the original image (left) and its translated companion (right), note how the template has to be fully contained within the window, and how Image B has a slight rotation present. Source image [1].

3.2.1 2-Dimensional Fourier transform

Both NCC and phase correlation contain Two Dimensional (2D) Fourier Transforms which form the core of the implementation. In both cases, the calculation makes use of discrete points in a 2D-grid, the *type* of Fourier Transform is the *discrete* Fourier/Inverse Fourier Transform. Computing this Discrete 2D-Fourier Transform is achieved by making use of the *Fast* Fourier Transform (FFT). Various implementations of the FFT exist. For instance Radix-2, Radix-4, Radix-8, where constraints are put on data length $\#_{data}$. These are such that $\#_{data} = n^x$ where $n \in 2, 4, 8$ for the various radices respectively. For this discussion, only the Radix-2 algorithm is used as it holds the lowest constraint on the data length ($\#_{data} = 2^x$) and therefore math IP-blocks and functions for this method are widely available.

It is important to note that the FFT is not an approximation of the discrete Fourier transform, it uses the periodicity and symmetry to define a different sequence of the calculations, termed the butterfly operator. The 2D-FFT is a simple extension of the 1D-FFT. Simply put, given a matrix with a columns and b rows, the a -point 1D-FFT is taken along each row and the result stored in a temporary matrix, then the b -point 1D-FFT is taken along every column of the temporary matrix. This is mathematically represented as

$$F[k, l] = \sum_{r=0}^{b-1} \sum_{q=0}^{a-1} f[q, r] e^{-j2\pi(\frac{kq}{a} + \frac{lr}{b})} \quad (3.10)$$

and for the inverse transform

$$f[q, r] = \frac{1}{ab} \sum_{l=0}^{b-1} \sum_{k=0}^{a-1} F[k, l] e^{j2\pi(\frac{kq}{a} + \frac{lr}{b})} \quad (3.11)$$

thus if a 2D-FFT is to be calculated on an $(M \times M)$ image, it will be equivalent to running $2M$ sets of 1D-FFTs.

3.2.2 Runtime and memory analysis

To provide a first-order estimate of runtime and resource requirements, some assumptions are made to simplify the problem.

Even though these are naive assumptions, given the complex and highly optimised mathematical hard-ware architectures, this Section is trying to estimate the *algorithm's* characteristic, and is done in such a manner to decouple its performance from the underlying hardware platform's ability. The assumptions made, are:

1. a basic sequential math unit with no inherent parallel ability performs the calculation

2. the time taken to calculate a square root, division and multiplication are all equal, and referred to as one Runtime Unit (RTU)
3. no memory resources are re-used
4. additions and subtractions are considered to have negligible running time

The algorithms total runtime and memory requirements are now discussed in Sections 3.2.2.1 and 3.2.2.2. Runtime and memory requirements are mostly discussed in the generic term using n , where n refers to the number of points to the generic dimension of either window or template. When the specific calculation occurs, n is swapped out with M or N , the dimension of the window or template.

3.2.2.1 Phase correlation

Recall from Eqs (2.3) and (2.4), the full algorithm for phase correlation:

$$\delta(x - x_0, y - y_0) = \mathcal{F}^{-1} \left\{ \frac{F(\xi, \eta)G^*(\xi, \eta)}{|F(\xi, \eta)G^*(\xi, \eta)|} \right\} \quad (3.12)$$

Consider a window $f(x, y)$ and template $g(x, y)$ with dimensions $(M \times M)$ and $(N \times N)$. Zero padding both windows to $(M + N - 1 \times M + N - 1)$ is required to ensure that the appropriate result is not aliased, yielding a true cross correlation result [52]. However, if only the template is zero-padded to dimension $(M \times M)$, significant memory resources may be saved, this is at a cost of the data at the edges being aliased, whereas the central $(N \times N)$ data will be untainted. Figure 3.3 represents the mathematical flow for this scenario.

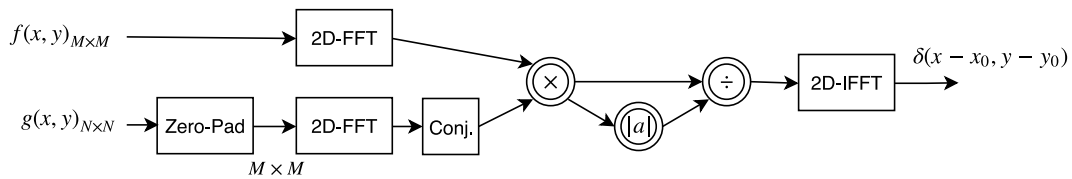


Figure 3.3: Phase correlation mathematical flow, note how the same 2D-FFT block may be used to calculate $F(\xi, \eta)$ and $G(\xi, \eta)$.

A breakdown of these various sub-parts is given in terms of runtime and memory requirements assuming 32bit (4 byte) floating point precision is used.

2D-FFT/IFFT: Runtime is $\frac{n}{2} \log_2(n)$ (number of complex multiplications for radix-2 FFT [52]) for the 1-dimensional case, thus $2n \times \frac{n}{2} \log_2(n)$ for the 2D case given a matrix of size $(n \times n)$. As this represents the number of complex multiplications, it is equivalent to 4 normal multiplications and 2 additions. Note the normalising factor of $\frac{1}{ab}$ in Eq. (3.11), this is ignored in the discussion,

as each element is divided by the same value, and not present in Eq. (3.10). In terms of memory resources required for on 2D-FFT/IFFT, for an $(n \times n)$ matrix, each resultant element will consist of a real and complex value, the memory required for the result is thus $4 \times 2 \times (n \times n) = 8n^2$ bytes.

Point-wise multiplication: Given a matrix size of $(n \times n)$, the number of complex multiplications is n^2 , therefore $4n^2$ multiplications are computed. Memory requirements: 1 real and 1 complex value is required per element, therefore $2 \times 4 \times n^2$ bytes required to store the result of $F(\xi, \eta) \times G^*(\xi, \eta)$.

Magnitude: this requires the calculation $\sqrt{re^2 + im^2}$, which given an $(n \times n)$ matrix, requires $2n^2$ multiplications, n^2 square root calculations and $1 \times 4n^2$ bytes of memory.

Point-wise division: Given an $(n \times n)$ matrix, the calculation requires $\frac{re+im}{x}$, equivalent to $2n^2$ divisions, and $2 \times 4n^2$ bytes of memory.

Zero-pad and conjugate: are ignored, since the contribution to running time is not significant. Consider the conjugate operation in C-code, just one bit flip per element, and the Zero-Pad can be implemented as just sending zero-data to the 2D-FFT block in the correct order.

$$\begin{aligned}
\text{Phase Runtime} &= 2 \times \text{2D FFT} + \text{Mult.} \\
&+ \text{Magnitude} + \text{Div.} + \text{2D IFFT} \\
&= 2 \times \{4 \times M^2 \log_2(M)\} + \{4M^2\} \\
&+ \{3M^2\} + \{2M^2\} + \{4 \times M^2 \log_2(M)\} \\
&= 12M^2 \log_2(M) + 9M^2 \text{ RTU's}
\end{aligned} \tag{3.13}$$

$$\begin{aligned}
\text{Phase Memory} &= 2 \times \text{2D FFT} + \text{Mult.} \\
&+ \text{Magnitude} + \text{Div.} + \text{2D IFFT} \\
&= 2 \times \{8M^2\} + \{8M^2\} \\
&+ \{4M^2\} + \{8M^2\} + \{8M^2\} \\
&= 44M^2 \text{ bytes}
\end{aligned}$$

3.2.2.2 Normalised cross-correlation

Consider Eq. (3.14) first derived in Section 2.4.2.2, where $f(x, y)$ is the $(M \times M)$ window and $t(x, y)$ is the $(N \times N)$ template, where $N = M/2$:

$$c_{norm}(x, y) = \frac{\mathcal{F}^{-1}\left\{\mathcal{F}\{f(x, y)\} \times \mathcal{F}^*\{t(x, y) - \bar{t}\}\right\}}{\sqrt{\sum (t(x, y) - \bar{t})^2} \sqrt{\sum (f(x, y) - \bar{f}_{u,v})^2}} \tag{3.14}$$

For convenience, the same colour-coding will be used throughout this document. Let's make the assumption that if $M = 2^a$ and $N = 2^{a-1}$, if t is zero padded to

dimensions $(M \times M)$, then acceptable aliasing happens. This is re-using the reasoning of zero-padding to $(M \times M)$ as with the phase correlation.

Figure 3.4 shows a simplified version of the mathematical flow of NCC. Consider the 2D-IFFT block: its input is $(M \times M)$ complex data, the output is $(N \times N)$ real data. Note that the output of the 2D-IFFT block is actually $M \times M$, however only the un-aliased $(N \times N)$ data is used. A breakdown

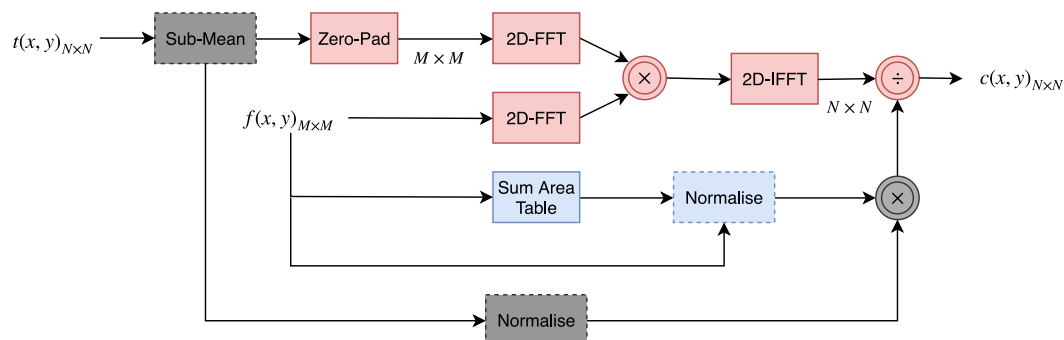


Figure 3.4: Simplified NCC mathematical flow

of the various computationally expensive math blocks is provided, the same assumptions are adhered to as in the discussion in Section 3.2.2. For the sake of brevity, mathematical functions already discussed in Section 3.2.2 will not be repeated.

Sum Area Table¹ (SAT): Blue [53] is a method to determine the average pixel value of a sub-section of a larger image. This is used to easily determine the mean of a sub-section of $f(x, y)$, namely $\bar{f}_{u,v}$. It consists of two parts, first the sum area table is *populated*. To populate this table, the table entries $s(x, y)$ are calculated using the original pixel values of the image in the following manner:

$$s(x, y) = f(x, y) + s(x - 1, y) + s(x, y - 1) - s(x - 1, y - 1) \quad (3.15)$$

the time required to populate is ignored, since it occurs once, and only consists of additions/subtractions. Secondly, in order to *determine the mean* of a certain sub-section, the four boundary coordinates are read from the table. To retrieve the mean of an arbitrary sub-section of $f(x, y)$ with boundaries x_0, x_1, y_0, y_1 where $x_0 < x_1, y_0 < y_1$, this simple formula is used:

$$\bar{f} = \frac{s(x_1, y_1) + s(x_0 - 1, y_0 - 1) - s(x_0 - 1, y_0) - s(x_0, y_0 - 1)}{(x_1 - x_0) \times (y_1 - y_0)} \quad (3.16)$$

This demonstrates the computational gain of implementing a sum area table: 3 additions and 1 division, to determine the mean of a sub-section of any size. This action of retrieving the mean of a sub-section of $f(x, y)$ occurs N^2

¹Also referred to as an ‘integral image’ in literature.

times, since *each* component of $c(x, y)$ is normalised with a unique sub-section mean. Although $c(x, y)$ is $M \times M$, only the central $N \times N$ part of $c(x, y)$ contains useful information. Since we read from it N^2 times, the runtime is N^2 divisions and N^2 multiplications, therefore $2N^2$ RTU. Compare this with $N^2 - 1$ additions with 1 division, required when calculating this mean the traditional way: where all the pixels in the relevant area are summed, then divided by the number of pixel in the area. Since the sub-section mean needs to occur N^2 times, the assumption that an addition is negligible becomes imprecise (in total $N^4 - N^2$ additions). Despite a significant gain in calculating various means at constant runtime, the downside to SATs is that memory needs to be allocated to store the table. The table size, assuming each entry is a 32bit float², $4 \times M^2$ bytes are required.

Normalisation: Grey this occurs once, runtime considerations are thus negligible.

Normalisation: Blue this is calculated N^2 times (once for every element in $c_{norm}(x, y)$). It consists of N^2 subtractions (subtracting the mean – calculated by the SAT – from each relevant pixel in the sub-section), 1 square root after which the result is multiplied with the grey normalisation factor. Thus only 1 RTU's and negligible memory usage. It is however crucial to note that for *each* element in $c_{norm}(x, y)$, this branch needs to calculate a specific normalisation factor. Hence occurring N^2 times.

Sub-Mean subtracts the template's mean from the template, it occurs once, therefore has an inconsequential runtime.

$$\begin{aligned}
\text{NCC Runtime} &= 2 \times \text{2D-FFT} + \text{Mult.} \\
&+ \text{2D-IFFT} + \text{Div.} \\
&+ \text{SAT} + \text{Norm.} + \text{Mult.} \\
&= 2 \times \{4 \times M^2 \log_2(M)\} + \{4M^2\} \\
&+ \{4 \times M^2 \log_2(M)\} + \{N^2\} \\
&+ 2N^2 + N^2 + N^2 \\
&= 12M^2 \log_2(M) + 4M^2 + 5N^2 \text{ RTU's}
\end{aligned} \tag{3.17}$$

$$\begin{aligned}
\text{NCC Memory} &= 2 \times \text{2D FFT} + \text{Mult.} \\
&+ \text{2D IFFT} + \text{Div.} \\
&+ \text{SAT} \\
&= 2 \times \{8M^2\} + \{8M^2\} \\
&+ \{8M^2\} + \{4N^2\} \\
&+ \{4M^2\} \\
&= 36M^2 + 4N^2 \text{ bytes}
\end{aligned}$$

²Using a 32bit float to hold this value is overkill, but is used for the generic case, this guarantees no overflow occurs.

3.2.2.3 NCC vs phase correlation: theoretical performance comparison

The runtime and memory, using Eq's (3.13) and (3.17), are used to visually compare NCC with phase correlation. This is shown in Figure 3.5, the influence of M 's magnitude on memory usage and runtime for both methods.

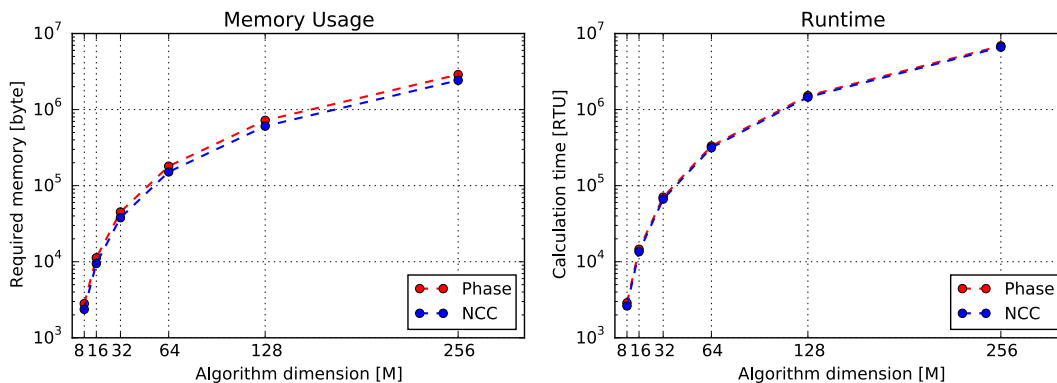


Figure 3.5: Memory usage (left) and running times (right) of both algorithms for varying sizes of M , where $N = M/2$. Note the logarithmic y-scale.

Note how both runtime and memory requirements are similar. This is because the methods' only significant difference is the manner in which they determine the normalisation factor. Phase correlation's runtime and memory is slightly higher since the normalisation factor is calculated on the complex value *before* the 2D-IFFT is taken, whereas NCC normalises *after* 2D-IFFT, where the data is assumed to only have a real component. Division on real-only data is $2\times$ faster than division on complex data. The memory disparity, is significant for the same reason as above: the normalisation of phase correlation happens on complex data, and thus requires more runtime.

3.3 Summary

Two specific methods have been introduced which will obtain the sub-pixel component of the translation. Both NCC and phase correlation have been investigated on a lower-level. Both expected runtime and resource requirements have been investigated and compared for various sizes of M . The following questions and statements arise, they are investigated in Section 4:

- The Taylor series sub-pixel method is computationally simpler than the polynomial least squares. Which of these methods provides better accuracy?

- Phase correlation's control structure for the data is much simpler than NCC's, this method does however pay for it with higher memory usage and runtime. Which algorithm provides better estimates?
- The size of M and consequently N will significantly influence the running times and memory usage of the implementation, to such a degree that a trade-off between accuracy and size of M, N needs to be made.
- The reuse-ability of the FFT blocks allow either a high throughput high resource utilisation, or a low throughput low resource utilisation implementation.

This chapter has laid the groundwork on how and what needs to be compared between phase correlation and NCC, and polynomial least squares and Taylor series sub-pixel methods in simulation. Thereby we will tweak parameters M and N to establish the fastest and leanest possible accurate option.

Chapter 4

Algorithm Benchmarking

This chapter furthers the investigation into NCC and phase correlation in a Python simulation setting. Comparisons are made between the two algorithms, where accuracy is the driving factor - addressing the findings from Section 3.3. An accuracy of total translation estimation error that is less than 0.1pixels, is considered a success.

First, the effect of SPITE accuracy on an end-point application – image stacking – is quantified. Thereafter the best sub-pixel method is matched to the superior integer translation estimation algorithm. To reduce the resource usage and runtime, suitable dimensions for the window and template are evaluated. These dimensions influence the expected range across which the algorithm may function, the accuracy and error over this range is reported. Lastly, to simulate effects of the environment on SPITE, accuracy is determined in the presence of noise and small rotations.

4.1 Preamble: generating test-images

A thorough test is only as good as the underlying test’s data and design. With regard to the test data: it is difficult to quantify what and how much useful information a ‘typical’ image may contain¹. Therefore, the following assumption is required as a starting point: if SPITE is required, the imager will be imaging a point of interest; features are inherent to a point of interest, this will provide significant information for the algorithm to function. In its most basic form, the test will provide an image A with a translated version thereof, image B. The translation will be a shift along both x and y axes. The SPITE algorithm then has to determine what the translation between the two images is. The error is defined as the difference between the applied translation and the estimated translation. Images A and B are captured through an ideal theoretical imager.

¹Chapter 11 of [11] gives an introduction to this field of study.

To produce simulated test images, a high resolution aerial photograph of the town of Stellenbosch² is utilised [1], subsequently *all* aerial image shown in this and linked Sections are sections of this image. See Appendix B.5 to view the various sections of the image chosen to represent a variety of input data.

The initial image has a Ground Sampling Distance (GSD³) of 0.5m and is converted to an 8bit grey scale format. This is treated as ‘ground truth’ denoted as GSD_{original} , the perfect noiseless image which represents the ground accurately. This ground truth image is then manipulated in such a way as to mimic the lower resolution image which the theoretical imager would capture.

This theoretical imager is assumed to have ideal optics: where distortion is negligible, such that the resolution and contrast of the captured image is a function of GSD only. A GSD of 2.5m is assumed for this theoretical imager. Note that $GSD_{\text{theoretical}} = 5 \times GSD_{\text{original}}$. The following set of operations are applied to the original image to produce the test images A, B; where a desired sub-pixel translation of $x_{\text{shift}}, y_{\text{shift}}$ is required where $x_{\text{shift}}, y_{\text{shift}}$ are multiples of 0.2:

1. The original image is cropped to (2000×2000) , a manageable size. Note that these dimensions are a multiple of 5.
2. The cropped image is then shifted by $5 \times (x_{\text{shift}}, y_{\text{shift}})$, an integer value shift using `numpy.roll()`. Thus for 0.2 pixel shift in resultant image, a full pixel shift is induced in this 2000×2000 image.
3. The image is now divided into super pixels of dimensions (5×5) , the average of these super pixels becomes the new down-sampled pixels’ value. This assumes that each of the CCD arrays individual sensitivity pads are geometrically square and equally sensitive to incoming light over the whole surface, as each of the 25 pixels are weighted equally.
4. The image is now converted back to 8bit format with range 0 to 255, as is typical for an optics system. The un-shifted image, image A is created in the same manner excluding step 2.

It is important to note that the smallest sub-pixel shift available to create in this fashion is a 0.2 pixel shift. All subsequent sections besides Section 4.5 require smaller shifts. To achieve this finer shift resolution, *after* step 2, pixel values are linearly interpolated. E.g. a resultant translation of 0.45 pixels would require a 2 pixel shift (step 2), then a 0.25 pixel linear interpolation, thereafter pixels are grouped (step 3) and compressed (step 4).

See Appendix B.2 for more detail regarding how the interpolation is achieved.

²The original image set is made publicly available by the City of Cape Town in accordance with the terms and conditions set out in the Open Data Policy and is available to the public at <https://citymaps.capetown.gov.za/EGISViewer> [1].

³GSD is the size which each pixel represents on the ground.

4.2 Influence of SPITE accuracy on image stacking

In order to properly compare the two sub-pixel methods, a sufficient accuracy metric is required. In Section 1.1, a starting point of ≤ 0.1 pixel translation estimation error (mis-registration) was given. What is the effect of this 0.1 pixel mis-registration on image stacking (introduced in Section 2.5.1), a potential end application?

Consider 11 images ($I_0 \dots I_{10}$) which are averaged, perfect registration and realignment will render a SNR gain of $\sqrt{11} = 3.32$ (Section 2.5.1). To determine the influence of a certain mis-registration $E_{reg.}$ on resultant post image stacking SNR, the total error of consecutive image registrations is set to $E_{reg.}$ pixel.

Therefore, translations between consecutive images before stacking, will model the effect of mis-registration on image stacking performance. Here, Err_{number} represents the mis-registration due to the SPITE algorithm between two consecutive images, such that $Err_0 = SPITE(I_0, I_1) = Err_1 = SPITE(I_1, I_2) = Err_2 = \dots = SPITE(I_9, I_{10}) = E_{reg.}$ pixels. This error is defined as:

$$E_{reg.} = \sqrt{x_{error}^2 + y_{error}^2} \quad (4.1)$$

where x_{error} and y_{error} is the mis-registration in both x and y directions in the image plane.

Assume the worst case: the error is biased in one *direction* and always has the maximum magnitude. After 11 images are stacked, a ‘blur’ of 1 pixel will be present when inter-frame mis-registration = 0.1 pixels. Note that this derivation assumes that no error occurs in the *realignment* process.

Aim: Determine the relationship between SPITE error (mis-registration) on image stacking’s ability to *improve* SNR.

Method: Conduct a simulation where 11 images are stacked. The mis-registration will be represented by a sub-pixel shift between consecutive images, as introduced above. Thus the SNR of the averaged image for various mis-registrations will be compared.

First, the image needs to be shifted at sub-pixel increments, after which noise is added.

Adding noise to the signal: Images with desired SNRs are generated from the noiseless, high-resolution Stellenbosch image *after* the appropriate shift was induced. The range of SNR over which the test runs is determined empirically and aesthetically. Only photon noise is modelled.

Photon noise can be modelled as a Poisson distribution, which approaches a Gaussian distribution for well-lit images [44], the images used are sufficiently well-lit. The component of photon noise is proportional to the square root of S

[46]. The effect thereof is scaled by a factor a . This allows us to create different SNR values due to photon noise.

To create a noisy image with a required signal to noise ratio of Eq. $\text{SNR}_{req.}$:

1. Determine the original image's average pixel intensity S_{avg} .
2. Determine the scaling factor a using the definition of SNR from Eq. (2.16):

$$\begin{aligned} \text{SNR}_{req.} &= \frac{S_{avg.}}{\sigma_{req.}} \\ &= \frac{S_{avg.}}{a\sqrt{S_{avg.}}} \end{aligned}$$

therefore

$$a = \frac{\sqrt{S_{avg}}}{\text{SNR}_{req.}}$$

3. Step through the original image 1 pixel at a time and update it to the following:

$$S(x, y)_{\text{new}} = S(x, y)_{\text{original}} + \text{Gauss} \left\{ \sigma = a\sqrt{S(x, y)_{\text{original}}}, \mu = 0 \right\}$$

where $\text{Gauss}\{\dots\}$ refers to drawing a sample from a Gaussian distribution with standard deviation σ and mean μ . The resulting pixel value is clipped to fit within the 0 to 255 range of an 8-bit image.

The following SNR's are selected: 2.5, 5, 10, 20, 50 and ground truth – the down-sampled image with no added noise. Figure 4.1 shows the various levels of SNR, for one of the test images. Test images have dimensions 400×400 , see step 3 of Section 4.1, however only the central 200×200 of the image was used. To include variation in image content, four image (Figures B.8 to B.11 in the Appendix) were used.

The four images are shifted in both x and y directions according to $E_{reg.}$, the direction and magnitude is constant to model the worst case scenario where the inter-frame mis-registration is constantly the largest magnitude in the same direction. Noise is then added, after which the images are stacked and averaged. The tabulated results show the SNR *after* the stacking process for a combination of various magnitudes of mis-registration along with initial SNR. The results for SNR post image stacking are averaged across all four test images. See Appendix B.3 discusses how the post image stacking SNR is determined.

Results: Figure 4.2 shows the gain in SNR due to image stacking, as a percentage of the maximum theoretical gain, $\text{SNR}_{\text{Initial}}\sqrt{n}$. The larger the initial SNR, the less is gained *initially*, however as the mis-registration approaches



Figure 4.1: One of the four test images with varying SNRs: 2.5, 5, 10, 20, 50, ground truth, from top left to bottom right respectively. Subsection of [1].

zero (0.001 pixel), the gain approaches the theoretical value. When $\text{SNR} = 2.5$, it overshoots the 100% mark, this could be due to the clipped pixel values – when a large noise is to be added to a large pixel value, it saturates to 255, therefore the SNR is already *larger* than it should be. The same is the case for pixels with values close to 0, if a ‘negative’ noise is added, it saturates at 0.

More importantly, the influence of the inter-frame mis-registration is evident. The larger this error, the less is gained from image stacking, and this follows an exponential curve. This is a result of the neighbouring pixels smearing the image – this effect increases as mis-registration increases.

Note: for images with high contrast and sharp edges, where low noise levels are present, averaging may reduce SNR for even small mis-registration. This effect is made plain in Figure 4.3, where the initial image has SNR 10, but when image stacked with 0.3 mis-registration, the SNR decreases to 7.11.

This effect may be reduced by determining SPITE in the following manner: $Err_0 = SPITE(I_0, I_1)$, $Err_1 = SPITE(I_0, I_2)$, $Err_2 = SPITE(I_0, I_3)$ etc. instead of $Err_0 = SPITE(I_0, I_1)$, $Err_1 = SPITE(I_1, I_2)$, $Err_2 = SPITE(I_2, I_3)$ etc., which was used for this simulation, to highlight the worst case scenario. Furthermore, if it can be proven that the SPITE error is not biased in any direction, this may curtail such effect.

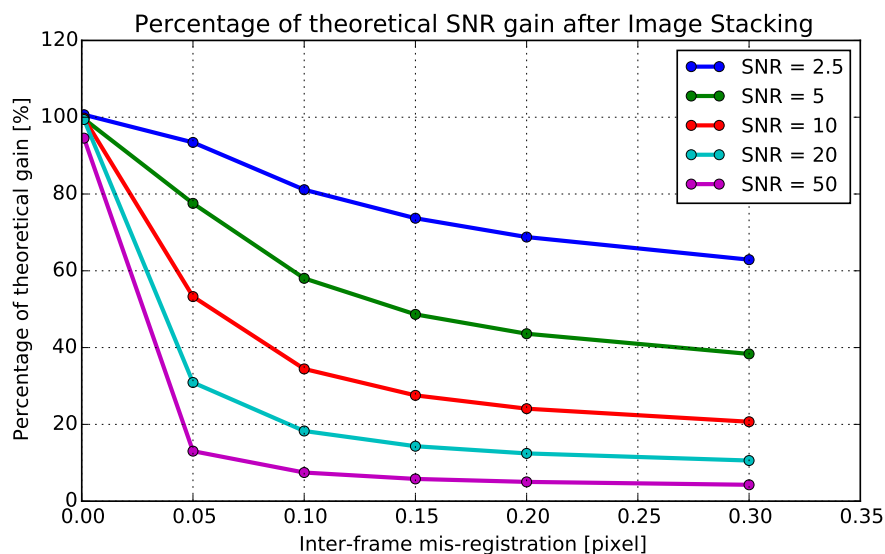


Figure 4.2: The gain in SNR due to image stacking 11 images, as a percentage of the maximum theoretical gain, $\text{SNR}_{\text{initial}}\sqrt{11}$. Any gain below $100(1/\sqrt{11}) = 30.15\%$, is a reduction in SNR.

Conclusion: Clearly, the smallest possible registration error is desirable for maximum gain from using image stacking. This provides a good starting point when considering this end-point application. It is now possible to defend why a 0.1 pixel error is desirable. For relatively noisy images which will benefit from stacking, $\text{SNR} = 20, 10$ and 5 , at 0.1 pixel mis-registration there is a significant change in slope. There is a more drastic change in slope at 0.05 pixel mis-registration – a more demanding target.

Thus Figure 4.2 is a starting point to determine: the *SPITE accuracy* required, to facilitate a certain *post-image stacking SNR*, for a set of images with a certain $\text{SNR}_{\text{initial}}$. The level of SPITE accuracy attainable for an image set with a certain SNR will be discussed in Section 4.10. Thereafter the suitability of image stacking using SPITE with a certain image set, will be established.

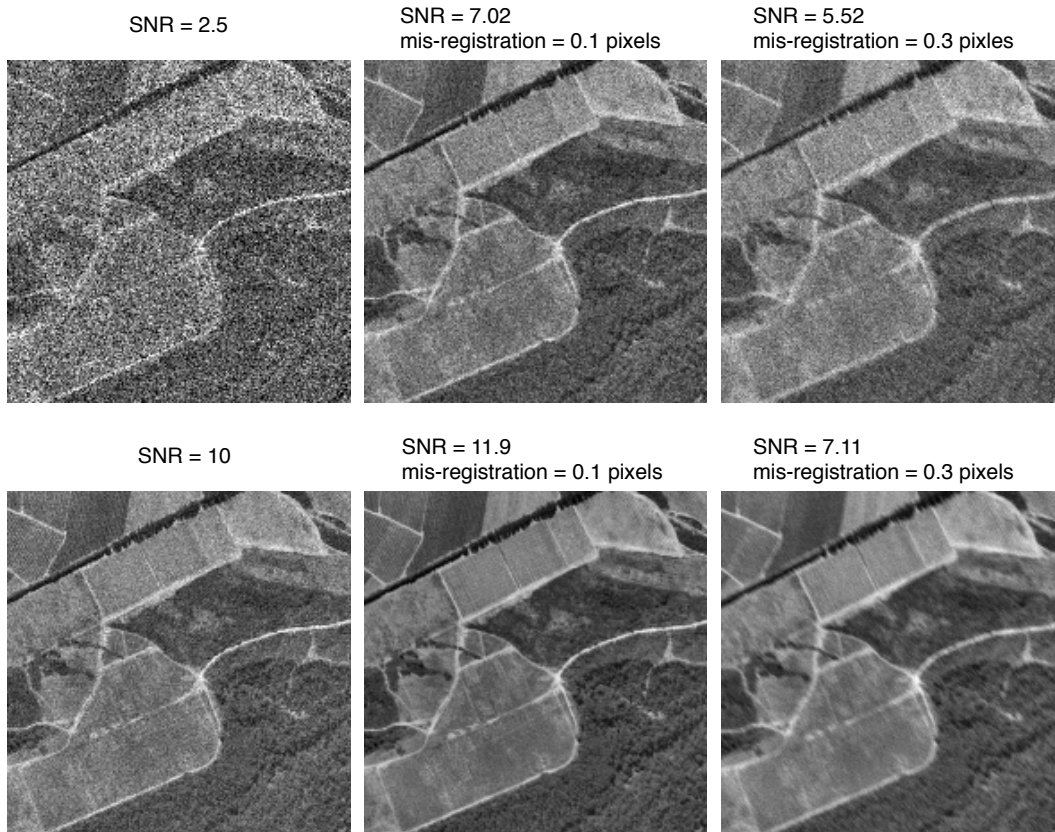


Figure 4.3: Dirty images before image stacking (left), and resultant images after image stacking, for various magnitudes of worst case inter-frame mis-registration (middle, right). For low SNR large registration errors may still increase the overall SNR after image stacking, however not the case for larger initial SNRs. Subsection of [1].

4.3 Surface fit and registration method comparison

In order to obtain the most accurate SPITE, we need to determine the best *combination* of correlation methods (NCC and phase correlation) with the sub-pixel methods (polynomial least squares regression and Taylor series expansion).

Aim: This simulation aims to match the best sub-pixel method with the best underlying correlation method by comparing the SPITE error.

Method: A set of sub-pixel shifts were induced in the x -direction only, these shifts were then estimated with the various correlation methods, using the various sub-pixel methods to determine the sub-pixel translation. To remove the uncertainty of the effect of the size of M and N , these were chosen as significantly large $M = 256$, $N = 128$. The test was conducted over a sweep of

sub-pixel shifts in increments of 0.2 pixels, over the range $[-1, 1]$. The error was measured as the Euclidean distance, such that

$$\text{Total absolute error} = \sqrt{x_{\text{error}}^2 + y_{\text{error}}^2} \text{ pixels} \quad (4.2)$$

Only one image, Figure B.8, is used. If a certain combination of correlation and sub-pixel methods does not yield desirable results with this one image which has with significant features present, then it is assumed for the general case to be less effective for the other images with less prominent features. If it is found later on in the investigation that the preferred combination is not adequate, this exercise would be revisited more thoroughly.

Results: From Figure 4.4 two observations are clear: NCC generally trumps phase correlation in terms of accuracy and the best sub-pixel method is the polynomial 3×3 least squares regression (Poly 3×3), regardless of the underlying image registration method. This small subsection of image shifts is representative of the general case, see Figure B.4 in the Appendix, it shows how the trend is present throughout a larger translation range.

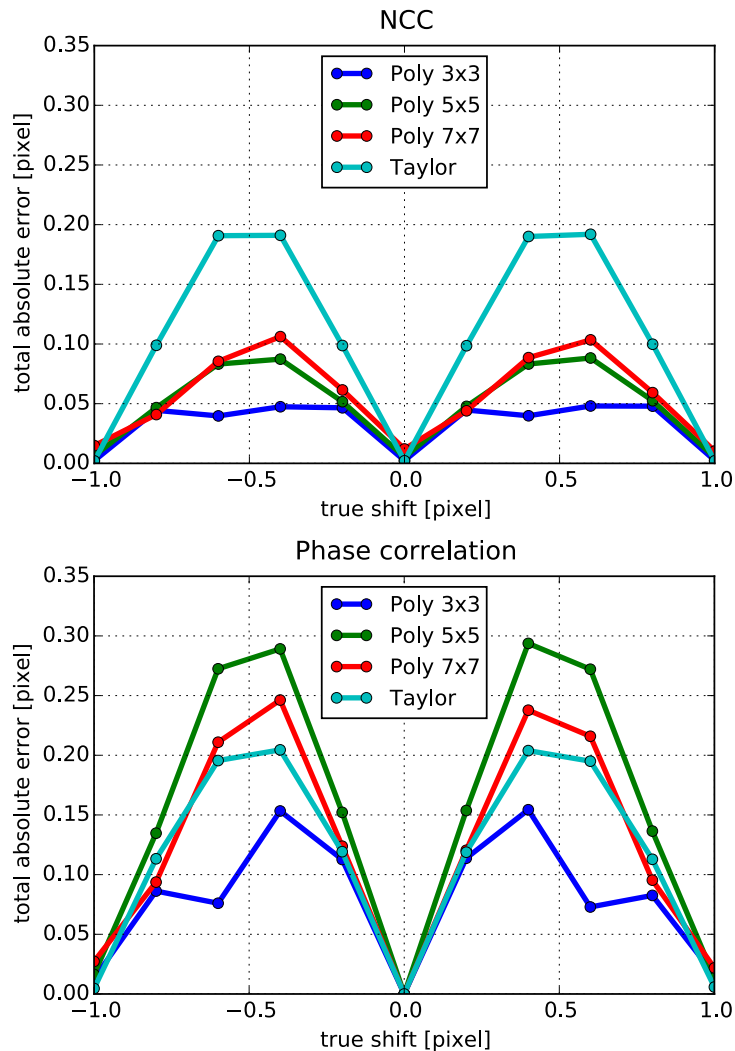


Figure 4.4: Comparison of two image registration methods along with two sub-pixel methods. Shift induced in x -direction only. Poly $N \times N$ refers to the polynomial fit with least squares method and the size of the matrix which is used to determine this fit.

Further observations may be drawn: even in this best case scenario, where there is no noise, rotation or large integer shift offset, the phase correlation method does not meet the desired metric of error < 0.1 pixel as consistently as NCC, given the sub-pixel methods. This will disqualify phase correlation from further consideration since NCC with Poly 3×3 is the most accurate in this test.

Interestingly, the Taylor series method has the worst accuracy with NCC, where this is not the case with phase correlation. This shows that the *shape* of the underlying data matters. Remember that the phase correlation results in a sharp maximum at the corresponding point, and is understandably not easily modelled by a polynomial.

Similarly, the sub-pixel prediction made by the *polynomial* method becomes *worse* the more points are used to train it (for NCC). This indicates that the implicit assumption made in Section 3.1, that $c_{norm}(x, y)$ is best modelled using a polynomial surface does not hold equally for larger sections. This is evidenced by the significant increase in error between Poly 3×3 and Poly 7×7 , i.e. the nature of the larger surface is not modelled accurately as that of a parabolic surface.

Lastly, it is interesting to note that the largest error is at the 0.5 pixel offset. This error is shown (in Section 4.5) to be a function of the algorithm, consistently yielding the largest error when the shift is at $0.5 + \text{integer}$. This maximum error trend at $x_{\text{shift}} = \pm 0.5 \text{pixel}$ is investigated further by conducting a similar simulation. This time however, the shift was induced in *both* x and y -directions simultaneously, with finer shift increments of 0.05 pixel. This was to reproduce (what would seem to be) the worst case scenario when $x_{\text{shift}} = y_{\text{shift}} = \pm 0.5 \text{pixel}$. Results in Figure 4.5 show how the error goes above 0.1 at both +0.5 offset and -0.5 offset for NCC with Poly 3x3, the significance thereof is addressed in Section 4.5.

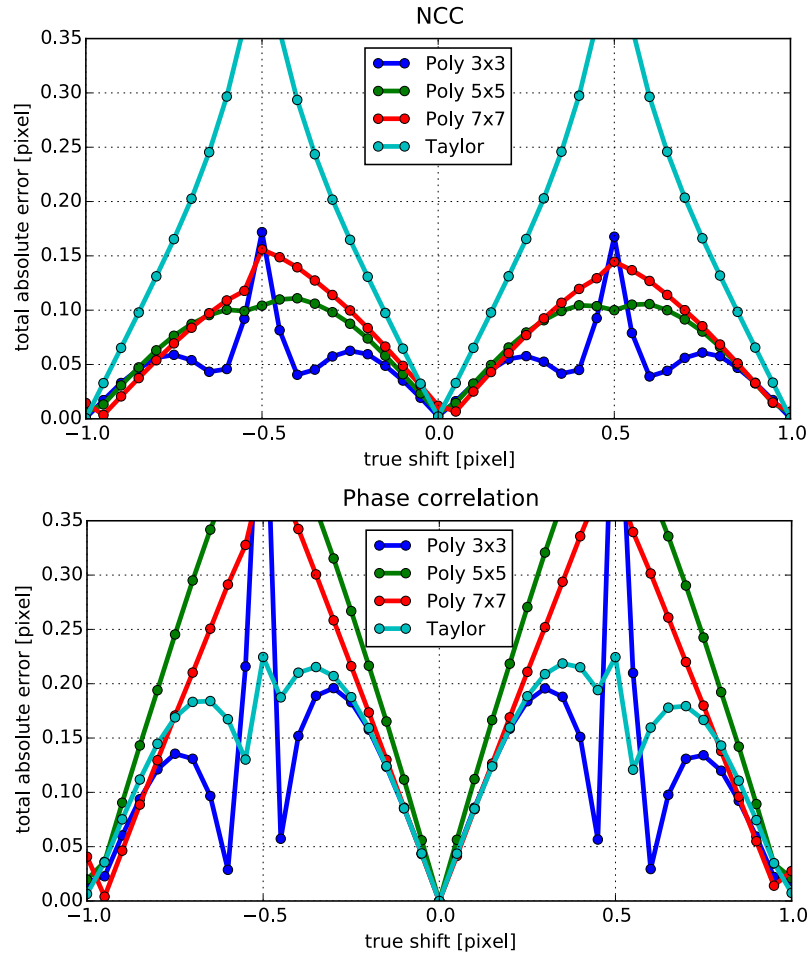


Figure 4.5: Comparison of two image registration methods along with surface-fitting methods where $x_{\text{shift}} = y_{\text{shift}} = \text{true shift}$.

Conclusion: NCC with Poly 3×3 is the best candidate for determining the sub-pixel translation, it meets requirements (error ≤ 0.1 pixel) in the range $[0, 0.45] \cup [0.55, 1]$ according to Figure 4.5. This conservative range will be used in future discussions. The worst error will be when the $x_{\text{shift}} = y_{\text{shift}} = \text{integer} + 0.5$ pixel.

4.4 SPITE required window and template dimensions

Section 3.2 shows that runtime and memory requirements are dependant on dimensions M and N . In order to minimise both of these requirements, the window size M and template size N need to be as small as possible⁴.

⁴See Appendix B.6 for an explanation why dimensions are always 2^n

Aim: This simulation compares translation estimation accuracy with size of M, N .

Method: The shifts were induced along both x and y -axes, in increments of 0.1 pixels in a range $[-1, 1]$ shift, where $x_{\text{shift}} = y_{\text{shift}}$ to induce maximum error. The translation was determined with NCC Poly 3×3 , where the sizes of M and N were varied. Absolute error was the yardstick with which the accuracy of the various sizes were compared. Four different images were used to obtain representative results over a wide range of inputs. The sub-image content for each of the images was chosen such that significant features were present in all the windows, regardless of M and N 's size.

The four images used are shown in Figures 4.6, 4.7, 4.8 and 4.9. Each is a subsection of the full Stellenbosch image [1]. The various blocks superimposed give an indication of the size of the various windows. The various colours of the blocks corroborate with the colour legend in Figure 4.10, where the smallest is 32×32 and the largest 256×256 .

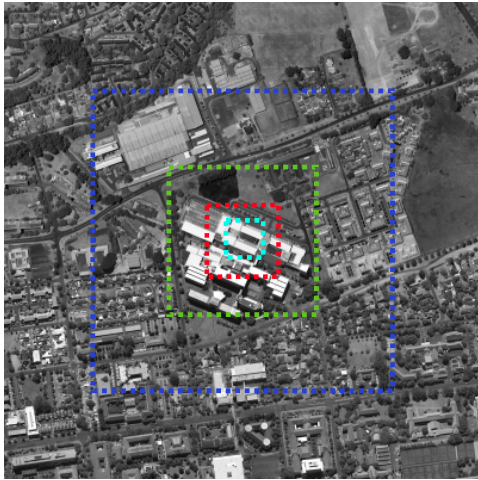


Figure 4.6: Test data, Image A: Engineering faculty

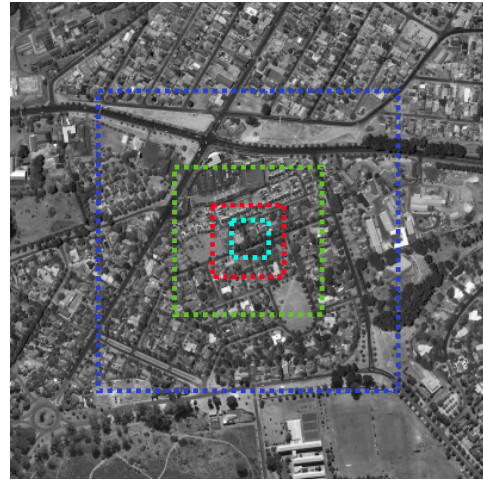


Figure 4.7: Test data, Image B: Simonswyk

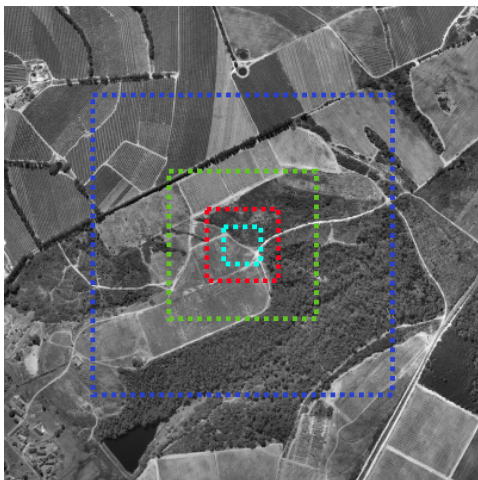


Figure 4.8: Test data, Image C: Agricultural land

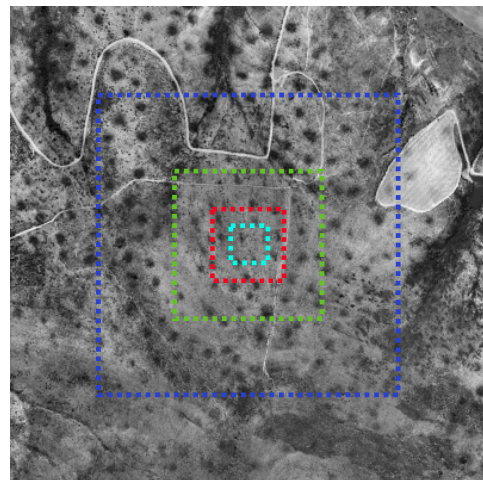


Figure 4.9: Test data, Image D: Bushes

Engineering faculty: Image A, a large building with a white roof provides multiple strong edges. *Simonswyk:* Image B, a suburban neighbourhood provides smaller, less bright features at somewhat regular intervals. *Agricultural land:* Image C, this contains cultivated land, borders consist of narrow farm ‘dirt-roads’. *Bushes:* Image D, this contains mainly weak edges, dull footpaths and clumps of shrubs.

Results: Two key observations are drawn from examining Figure 4.10 for images A, B and C: when the size of $M = 256, 128$ or 64 the trend of the absolute error is similar, whereas for $M = 32$ some undesirable trends are observed. Lastly, it is seen that significant accuracy gains are not made when increasing M from 64 for either of the images. Surprisingly Image D, the image

with the least ‘strong’ edges is the most stable. It could be that the images with harder edges will yield a sharper c_{norm} , which is not easily modelled by Poly 3×3 .

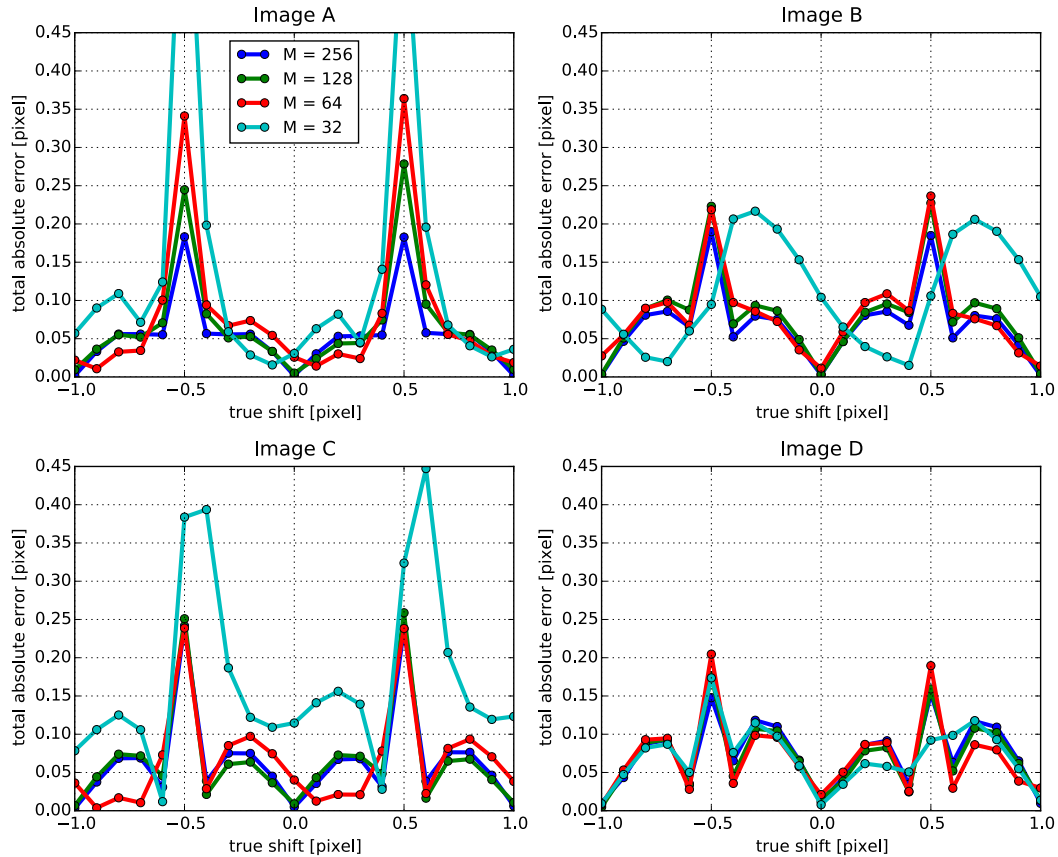


Figure 4.10: Evaluating the effect of various sizes of M on the accuracy of NCC Poly 3×3 , $N = M/2$. Results are shown across four different images.

Conclusion: $M = 64, N = 32$ is a good compromise between size and accuracy. When $M = 128, N = 64$ the accuracy gained is not much, while $M = 32, N = 16$ showed a significant reduction in accuracy.

4.5 SPITE range, offset and error

What is the effective range of pixel offset over which NCC SPITE with $M = 64$ is viable? Consider the case of NCC in 1-Dimension, where $N = 8$ and 1 pixel either side of the maximum of $c_{norm}(x)$ is still required to surface fit (Poly 3^5). Figure 4.11 shows how this results in the effective range of $[-\frac{N}{2} + 2, \frac{N}{2} - 1]$.

⁵Poly 3 and not Poly 3×3 as we are talking about the 1D-case.

Due to Poly 3×3 , the range is reduced by $\text{floor}(3/2)$, on each side, resulting in the -1 .

Naturally this can be extended to 2-Dimensions: shifts in both x and y directions, then Poly 3×3 will have range $[-\frac{N}{2} + 2, \frac{N}{2} - 1]$. This reasoning extends to the other polynomial methods, i.e., for Poly 5×5 the range would be $[-\frac{N}{2} + 3, \frac{N}{2} - 2]$ and for Poly 7×7 $[-\frac{N}{2} + 4, \frac{N}{2} - 3]$.

$$c_{\text{norm}}(x)$$

| | | | | | | | |
|--|----|----|---|---|---|---|--|
| | -2 | -1 | 0 | 1 | 2 | 3 | |
|--|----|----|---|---|---|---|--|

Figure 4.11: Effective range for 1-Dimensional NCC where $N = 8$ and Poly 3×3 used for surface fit. This extends to the 2-Dimensional case.

Aim: Determine whether Poly 3×3 with $M = 64$, $N = 32$ is stable across its defined range. Secondly, determine whether the error is biased in any given direction. Discuss what, if any, error trend is observed.

Method: Window and template size were kept constant with $M = 64$, $N = 32$. A range of $[-14, 15]$ pixel shifts in increments of 0.2 pixels were induced, where $x_{\text{shift}} = y_{\text{shift}}$. This corresponds to the range over which the algorithm is defined: $[-\frac{N}{2} + 2, \frac{N}{2} - 1]$, specifically the worst case *will* be tested since $x_{\text{shift}} = y_{\text{shift}}$.

Results: The x , y and absolute errors are shown over the translation range $[-14, 15]$ in Figure 4.12. Both x and y errors oscillate around 0, indicating that the error is unbiased.

Note the mostly periodic, unbiased behaviour interrupted by significantly large errors for Images B and C. This challenges the stability/reliability of the estimations.

In Section 4.2 the specification for $Total\ Error < 0.1$ pixel was discussed for an image stacking application, given a *worst case* scenario where there is an offset bias. Since there is no bias, this allows the specification to be relaxed somewhat. Furthermore there is no change in accuracy due to integer offset, since only the untainted section of $c_{\text{norm}}(x, y)$ is operated on, namely the central $(N \times N)$ section.

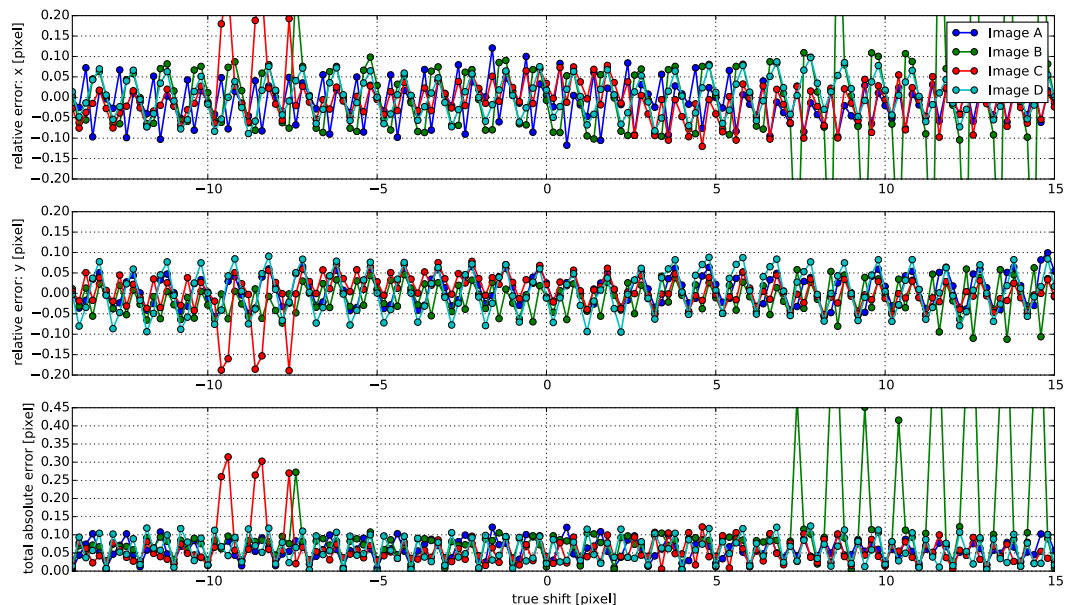


Figure 4.12: Various errors: x (top), y (middle) and total absolute (bottom) error is shown over a wide range of shifts for the 4 images : A, B, C and D.

It is apparent that the largest *periodic* error occurs when either x_{shift} or $y_{\text{shift}} = 0.5 + \text{integer offset}$. In this case, the error oversteps the 0.1pixel error requirement occasionally, and this happens periodically across the full range. It is now safe to assume that this odd behaviour is due to the nature of the problem: when a 0.5 pixel shift occurs, there seems to be a maximum uncertainty of what the precise translation is. To determine the precise cause of this mathematically falls outside of the scope of this project. The behaviour is however investigated further in Section 4.6⁶. This behaviour is amplified when shifts in *both* x and y happen simultaneously as was shown in the progression between Figures 4.4 and 4.5.

To address the influence of this periodic error on overall accuracy, it may be profitable to know how often such a periodic error occurs. Consider the presupposition that each subsequent shift will be random (as stated in the Scope), and thus each shift combination has an equal likelihood of occurring. Consider the case where only sub-pixel shifts in range $[-1, 1]$ in both axes are investigated (as an integer offset has no real impact). From a conservative observation, this periodic error is only > 0.1 pixel when either x or y translation is in the range: $E_{\text{range}} = [0.45, 0.55] + \text{integer offset}$ (see Figure 4.5). The following probability approach can be taken. Thinking of these probabilities as areas in the Cartesian plane, and seeing that it is equivalent to $[0.45, 0.55]$ plus

⁶One may think this is caused by the linear shifts method (Appendix B.2), this is however not the case since these shifts are only increments of 0.2 pixels, so the linear shift method is not performed here.

any integer, Figure 4.13 makes it more intuitive:

$$\begin{aligned}
 P(\text{SPITE fail}) &= \frac{\text{Occurences of failure}}{\text{Total occurences}} \\
 &= \frac{A_{\text{Fail}}}{A_{\text{Total}}} \\
 &= \frac{2 \times (1 \times 0.1) - (0.1 \times 0.1)}{(1 \times 1)} \\
 &= 0.76/4 = 0.19
 \end{aligned} \tag{4.3}$$

in layman terms, roughly every fifth SPITE will yield an error > 0.1 pixel. We define this translation dependant error as the *periodic* error.

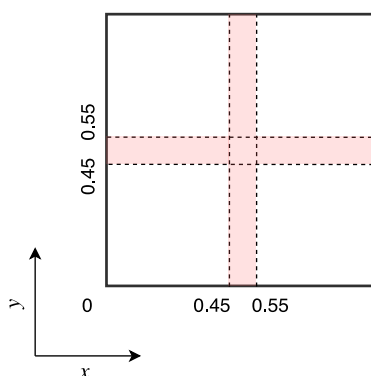


Figure 4.13: The red area represents there where SPITE consistently fails, referred to as periodic error.

Discussion: stability Another observation may be made: the algorithm is stable across the dimensions which it is defined by: shifts lying within the boundary $[-\frac{N}{2} + 2, \frac{N}{2} - 1]$ for both x and y shifts. This is with the exception of images B and C which display some undesired behaviour in the regions: Image B: $[-9.6, -7.6]$ and Image C: $[7.4, 14.8]$.

The underlying window and template when the undesired behaviour occurs is shown in Figure 4.14, surprisingly this shows that there *are* significant features present. Intuitively, low accuracy is the result of an image with no significant features. Since there are significant features present, which part of the method is under-performing? Either the underlying *correlation* or *sub-pixel* technique? Since it is not possible to decouple these two parts of the algorithm, both NCC and phase correlation with various surface fitting techniques are implemented for the scenario of images B and C, at the specific translations of failure. The same surface fit techniques as in Section 4.3 are used. The performance thereof is shown in Figure 4.15.

Clearly the undesirable behaviour is due to the surface fitting technique of Poly 3×3 for NCC. Both Poly 5×5 and Poly 7×7 show increased accuracy with

NCC. Note how phase correlation coupled with Poly 3×3 also has a smaller error than Poly 3×3 with NCC. Figure 4.16 shows the points of $c_{norm}(x, y)$ which are used to model for the Poly 3×3 , Poly 5×5 and Poly 7×7 surface fit techniques for Image C. Clearly the 5×5 and 7×7 better approximate a parabolic surface than the 3×3 , the reason why better accuracy is expected with Poly 5×5 and Poly 7×7 . We define this error as the *fitting* error.

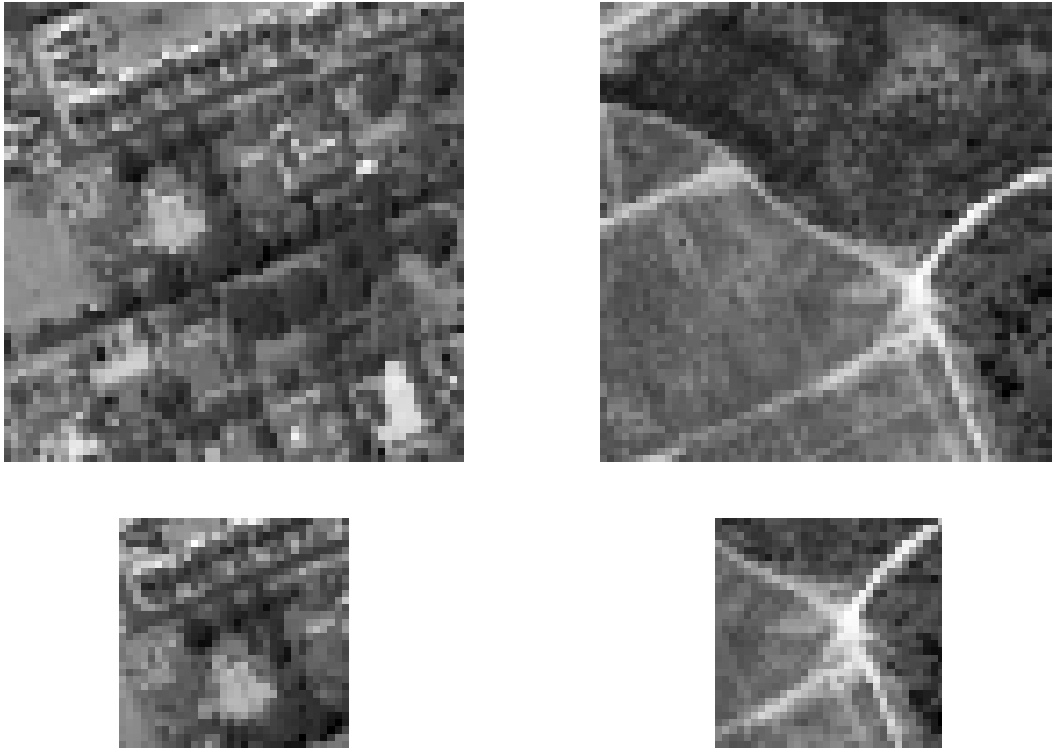


Figure 4.14: Image B's window and template (left top and bottom) and Image C's window and template (right top and bottom) with translations 9.4 and -9.4 respectively. Subsections of [1].

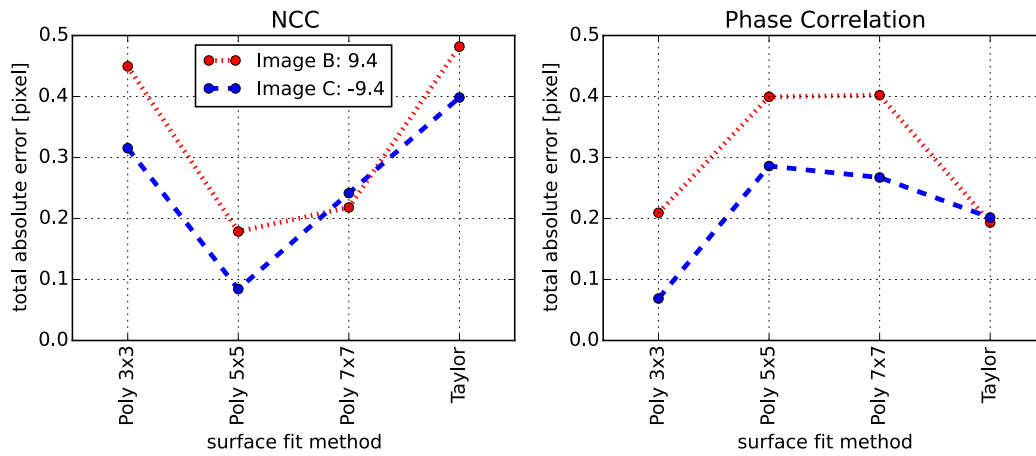


Figure 4.15: Worst case error from Figure 4.14 Image C and B with their respective translations -9.4 and 9.4., using different underlying correlation (NCC left, phase right) and sub-pixel technique.

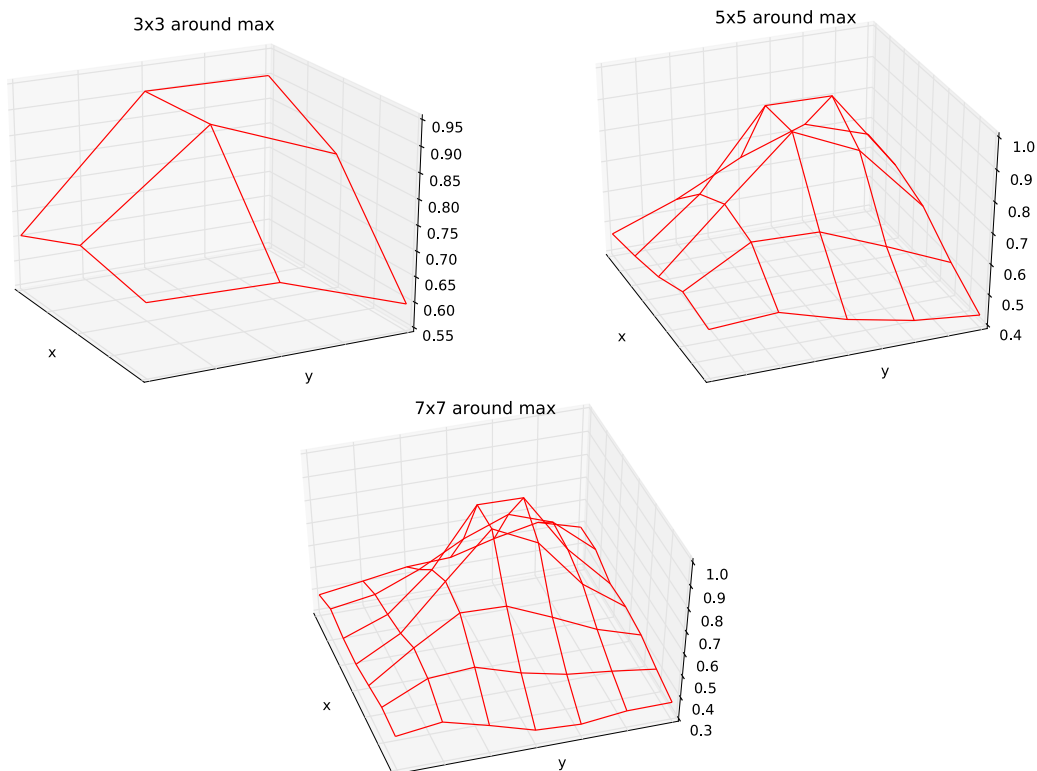


Figure 4.16: The surface surrounding the maximum of Image C's $c_{norm}(x, y)$. Evidentially the 5×5 and 7×7 provide better data to approximate the underlying parabolic surface.

Conclusion: SPITE is shown to display expected behaviour within its defined working range. It generally provides estimations within the required accuracy. Under two scenarios however, this accuracy is not met, the resulting errors are referred to as *periodic* and *fitting* errors. Periodic errors occur when the sub-pixel shift is in the range $[0.45, 0.55] + \text{integer pixel}$. This occurs roughly every fifth SPITE, but does not yield a too large error. The second scenario, the *fitting* error, provides a larger error, and is trickier to predict. A larger simulation is required to determine the influence of the fitting error, and is discussed in Section 4.6. Lastly, the error is shown to be unbiased, this implies that when stacking images – as in Section 4.2 – with these shift approximations, there will be no bias in any given direction. Thus, significantly less, and not biased inter-frame mis-registration is expected, rather than the worst case which was modelled in Section 4.2.

4.6 SPITE mis-registration investigation: Monte Carlo simulation

The periodic and fitting error were introduced in Section 4.5. It is favourable to better quantify the probabilities of these errors from a large simulation.

Aim: Quantify the behaviour of periodic and fit errors. This will provide a better stability metric.

Method: Induce a set of random shifts selected from the uniform distribution, form the operating range $[-14, 15]$ in both x and y directions. Induce a large number - 1000 - of these random shifts across all four test-images. Quantify the error as $E_{\text{periodic}} : 0.1 < \text{Error} \leq 0.15$ and $E_{\text{fit}} : 0.15 < \text{Error}$

Results: Table 4.1 contains the average pixel error along with the percentage of various errors' occurrences. Figure 4.17 shows the distribution trends in a histogram format.

The periodic error occurs at a maximum of 17%, close to the assumed rate mentioned in Section 4.5, however it is generally less than half of this value. Secondly, E_{fit} constitutes less than half of E_{fail} , implying that this is a seldom occurrence. Lastly the largest error for image A is 1.59 pixel. This occurs at translation $(6.752, -4.593)$. Appendices B.5, B.6 and B.7 show how the various sub-pixel techniques would reduce this error (as in Figure 4.15), another example of fit error.

Table 4.1: Absolute error spread for the Monte Carlo simulation.

| Error spread | Images | | | |
|---|--------|-------|-------|-------|
| | A | B | C | D |
| Mean [pixel] | 0.083 | 0.074 | 0.061 | 0.073 |
| $E_{\text{fail}} : E > 0.1[\%]$ | 16.1 | 12.2 | 11.5 | 17.4 |
| $E_{\text{periodic}} : 0.1 < E \leq 0.15[\%]$ | 8 | 8.8 | 6.6 | 17 |
| $E_{\text{fit}} : E > 0.15[\%]$ | 8.1 | 3.4 | 4.9 | 0.4 |
| E_{largest} [pixel] | 1.59 | 0.45 | 0.42 | 0.18 |

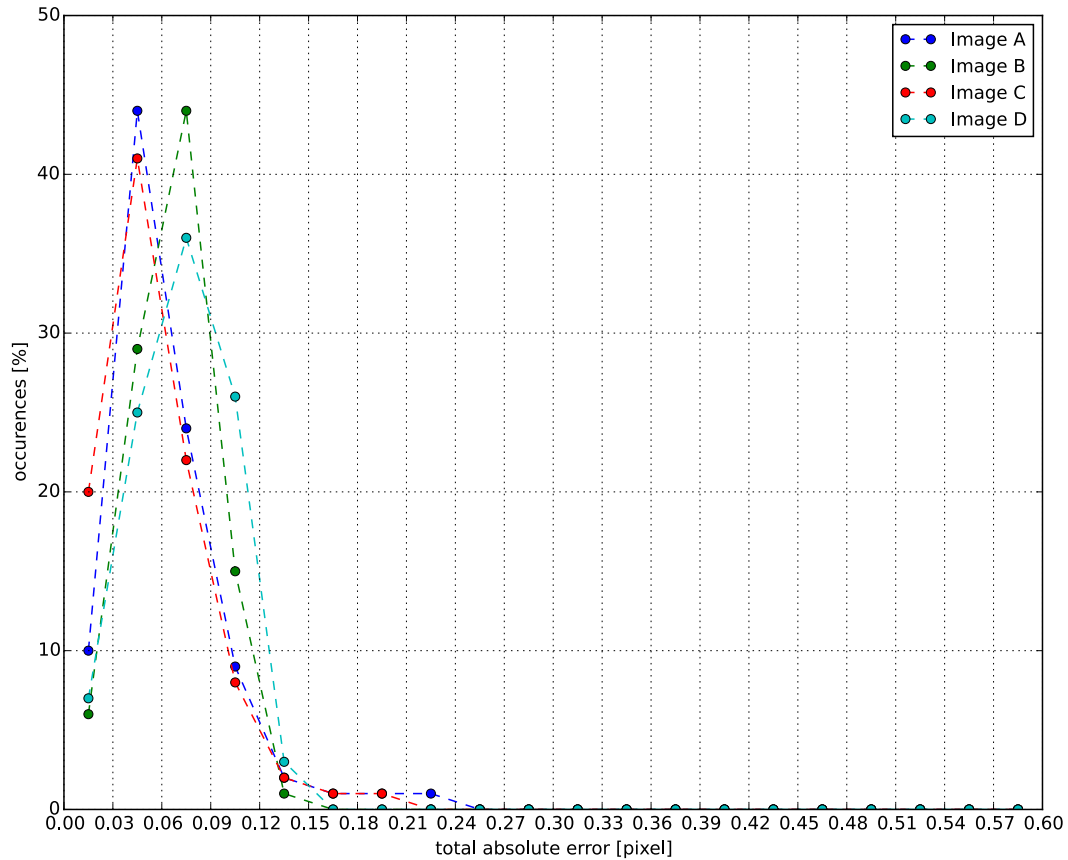


Figure 4.17: Histogram of Monte Carlo simulation, bins are aligned to vertical grid lines.

Conclusion: SPITE (in the worst case) fails the 0.1 pixel specification 17.4% of the time, which confirms that the ‘1 in 5’ SPITES will fail prediction (Section 4.5) is a conservative estimate, since this accounts for E_{periodic} only, whereas in reality the *total* error is less frequent than ‘1 in 5’.

Only 8.1% of translations breach the 0.15 pixel error mark (the fit error).

When considering the worst case⁷ (Image A) v.s. the second worst case (Image B), the values are significantly far apart. Clearly the surface fit error is scene content specific. SPITE using NCC with Poly 3×3 , where $N = 32$, $M = 64$ seems reasonably robust and well suited for the intended application, especially if we are concerned with mean error, as opposed to number of failures.

4.7 SPITE accuracy in noisy conditions

In practice, any image captured by a satellite will have a certain amount of noise superimposed on it. It is thus imperative that the effect of noise on the performance of the algorithm is tested.

Aim: Determine the accuracy to which the algorithm works in the presence of photon noise. It is of interest to determine at which noise levels the algorithm yields unsatisfactory results.

Method: 100 translations were induced and estimated with NCC Poly 3x3 where $M = 64$, $N = 32$. These were calculated over various SNR's for the four images A, B, C and D. These translations are random (not bound to increments of 0.2) but re-used between the various SNR values to have a consistent test. Note these same translations are used for subsequent tests to maintain consistency across the board. These translations fall within the range $[-14, 15]$. The noisy images were generated in the same manner as in 4.2. Figures B.8 to B.11 in the Appendix, show all four test images, with their equivalent SNR's.

Results: Table 4.2 contains the average pixel error, and percentage of those errors over the required specification of 0.1 pixel error and Figure 4.18 shows the trends in histogram form as the noise increases. Note how for SNR's ≥ 20 for all four images, the average error is below 0.1 pixel error. Note how Images A, B and C are quite robust to low SNR. Image D's accuracy gets significantly worse as SNR decreases, given its low contrast features as predicted in [32].

⁷Worst case is defined as the maximum mean error.

Table 4.2: Average absolute SPITE error in pixels and the percentage of these errors that do not meet the desired specification for various amounts of noise in the image.

| SNR | Images | | | | | | | |
|-----|--------------|---------|--------------|---------|--------------|---------|--------------|---------|
| | A | | B | | C | | D | |
| | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 |
| 2.5 | 0.278 | 74 | 0.645 | 84 | 0.822 | 94 | 8.5 | 100 |
| 5 | 0.118 | 32 | 0.107 | 48 | 0.193 | 69 | 0.651 | 86 |
| 10 | 0.087 | 19 | 0.084 | 29 | 0.078 | 27 | 0.137 | 63 |
| 20 | 0.083 | 15 | 0.077 | 18 | 0.065 | 14 | 0.084 | 31 |
| 50 | 0.084 | 18 | 0.076 | 12 | 0.064 | 12 | 0.074 | 15 |
| GT | 0.083 | 18 | 0.076 | 12 | 0.059 | 9 | 0.071 | 14 |

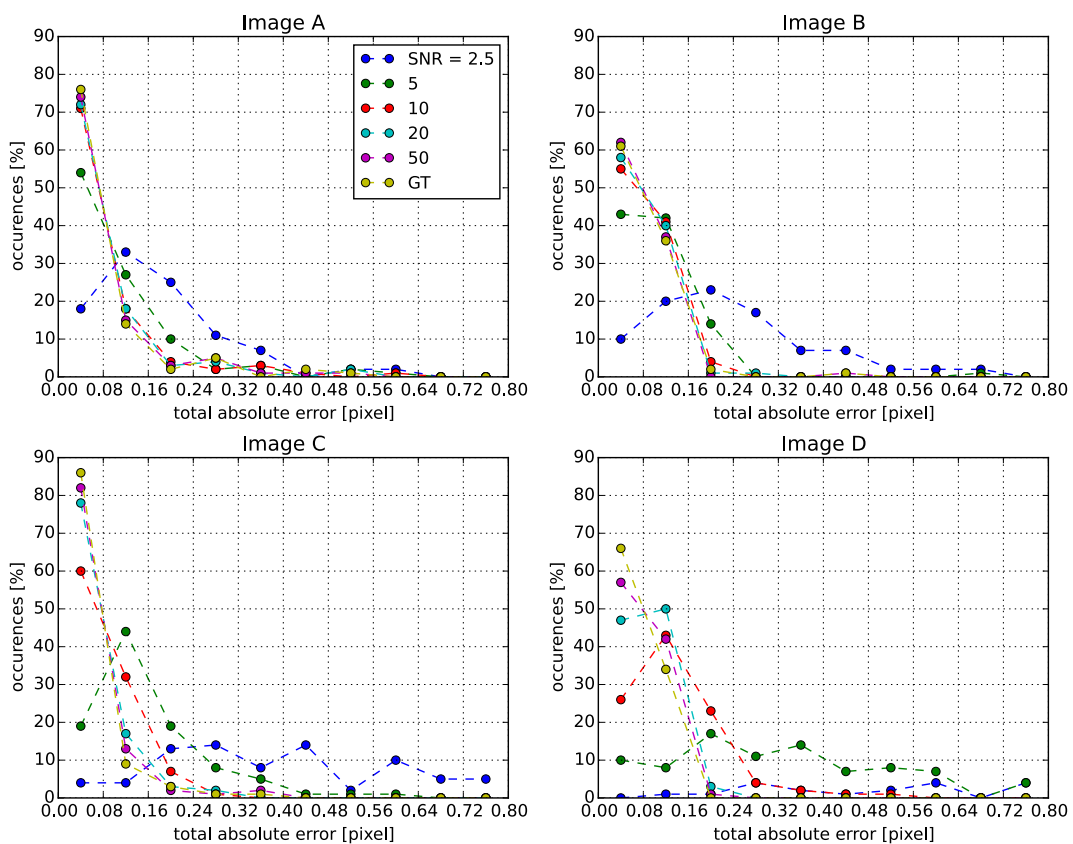


Figure 4.18: Histogram of the various images with the varying SNR's: 2.5, 5, 10, 20, 50 and ground truth. Bins are aligned to vertical grid lines.

Conclusion: Table 4.2 proves that NCC Poly 3x3 *generally* meets specification for all representative images where $\text{SNR} \geq 20$.

4.8 SPITE accuracy in rotations

In practice, a rotation may be present between subsequent images. The extent to which SPITE behaves favourably in such a scenario is an important metric when benchmarking SPITE for practical applications.

Aim: Determine the accuracy to which the algorithm works in the presence of rotations. Since NCC is sensitive to rotations, determine to what degree it functions well enough.

Method: SPITE was performed over various angles for the four images A, B, C and D. Window and template sizes were kept constant at $M = 64$, $N = 32$. Shifts were applied randomly in the range $[-14, 15]$ pixels, after which the rotation is applied to the shifted image around its new centre as illustrated in Figure 4.19. The random shifts were re-used for all the various rotation angles, to keep the test consistent.

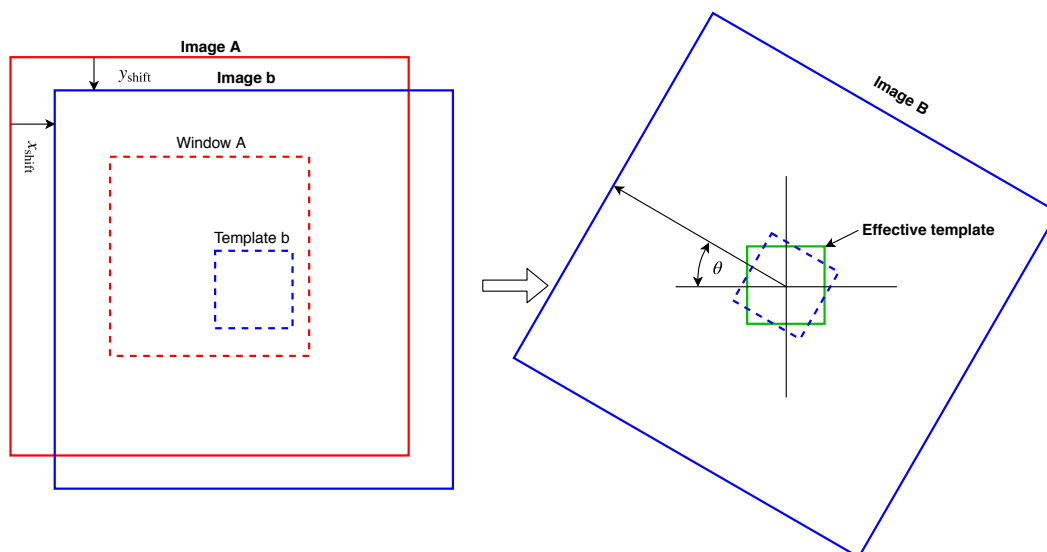


Figure 4.19: This figure shows the effective chosen template from Image B, the shifted and rotated version of image A.

Due to the nature of this problem, a large emphasis is placed on the generation of the rotated image, Image B from the translated image, Image b. This is largely dependant on the *type* of interpolation used in the rotation method. As a proper study of interpolation methods is out of scope for this project, only one method is used: third order spline interpolation. This is also the only interpolation technique currently available with `scipy.ndimage.interpolation.rotate`.

⁸

⁸In the process `scipy.ndimage.interpolation.rotate` was found to be superior in functionality to `scipy.misc.imrotate` when rotations around small angles were required.

Results: Table 4.3 contains the average pixel error, and percentage of those errors over the required specification of 0.1 pixel error and Figure 4.20 shows the trends in histogram format as the rotations increase.

Consider Figure 4.20, the first observation is that: `scipy`'s `ndimage.rotation` works well with the spline interpolation since the 0.125° rotation yields very similar (yet not the same) results to the 0° rotation. This implies that a rotation was undergone for this small angle. With this confidence that the simulation undergoes the rotation for small angles, further observations may be made. Rotations in the range $[0^\circ, 0.5^\circ]$ yield mean errors < 0.084 pixels for all four images, meeting specification. When rotated by 1° , accuracy gets significantly worse for images A, B and C, however the worst case mean pixel error is only 0.103 pixels. Therefore, up till 1° rotations, the full data set is stable, generally yielding results under 0.1 mean pixel error.

Table 4.3: Average absolute SPITE error in pixels and the percentage of these errors that do not meet the desired specification (> 0.1), where various rotations are induced on the template.

| Rot. θ [deg.] | Images | | | | | | | |
|-------------------------|-----------------|--------------|-----------------|--------------|-----------------|--------------|-----------------|--------------|
| | A | | B | | C | | D | |
| | Avg. [pixel] | % $>$ 0.1 | Avg. [pixel] | % $>$ 0.1 | Avg. [pixel] | % $>$ 0.1 | Avg. [pixel] | % $>$ 0.1 |
| 0.0 | 0.083 | 18 | 0.076 | 12 | 0.059 | 9 | 0.071 | 14 |
| 0.125 | 0.082 | 17 | 0.077 | 13 | 0.059 | 10 | 0.071 | 13 |
| 0.25 | 0.082 | 18 | 0.077 | 15 | 0.061 | 12 | 0.071 | 13 |
| 0.5 | 0.084 | 21 | 0.078 | 16 | 0.067 | 15 | 0.071 | 12 |
| 1 | 0.103 | 30 | 0.087 | 33 | 0.076 | 24 | 0.07 | 13 |

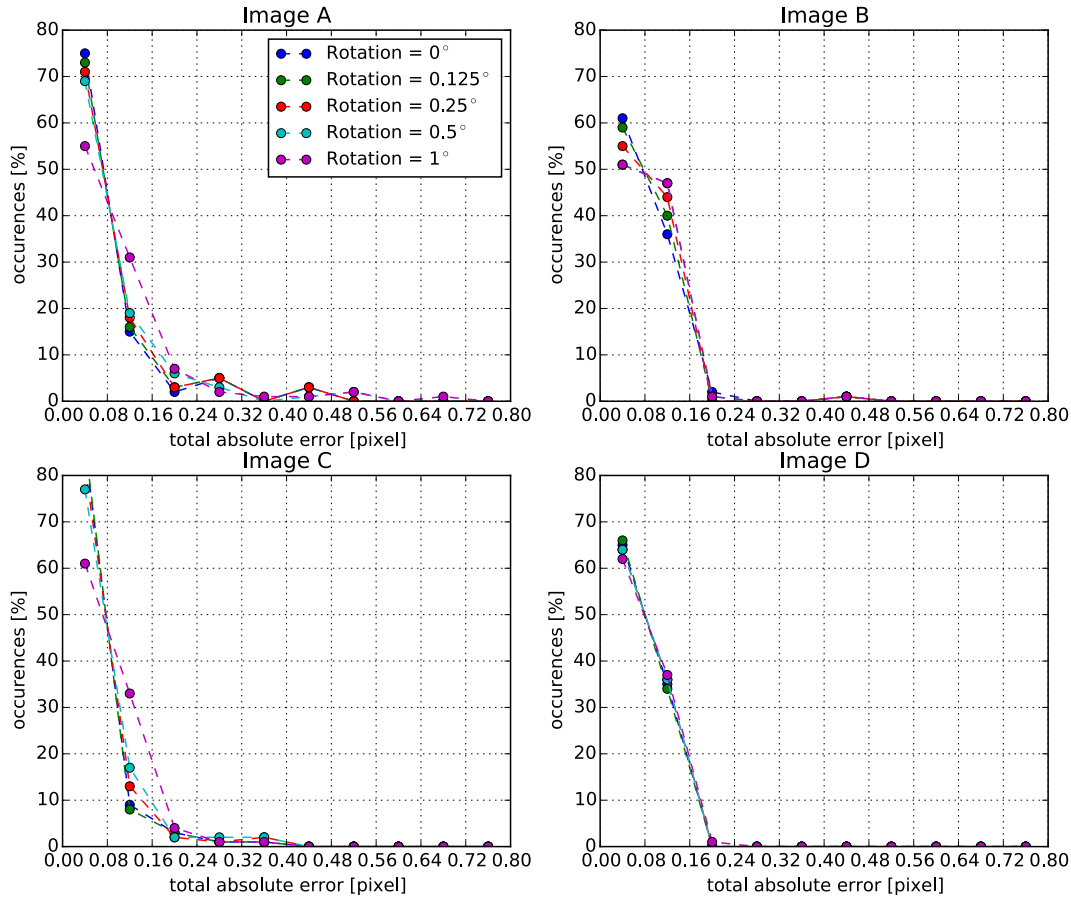


Figure 4.20: Histogram of NCC Poly 3×3 accuracy in presence of various rotations. Bins are aligned to vertical grid lines.

Conclusion: NCC Poly 3×3 SPITE with dimensions $M = 64, N = 32$ is reliable for 3 of the 4 input images up till rotations as large as 1° . Therefore, the largest expected rotation has a noticeable influence on the accuracy, but not to magnitudes significantly over specification – this proves a relatively stable system under small rotations across a range of input images.

4.9 SPITE accuracy in noise and rotations

Noise and rotations may be present in subsequent images simultaneously, it is important to determine the robustness of SPITE under such conditions.

Aim: Determine the effect of noise and rotations on SPITE’s accuracy.

Method: NCC Poly 3×3 was shown to be robust for $\text{SNR} \geq 20$, now we need to determine the impact of various rotations thereon. 100 random shifts in the range $[-14, 14]$ pixels were applied. The rotation is induced, after which the

noise is applied. The noise and rotation is generated in the same fashion as in the previous Sections 4.2 and 4.8. SPITE was then performed, over the various angles for the four images A, B, C and D. Window and template sizes were kept constant at $M = 64$, $N = 32$.

Results: Table 4.4 contains the average pixel error, and percentage of those errors over the required specification of 0.1 pixel error. Figure 4.21 shows the trends in histogram form as the rotations increase for SNR of 20. The pairing of the results show that at this noise level, with the various rotations induced, the algorithm responds favourably. For rotations 0.5° and lower, the total average error is ≤ 0.1 pixel.

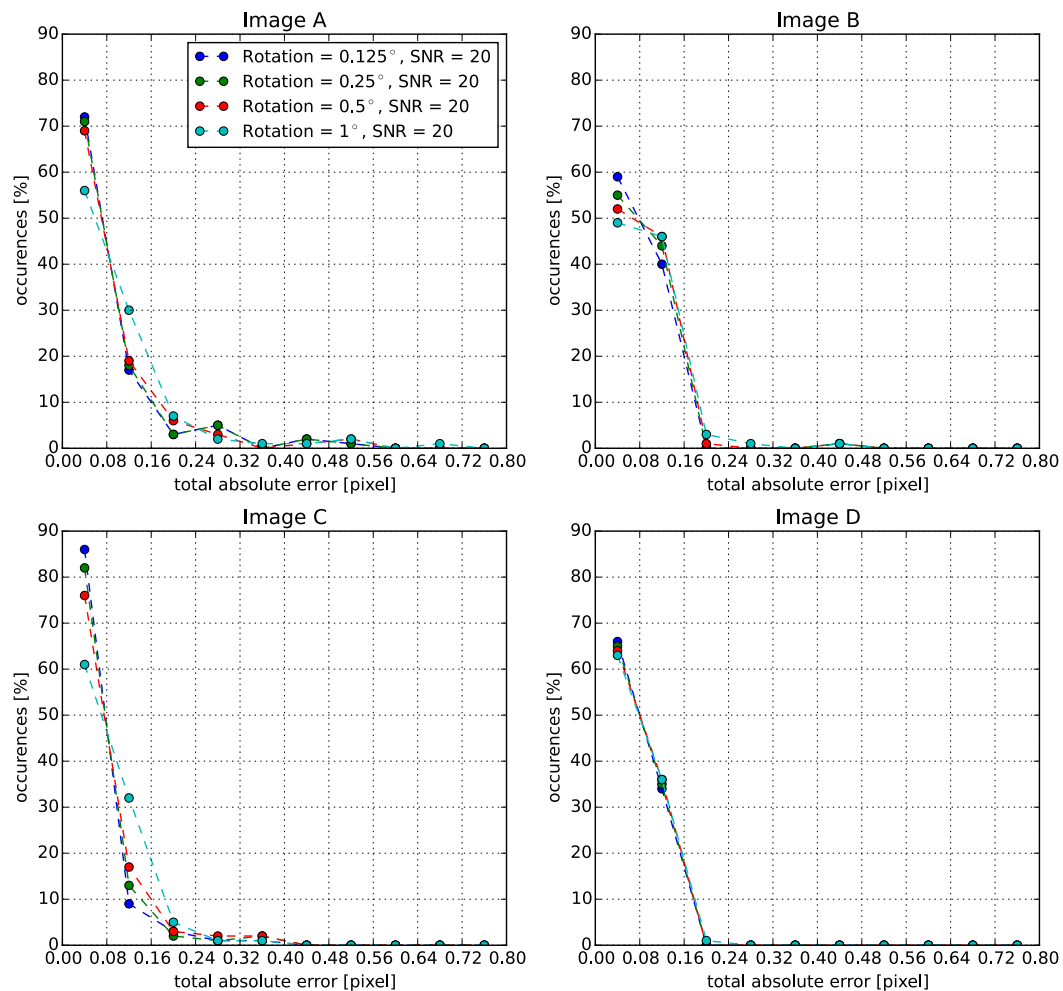


Figure 4.21: Histogram of NCC Poly 3x3 accuracy in presence of both noise and rotations. Bins are aligned to vertical grid lines.

Conclusion: Figure 4.21 and Table 4.4 show that NCC Poly 3x3 gives mean error < 0.1 pixels up to 0.5° rotation for a SNR = 20. The implication thereof:

Table 4.4: Average absolute SPITE error in pixels and the percentage of these errors that do not meet the desired specification, where $\text{SNR} = 20$ paired with various rotations.

| Images: SNR = 20 | | | | | | | | |
|-------------------------|-----------------|------------|-----------------|------------|-----------------|------------|-----------------|------------|
| | A | | B | | C | | D | |
| Rot. θ [deg.] | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 | Avg. [pixel] | % > 0.1 |
| 0.125 | 0.082 | 17 | 0.076 | 13 | 0.059 | 11 | 0.071 | 14 |
| 0.25 | 0.082 | 18 | 0.077 | 15 | 0.061 | 12 | 0.071 | 12 |
| 0.5 | 0.084 | 21 | 0.078 | 16 | 0.067 | 15 | 0.071 | 12 |
| 1 | 0.103 | 30 | 0.087 | 33 | 0.078 | 25 | 0.07 | 11 |

NCC Poly 3x3 with $M = 64$ and $N = 32$ is stable across a wide range of typical operating scenario.

4.10 Discussion

In this Chapter, NCC with Poly 3x3 has been shown to be the superior SPITE correlation and sub-pixel combination. Appropriate sizes for the window and template dimensions are determined to be $M = 64$, $N = 32$. This enables a lean implementation, with estimates in the translation range of $[-14, 15]$ pixels in both x and y directions.

An error of magnitude $[0.1, 0.15]$ is expected when either x or y shifts have a sub-pixel component in range $[0.45, 0.55]$, this is referred to as the periodic error ($E_{periodic}$). Furthermore, a less predictable error – E_{fit} with magnitude ≥ 0.15 pixel – is driven by scene content. A Monte Carlo simulation was conducted to determine the probability of both of these errors occurring, 8.8% and 8.1% respectively for the worst case. The error behaviour was also observed to be unbiased towards either direction.

A smaller simulation determined the impact of noise, rotations and the combination of noise and rotations on SPITE accuracy. In the presence of noise up to $\text{SNR} = 20$, 69% of SPITEs made specification with a mean error < 0.084 pixel (worst case). In the presence of rotations up to $\theta = 1^\circ$, 70% of SPITEs made specification with a mean error < 0.103 pixel (worst case). In the presence of both noise and rotations where $\text{SNR} = 20$ and $\theta = 1^\circ$, 70% of SPITEs made specification mean error < 0.103 pixel (worst case). Clearly the addition of noise had no significant impact on the *worst* rotation data-set (image A).

Data-sets which behave well under noisy conditions will not necessarily behave well in presence of rotations and vice versa – see results from images A and D for noise only and rotations only simulations to be convinced. E.g. image D, which is the most susceptible to noise, is the most stable under rotations.

On the contrary, image A which is least susceptible to noise is most susceptible to rotations. This shows why there is a need to test across a large range of input data, with varying detail.

These simulation results show that the SPITE algorithm will behave favourably under the expected circumstances, as results generally meet the criteria for the given noise, rotations and combinations thereof.

Lastly, if we judge by *average* absolute error, Figure 4.2 shows that 3/4 of the images would yield a satisfactory *increase* in SNR when implementing SPITE for an image stacking end-point application, where the initial images' SNR is as low as 10. This is since Figure 4.2's initial SNR of 10 at 0.1 mis-registration still yields a gain in SNR post image stacking, since a 10 SNR image's percentage gain is above 30.15, the criteria introduced in Figure 4.2 for a successful image stack⁹. This 0.1 pixel mis-registration is guaranteed for 3/4 images at SNR = 10 in Table 4.2.¹⁰ Therefore, successful image stacking is a viable end-point application with the current SPITE algorithm.

Now that the specific candidate algorithm, its dimension and ability is known, and we have been convinced that it will generally meet the criteria under various re-enacted mission scenarios, a custom hardware *implementation* is sought.

⁹Image stacking can further be tweaked by varying the number of input images. A full study into the dependencies of SPITE accuracy, SNR levels, number of input images and SNR gain is out of scope. It does however highlight the manner of investigating this co-dependency.

¹⁰This was only tested in presence of noise, not rotations.

Chapter 5

Hardware Investigation

In Section 2.3, the notion of various possible platform classes for the space environment were introduced. Two platforms which show initial promise – the MicroZed 7020 (CPU + FPGA SoC) and the Nvidia Jetson TK1 (CPU + GPU SoC, a.k.a GPGPU SoC) – are now discussed in more detail. A full architecture and sub-system analysis is too verbose, only aspects significant to the design are discussed. The investigation focusses on three aspects: a brief high-level architecture discussion (focussing on power vs. performance malleability), a 2D-FFT benchmark which will better quantify such performance, and available radiation mitigation options. The aim: to make an informed decision on which is the better hardware platform for *this specific* context.

5.1 MicroZed 7020

5.1.1 Architecture

The MicroZed 7020 (development-board) consists of a SoC, the Zynq7000, along with various memories and interfaces. The Zynq7000 contains two ARM Cortex-A9 CPUs (50 - 667MHz clock), FPGA fabric (0.1 - 250 MHz clock), hardcoded logic blocks, along with a host of peripherals, memory interfaces and clock generating logic (see Figure 5.1). Of importance to this project are the number of BRAMS ($140 \times 36\text{Kbit}$) and hardcoded math blocks ($220 \times \text{DSP48s}$). SPITE will be broken down into a data and control path, where the data will be stored in BRAMS and sent through DSP48s for mathematical operations. Thus the number of BRAMs and DSP48s indicate the types of algorithm expansions that are possible. Each Cortex-A9 CPU has a single-instruction-multiple-data (SIMD) NEON floating point co-processor, useful for implementing fast math on the CPU.

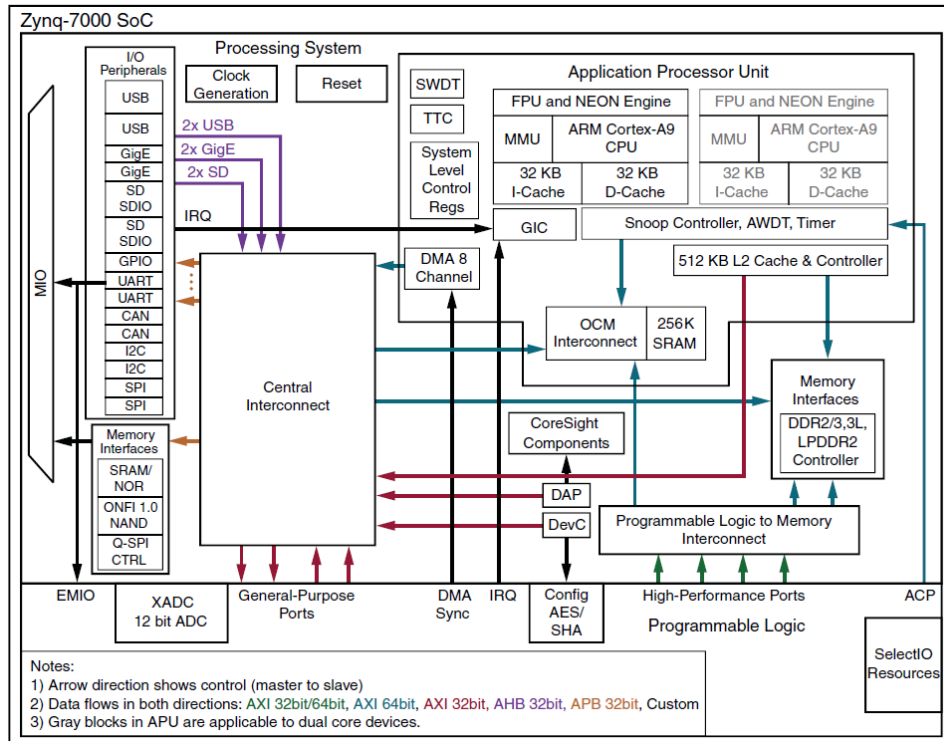


Figure 5.1: Zynq 7000 SoC architecture [2].

The logic in the fabric is driven by a certain clock. Besides sophisticated clock throttling techniques which may be used, fabric clocks may also have their frequency altered (by writing an appropriate value to the divisor register), thereby altering the dynamic power. Figure 5.2 shows the various clocking options. Dynamic Voltage Frequency Scaling (DVFS) – discussed in Appendix A.2 – reduces power by reducing the frequency when lower throughput is required.

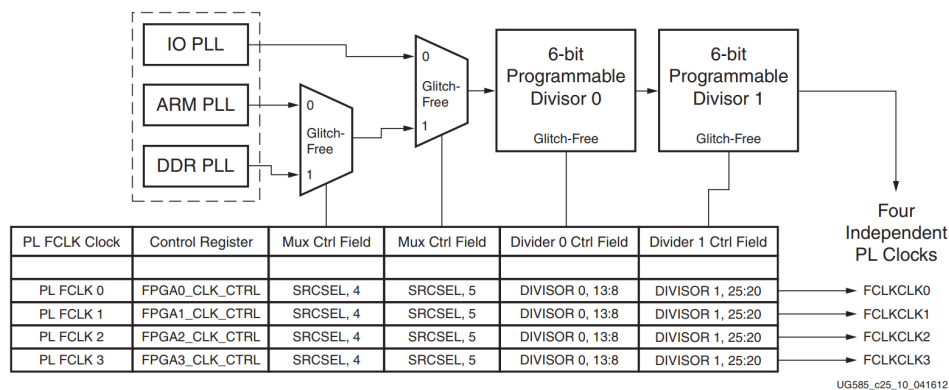


Figure 5.2: The clocking logic available to the fabric (referred to as Programmable Logic (PL) in Xilinx’s manuals), note how four independent clocks are available to the fabric, and on each the frequency may be scaled [2].

5.1.2 Benchmark: 2D-FFT

Section 3.2 shows that the 2D-FFT forms a significant part of the SPITE calculation. Since the 2D-FFT is a compute-intensive calculation and is inherently parallel, it is a good candidate to benchmark the performance of both MicroZed 7020 and Jetson TK1.

Aim: Determine the performance of the MicroZed 7020, with regards to calculation speed over various sized 2D-FFTs.

Method: The following *theoretical* 2D-FFT implementation is investigated: an un-pipelined implementation where all the rows’, then all the columns’ 1D-FFTs are calculated sequentially (Section 3.2.1). This requires only a single 1D-FFT core, and makes little requirements on data handling (un-pipelined approach). Thus this is an easy to implement, robust, sub-optimal, theoretical, first-order design.

Results are from Xilinx’s Fast Fourier Transform (9.0) core, selecting the following IP-core options: number of channels: 1, target frequency: 100MHz, architecture: Radix-2 Burst I/O, data format: 32bit float, precision option: 24 phase factor width, output ordering: natural, throttling scheme: real time. The rest of the options are left as default, only the transform length is changed to determine requirements for various sized 2D-FFTs.

Runtime is calculated using the latency provided by the cores configuration window, using the same methodology as in Section 3.2, where $2M$ 1D-FFTs are calculated:

$$\text{Runtime}_{M \times M \text{ 2D-FFT}} = 2M \times \text{latency}_{1\text{D-FFT}} \quad (5.1)$$

Scratch pad memory required to store results is $8M^2$ bytes, plus the memory required by the 2D-FFT IP-core. This assumes the scratch pad memory uses BRAMs.

Results and Conclusion: Table 5.1 shows that 2D-FFT is only achievable for sizes up to 256×256 , as the number of available BRAMs is the limiting factor. The required 64×64 2D-FFT (size proven in Section 4.4), has a competitive runtime of 0.65ms. Note that various combinations of settings for the FFT-block along with different architectural choices and optimisations will influence this value, it is however a good starting point to determine that: 2D-FFT *is* possible on a MicroZed 7020 in *real-time* for the required size.

Table 5.1: Theoretical runtime of 2D-FFT with various sizes M , where resource usage is reported as percentage of MicroZed 7020 resources.

| Size | t [ms] | BRAM [%] | DSP48[%] |
|------|--------|----------|----------|
| 64 | 0.65 | 8 | 3.63 |
| 128 | 2.5 | 23.6 | 3.63 |
| 256 | 9.95 | 86 | 3.63 |
| 512 | 41 | 336 | 3.63 |

In Section 7.3, the MicroZed 7020’s total algorithm power is measured and discussed. There a energy per calculation for *that* 2D-FFT implementation is deduced from the total algorithm power, for reference sake, it is 0.35mJ/2D-FFT calc.

5.1.3 Radiation mitigation

Xilinx’s BRAMs allow easy implementation of error correction codes – these are briefly discussed as they can increase the implemented design’s reliability in the case of radiation events.

Xilinx’s BRAMs can be configured as a standard block of random access memory, where each data point is accessed at a certain address, or as a standard FIFO buffer. They can be configured such that they accept 64bit wide data, which it assigns 8 parity bits. This Hamming error correction code, enables functionality of Single Error Correction Double Error Detection (SECDED). The operation of the parity bits are abstracted from the user. For simulation purposes however, single and double bit errors may be injected, to test the functionality. Every time data is read from a BRAM, the various statuses are provided: *no error*, *single bit error corrected* and *double bit error*¹. Note that these statuses refer to the specific output data at a specific 36Kbit BRAM Macro. Since multiple macros may be chained together (to create a FIFO/BRAM that is larger than 36Kbit), if a double bit error is detected, the user will not know *in which* BRAM it occurred, since all error signals are OR’d. Furthermore, it

¹An in depth study of Hamming code is out of scope for this project, it is however important for future developers to know that status *no error* can also imply 3+ errors, since this method won’t be able to detect more than two errors.

is suggested to use ‘soft ECC’ option when instantiating FIFO/BRAM which has a data width less than 64 bits, since it uses the interconnect to generate the ECC logic, and is more efficient than the internal logic at such widths.

Therefore, besides the normal TMR which is easily implemented on an FPGA, these *data* corruption mitigation options are available for the designer to use, and its convenience is demonstrated in Section 6.7. Note that the CPU was not discussed, only the fabric.

5.2 Jetson Tegra K1

5.2.1 Architecture

The Nvidia Jetson Tegra K1 is a development-board for the Tegra K1 (TK1) SoC which is utilised in various mobile efficient consumer electronics, such as tablets and chromebooks [54]. It comes preloaded with the Linux4Tegra operating system which has a similar aesthetic to Ubuntu 14.04. The board also has a whole host of peripherals and functions. The architecture is shown in Figure 5.3. This discussion is however limited to architectural choices which influence performance.

The TK1 SoC boasts an impressive ‘4-plus-1’ CPU architecture. Four clustered 32-bit ARM Cortex-A15 cores share a 2MB L2 cache, along with the Low Performance Core (LPC), a single 32-bit ARM Cortex-A15 core which has its own 512KB L2 cache. This allows the operating system to hand-over responsibility between the high-performance four core cluster (HPC), and the single LPC as it deems necessary. All five Cortex-A15 cores come with their own NEON floating point co-processors. A variety of clock frequencies are available for all the cores, the cluster can go between 204 and 2065.5 MHz, whereas the LPC clock ranges between 51 and 1092 MHz. This allows for added performance control, scaling the clock as the work loads varies (DVFS). By default, this is handled by the Kernel.

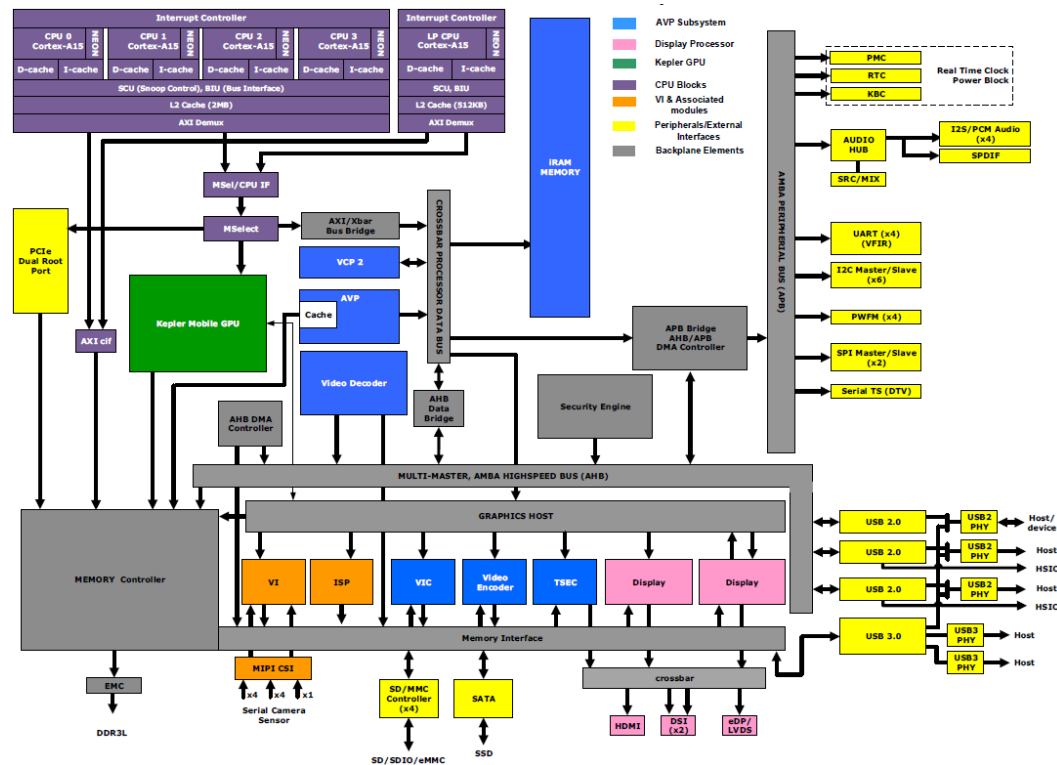


Figure 5.3: Architecture of TK1 SoC [3], showing the various available peripherals and interfaces (yellow), dedicated co-processors (blue), GPU (green) and ARM processor cores (purple).

In addition to the ‘4-plus-1’ CPUs, the TK1 has a GK20A: a 192 core Kepler GPU whose clock can scale between 72 and 852 MHz. In order to make use of this massively parallel resource, we use Nvidia’s Compute Unified Device Architecture (CUDA), a C-application programming interface.²

The HPC, the LPC and the GPU are all part of the same chip, however they are part of distinct power domains shown in Figure 5.4. The ‘CPU’, ‘GPU’ and ‘Core’ power domains are powered by three separate power stages (AS3728), these are controlled by the AS3722 power management integrated chip. Required output current values (for each AS3728) may be interrogated with I2C via their controller: AS3722³. Furthermore, voltages across the various power domains may be read via the ‘/sys/kernel/debug/clock/dvfs’ file. Therefore, since both voltage and current values may be interrogated, instantaneous power may be determined in semi-real time for each domain. This is extremely useful for benchmarking purposes.

²Since the TK1 shares the same physical chip, it is one of the first architectures where data does *not* need to be transferred from CPU to GPU via a bus (typically the PCI-express) as in desktop rigs. The shared memory controller makes this possible.

³The RTC power domain is powered directly by the AS3722.

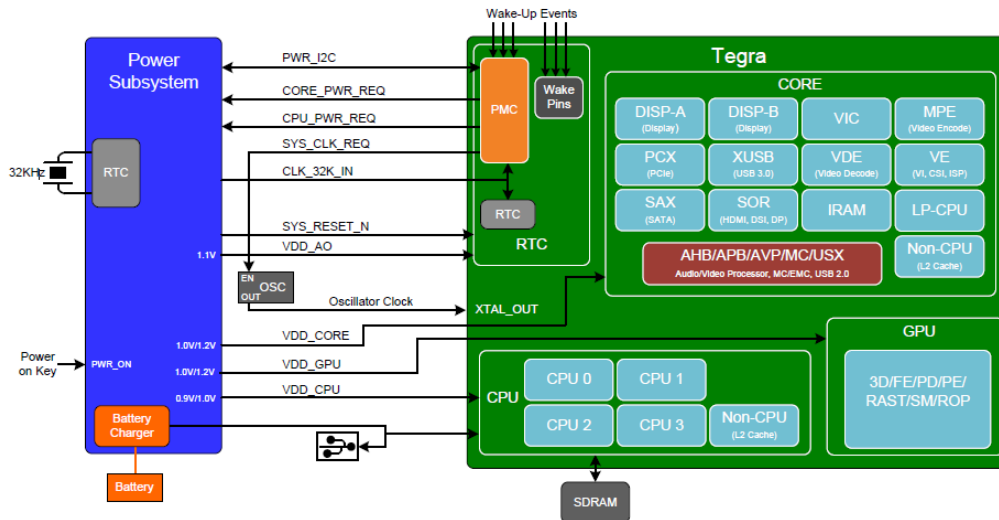


Figure 5.4: Power system of Jetson TK1 [3]. Note the various power domains: Core, CPU, GPU and RTC.

5.2.2 Benchmark: 2D-FFT

Aim: Determine the performance of the Jetson TK1, with regards to calculation speed and power, over various sized 2D-FFTs.

Method: To represent the Tegra K1's versatility, we use the following three configurations to determine a range of operational scenarios, each tested at minimum (LF) and maximum (HF) clock frequencies:

- a C-serial-only application running on the single LPC, at 51 and 1092 MHz, referred to as *C-serial*
- a parallel C-application (achieving 4x parallelism using OpenMP⁴, four cores allow four concurrent threads), referred to as *C-parallel*
- a CUDA implementation (with parallelism), using the 'cufft' library (CUDA Toolkit 6.5), where the LPC does housekeeping at 1092MHz, referred to as *CUDA*

These are the edges of operation, since both minimum (LF) and maximum (HF) frequencies are tested, the reader may extrapolate that in-between values are attainable. The sizes (dimensions) of the 2D-FFT are varied to determine the change in performance over a wide range, for all the variations the length and breadth are equal.

⁴OpenMP is an application programming interface which enables shared memory multiprocessing by use of threads, thus speed-up in the C-code is achieved via OpenMP not NEON.

A Python script manages the testing procedure from a high-level of abstraction. First the appropriate configuration is set-up, forcing a certain operating frequency (thus overriding the default clock scaling), and enabling a certain set of cores. Next, Python spawns two threads (via the ‘threading’ module), one to determine the instantaneous power (Thread 1) and one which runs the application to be benchmarked (Thread 2). Two high-level threads in Python are utilised such that instantaneous power-polling may be executed independently of the application being benchmarked. This reduces the coding complexity of the code which benchmarks the algorithm (C-code) to the Python script. Although it may seem counter-intuitive to run two threads on the single LPC (the C-serial test), the two-threaded functionality was still realised via context switching. Functionality for the C-serial LF does however become problematic due to the overhead of both the operating system and two threads. The Python script overview is shown in Figure 5.5.

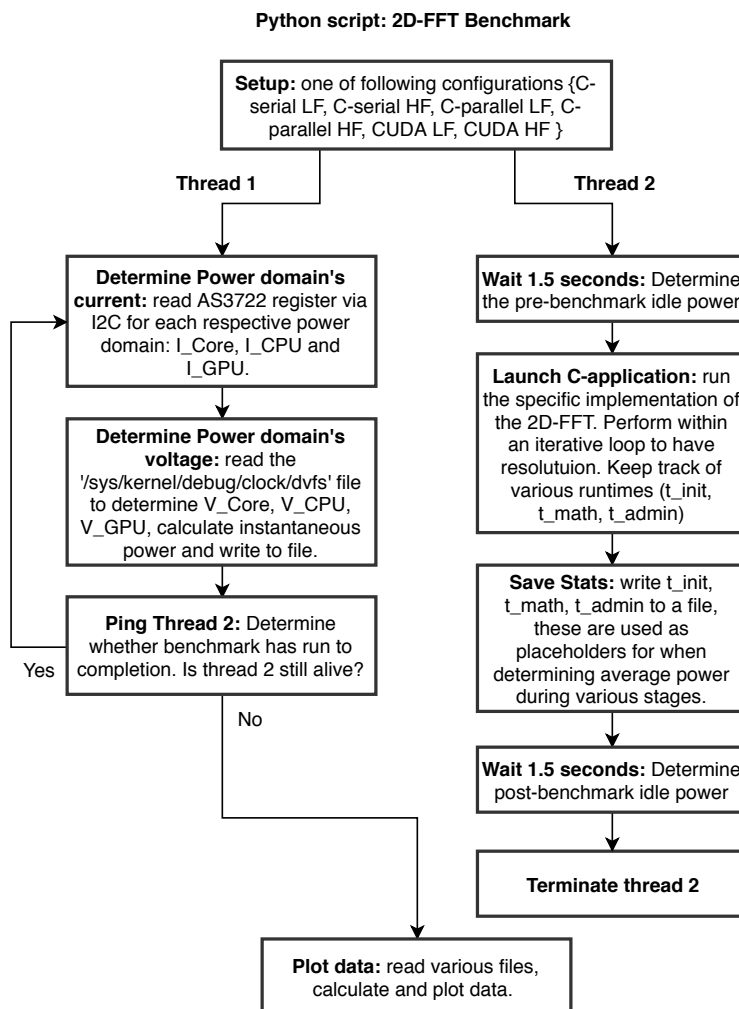


Figure 5.5: Python script for 2D-FFT benchmarking the Jetson TK1.

Note, the instantaneous power measured is the *SoC* power, not the total Jetson board power. Therefore, all power results refer to the SoC's response to the algorithm, and not the boards response. To quantify idle board power without the SoC, we measured the board idle power with a voltmeter, and subtract the SoC idle power, determined in *Thread 2* of Figure 5.5. Thereby we determine the offset power which needs to be added to SoC power results when making *platform* power estimates.

Platform idle power was determined for C-serial HF, C-parallel HF, and CUDA HF – 4.96, 6.06, and 5.17W respectively. Each of these fluctuate no more than of 2.5%. These values differ, most likely due to the change in *RTC* power usage (in Figure 5.4) for the various configurations. Fully characterising the platform power-response is out of the scope of this investigation. For more detail regarding power and timing measurements, see Appendix B.7.

Results: Figure 5.6 shows the typical results of the benchmark, where the instantaneous power domain values are plotted throughout. The vertical black lines are the place-holders of the various portions of the benchmark: *idle* (the first 1.5 seconds where Thread 2 sleeps), *t_init* (time taken to allocate memory), *t_math* (total calculation time) and *t_admin* (time to free resources post-math). Naturally the benchmark of the 2D-FFT operation is run in a for-loop to increase resolution and reduce the effect of overhead. Power is determined by taking the average power of the *math* operation: the operation within the vertical black dotted and dashed lines.

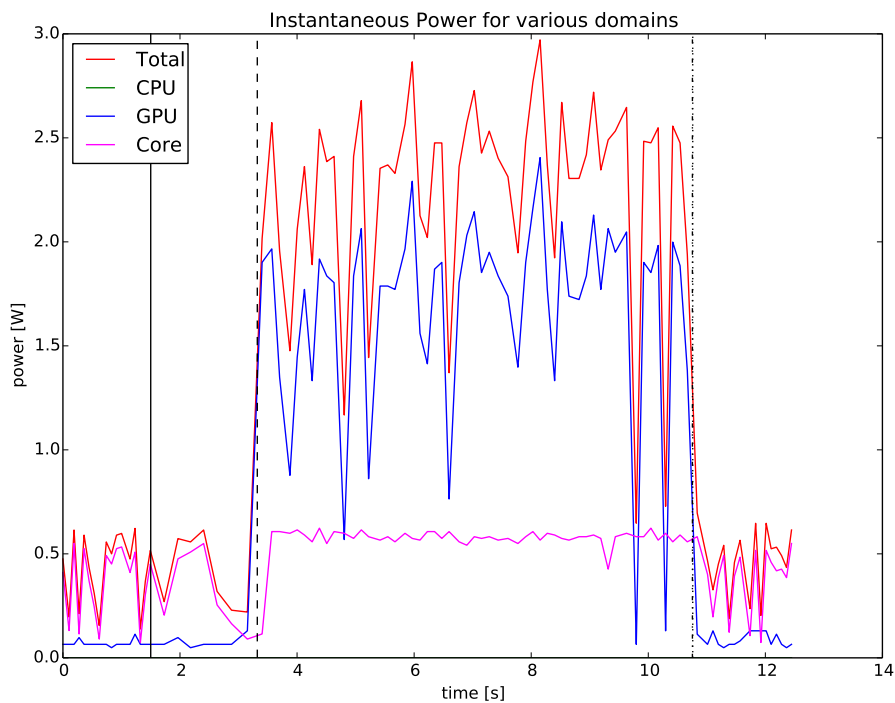


Figure 5.6: Various power domains while running the 2048×2048 2D-FFT CUDA HF implementation. Note how RTC's power domain is not plotted, this is so that the applications power is measured (where the calculation is taking place). Further, note how CPU power is zero due to intelligent power gating. Between solid and dashed vertical lines: t_{init} , between dashed and dotted: t_{math} , t_{admin} is so small that it lines up with dotted vertical.

The current values read via I2C, are provided by an analogue to digital converter, one of the bottlenecks for the time-resolution in Figure 5.6. Furthermore, the overhead of the operating system and C-application make for irregular time intervals of power readings. Thus the power measurements are a good indication of true use, but not perfect.

Figure 5.7 displays time per 2D-FFT calculation and average power. Only two data points are displayed for the C-serial application when running at LF, since the overhead of the operating system coupled with the low operating frequency of the single active CPU, renders inter-run results inconsistent for a reasonable number of iterations. Clearly the parallel CUDA implementation significantly outperforms both C-serial and C-parallel implementations when considering runtime. This is to be expected.

Note the jump in power for the CUDA implementation between length, breadth of 256 and 512. The number of floats required to store a 2D-FFT is $8n^2$ bytes (proven in Section 3.2.2.1) Where $n = 256$ this amounts to 512KB.

Therefore when $n = 256$ it still fits in the convenient 2D-‘texture’ memory, thus when $n > 256$, calculation is moved away into the RAM, increasing the power draw significantly. This is interesting, since the manner in which the memory is allocated in the application is general, however the cufft library calls know the dimensions of the calculation, most likely it makes use of this prior knowledge to allocate the calculation to the texture memory. Lastly, there is a significant reduction in power when going from HF to LF, this is to be expected due to the reduction in clock speed. The power vs. frequency relationship is shown in Eq. 5.2 [55]:

$$P_{\text{switching}} = C_L V_{dd}^2 f_{clk} \quad (5.2)$$

This is the power dissipated by the transistor due to transitioning from one logic state to another. V_{dd} is the transistors voltage, C_L is the load capacitance, and f_{clk} the frequency at which the transistor transitions. The switching power is a linear product of the clock’s frequency, hence power is directly related to the frequency, and exponentially influenced by the voltage. See Appendix A.2 for a more detailed explanation of the CMOS-transistor’s power consumption.

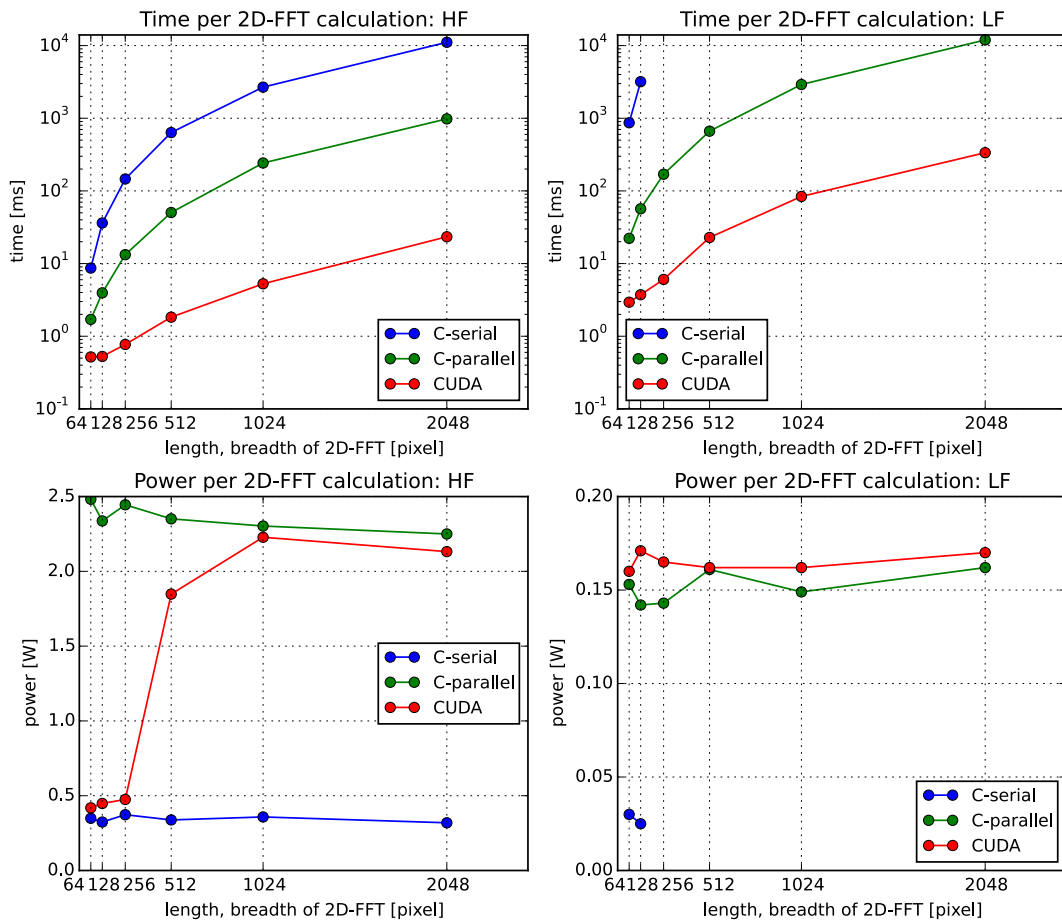


Figure 5.7: 2D-FFT performance results on the Jetson TK1, runtime (top left and right) and average power (bottom left and right).

Figure 5.8 combines all the results from Figure 5.7, comparing the energy per calculation for the various configurations. In all cases (except the C-parallel implementation), it is more energy efficient to perform the calculation at a high-frequency for a short time period, rather than at low-frequency over a longer time period. This could be explained by the static power dissipation:

$$\begin{aligned} P_{\text{transistor}} &= P_{\text{dynamic}} + P_{\text{static}} \\ &= P_{\text{switching}} + V_{dd}I_{\text{leakage}} \end{aligned} \quad (5.3)$$

Since a CMOS-transistor's average power is the summation of both *dynamic power* (Eq. 5.2) and *static power*, and since a longer calculation time (due to lower operating frequency) will result in a larger leakage energy which does *not* scale down with the frequency, this results in a less efficient implementation. From Figure 5.7 it shows that these results are relatively close (yellow-purple and red-turquoise), e.g. the CUDA 2048 HF energy/calc. is a 14.2% increase from LF's energy/calc.

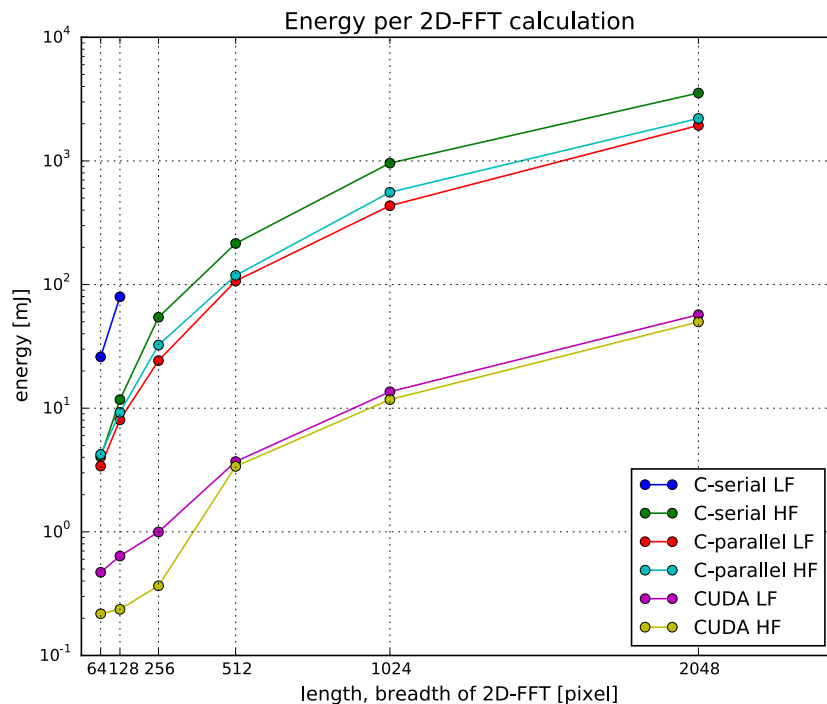


Figure 5.8: 2D-FFT performance results on the Jetson TK1: where Energy per 2D-FFT = average power \times runtime.

Conclusion: The Jetson Tegra K1 is an extremely versatile SoC. From a coding perspective it is extremely simple to scale up the 2D-FFT for C-serial, C-parallel and CUDA. This implies the same is possible with well-written SPITE implementations. Figures 5.7 and 5.8 are a good reference point for designers

who require a certain throughput per energy for various sizes. Needless to say, for the required 64×64 2D-FFT, Table 5.2's summary shows that all configurations (except C-serial LF) may be classified as near real-time⁵, where the single LPC + GPU configuration provides the fastest and most energy efficient solution.

Table 5.2: Power consumption, runtime and energy per 64×64 2D-FFT for the SoC with the three configurations.

| Cfg. | HF | | | LF | | |
|------------|----------|-----------|-----------|----------|-----------|-----------|
| | P [W] | t [ms] | E [mJ] | P [W] | t [ms] | E [mJ] |
| C-serial | 0.35 | 8.67 | 3.03 | 0.03 | 867 | 26 |
| C-parallel | 2.48 | 1.7 | 4.21 | 0.15 | 22.3 | 3.34 |
| CUDA | 0.42 | 0.52 | 0.21 | 0.16 | 2.94 | 0.47 |

5.2.3 Radiation mitigation

George and Milluzii [56] mention common radiation mitigation techniques, namely: Algorithm-Based Fault Tolerance (ABFT), time-based and concurrent Triple Modular Redundancy (TMR). ABFT can only correct for *data* errors, and is thus perceptible to logic errors. Furthermore, data needs to be encoded, the algorithm modified, and output decoded to determine whether or not an error occurred [57].

Both the time-based and concurrent TMR are susceptible to scheduler and core errors because Nvidia's scheduler does not guarantee determinism. In the case of TMR, all three tasks could be scheduled to run on the same set of cores. This may happen in the instance where the rest of the GPU cores are busy. Thus if a certain core is malfunctioning, the same behaviour would be visible in all three TMR answers, nullifying the characteristic usefulness of TMR.

Nvidia GPUs use *streams* to calculate concurrent data. A kernel (a group of threads) can be assigned to a stream. If multiple streams are instantiated, and the various kernels assigned to various streams, then kernels (which have the same operation) will be able to run concurrently.

George and Milluzzi [56] then build on a the concept of Persistent Threads (PTs). PTs forces the kernel's threads to perform the same task, either until the task's completion or reset. This means the kernel 'hogs' the stream, which means it is allocated certain hardware. This forces a certain calculation to be calculated on a certain section of the GPU, as it has removed control of

⁵The term 'real-time' is context dependant. One such application which is investigated in some detail - image stacking - could get data in as fast as video frame-rates. Depending on the overall system and requirement of the end application, video frame rate would be a ballpark value for real-time.

the scheduler. By means of streams and PTs, concurrent TMR is achieved on the hardware platform, since there is a guarantee that redundant kernels will execute on different hardware blocks.

This was implemented on a Nvidia TX1 development-board by George and Milluzzi, in their findings they mention there is scope for such an implementation on the Jetson TK1. This was however not investigated further. It is highly recommended that such (or a similar approach) is implemented if a proper space-redundant solution is required, since the GK20A has no radiation mitigation schemes.⁶ Note, that a proper study will still have to be conducted on the specific CPU configuration, since the current approach assumes the CPU will not malfunction due to radiation events.

5.3 Discussion

Both the MicroZed 7020 and the Jetson TK1 have convenient clock scaling options, facilitating DVFS.

Comparing speeds between the two platforms for the 64×64 2D-FFT, both perform well, 0.52 and 0.65ms for the Jetson TK1 CUDA-HF and MicroZed 7020 implementations respectively. It is however more sustainable to scale up to larger sizes with the Jetson TK1, since writing C-code for a computer-like architecture abstracts complexity when scaling up. An FPGA on the other hand will have to do low-level external memory interfacing for sizes larger than 256×256 , thus reducing the ability to easily scale up.

From a radiation perspective, it is easier to implement TMR on the MicroZed 7020 in the fabric – along with local memory correction – as opposed to the Jetson’s TMR approach.

Although it is possible to implement TMR with the Jetson TK1 using persistent threads, it is riskier since the developer has less intuition of precisely what the underlying hardware is doing, and how it will react in a radiation event. Furthermore, such a solution is not easily portable, since we need to know what the CPU’s radiation ability is, since housekeeping will most likely be implemented by one of the on-chip CPUs. This is not the case for IP generated for an FPGA, as it can easily be ported to other 7th Generation Xilinx devices.

Therefore, since a *specific* sized SPITE is required, and simple radiation tolerance is desirable, SPITE for only the MicroZed 7020 platform will be developed. The Jetson TK1 is however the better candidate for designs where: size may vary, a quick time to market is required and radiation is not an issue, because it is much easier to develop at a higher level (C or CUDA libraries), than at a lower level using VHDL and logic blocks.

⁶This is seen when querying the device properties, it states that no support for ECC is implemented.

Chapter 6

Hardware Implementation

SPITE is developed for the MicroZed 7020. The methodology, architecture, detailed design, error propagation, resource usage, runtime and ability to increase robustness of its specific implementation is discussed in this chapter. It concludes with a discussion on the success of the full hardware implementation.

Terminology: The following terminology is used for the discussion.

1. IP-core: Any FPGA logic provided by the vendor, Xilinx. No proprietary cores were used.
2. Math-unit: a *math* IP-core
3. Math-block: Consists of one or more Math-units, has defined in/outputs, one or more Finite State Machines (FSMs) for control, and its *own* dedicated memory: either BRAM, FIFO or both.
4. Branch: The algorithm is divided into three sections referred to as branches: blue, red and grey, this will be introduced fully in Figure 6.1
5. Fabric: The reconfigurable logic on an FPGA.
6. Zynq CPU: The name we use for the CPU on the MicroZed.
7. ‘pulsed’: This refers to making a signal logic high for one clock cycle, dependant logic is designed such as not to miss such a pulse.
8. row-wise, column-wise: Image processing consists of 2D data-sets, it is important to keep track of which order the data-set is transferred between Math-blocks.
9. Data-set: This refers to all the data required for a specific calculation. Each element in the data-set is referred to as a data-element.
10. M_{number} : This refers to the functional math-block (Figure 6.1) and $M_{number_number\dots}$ a grouping of such math-blocks. These are grouped together as a unit when it is convenient to join them functionally.

Text in italics refers to signals and functional blocks in a corresponding Figure. Appendix C.3 has an overview on FPGA development steps and processes.

6.1 Methodology

A bottom-up implementation of the SPITE algorithm was performed for the MicroZed 7020, using Xilinx's 2016.4 Vivado design suite.

A hardware design should be manageable, therefore *determinism* and *adaptability* were key when developing the hardware.

Determinism implies a thorough understanding of underlying functionality. VHDL was coded directly instead of making use of Vivado's High Level Synthesis (HLS) which takes C-code and converts it to register-transfer level code. Use of HLS tools would significantly reduce development time, but abstract the detail to some degree, thereby abstracting *intuitive*¹ optimisations, improvements and changes. A control and data path technique was used, as the algorithm fits this methodology. All control was implemented with a Finite State Machine (FSM), while data moves between memory buffers and math-units. Therefore, all calculations are controlled by custom FSMs, which control specialised IP-cores provided by Xilinx.

Adaptability implies scalability, which is convenient for debugging functionality in simulation. To this end, a scaled down data-size where $N = 4$ and $M = 8$ was first developed, fully simulated and debugged. Thereafter the final scaled up version, where $N = 32$ and $M = 64$, was tested on hardware only. This was done with the MicroZed 7020 development board, thereby removing the necessity to either debug a large design in simulation (extremely long simulation time) or excessively debug the large design in hardware (multiple synthesis and implementation runs). Scalability is also convenient for porting modules to other projects where different sizes are required.

In order to create a *suitable* implementation, the following design drivers – listed in order of priority – guided the development process:

1. Accuracy: to attain results representative of the Python simulation, we pay careful attention to the accuracy.
2. Resources: the implementation needs to fit onto a medium-range FPGA, Section 4.4 determined the smallest possible window and template combination, thus an efficient design needs to capitalise on these small window and template dimensions. A self-contained design will restrict the designer to use resources common to medium-range FPGAs, this facilitates portability.

¹HLS does allow significant optimisations to be applied easily, however low-level implementation gives that extra bit of control, desirable for ad hoc adaptations.

3. Runtime: using an FPGA to determine SPITE is only attractive if it is possible to do it faster than a pure microcontroller implementation.

Adherence to these three design drivers are implied during the design process, and will be elaborated in Sections 6.4, 6.5 and 6.6 respectively. Note that they are not always in conflict with one another, so a trade-off is not always required.

6.2 Hardware architecture

The first step to determining the best underlying architecture for the FPGA implementation, is to break down the SPITE algorithm – Eq. (6.1) – into its smallest mathematical components.

$$c_{norm}(x, y) = \frac{\mathcal{F}^{-1}\left\{\mathcal{F}\{f(x, y)\} \times \mathcal{F}^*\{t(x, y) - \bar{t}\}\right\}}{\sqrt{\sum (t(x, y) - \bar{t})^2} \sqrt{\sum (f(x, y) - \bar{f}_{u,v})^2}} \quad (6.1)$$

This can then be arranged into a flow diagram shown in Figure 6.1.

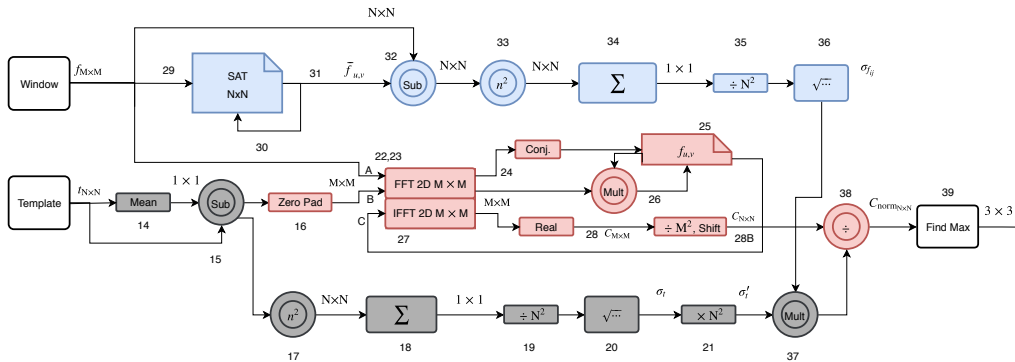


Figure 6.1: Low level math block layout, the various math-blocks' names will be the number next to them, e.g. the sum area table in the blue branch is equivalent to $M_{29_30_31}$. Note that math-blocks M_{19} , M_{21} and M_{35} are superfluous, as they cancel out after M_{37} .

Consider the various branches of the algorithm. From the investigation in Section 3.2.2.2, it was shown that the grey branch is the 'shortest'. It occurs once for *Template*. A low-throughput streaming methodology will be applied for this branch of the calculation to minimise resource usage. The blue branch, which repeats N^2 times, will require a high-throughput streaming methodology is also used for the red branch, as it contains the 2D-FFT, along with other computationally intensive calculations.

We use modular development principles, therefore the same streaming control will be used throughout the various branches. This streaming Finite State Machine (*FSM-streaming*), will be discussed in more detail in Section 6.3.1.

Note also that some operations require large temporary data to be stored during a computation, this will be done in BRAM's, convenient local memory. The typical example is the 2D-FFT which stores the result of the column-wise 1D-FFT, after which the row-wise 1D-FFT is calculated. These somewhat more complex math-blocks will be discussed individually in Section 6.3.

But before discussing the low-level details in-depth, consider how the algorithm will *interface* with the real world.

6.2.1 System-level interfaces

Consider the high-level diagram in Figure 6.2, a continuation of Figure 3.4, as well as taking some intuition gathered from Figure 6.1. The convenient CPU + fabric resources available in the MicroZed 7020 are utilised to divide the design into two segments: the bulk of the calculation which happens in the fabric (Blue, Red, Grey) and the loose ends which are tied in the *Zynq CPU* via a C-application. *Zynq CPU* also fulfils a second function: it allows easy integration with other peripherals from a high level of abstraction. As this is a prototype, this functionality is extremely useful for testing and future work scenarios. The system-level sub-components are shown in Figure 6.2:

Zynq CPU: The formal definition of the C-application which runs on *Zynq CPU*:

1. High-level abstracted control. *Zynq CPU* sends the following commands via the memory-mapped *AXI-lite* bus to *FSM Mom*, *reset* and *start*. The ability to pause mid-SPITE calculation was initially incorporated, however the added complexity for such a short calculation makes it infeasible.
2. Once the calculation in the fabric is complete, the maximum of $c_{norm}(x, y)$ with its surrounding 8 pixels is returned via *AXI-lite*'s interface. *Zynq CPU* does the last part of the computation, the polynomial least squares regression (Figure 3.1), after which it determines the turning point which provides the sub-pixel translation.
3. Housekeeping for further prototyping. This culminates in a UART to PC link in Section 7.1 required for streaming images to the MicroZed 7020, and responding with the appropriate calculated SPITE vectors.

FSM Mom: Receives the reset/start command from *Zynq CPU*. Based on this input and the various statuses from *FSM Blue*, *Red*, and *Grey*, and their

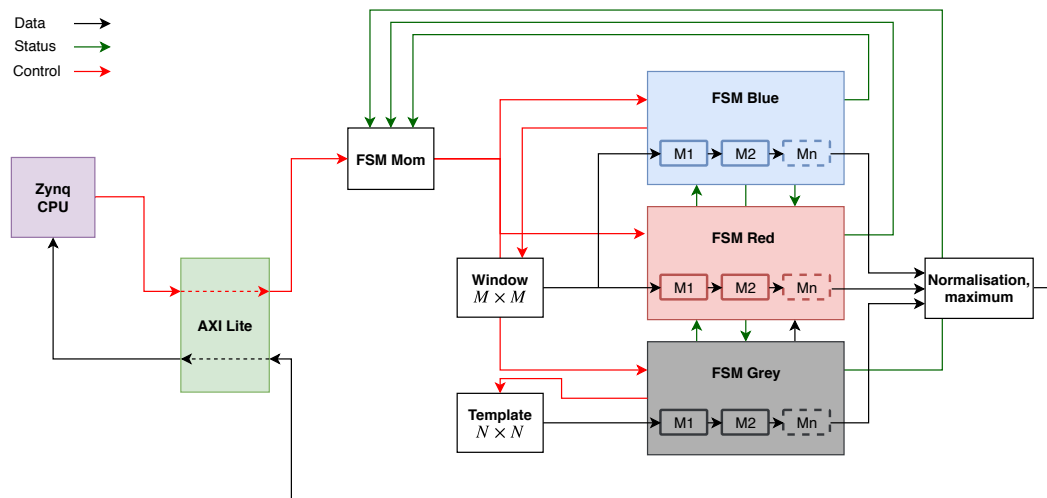


Figure 6.2: System-level architecture, showing how data, status and control is manifested in design. *Zynq CPU* sends commands via the *AXI-lite* bus to the highest level state machine, *FSM Mom*. *FSM Mom* then appropriates oversight to the lower level ‘children’ FSMs: *Blue*, *Red*, and *Grey*. These in turn control the data flow, and eventually send the result (pre sub-pixel) back across the *AXI-lite* bus, where *Zynq CPU* calculates the sub-pixel result.

respective math-blocks’ statuses, varying control signals are sent to the various branches’ main FSM.

Window: is the BRAM which contains the data from the first image, namely the 8bit $M \times M$ window data. This data is multiplexed between the Blue and Red branch by *FSM Mom*.

Template: is the BRAM which contains the data from the second (translated) image, namely the 8bit $N \times N$ template. This data only flows to *FSM Grey*.

FSM Red: controls the process of determining the 2D-FFT along with subsequent calculations, M_{23} , M_{23} , and M_{27} . It reads the full *Window*, then relinquishes control of *Window*.

FSM Blue: controls the process of determining the iterative part of the normalisation. It starts after M_{16} is completed since *FSM Red* will be finished using the *Window* data, since it performs M_{22} (2D-FFT of *Window*), then M_{23} (2D-FFT of zero-padded *Template*). Therefore, when M_{16} (the zero-padding process) is complete, *FSM Blue* takes control of *Window*’s data till the end of the full calculation.

FSM Grey: controls the process of determining the normalisation factor component of the template. It has dedicated control over *Template*.

Normalisation, maximum: gathers the answers from the Blue, Red and Grey

arms of the calculation, applies proper normalisation to produce $c_{norm}(x, y)$, determines the maximum, then sends the maximum with its 8 surrounding values (with the centre's coordinate) to *Zynq CPU* via *AXI-lite* bus.

6.2.2 Data-Level

The data flow is discussed, both the order which is required to obtain accurate results, as well as the control thereof.

6.2.2.1 Required data-flow

Consider the appropriate data flow required for the proper functioning of Figure 6.1:

1. *Template* is loaded into M_{14} . Simultaneously *Window* is loaded into $M_{22,23,27}$ for the first 2D-FFT calculation.
2. When M_{14} has determined the mean, *Template* is re-read so that the mean can be subtracted and sent to M_{16} pixel-wise, simultaneously this pixel-stream is sent down to $M_{17,18}$.
3. When the first 2D-FFT is complete, its result is stored in M_{25} , when M_{16} is ready, the zero-padded data is sent to $M_{22,23}$ for the second 2D-FFT calculation. When M_{16} has sent out its last data-element, *Window* is loaded into M_{29} in order to create the *SAT*². When the *SAT* is completed, the various sub-windows are read into M_{32} . In the meantime, $M_{25,26}$ is calculating the complex conjugate (M_{24}) and multiplying between the first two 2D-FFT results, and sending this result to M_{27} , to determine the 2D-IFFT.
4. The 2D-IFFT result streams pixel-wise into M_{28} and subsequently M_{28B} . After which M_{28B} sends the re-ordered data out pixel wise to one of M_{38} 's two input buffers, meanwhile, the output of M_{37} is streamed to M_{38} 's second input buffer. This is where the data is 'synchronised', so that the first data-element in the first buffer is divided by the first data-element in the second buffer and so forth. Thus the first buffer forces M_{28B} not to send out all its data immediately as the throughput of $M_{33,34,35,36}$ is significantly lower than that of M_{28B} .
5. Hereafter, the data is one single stream of normalised data-elements. The maximum along with its coordinates is found, and sent out via *AXI-lite*'s bus to the CPU for the final sub-pixel computation (polynomial least squares regression).

²Technically speaking, this could already occur after the first 2D-FFT was completed, this is slightly less efficient but not incorrect.

6.2.2.2 Data coordination

This control of the various calculations is achieved by having a strict *hierarchical* structure. At the lowest level are the individual math-blocks such as M_{28B} , M_{16} and $M_{19.20.21}$. They require their full data-set to fulfil their functionality. The pipelined fashion ensures that no data-elements are lost.

One hierarchy level higher is the FSM for each branch of the calculation namely: *FSM Blue*, *FSM Red*, and *FSM Grey* (Figure 6.2). These ensure that the math-blocks (within their colour grouping) are started at the right time, as well as relaying the control from *FSM Mom*, which is another hierarchy level higher. *FSM Mom* ensures smooth operation between different branch's of the calculation and *Zynq CPU*, which is at the top of the hierarchy.

The high level coordinator between the three branches of the calculation from a functional point of view is *FSM Mom*. It defines the following control:

1. when *FSM Mom* receives the start command from *Zynq CPU*, it sends start commands to both *FSM Red* and *FSM Grey*. These in turn start *Template* and *Window* to send template data to M_{14} , M_{15} , and window data to $M_{22.23.27}$ for the first 2D-FFT calculation.
2. *FSM Mom* waits for M_{16} to complete, after which it hands over control of *Window* to *FSM Blue*, as *FSM Red* will have finished loading window data. The first 2D-FFT will be complete by the time the second 2D-FFT has started.
3. *FSM Mom* waits for M_{39} to pulse its finished flag, after which fabric part of the calculation is complete.

The simplicity of the control structure is due to the robust design of each math-block which keeps track of where in the calculation it is as well as the back-presssure provided by their input-buffers (discussed further in Section 6.3). The ability to design the math-blocks in such a manner is greatly simplified by making the following design choice: only one SPITE calculation will be processed at one time. This means the 'streaming' which is referred to is on a pixel-by-pixel level within one SPITE, not between full data-sets.

A hierarchical approach to control, coupled with the simplicity of *FSM Mom*, enables custom functionality to be added easily. The following three options were *not* implemented in this project, however easy implementation thereof is facilitated:

1. *FSM Mom* could monitor the states of the various math-units, if they enter an undesired state (ERROR state), the whole calculation could be restarted. This concept is expanded on in Section 6.7.
2. Counters in the fashion of a watchdog timer could be implemented for certain parts of the calculation, if the calculation has not been completed in adequate clock cycles, the whole calculation could be restarted.

- On a higher level, since the clock is propagated through *FSM Mom*, and given the nature of *Zynq CPU*, frequency scaling could be implemented between subsequent calculations if an increase or decrease in throughput is desired.

6.3 Detailed math-blocks

This section discusses the various designed math-blocks. Unless explicitly stated, 32bit floating point precision is used for the math-units.

In the discussion, signals and functional blocks from figures are written in italics.

6.3.1 Streaming math-blocks

A generic streaming design is implemented for the following blocks, as they conveniently map to streaming control. The difference between the functionality which they perform, has no influence on their control. They are listed by branch affiliation, as in Figure 6.1:

- Blue: M_{32} , M_{33_34} , M_{35_36} , input of $M_{29_30_31}$
- Red: M_{38} , the inputs of M_{22} , M_{23} , M_{27}
- Grey: M_{14} , M_{15} , M_{17_18} , $M_{19_20_21}$, M_{37}

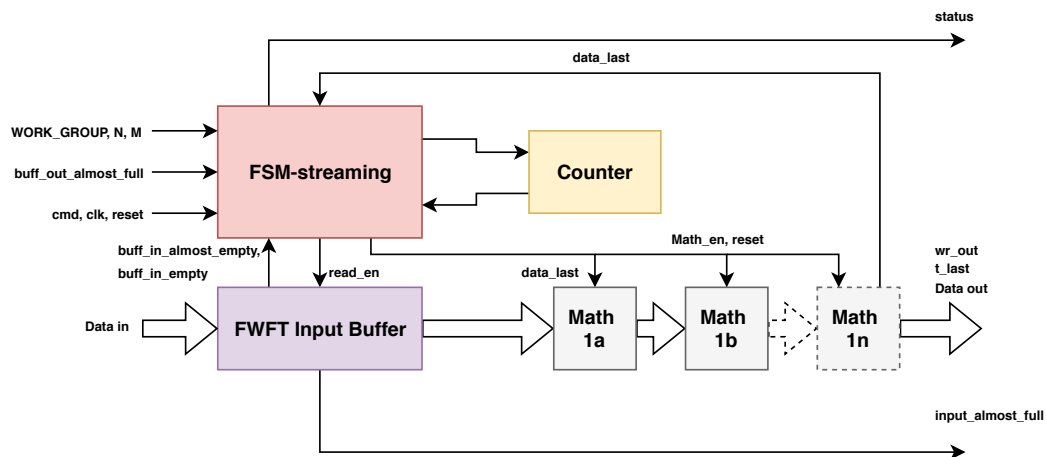


Figure 6.3: Generic layout of *FSM-streaming*. A full streaming math-block generally has, 1x *FSM-streaming*, 1x *Counter*, 1x *FWFT Input Buffer* and various Math-units e.g. *Math 1a*.

FSM-streaming will now be discussed for the general case, as shown in Figure 6.3. The discussion will be abstracted to better communicate the available functionality.

FSM-streaming is arranged in a serial pipeline: data comes in from outside the math-block (the left), and is stored in the First Word Fall Through (*FWFT*) *Input buffer*³. This buffer is chosen to be 32bit wide and 512 deep, such that no more than 1x18Kbit BRAM is utilised fully.

WORK_GROUP is the number of data-elements in one full data-set. When the start command is issued (via *cmd*), data is read from *FWFT input buffer* into the first math-unit *Math 1a*. If no data is available, it waits until a data-element enters the buffer. Therefore, a data streaming approach was chosen instead of a burst data approach.

Every time a data-element is read from *FWFT Input Buffer*, *Counter* is incremented, it keeps track of how far into the data-set the calculation is. *Counter* also simplifies the issue of coordinates, coordinates do not need to be sent along with the data if the size of the template/window is known and a data order convention upheld. Consider e.g. the window which is $M \times M$, it requires a $2 \times \log_2(M)$ bit counter to count through exactly all M^2 data-elements. When the last data-element is read, *data_last* is asserted, this flag propagates through the various math-units. *FSM-streaming* asserts *Math_en* until *Math 1n* asserts *data_last*, which indicates that a full data-set of size *WORK_GROUP* has been processed.

The streaming math-block design

- allows for convenient cascading of these math-blocks, the output of the first math-block is coupled to the subsequent math-block's input.
- is aware of the output buffer status (the input buffer of the next math-block, *FWFT Input Buffer*). When *buff_out_almost_full* = 1, the whole math-block waits until the flag is cleared. This back-pressure ensures that data is never written to a full buffer. Thus no data-elements are lost. Handshaking between output and the next math-blocks input buffer is not enabled.
- has a 20-bit counter system which allows *WORK_GROUP* to be any size, from 1 to 1048575, therefore streaming math-blocks are easily scaled.
- allows a large data-set to be broken into parallel sets (where functionality allows), this is made possible by the *WORK_GROUP* and passing of window and template sizes M , N separately. If for example a data-set is to be broken into four parallel lines, each math-block would have: $WORK_GROUP_{parallel} = WORK_GROUP_{Serial}/4$.

³This buffer has a read latency of 0 clock cycles

- allows immediate reset, as *FSM-streaming* can respond to a reset command during any state (this is true for all FSMs throughout the project)
- provides a unique 8bit *status* for the current FSM state, useful for control and hardware debugging purposes.

Appendix C.8.1 shows an example of the typical streaming case which displays an example of input/output buffer awareness, Appendix C.8.2 and C.8.3 attest to the inherent versatility which the separate control and data path facilitates: easy modification of the math-data-path where necessary. Specifically $M_{17,18}$, where fixed-point optimisations are utilised, and $M_{19,20,21}$ where a lower throughput is made possible. In order not to duplicate information, Appendix C.8 should be read in order.

6.3.2 $M_{29,30,31}$: sum area table

This math-block determines the window's 'under the template' mean: $\bar{f}_{u,v}$. It uses three linked FSMs. In Figure 6.4 the colour of the FSM relates to its control of the data, this is achieved via multiplexed data and address lines.

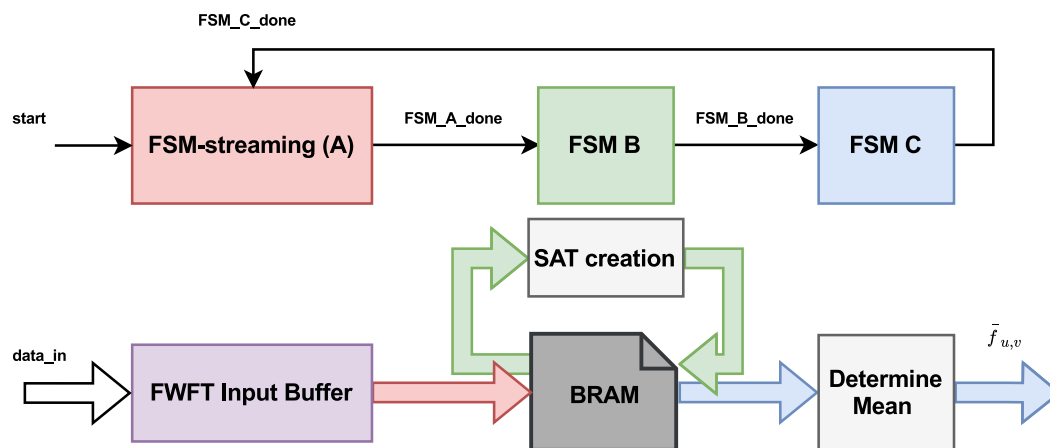


Figure 6.4: Architecture of sum area table math-block ($M_{29,30,31}$)

Its modus operandi is:

1. After *FSM-streaming (A)* receives a *start* command, it reads the data from *FWFT Input Buffer*, converts it from 32bit float to 24bit unsigned fix-point (ufix), and stores it in *BRAM* row-wise. This is sufficient as data is still just representing pixel values ranging from 0-255. When the whole data-set is read through, *FSM_A_done* is asserted.
2. *FSM-B* now updates *BRAM* with the definition of the SAT: $s(x, y) = f(x, y) + s(x - 1, y) + s(x, y - 1) - s(x - 1, y - 1)$ (3.15) and asserts *FSM_B_done* when finished.

3. *FSM-C* now reads out the four required data points, calculates the numerator of Eq. (3.16), converts back to float₃₂ and finally divides by N^2 , the denominator of Eq. (3.16). This is performed N^2 times, to read out all N^2 $\bar{f}_{u,v}$. After it has read out all N^2 values of $\bar{f}_{u,v}$, it asserts *FSM-B_done*. This enables *FSM-streaming* to respond to a *start* command, the starting point of the whole process.

Consider the nature of the SAT, always adding pixel values, even when the *FSM-B* updates the values with equation Eq. (3.15), the values are always positive integers. A 24bit unsigned integer is used, as it has range of 0 to 16777215 before overflowing. Therefore in the worst case where every pixel has a value of 255, $255 \times M^2 \leq 16777215$, $M = 256$, therefore the SAT Math-block will scale well till $M = 256$ in this configuration.

Memory requirements ($M = 64$): BRAM: (24bit $\times M^2$), FIFO: 32bit $\times 512$, Total: 3.5×36 Kbit BRAMs.

6.3.3 Sub-window loader

This math-block does not actually do any math, it merely loads the relevant ‘under the template sub-window’, and sends it to M_{32} . Since the template ‘moves’ across all possible N^2 locations, all the pixels in *Window* which happen to fall under the template at each increment and need to be appropriately read out.

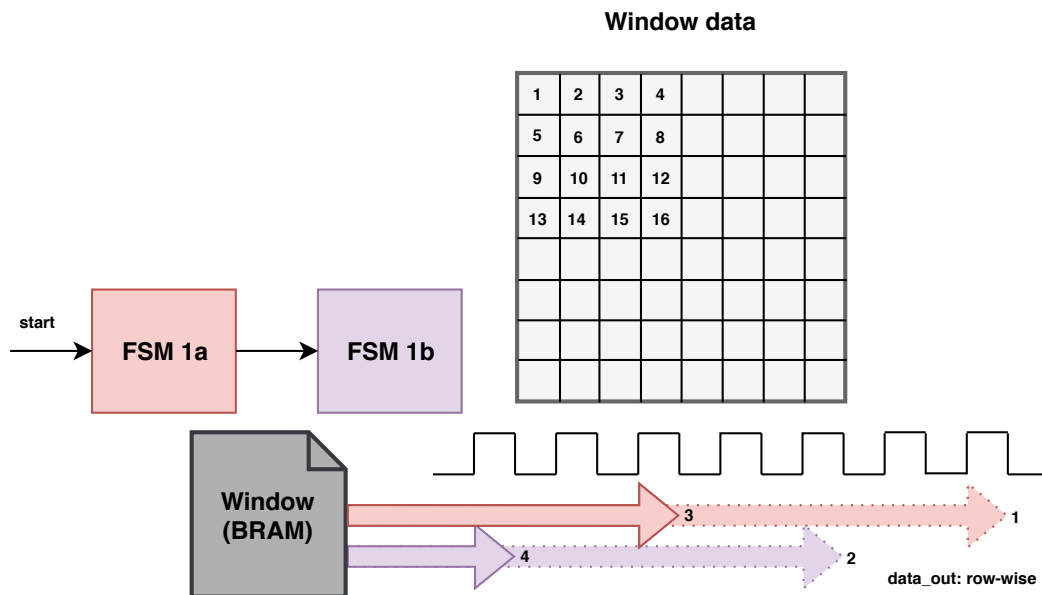


Figure 6.5: Architecture and order of sub-window pixels being read by sub-window loader, where $N = 4$, $M = 8$. Pixels numbered in *window data* are the first sub-window to be read out. Note how only 2 data-elements are read out every 4 clock cycles, and how FSMs read alternating pixels.

Figure 6.5 shows that:

1. When *start* is received from *FSM Blue*, and then proceeds to read the first ‘under the template sub-window’. The order is represented in Figure 6.5. Note how *FSM 1a* and *FSM 1b* both read from the same BRAM, in synchronised out of order fashion, an optimisation which was implemented to reduce runtime (see Section 6.6.1). The window is stored 8bits per pixel. Therefore as each pixel is read out, it gets sent through a 8bit ufix to 32bit float converter.
2. After the first sub-window is read out, the next sub-window is read out, this is the whole sub-window shifted one pixel position to the right.

Memory requirements ($M = 64$): BRAM: $8\text{bit} \times M^2$, Total: $1 \times 36\text{Kbit}$ BRAM.

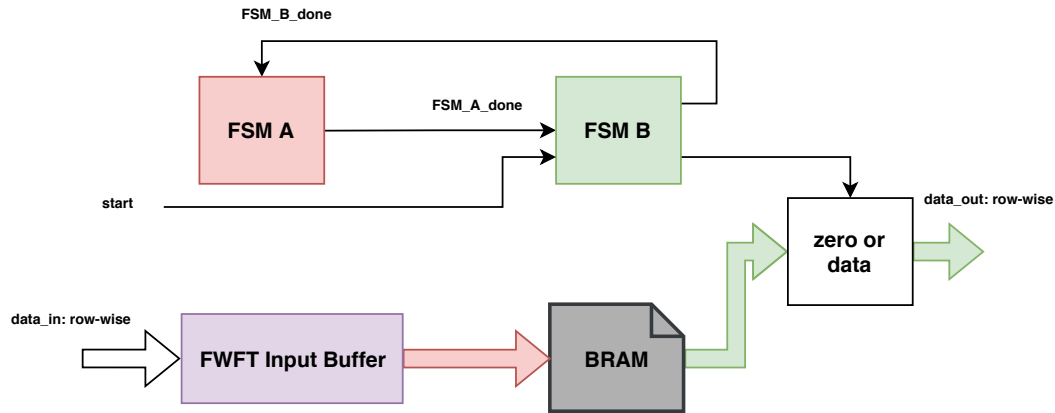
6.3.4 M_{16} : zero pad

This math-block zero pads the $N \times N$ template, where its mean has been removed by M_{15} , to dimensions $M \times M$. This is so that a $M \times M$ 2D-FFT may be applied to it. The colour of the FSM once again refers to its control of the data.

Figure 6.6 shows that:

1. The data enters the math-block row-wise and is stored in *FWFT Input Buffer*. *FSM A* does *not* wait for a *start* signal, it immediately reads the data out of *FWFT Input Buffer* and writes it to *BRAM* row-wise. When the whole data-set has been read through and written, *FSM_A_done* is pulsed.
2. *FSM B* waits for *FSM A* to be done as well as *FSM Red* to provide it with the *start* signal, thereby the output data of M_{16} is controlled, as it needs to wait for the first round of 2D-FFT to be complete before being read out. It has a counter which counts through a full $M \times M$ image: when the counter represents a coordinate which is in the zero-region, a zero is sent to the output. When its coordinates represent the non-zero region, it reads out the value from *BRAM*. The data is also sent out row-wise. When it is finished, *FSM_B_done* is pulsed to enable *FSM A* to read and write data through again, the starting point of the whole process.

Memory requirements ($M = 64$): BRAM: $32\text{bit} \times N^2$, FIFO: $32\text{bit} \times 512$, Total: $1.5 \times 36\text{Kbit}$ BRAMs.

Figure 6.6: Architecture of Zero-Pad Math-block (M_{16})

6.3.5 $M_{22_23_27}$: 2D-FFT

This math-block determines the two dimensional FFT/IFFT. It requires data to be sent in row-wise, and sends the result column-wise. It is configured for 64×64 2D-FFTs. Xilinx's FFT IP-core makes use of a 24bit phase factor as opposed to 25bit, the difference according to [58] is roughly 5dB in accuracy. Whether this has a significant influence on the final answer is only discussed in Section 7.1. It is configured to Forward or Inverse mode during operation, and for a target frequency of 250MHz, the maximum possible clock output frequency from *Zynq CPU*⁴.

The operation is:

1. When *start* is pulsed, *FSM-streaming(A)* configures *1D-FFT* block to either forward or inverse mode, depending on what *FSM Red* requires. Thereafter it streams data from *FWFT Input Buffer* into the *1D-FFT* row-wise. This result is written to the *BRAM*, stored row-wise. When the full data-set is streamed through, *Dimension_1_done* is pulsed.
2. *FSM-B* now burst-reads 1 full column from the *BRAM* at a time, and writes it to *Buffer intermediate*. *FSM-streaming(A)* now reads data from *Buffer intermediate* (while *FSM-B* feeds *Buffer intermediate*) and sends it to *1D-FFT*. The result thereof is then sent out, to the next math-block column-wise. *FSM-B* will pulse *FSM_B_done*, when it has read out all the columns, to indicate when a full data-set has been sent to *Buffer intermediate*.
3. *FSM-streaming(A)* pushes the remaining data through *1D-FFT*. When it is finished, an internal flag is set to switch *FSM-streaming(A)* back to *FWFT Input Buffer*, and waits for a *start* command.

⁴The fabric gets its clock from *Zynq CPU*.

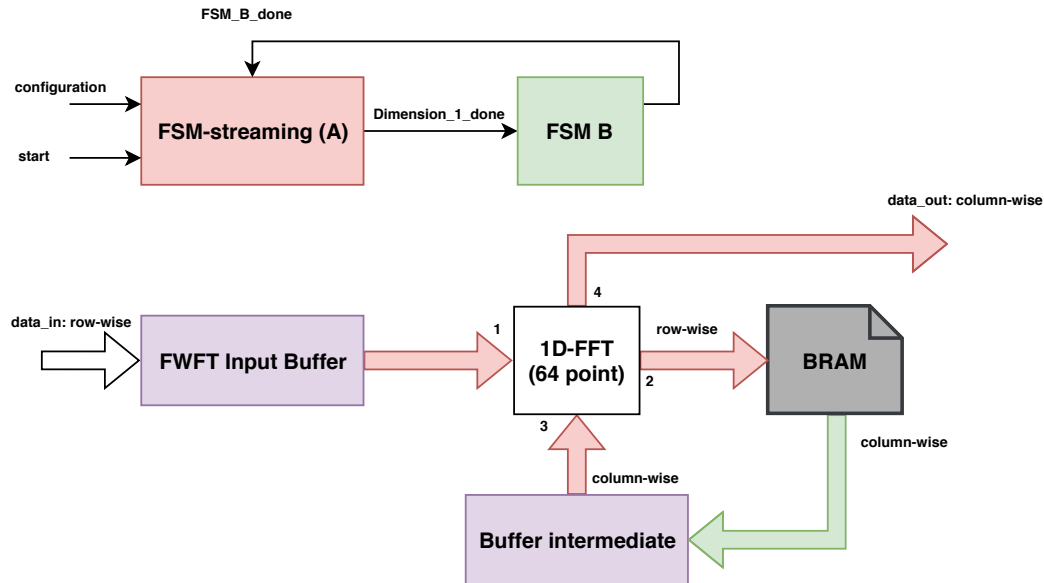


Figure 6.7: Architecture of 2D-FFT Math-block ($M_{22_23_27}$), note numbering 1 to 4 indicates the order in which the data flows.

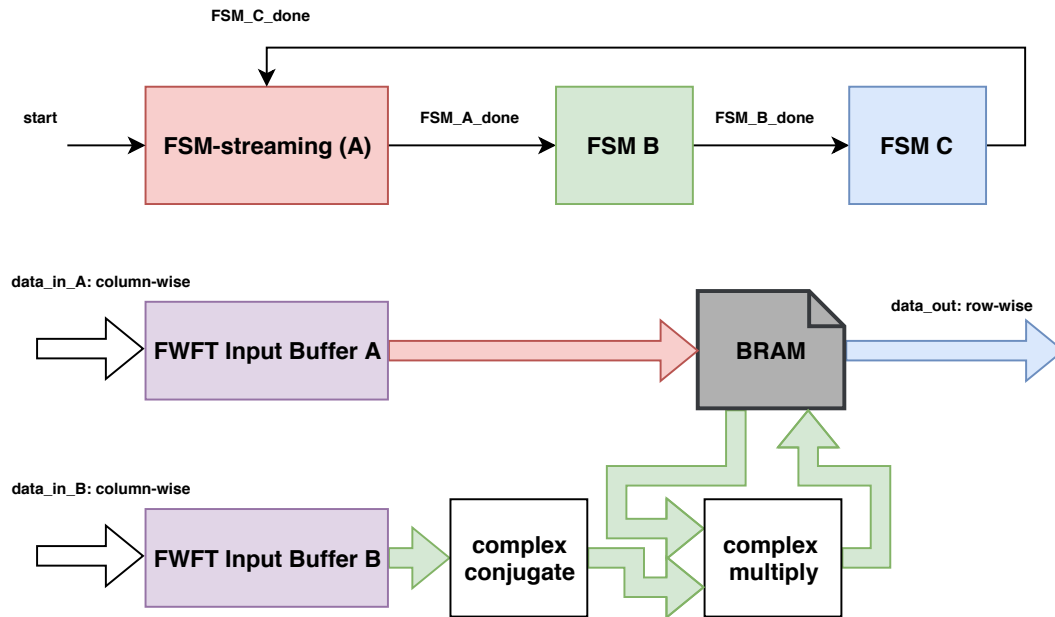
Memory requirements ($M = 64$): BRAM: $64\text{bit} \times M^2$, FIFO: $2 \times 64\text{bit} \times M$, Total: $8.5 \times 36\text{Kbit}$ BRAMs.

6.3.6 $M_{24_25_26}$: conjugate and multiply

This math-block receives the column-wise result from the 2D-FFT Math-block: the 2D-FFT of the window and zero-padded (template - mean) data. It determines the complex conjugate of the zero-padded 2D-FFT, and then complex multiplies it with the window's 2D-FFT.

Figure 6.8 shows that:

1. when *start* is pulsed, *FSM-streaming(A)* reads the data from *FWFT Input Buffer A*, and then stores it column-wise into *BRAM*. It pulses *FSM_A_done* when complete.
2. *FSM B* now reads the data from *FWFT Input Buffer B*, sends the data through *complex conjugate*, collects its corresponding data points from *BRAM*, and sends them to *complex multiply*. The result is then written back to the same location of *BRAM*, all happening column-wise. It pulses *FSM_B_done* when complete.
3. *FSM C* now reads out the data from *BRAM* row-wise, since the subsequent math-block (2D-FFT) requires the data in row-wise fashion. When it has completed, *FSM_C_done* is pulsed to enable *FSM-streaming(A)* to wait for *start*, to repeat the whole process again.

Figure 6.8: Architecture of Math-block ($M_{24,25,26}$)

To increase performance in this high-activity region, a simple, high-throughput complex-multiplier was implemented (Figure 6.9) to perform:

$$\begin{aligned} \text{Complex Multiply} &= (a + bj)(c + dj) \\ &= (ac - bd) + (bc + ad)j \end{aligned}$$

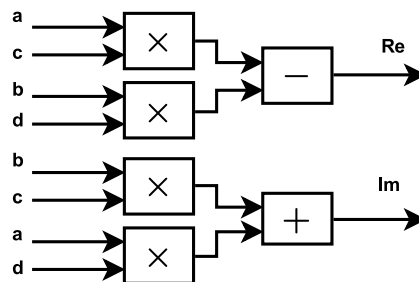


Figure 6.9: High throughput complex multiplier: each square represents a 32bit floating point math-unit.

Memory requirements ($M = 64$): BRAM: $64\text{bit} \times M^2$, FIFO: $2 \times 64\text{bit} \times 512$, Total: $9.5 \times 36\text{Kbit}$ BRAMs.

6.3.7 M_{28B} : data shift

This math-block re-orders the $M \times M$ data elements which come from 2D-IFFT, to the $N \times N$ data elements present in $c(x, y)$, it is equivalent to

`numpy.fft.fftshift()`. It also divides each element by M^2 . This re-ordering is crucial, since the output of M_{37} – the normalisation factors – corresponds to a specific sequence. After re-ordering and normalisation, a zero-shift will correspond to a maximum in the centre of $c_{norm}(x, y)$, as an example for a simple $M = 8, N = 4$ scenario shown in Figure 6.10, the maximum would be at ‘K’.

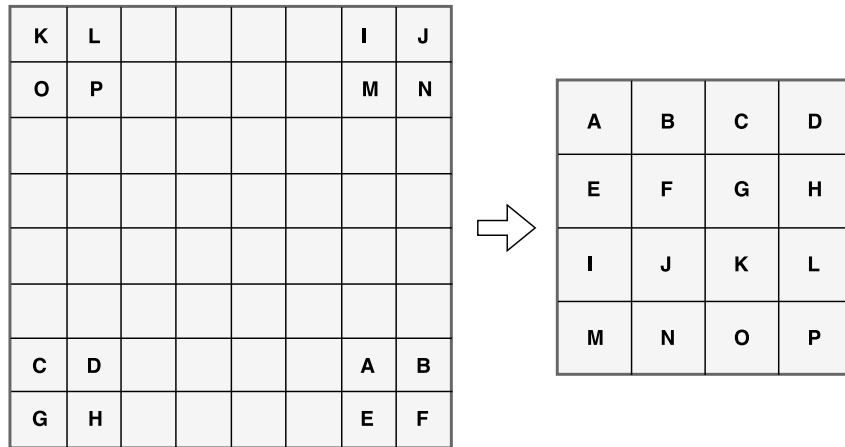


Figure 6.10: A diagrammatic representation of M_{28B} 's reordering, a zero translation would result in the maximum being present at ‘K’ after normalisation.

Figure 6.11 shows that:

1. when start is pulsed, $FSM-streaming(A)$ reads the data from the $FWFT$ Input Buffer, sends it through the division IP-core and stores in $BRAM$ column-wise. FSM_A_done is pulsed when all $M \times M$ data elements are stored in the $BRAM$.
2. $FSM B$ now reads out the $N \times N$ data elements in the required order. Data is sent out in row-wise order. FSM_B_done is pulsed to enable $FSM-streaming(A)$ to wait for $start$ to repeat the whole process again.

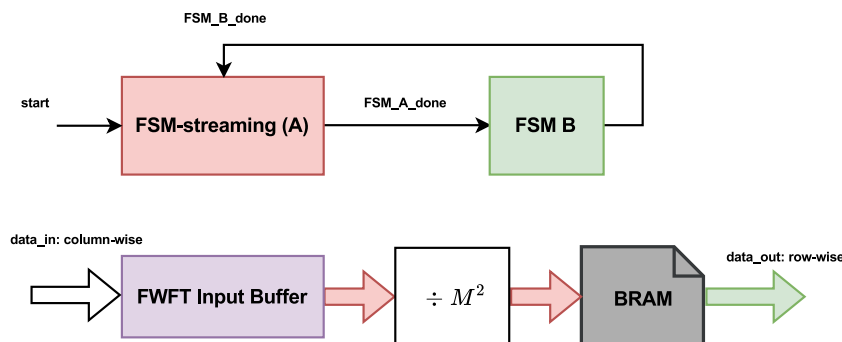


Figure 6.11: Architecture of M_{28B} .

Memory requirements ($M = 64$): BRAM: $32\text{bit} \times M^2$, FIFO: $32\text{bit} \times 512$, Total: $4.5 \times 36\text{Kbit}$ BRAMs.

6.3.8 M_{39} : max 3x3

This Math-block determines the maximum value of $c_{norm}(x, y)$, and sends out the maximum value with its coordinates, as well as the 8 values that surround it. It receives the $N \times N$ data row-wise, and sends out the data in parallel to *AXI-lite*, such that each element corresponds to a different address⁵.

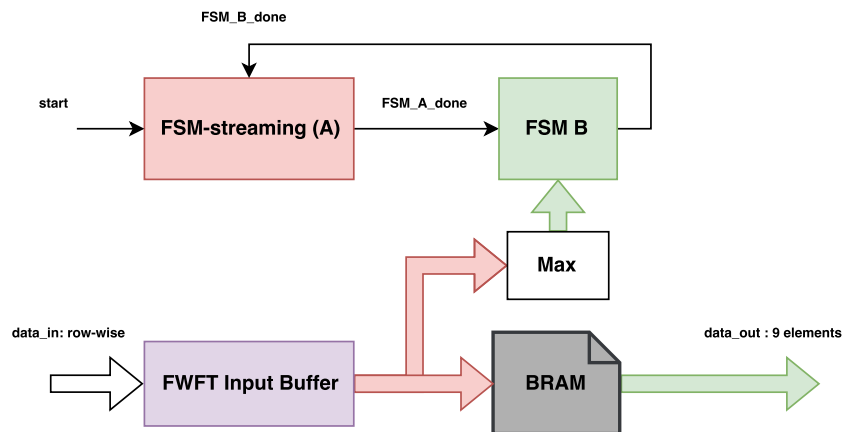


Figure 6.12: Architecture of M_{39} .

Figure 6.12 shows that

1. when *start* is pulsed, *FSM-streaming(A)* reads the data from *FWFT Input Buffer* into *BRAM*. Simultaneously it sends the data to logic which determines whether it is greater than the previous data element or not. When it has run through the whole data-set: *Max* holds the coordinate of the maximum value and *FSM_A_done* is pulsed.
2. *FSM B* reads out the relevant 9 values by using the maximum value's coordinates as an offset, then pulses *FSM_B_done* when finished to wait for *start* to repeat the whole process again.

Memory requirements ($M = 64$): BRAM: $32\text{bit} \times N^2$, FIFO: $32\text{bit} \times 512$, Total: $1.5 \times 36\text{Kbit}$ BRAMs.

6.3.9 M_{40} : surface fit

Lastly, the surface fit algorithm is implemented. It retrieves the 9 data points from M_{39} with its centre pixel coordinate. Eq. (6.2) – introduced in Section

⁵This is not the most elegant approach, a serial implementation could also be implemented.

3.1.1 – is used to determine the sub-pixel shift, after which it incorporates the coordinate to accommodate the integer offset.

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{z} \quad (6.2)$$

We implement this in a C-application on *Zynq CPU*, since it is easier to develop recursive operations such as Eq. 6.2, in C instead of VHDL. Recursive functions such as matrix inversion, will tend to use n times more FPGA resources when the recursion takes place n times, resulting in bloated FPGA implementations. Conveniently, M_{40} is the last calculation in SPITE, therefore data only transfers from the fabric to *Zynq CPU* once. *Zynq CPU* is already in use for rapid prototyping, therefore it is easy to add and debug this code.

The C-equivalent of Eq. (6.2) is constrained to use 64bit doubles, so that no further error is introduced due to lack of resolution. The various matrix operations – included in Appendix C.5 – are `mult_mat(A, B)`, `transpose(A)`, and `inverse_mat(A)`, which has its sub-functions `adjoint(A)`, `determinantOfMatrix(A)`, and `getCofactor(A)`.

Although not all the math-blocks and FSMs are discussed here, this Section has aimed to provide the reader with a feel for what is occurs on the FPGA from a math, control and data-flow perspective.

6.4 Hardware accuracy comparison

The HW implementation, where floating point math is required, makes use of 32bit floats, whereas the SW simulation uses 64bit floating point math. It is important to note the *magnitude* of the error that is introduced due to such limitations as well as determining its *influence* on the final answer. Similarly, the phase factor in Xilinx’s FFT IP-core uses 24bit resolution instead of 64bit.

In order to capture various stages of the calculation, outputs from the math-blocks are stored in a FIFO. Afterwards, these are read out to the PC and compared to the SW’s answer. The precise method of achieving this is: the data captured in the FIFO is read from FIFO to the Integrated Logic Analyser (ILA), this read is initialised with a Virtual Input/Output (VIO) via JTAG connection, then ILA data – which is read via JTAG – is dumped to a .csv file, which is then interpreted by a Python script. This is one way of performing HW debugging on the MicroZed 7020 without having to use abstraction layers for higher level data download.

A known test data-set from the Stellenbosch image set [1] – Figure 6.13 – was used to ensure that functionality was correct and coordination was in sync during development phase. It had the secondary purpose of documenting the accuracy.

Three specific outputs, namely each branch’s output is shown in Figure 6.14. It holds the SW simulations result, the HW result and the error, for the

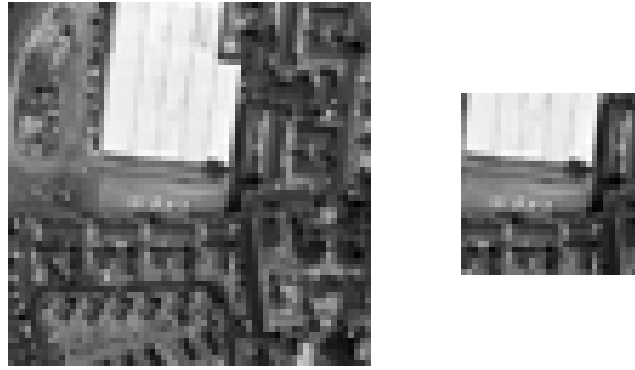


Figure 6.13: 64×64 window (left) and 32×32 template (right) test data-set [1] used to calibrate, check synchronisation, and determine resolution error of HW implementation. Subsection of [1].

three math-blocks: M_{28B} , M_{37} and M_{38} . The error present in M_{37} is purely a function of 32bit resolution, whereas M_{28B} 's error is also as a result of the FFT implementation. Refer to Figure 6.1, it shows that M_{38} is the result of a point-wise division of M_{28B} by M_{37} , thus merging the two errors. Table 6.1 shows the percentage of the errors' magnitudes. The small error might be an insignificant error, as its influence on the final SPITE result might be negligible. Since the HW implementation provides results that have acceptable error margins, the design is deemed adequate. A small, insignificant error negates the necessity of further characterisation and study thereof. This speculation is confirmed in Section 7.1, where we validate the insignificance of the induced error.

Table 6.1: Percentage of error due to 24bit phase factor in FFT math-unit, and 32bit floating point HW implementation as opposed to 64bit floating point SW implementation. This is over all pixels in the single test image.

| Math-block | Max error [%] | Average error [%] |
|------------|------------------------|-----------------------|
| M_{37} | 1.455×10^{-5} | 4.37×10^{-6} |
| M_{28B} | 3.24×10^{-5} | 8.83×10^{-6} |
| M_{38} | 3.95×10^{-5} | 1.14×10^{-5} |

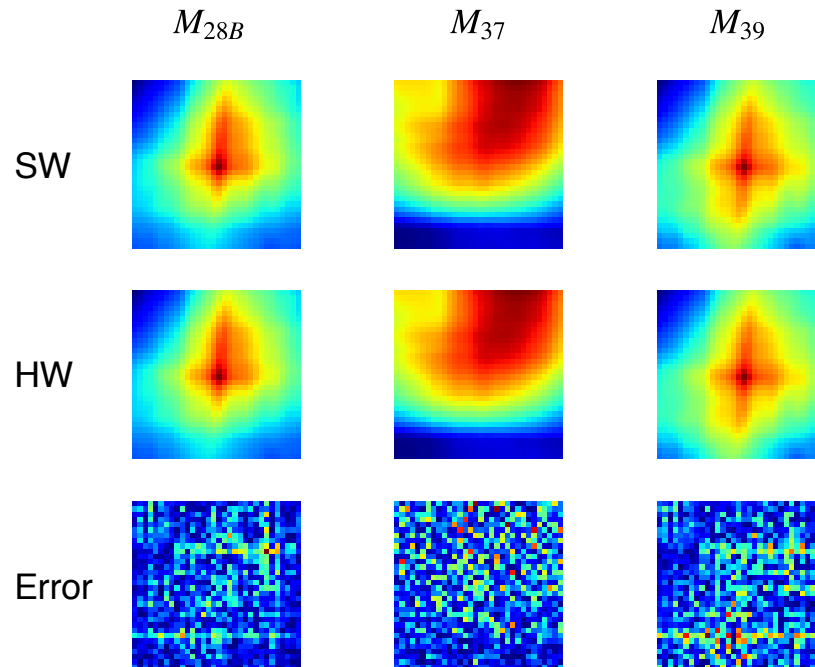


Figure 6.14: The error between M_{28B} , M_{37} and M_{39} , where SW's 64bit resolution is treated as 'Ground Truth', and the HW induces an error due to lack of resolution. Note how the maximum of M_{38} 's output is in the centre, this is because the data-set used has a zero shift in both axes.

6.5 Design for resource scalability

In Sections 3.2.2.2 and 4.4, it was shown that the dimensions of the template and window are of utmost importance for both accuracy, robustness and feasibility. Easy scalability of M and N is therefore an important part of the design and implementation, not only for debugging purposes. The aim was to make the implementation easily scalable.

This is clearly seen in the way that the streaming math-blocks are designed, as discussed in Section 6.3. Note how neither M_{14} , M_{15} , M_{17-18} , $M_{19-20-21}$, M_{32} , $M_{33-34-35}$, M_{37} , M_{38} or M_{40} need to increase their resource usage when scaling up. A study however of the block diagram of $M_{22-23-27}$ (in Appendix Figure C.3), is representative of how $M_{29-30-31}$, M_{16} , $M_{24-25-26}$, M_{28B} and M_{39} are laid out. It shows that some manual reconfiguration will need to occur before scaling these specific math-blocks, in specific counters, data/coordinate multiplexor lines, BRAM depths, and various FSM's generics will need to be altered.

As was shown in Section 6.3, the design methodology of these math-blocks rely heavily on BRAMs as a means of storing intermediate data. As a guide for future designs, Table 6.2 provides a good general BRAM usage per set of window and template dimensions. DSP48, LUT and other resource usage is not

reported on as designers can easily modify the various math-units to trade-off between using LUTs and DSP48s. The only math-block which will require significantly more DSP48s and LUTs when scaling up is the 2D-FFT: $M_{22.23.27}$. Using the data from Table 6.2 together with FPGA family datasheets, an optimal FPGA can be selected for the final scaled design.

Table 6.2: Memory required for various math-blocks of various sizes M , where $N = M/2$. Memory is in terms of 36Kbit BRAMs required. For an approximate breakdown see Appendix C.1

| M-B | M = 256 | M = 128 | M = 64 |
|----------------|---------|---------|--------|
| Window | 15 | 4 | 1 |
| Template | 4 | 1 | 0.5 |
| M_{14} | 0.5 | 0.5 | 0.5 |
| M_{15} | 0.5 | 0.5 | 0.5 |
| M_{16} | 15 | 4 | 1.5 |
| M_{17-18} | 0.5 | 0.5 | 0.5 |
| $M_{19-20-21}$ | 0.5 | 0.5 | 0.5 |
| $M_{22-23-27}$ | 115 | 30 | 8.5 |
| $M_{24-25-26}$ | 115 | 30 | 9.5 |
| M_{28B} | 58 | 16 | 4.5 |
| $M_{29-30-31}$ | 44 | 12 | 4 |
| M_{32} | 0.5 | 0.5 | 0.5 |
| $M_{33-34-35}$ | 0.5 | 0.5 | 0.5 |
| M_{37} | 0.5 | 0.5 | 0.5 |
| M_{38} | 0.5 | 0.5 | 0.5 |
| M_{39} | 16 | 5 | 1.5 |
| Total | 386 | 106 | 35.5 |

When M doubles to 128, memory usage goes up by a factor of three, therefore a high memory penalty exists for a small gain in accuracy.

6.5.1 Further Optimisation

Consider the case where a larger N, M is required, Table 6.2 clearly says a significantly larger FPGA will be required to facilitate such a design with the current design methodology. Two ways of reducing BRAM usage *without* altering the design significantly are discussed, they do however violate the initial design drivers of being less accurate and not self-contained:

1. The simplest way of reducing the BRAM usage, is to use 16bit floats instead of 32bit floats. Consider $M_{24.25.26}$ as an example. It requires storage for $(M)^2$ complex numbers. This is equivalent to $4M^2$ floats. Reducing the resolution from 32 to 16 bit per float will give a $2\times$ reduction

to $2M^2$. A $2\times$ reduction will be valid for the following math-blocks: $M_{22_23_27}$, $M_{24_25_26}$, M_{28B} and M_{16} . This will roughly half BRAM usage shown in the table. The resolution error will need to be monitored carefully, as in Sections 6.4, as well as proving that the resulting accuracy is sufficient, as in Section 7.1, since a significant increase in error is expected.

2. The MicroZed 7020 has access to external RAM. In the scenario where reducing from 32 to 16bit float resolution is not sufficient or viable, large portions of intermediate data may be stored in this RAM. An AXI DMA may be used to achieve this. This approach will be significantly tedious when compared to the previous suggestion, and less portable as we are not guaranteed a certain – if any – RAM across different platforms. Such an implementation may incur a reduction in throughput, since interfacing with external RAM will most likely incur lead times significantly greater than two clock cycles which is the case with BRAM. Optimised pipeline data access will be beneficial to such a design.

6.6 Runtime: individual math-blocks

It is useful to determine the precise runtime of the various math-blocks, especially when they are designed with re-use in mind. This provides a better intuition of what the FPGA implementation enables, and hints at final total system runtime.

To determine runtime, clock cycles need to be counted and maximum operating frequency be determined⁶.

Aim: Determine the runtime of various complex math-blocks.

Method Fabric Math-Blocks: The math-blocks are tested individually, with a data-stream representative of what it might experience if fed directly by the *Template/Window* read controllers, therefore the performance is not hindered severely by a slow data stream⁷. The total runtime starts when first data element is loaded into math-block and ends when the last data element leaves math-block. The number of data elements is the amount that is expected for this application ($M = 64, N = 32$). The fabric math-blocks (all math-blocks except M_{40}) use a simple counter implemented in HW: it starts counting when the first data-element enters the math-block, and stops when the last data-element leaves the math-block. This counter runs on the same clock as

⁶If terms such as Synthesis, Implementation, Place and Route are not familiar to the reader, see Appendix C.3 for an overview of FPGA design steps and processes.

⁷The influence of a slow data stream is most prevalent in M_{37} , as it needs to wait roughly 2144 clock cycles for each output from M_{36} , N^2 times.

the math-block, and since the progression of each math-block is deterministic, it provides deterministic results.

The maximum frequency is determined using Vivado's default synthesis options for the MicroZed 7020. Implementation results (place and route) are purposefully negated for timing analysis in this section, as a generic metric is desired to represent the coding-style, and not the timing characteristics of the underlying FPGA resources⁸. Maximum frequency is determined by setting the clock constraint to 100MHz, and then determining the worst negative slack post-synthesis, subtracting that from 10ns (period of 100MHz), and determining the resultant frequency.

Method C-Math Blocks: Timing M_{40} is somewhat more nuanced as it runs on the Zynq's CPU. Appendix C.7 shows all the steps involved in determining C-code runtime. Precisely the same methodology is followed in all subsequent C-application timing unless stated otherwise.

Table 6.3: Runtime of various important math-blocks, in clock cycles along with time (t) in milliseconds. Frequency given is the maximum possible frequency the math-block can be operated at and the times listed is the minimum runtime achieved at this maximum frequency. Various data-set sizes.

| Math block | Clk cyc's. | Freq. [MHz] | t [ms] | Size |
|--------------------------------------|--------------------|-------------|---------|--------|
| $M_{29.30.31}$ | 97786 | 164 | 0.59625 | M^2 |
| $M_{32}, M_{33.34.35.36}$ | 2144×1024 | 190 | 11.56 | N^4 |
| M_{16} | 24561 | 204 | 0.1203 | N^2 |
| M_{28B} | 16397 | 170 | 0.096 | M^2 |
| $M_{22.23.27}$ | 17515 | 173 | 0.101 | M^2 |
| $M_{24.25.26}$ | 45087 | 215 | 0.209 | $2M^2$ |
| $M_{14.15}, M_{17.18}, M_{19.20.21}$ | 6291 | 152 | 0.0413 | N^2 |
| M_{40} | 1027940 | 666.66 | 1.54 | 11 |

Results: The calculation time per math-block is displayed in Table 6.3. It is quantified in clock cycles, as the maximum frequency may vary between various FPGA platforms⁹. Surprisingly the 2D-FFT math-block is not the slowest: this shows the benefit of a pipelined approach, although the latency of a 1-row FFT calculation is 360 clock cycles, when properly pipelined the final runtime is significantly less than $360 \times 2M$, the number of clock cycles if each

⁸The post-synthesis timing results will make use of hardware models which will be dependant on speed-grade, however the magnitude of the resource usage will not influence the result (as post-implementation timing analysis would). Consequently reasonable projections may be made for FPGAs with differing amounts of resources.

⁹This is dependant on the speed-grade of the platform, as well as the amount of resources available, which may lead to more optimal routing, resulting in a higher possible frequency.

row/column of FFT was performed un-pipelined as the theoretical 2D-FFT block in Section 5.1.2.

M_{32} , M_{33} , M_{34} , M_{35} , M_{36} , the loading of the ‘sub-window under the moving template’ is the longest part of the calculation of the fabric. The reason is that 1024 data elements (number of pixels in sub-window) need to be read 1024 times (number of locations it moves about).

M_{40} , the surface fit algorithm, takes a significant number of clock cycles, but makes up for it by having a high clock frequency, 666.67MHz. Furthermore, even if the implementation is scaled up, M_{40} will be the only math-block which will not have an increased runtime.

Conclusion: Considering the running times of the individual math blocks, at first glance it seems that this algorithm will perform at sub 30ms. The precise value of this will be dependant on the final clock frequency, the influence of the utilised parallelism, the inter-block pipelining and the AXI-Lite bus transmission time. So far the benefit of a pipelined approach is clear from M_{22} , M_{23} , M_{27} ’s runtime, furthermore the number of clock cycles needed to perform M_{40} illustrates the non-triviality of performing complex matrix algebra on embedded systems.

6.6.1 Possible optimisations

Low-level embedded systems nearly always have some possible runtime improvement opportunities. Possible improvements are discussed here in order of most gain and least effort. This shows the benefit of designing an embedded system with a bottom up approach. Only the sub-window loader optimisation was implemented, since the remainder of the current implementation is fast enough.

The Sub-Window Loader is the module responsible for loading the ‘sub-window under the moving template’ into M_{32} (Section 6.3.3), it initially loaded 1-data-element every 4 clock cycles due to the nature of the control logic. Why is the read-out so inefficient? It is non-trivial to read from a BRAM (latency of 2 clock cycles), and check the output buffer’s state (if it is nearly full), and keep count of the current address. Although a more efficient implementation is possible, simplicity and determinability were opted for initially to get the math-block working. After determining that this module was by far the slowest in the system, an optimisation was implemented. A true dual port BRAM was used to store the window, this means that two simultaneous reads are possible from the same BRAM module. The same FSM control logic was replicated to read out every second data-element. This improved the read out to the current rate of 1-data-element every 2 clock cycles. This cycle may be repeated, if 1 extra $8\text{bit} \times M^2$ dual port BRAM is included to hold the window data, and two extra FSM control logic modules were replicated to read every fourth data-element, a throughput of 1-data-element per clock cycle would be achieved.

This was not implemented. This module can no be further optimised without having to create a parallel branch of $M_{32,33,34}$ with an extra set of accumulators after M_{34} . Consider the gains when implementing this parallel optimisation (throughput: 2 data per clk cycle): the expected clock cycles will reduce by a factor of 4, from the current 2 million to roughly 0.5 million. This number is still significantly higher than any of the other fabric math-blocks' runtime, and would be a worthwhile optimisation.

In a similar fashion as the Sub-Window Loader, the *Template* and *Window* data read-out also has a low-throughput. It runs at a throughput of 1 data-element every 3 clock cycles, as evidenced by its flow-diagram-like FSM in Figure 6.15. This is referred to as the *un-pipelined BRAM reading*. It was developed in this way so as to make reading from the BRAM as simple and reliable as possible. When it was developed, the following were still unknown, size of M and N , equivalent throughput of various subsequent math-blocks, and size of their input buffers. These three unknowns, informed the design to: check the output buffer if it is nearly full¹⁰ before writing to it, since the buffer would fill up at various rates due to any combination of the three aforementioned unknowns. Similar BRAM read-out FSM's are employed in most of the math-blocks which have to read from BRAM. Improving this read-out, even if only for *Template* and *Window*, will reduce the runtime of the various math-blocks. Note that this FSM was used to load the various math-blocks for the timing test in Section 6.6, so as to provide realistic results.

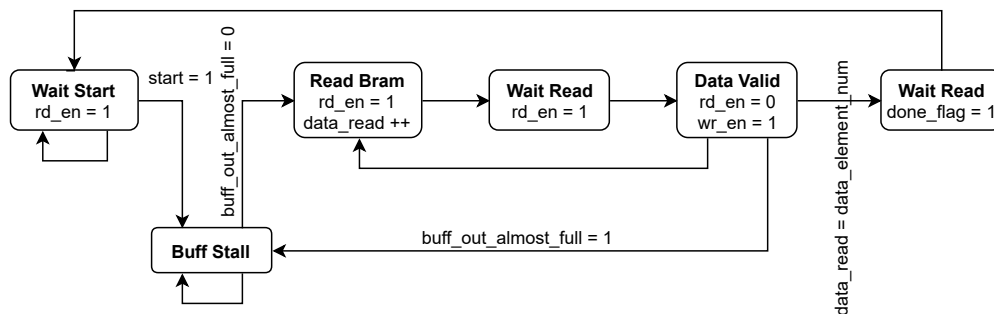


Figure 6.15: Typical BRAM reading FSM, throughput of 1/3.

De-constructing M_{40} , and implementing the various parts efficiently in fabric may further improve the speed. For the implementation to be successful in reducing runtime, the final fabric implementation will require less than 6 times the C-implementation's clock 1 million clock cycles since the CPU's clock is roughly 6 times faster than the fabric's clock. Furthermore, if a significant

¹⁰'Nearly full' instead of 'full' is checked, since responding to the full flag only happens 1 clock cycle after it is asserted.

reduction in accuracy is acceptable, implementing the Taylor series method in Section 3.1 will be significantly simpler and use less clock cycles.

All the math in the fabric could also be implemented using fix-point instead of floating point arithmetic. Initially this seemed like a good idea, but since the data is pipelined and the data-set sizes are relatively large, the gain from fixed-point would not be worth the effort and loss in accuracy. Consider as an example the Xilinx 64 point-FFT. In 32bit floating point, it has a latency of 360 clock cycles, whereas its fix-point counterpart has a latency of 275 clock cycles. If both math-blocks are perfectly pipelined with 4096 data points (data feeder = 1 data per clk cycle), this change in latency is insignificant, since the fixed point math-block will finish 85 clock cycles earlier than the floating point math-block, a mere 2.1% improvement. Therefore a large latency is insignificant if a large data-set is worked upon, when properly pipelined. Places where such a fix-point implementation may be fruitful are parts of the calculation where a significant number of repetitions are performed, as each clock-cycle saved is multiplied by the number of iterations. This is the case when a calculation (possibly pipelined) occurs, and may not accept data for the next calculation until it has completed its current calculation set. This is the case for the accumulator in $M_{33,34,35,36}$. Consider this accumulator, if a similar approach is taken as for $M_{17,18}$ (see Appendix C.8.2), the number of clock cycles saved overall is:

$$CLK_{\text{Gain}} = \#_{\text{Iterations}} \times \{CLK_{\text{float_accumulate}} - (CLK_{\text{float_to_fix}} + CLK_{\text{fix_to_float}} + CLK_{\text{fix_accumulate}})\} \quad (6.3)$$

a significant number when $\#_{\text{Iterations}}$ is as large as 1024 as in this scenario. This demonstrates the gain when alternating between float and fix-point arithmetic when accuracy is decoupled.

6.7 Increasing robustness

The major outcome of Section 5.3, was that the solution will be implemented for an FPGA. This is to allow a potential error-mitigation scheme to be implemented. This design does not implement such techniques, however a discussion is held on how the architecture *facilitates* such modifications. M. Berg's slides [59] provide a good overview into radiation mitigation techniques on an FPGA. An in-depth study of how the underlying hardware is effected by radiation is *not* the aim of this Section. As this project is mainly IP-based, and not a physical hardware layout, only SEUs and MBUs will be considered. The findings should then be generic to various FPGA vendors¹¹.

¹¹The assumption that the minimal functionality provided by Xilinx's BRAM for instance is similar across vendors.

Only the following scenarios are identified and solutions provided for: data corruption during storage and FSM entering an unwanted state. Data corruption during math calculation is not addressed¹².

As the streaming architecture on which the design is based does *not* make use of hand-shaking between data transfers, data can not be retransmitted once it is in the math pipeline. This implies, if data corruption occurs at any point in the calculation, the whole calculation needs to be restarted. Similarly if a FSM enters an unwanted state, a full fabric reset is required and the calculation restarted.

Three somewhat unique detailed solutions are discussed¹³, and then compared to classic TMR, each with a brief description on its predicted robustness, effect on throughput, and resource usage. The discussion only considers the fabric, for this discussion the *Zynq CPU* is considered to be a radiation tolerant. The proposals lean heavily on *FSM Mom*'s oversight and easy extension as mentioned in Section 6.2.2. For the remainder of this section, data in memories and FIFOs utilise SECDED for robustness.

6.7.1 Proposal 1

Figure 6.16 proposes the simplest to implement radiation tolerant system: if any of the memory blocks (BRAM or FIFO) encounter a *double_bit_error*, the whole system is reset. This reset is driven by the *FSM Mom*, the highest hierarchical FSM, ensuring all blocks start afresh. Furthermore, all FSM's states are synthesized to one-hot encoding so that each state can be inspected constantly. If more than one state bit is logic-high at any one time, or all state bits are logic-low, the FSM has encountered a SEU. Logic needs to be implemented in a similar fashion to toggle a reset from *FSM Mom* when *any* FSM state encounters a SEU. *FSM Mom* will need to be setup in a TMR fashion, this is to ensure that *FSM Mom* never produces an unwanted output. The TMR section can be reset every n time periods by *Zynq CPU*, to ensure all three copies of *FSM Mom* start from an identical state.¹⁴

Pros: comparatively low resource overhead

Cons: If any data or FSM in the whole system is corrupted, the *whole* calculation and system needs to be reset. For this reason, harsher radiation environments will result in lower overall throughput.

¹²Corruption during math calculation can be covered by TMR-based mitigation techniques.

¹³The author does not claim these are novel or commonly used methods, rather, that they are common sense approaches.

¹⁴A SECDED implementation will know if two bits are flipped, but more than two flipped bits will go undetected. Additionally, if the state register, which should only hold one bit, experiences a double bit flip, it could also go undetected, thus 'the system will know' means that the system will only be confident that no error has occurred.

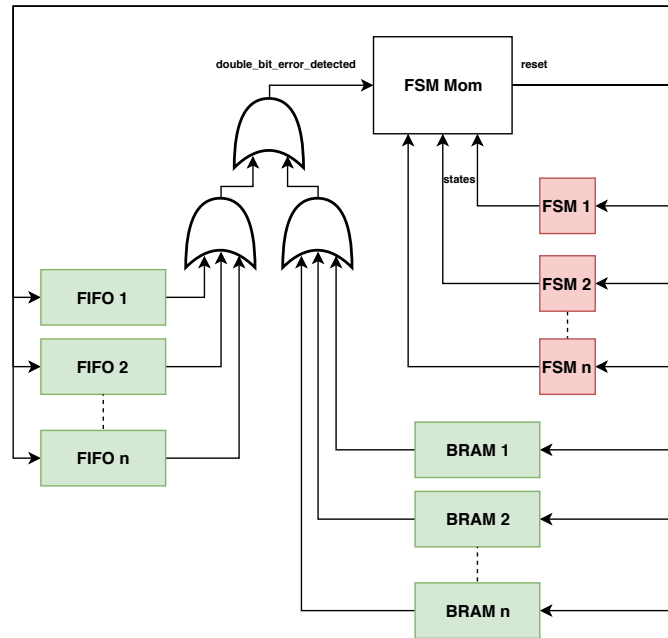


Figure 6.16: Architecture for Proposal 1, any double error detected in BRAM/-FIFO or any suspicious state from a FSM will trigger a reset via *FSM Mom*.

6.7.2 Proposal 2

Drawing on similar ideas from TMR, two sets of FIFOs/ BRAMs can be implemented side by side in lock-step. Logic is used to implement the following: a multiplexer which always chooses uncorrupted data. The logic will toggle a reset when both memories have undergone a double error detection. To further reduce probability of reset by both memories being influenced by one radiation event, place and route options can be utilised to geographically place the memories at different locations on the die. This double memory will be paired with the same FSM strategy as in Proposal 1.

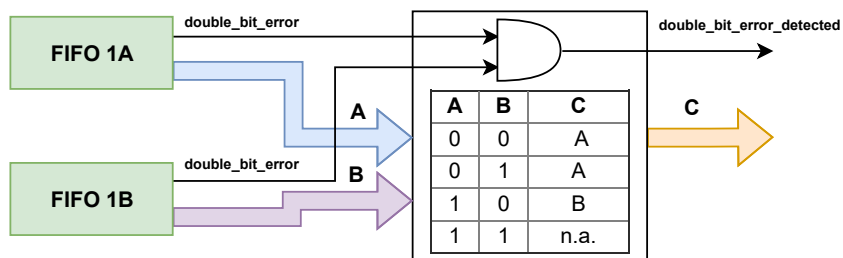


Figure 6.17: Memory logic for Proposal 2. The truth table refers to which data is outputted (C) for a certain set of double_bit_error signals. If both double_bit_error signals are asserted, a double_bit_error_detected will ping *FSM Mom* for a reset.

Pros: Less reset occurrences are expected than Proposal 1, due to better tolerance to double_bit_error scenarios (higher overall throughput due to less resets).

Cons: Twice as much memory is required, the maximum operating frequency may be reduced if the twin memories are placed far from one another, as this will increase routing delays.

6.7.3 Proposal 3

Consider the runtime of the various units in Table 6.3, clearly the total system runtime will be $< 30\text{ms}$. Assuming a runtime of $< 90\text{ms}$ ($3 \times 30\text{ms}$) is fast enough for the designer, a time-based TMR approach may be taken. The same data-set is calculated on the same hardware, three times, after which a majority vote is passed on the final three copies of M_{39} 's results. A watchdog timer will need to be implemented in case of the hardware hanging, as there is no indication given to *FSM Mom* when an error occurs.

Pros: Very little extra resources are required, only some high-level control logic along with voting logic on M_{39} 's three outputs.

Cons: A reduced time requirement for SPITE update rate. The system may provide consistently incorrect results if multiple MBUs are expected in a 90ms period.

Table 6.4 concludes the discussion and attempts to quantify the various aspects of each implementation. Although speed and resource usage is easily quantified, perceived reliability and implementation effort are not.

Table 6.4: A comparison of the various proposals, including classic TMR, ranked in descending order of perceived reliability. Effort is an estimate of how tedious it will be to implement for the designer on a linear scale of 1 (no effort) to 5 (maximum effort).

| Method | Mem | Logic | Speed | Effort |
|---------|-----------|---------------------|-----------|--------|
| TMR | $3\times$ | $3\times$ | $1\times$ | 2 |
| Prop. 2 | $2\times$ | $\approx 1.1\times$ | $1\times$ | 5 |
| Prop. 1 | $1\times$ | $\approx 1.1\times$ | $1\times$ | 4 |
| Prop. 3 | $1\times$ | $1\times$ | $3\times$ | 2 |

For Proposals 1 and 2, the hierarchical control approach coupled with the segregated control and data path reduces effort on the designers side to implement low-level radiation mitigation. Likewise the design choice in Section 6.3.1 pays off: entering reset state from *any* current state, enabling easily integrated reset functionality.

6.8 Discussion

Three major outcomes of the design are required in order for it to be considered suitable: accurate, resource aware and fast. These were all addressed in Sections 6.4, 6.5 and 6.6 respectively. From a first iteration, the various components are shown to meet these requirements. However, that does not mean they are optimal. The design is perceived to be accurate enough, use tolerable amount of resources (roughly one third of the MicroZed 7020 BRAMS, a low-to-mid range FPGA), and be sufficiently fast: a first estimate runtime of sub 30ms. The significance of these three outcomes are now *confirmed* for the *whole* system (Section 7), as this Chapter dealt with the segregated sub-parts: the math-blocks.

Chapter 7

Experimentation and Results

Tests are designed to determine the accuracy, runtime and power requirements for the final Poly 3×3 , $M = 64$, $N = 32$ SPITE implementation for the MicroZed 7020 platform. Throughout this section, comments will be made with regards to the Jetson TK1's performance in comparison to the measured results of the MicroZed 7020 implementation.

7.1 Accuracy

It is imperative to determine the accuracy of the HW solution so that the error introduced in the HW implementation may be quantified. This error was discussed in Section 6.4.

Aim: Compare the accuracy of the HW implementation with that of the SW simulation. Provide some confidence metric, to quantify the difference between the SW and HW results. The specific value to compare is the final answer, the SPITE result: $(x_{\text{shift}}, y_{\text{shift}})$

Method: To replicate the SW simulation on the HW, data-sets consisting of windows and shifted templates – referred to as the test-vector – were transmitted between a PC and the MicroZed 7020, thereafter the answer was transmitted back to the PC for logging and investigation.

A simple Universal Asynchronous Receiver-Transmitter (UART) link was established between the PC (via a Python script) and the MicroZed 7020 via its Zynq CPU. Conveniently, C hardware abstraction layers provided in its board definition files were used. Each data-set, is sent byte-wise to the Zynq CPU via the UART. Each byte is then sent to the Fabric to be stored in the relevant template and window BRAM via the memory mapped AXI-Lite bus. Once a data-set (one window and template) is transmitted, the calculation is initiated. When the calculation is complete, the estimated shift is sent back to the PC via UART. This process was repeated for the full number of data-sets in the test-vector.

The range test from Section 4.5 was replicated with this UART link, sending the full test-vector to the HW. Thus, four different images were tested, over the full operation range.

Results: Figure 7.1 indicates that the difference between the error of the HW and SW is small. Image B's results are specifically shown to illustrate that the HW follows the same trends as were observed in the SW simulation. It also follows the 'fit error' behaviour at true shift = 7 to 15 along the x-axis.

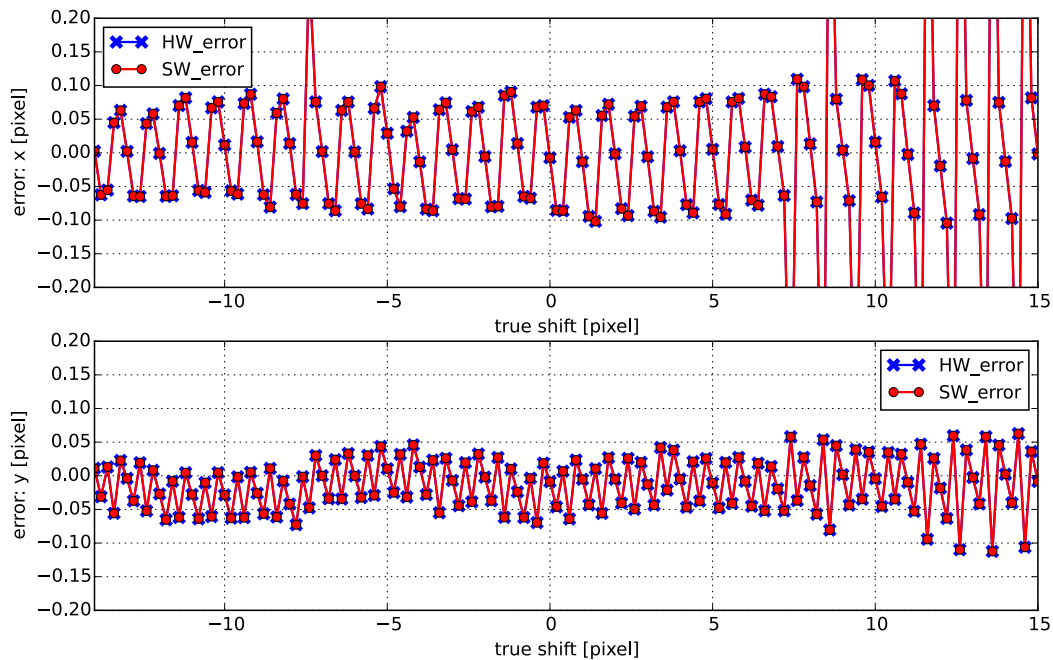


Figure 7.1: HW vs. SW error in both x and y axes for Image B.

Figure 7.2 shows the absolute distance error (x and y combined) in percentage of the maximum allowable pixel error as defined in Section 4.2: 0.1 pixel. The majority is below 0.024%, which provides an intuition as to how insignificant this error is in comparison to the maximum allowable error. It shows this consistency across all four images.

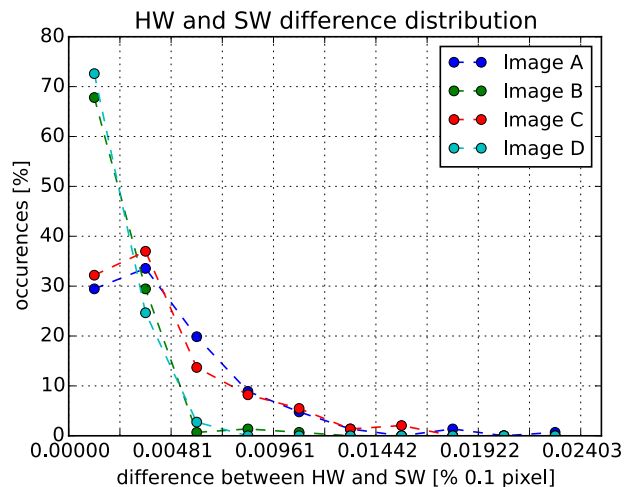


Figure 7.2: Distribuion of the absolute difference between HW and SW result in % of 0.1 pixel, the largest allowable error. Bins are aligned to vertical grid lines.

Conclusion: Since the HW error follows the same trends as the SW error, shown in Figure 7.1, whilst being negligibly small, it is safe to say that the HW implementation lines up well with the predicted SW implementation. This result validates the design, that no saturation occurred due to integer fix-point optimisations. Furthermore, clearly no significant error is introduced by using 32bit floats instead of 64bit doubles. Lastly, the 24bits used to represent the phase factor in Xilinx’s FFT core is enough resolution not to induce significant errors.

The following conclusions are drawn:

1. The HW is behaving as it should, providing consistent, accurate answers. The whole test setup tested 548 sets of images without hanging or delivering inaccurate results.
2. Since the error-trend follows the SW so closely for the range test, and since nothing else changes between subsequent tests in Section 4 besides input data, it may be argued that all of Section 4’s results will be similar to HW results. Based on this assumption, the SW can be used in aiding high-level optical mission analysis and planning. Furthermore the simple and fast UART interface to the HW enables this platform to be used to test various input images ¹.

Compared to the Jetson TK1 benchmark in Section 5.2, since it was also conducted in 32bit floats on the GPU, accuracy will be similar.

¹ The link runs at roughly 1 answer per image data-set every 1.2 seconds. If a significantly higher throughput is required, this link has scope for increased throughput through optimisations.

7.2 Runtime analysis and results

An important specification for time-critical embedded HW systems is runtime and resource usage. Both runtime and resource usage are investigated. In order to properly determine these, a final HW implementation run is required. This Section consists of three parts: *Final implementation: Frequency and resources*, *Runtime*, and a final *Runtime comparison*.

7.2.1 Final implementation: frequency and resources

The final implementation's maximum frequency directly influences the final runtime, and the resource usage describes how lean the final implementation is.

Aim: Determine the maximum operating frequency for which the HW design will work properly on the MicroZed 7020. Document final resource usage.

Method: All debug IP cores such as superfluous FIFO's and ILA's were removed from the design, allowing shorter routes between components, which improves maximum operating frequency. The MicroZed 7020 provides the following clocks via its Zynq CPU to the fabric: 100, 111.11, 125, 142.87, 166.67, 187.5, 200, 214.28 and 250MHz. Vivado's Implementation procedure tries to meet a certain timing constraint and does not directly inform the user what maximum clock frequency is possible. Therefore, multiple implementation runs with varying clock frequencies (increasing in frequency) were performed².

Each implementation run can be guided by various directives. The following strategies were utilised for each frequency attempt: Vivado's 'Default' and 'Performance Explore' strategies, along with power conscious variations of these strategies 'Default-Power' and 'Performance Explore-Power'. The latter two enable power optimisation by clock gating which reduces unnecessary switching activity³.

Results: First order power estimates and Worst Negative Slack (WNS), a value which needs to be positive to pass timing, is shown in Table 7.1 for all four implementation runs with frequency 166.67MHz. According to these runs, the maximum frequency that is attainable on the fabric, using 'Default power' implementation options and the Zynq CPU clock, is 166.67MHz. Note how this final frequency is higher than the slowest predicted frequency given

²Care should be taken when using such an approach, some IP cores are given a target frequency when instantiated, such as Xilinx's FFT IP core, this will mean that unless these are updated as well, faster timing will fail.

³A full study and knowledge of implementation options are out of scope for this project. FPGA Engineers often make use of a set of .tcl scripts and multiple variations of directives, to determine the most optimal high frequency to power ratio. Unfortunately a small gain in frequency increase may not necessarily benefit the fabric, as it is limited to the various frequencies provided by the Zynq CPU. Unless a separate clock circuitry is to be included, such an exercise may be superfluous if small frequency gains are made.

post-synthesis in Section 6.6, this shows that the implementation optimisations, done by the software suite, still play an important role. Furthermore, they also show that choosing accurate directives is not straight forward. Resources tabulated in Table 7.2 use ‘Default power’s implementation results.

Table 7.1: Timing characteristics WNS (negative is timing failure), and first order power analysis of the four proposed implementation runs.

| Run | WNS [ns] | Power [W] |
|---------------------|----------|-----------|
| Default | -0.069 | 0.892 |
| Default power | 0.095 | 0.890 |
| Perf. Explore | -0.064 | 0.893 |
| Perf. Explore power | -0.016 | 0.890 |

Table 7.2: The resource usage of the final design using ‘Default power’s implementation run, with the utilisation percentage of the MicroZed 7020.

| Resource | Number | Microzed [%] |
|-----------|--------|--------------|
| BRAM 36Kb | 48.5 | 34.6 |
| DSP48 | 57 | 25.9 |
| LUT | 18473 | 34.7 |
| LUTRAM | 1675 | 9.6 |
| Flip-Flop | 25160 | 23.6 |

Conclusion: 166.67MHz is the fastest that this solution will run on the MicroZed7020, a reasonable overall first order power ($< 0.9W$) is documented from the design tools, this will be investigated further in Section 7.3.

Furthermore, the resources required for the MicroZed 7020 SPITE implementation, are less than 35% of those available. This allows significant room for the implementation of additional algorithms that may use SPITE as a preprocessing step. Algorithms such as image averaging [8] to boost SNR or super-resolution to boost detail are just two examples. Furthermore, the lean SPITE implementation allows designers to port and place the IP onto complex, pre-existing, compatible imaging systems.

7.2.2 Runtime analysis

The runtime of both the fabric implemented design, and the Zynq CPU’s C-application, is determined.

Aim: Determine the runtime of one full SPITE operation. Tabulate the results in terms of both clock cycles and time, so as to model both behaviour of design,

and provide an intuitive value for runtime. Measure the fabric runtime and the C-application runtime separately, since these operate at different frequencies.

Method for fabric: All FSM in the project output a unique status comprised of 8bits for every state. It was a trivial exercise to route the status bus to a counter which starts counting when it receives a certain status, and ends counting after a certain status. The counter runs at the same clock frequency as the rest of the fabric, and will calculate an exact number of clock cycles. This final count value can then easily be read via JTAG via an ILA in Vivado’s HW manager mode. Counting starts when FSM Mom commands the various math units to start (in specific FSM Grey), counting stops when the last data packet from M_39 is ready to be read via the AXI-Lite bus. This incorporates the complete fabric calculation, but assumes that the window and template data is pre-loaded into the BRAM.

Method for Zynq CPU: The full C-application runtime is determined, which consists of the AXI-Lite bus transaction to read M_39’s 3×3 data as well as curve fit M_{40} . Specific details for the timing methodology are shown in Appendix C.7.

Results: The full fabric solution comprises of 2 222 928 clock cycles, this is a mere 27 472 clock cycles away from the reported $2144 \times N^2$ clock cycles of M_{32} , $M_{33.34.35.36}$. The results highlight two characteristics of the implementation. The sum of all math-block runtimes – 3.4×10^6 from Table 6.3 – is significantly larger than the final running time. This highlights the performance gain of implementing both pipelined and somewhat parallel data control, where parts of the red branch’s calculation occurs simultaneously to the Blue branch, essentially *masking* the runtime of the faster math-blocks. Secondly, since M_{32} , $M_{33.34.35.36}$ are by far the longest calculation, they slow down the full calculation to roughly its runtime. Consequently, reducing the runtime of M_{32} , $M_{33.34.35.36}$ is *crucial* for the speed up of the full calculation. This occurs when the sub-window loader’s throughput is increased (Section 6.6.1 and 6.3.3).

The C-application, despite containing many clock cycles, completes rapidly due to its fast clock. Furthermore the AXI-lite bus transaction time is essentially insignificant since the 1.54ms is equivalent to M_{40} ’s solo runtime (Table 6.3).

Table 7.3: Full system runtime: average runtime of the Zynq CPU application, along with the precise fabric runtime.

| Part | Clock Cycles | Operating Freq. [MHz] | Time [ms] |
|--------------------|--------------|-----------------------|-----------|
| Fabric | 2 222 928 | 166.67 | 13.34 |
| Zynq CPU | 1 032 214 | 666.67 | 1.54 |
| Full System | | | 14.88 |

Conclusion: The SPITE implementation on this platform is fast. Consider a standard video rate of 24 frames per second, with an update rate of $\frac{1}{14.88 \times 10^{-3}} = 67.2$ Hz, the system could comfortably perform SPITE on subsequent frames of a standard video feed.

The fast execution time can have various other benefits. If the power budget allows it, multiple instances may run in series on different parts of the same image data-set, due to its short runtime. These answers may then be averaged in order to increase confidence of the result. Furthermore if enough instances are run, a simple threshold detection algorithm may be implemented, one method of possibly minimising the fitting error seen in the Python simulation in Section 4.5.

7.2.3 Runtime comparison vs MCU based system

In order to determine whether the choice to develop a SPITE accelerator on an FPGA platform is justifiable, a comparison with a standard platform is performed.

Aim: Compare the runtime of the FPGA implementation with the runtime of a single core, microcontroller-only implementation.

Method: The Zynq CPU will be used to run the equivalent platform: a C-application. This microcontroller is chosen since a suitable substitute for an FPGA/GPU accelerator on a satellite will be its OBC. The Zynq CPU is a reasonable choice, since recent CubeSat OBCs often have ARM-based architecture, with additional hardware based Floating Point Units (FPUs). See Appendix C.7 for precise C-timing and setup methodology.

Four specific math-blocks were replicated in C to represent the wide range of functionality expressed on the FPGA. M_{28B} , a *data-intensive* operation, along with three *calculation-intensive* operations: M_{32} with $M_{33.34.35.36}$, $M_{24.25.26}$ and $M_{22.23.27}$. In order to keep the comparison fair, 32 instead of 64bit floating point operations were implemented.

The C-equivalent for M_{28B} is self evident from Figure 6.10, the various pixel positions are shifted, shown in Appendix C.6.1.

$M_{24.25.26}$'s C-equivalent in Appendix C.6.2 was implemented using a M^2 long, 1D-array, thus only one for loop was required. It also implements the complex conjugate.

The C-equivalent for M_{32} , $M_{33.34.35.36}$ in Appendix C.6.3, shows that a 'fake_mean' was used. This modelled the same sum-area-table optimisation which the FPGA implementation makes use of. Only one iteration of 'under-the-template-standard_deviation' was calculated, multiply by N^2 to calculate the total runtime.

The FFT equivalent C-code [60] in Appendix C.6.4 was modified to use ‘float’ instead of ‘double’. One 64-point FFT was determined, since a full 2D-FFT consists of 64 row-wise FFT’s and 64 column-wise FFT’s, the final answer was multiplied by 128 (i.e. $2M$), to determine the 2D-FFT’s runtime.

Results: Consider the C-equivalent runtimes in Table 7.4.

Table 7.4: Runtime comparison: FPGA vs. C-equivalent math-blocks. Gain is defined as $\frac{\text{C-runtime}}{\text{FPGA-runtime}}$

| Math-block | CPU C: 666,67MHz | | FPGA: 166,67MHz | | Gain |
|------------------------------|-------------------|-------|-------------------|-------|-------|
| | clk | t[ms] | clk | t[ms] | |
| M_{28B} | 6858 | 0.01 | 16397 | 0.1 | 0.083 |
| $M_{24_25_26}$ | 90633 | 0.14 | 45087 | 0.27 | 0.52 |
| $M_{32}, M_{33_34_35_36}$ | $7254 \times N^2$ | 11.14 | $2144 \times N^2$ | 13.1 | 0.85 |
| $M_{22_23_27}$ | $17590 \times 2M$ | 3.38 | 17515 | 0.11 | 30.73 |

M_{28B} runs roughly $10\times$ faster on the CPU. This can be attributed to un-pipelined BRAM reading (discussed in Section 6.6.1).

$M_{24_25_26}$, which has the second longest runtime on the FPGA, runs $1.92\times$ faster on the CPU. This slow FPGA comparison can also be attributed to the inefficient BRAM reading.

$M_{32}, M_{33_34_35_36}$, the longest path of the FPGA calculation, is also shown to be $1.17\times$ faster on the CPU. This could be attributed to the nature of the calculation, where each data-element being read out, has the same operations happening: subtract the mean, multiply with itself and add to a sum, these operations don’t challenge the the micro-controller’s ability, since minimal registers are required to fulfil the operation.

The last math-intensive operation, the 2D-FFT, modelled by $2M$ 64-point C-equivalent 1D-FFTs, is $30.73\times$ faster on the FPGA. Clearly the FPGA is more proficient at heavy mathematical operations which have parallelism, than the CPU, this is to be expected. The result for the 2D-FFT is significant since it occurs three times – two 2D-FFTs, one 2D-IFFT – therefore an expected total running time of only the 2D-FFT/IFFT is 10.14ms.

Comparing equivalent math-blocks between different platforms may be counter-intuitive. Even though the CPU is somewhat faster in some instances, all these calculations have to happen in *serial*. Thus all the runtimes need to be added to determine the CPU’s final runtime. The FPGA’s math-blocks are all pipelined and streamed, this means even when math-block A is not completely finished yet, the next math-block in the pipeline – math-block B – may already start with completed data elements of A. The most illustrative example, is the output of $M_{22_23_27}$ connected to $M_{24_25_26}$ ’s input. The completed data-elements are already ordered into $M_{24_25_26}$ ’s BRAM while $M_{22_23_27}$ is still busy pushing out data-elements, resulting in overlapping runtime. Such streaming

operation is not really possible on a CPU, whereas FPGA and GPU coding style's are such that the designer may exploit pipelined operation. One can not simply determine individual runtimes and accumulate them to determine the FPGA's final runtime. A *full* FPGA implementation will need to be assessed to determine representative results.

Conclusion: The following conclusions may be drawn from this comparison:

1. When undertaking such a project, the possibility of using an ARM microcontroller with FPU availability as a valid platform, should not be dismissed. The development time is significantly less: the trickiest code, the FFT took a few hours to integrate onto the micro-controller due to its availability online. Doing the same on HW, even though the Xilinx provides an FFT IP-core, would take a few days to a week to properly plan data-flow and implement, for a novice embedded engineer.
2. The runtime of the CPU platform will be roughly

$$\begin{aligned} \text{C-Runtime} &= t_{M_{32}, M_{33.34.35.36}} + 3 \times t_{M_{22.23.27}} \\ &\quad + t_{M_{24.25.26}} + t_{\text{Poly } 3 \times 3} + t_{\text{auxiliary}} \\ &\geq 21.42\text{ms} \end{aligned} \quad (7.1)$$

which results in a 46.7Hz update rate, comfortably meeting the 24fps video update rate. Clearly the CPU is *fast enough*⁴, the question then becomes: can the OBC *accommodate* such an implementation. It is therefore not a question of final update speed, but rather what resources are available on the satellite, what requirements are made of the OBC, along with what development capacity is available.

3. Clearly the FPGA solution trumps the C-solution, where the FPGA takes: 14.38ms vs 21.42ms, which is 1.5× times faster. An experienced C-programmer will be able to make significant improvements on the C-code – rewriting it to accommodate the NEON floating point co-processor⁵. Similarly, at least one obvious, easy to implement optimisation can be performed on the FPGA implementation – discussed in Section 6.6.1 – which can reduce runtime by factor of 4.

Consider how the Jetson TK1 benchmark in Table 5.2 compares with the MicroZed's FPGA based $M_{22.23.27}$: 0.52ms for the CUDA-HF in comparison with the 0.11ms of the MicroZed 7020 implementation, roughly factor five times

⁴Significant optimisations may still be made, utilisation of the NEON core amongst others.

⁵Optimal coding for standard C v.s. NEON optimised C is different [61], only standard C was investigated, using the FPU as a hardware floating point accelerator, but not utilising NEON's SIMD ability.

faster. Note that this 2D-FFT implementation also surpasses the theoretical 2D-FFT runtime from Section 5.1.2, even though it does not account for memory interfacing. This shows the benefit of a well pipelined calculation.

7.3 Power

An embedded system or sub-module requires a good estimate for energy consumed per calculation as this is an important metric when integrating with other modules, specifically where only limited energy in the form of a battery is available.

There are multiple ways in which power consumption can be calculated:

1. Most development boards provide a sense resistor, where power observance is its intent: if the voltage across the known resistor can be determined, the instantaneous power is $P = V^2/R$. Although the Zedboard, a development board provided by AVNET⁶, has a sense resistor, the MicroZed 7020 (which has the same Zynq chip present) unfortunately does not. This makes such an approach infeasible⁷ with the MicroZed 7020. See the technical manual [62] for more details regarding sensing circuit fundamentals.
2. In a similar style, an ammeter may be placed in series with the power circuit. This will then measure the current flowing to the regulators.
3. Complex embedded systems often have intelligent power-regulators which provide power to various parts of the system. These have registers which contain current / power / voltage information. See PMbus on Zedboard, which makes use of Texas Instruments' power regulators which can communicate their values via I2C bus, similar to the Jetson TK1 current polling (Section 5.2.2). If such regulators were present, per module results could be obtained, unfortunately this is not the case.
4. For the fabric part of the design, power can be determined via simulation. Vivado provides two power estimator options, *Vector-based* and *vector-less*. The vector-based approach is more accurate, it requires the user to provide the simulator with expected operation. This is generated by using a test-bench which defines typical operation, this in turn generates a .saif⁸ file. This file contains the cumulative switches which various signals undergo, knowing how much a specific resource is used is directly related to accurate power consumption modelling.

⁶The same company which sells the MicroZed 7020 board.

⁷To be precise, the MicroZed 7020 board has a resistor 'R50', if one uses the barrel DC-Jack power supply, one could attempt to use this resistor. The only problem is it has an incredibly small form factor, which means its initial intent is not current sensing.

⁸Switching Activity Interchange Format

Aim: Determine the power rating for the typical SPITE operation on the MicroZed 7020 platform. This will try to characterise the *algorithm's* power and not the *platform's*. This distinction is nuanced yet important, although the algorithm is *on* the platform, this allows a comparison with different implementations on the same hardware. Furthermore, the design is such that it can easily be ported to different Xilinx FPGAs of the same generation, providing a good estimate of it's performance if it were to be ported to a different FPGA.

Method: A simple ammeter test (method 2) is devised to determine the total current flowing to the power-regulators. Although this is not the most accurate, the tests are designed to determine a reference power due to regulators and peripherals, C-application power, fabric power, and lastly the SPITE operations power. Determining these is achieved through the following tests.

Test 1a: Create a project where only the Zynq CPU is instantiated, thus no fabric logic. Run a C-application, in an infinite while loop of no-operations (NOPs⁹). Measure the current drawn by placing an ammeter (FLUKE 15B+ digital multimeter) in series with the power supply. This will be considered the 'reference-power', the power being drawn by the platform while performing nothing substantial. Note that this power does not need to be low¹⁰, but rather determined for subsequent tests.

Test 1b: Use the same project as Test 1a, instead of running NOPs in an infinite while loop, determine the power drawn by running the polynomial least squares algorithm (M_{40}) in an infinite loop. This *change* in power is the value which characterises the power needed for M_{40} to run.

Test 2a: Create a project where the Zynq CPU is instantiated along with the full SPITE algorithm. Put and leave the fabric in reset mode, and then run the C-NOP application again. This will determine more or less the power used by the FPGA when it is not in use.

Test 2b: Use the same project, this time run the C-application such that it keeps the fabric busy by constantly restarting the fabric calculation after it has completed. This mimics the idea of putting the fabric in an infinite loop. Besides restarting the fabric calculation, the C-application idles. This change in power between Test 2a and Test 2b is the power required for the fabric part of the calculation.

⁹An instruction which achieves nothing except use up a one assembly instruction, such as the following inline assembly: `asm("mov R0, R0");`

¹⁰If a proper power-savy design was required for the C-platform, a proper throughput vs power trade-off would need to take place for the C-platform. Questions such as 'do we need cache', what frequency do we require for Zynq CPU etc. will then need to be determined. This would require a hard value for SPITE throughput. Such a design is out of scope, as this is only a first order prototype, where the power used by the fabric-implementation is important.

Test 3: Lastly, use the same project as in Test 2. In order to determine an *average* power, the standard C-application (M_{40} and control) along with the standard fabric calculation is run in ‘normal’ operating mode. This will give a good indication of what the system draws on average.

Results: Table 7.5 has the results of these various tests. It uses Test 1a’s result, which deviates a maximum of 2.6% from the stated 2.708W, is the reference power (0W). Thus we determine the *algorithms* impact on the power, not the underlying platform.

Table 7.5: MicroZed power measurements for various tests. Results show measured (ammeter) total power, the derived power from the total power, and the method by which it is derived. The total power values are averages over five runs.

| Test | Total power [W] | Method | Derived power [mW] |
|------|--------------------|------------------------------|-----------------------|
| 1a | 2.708 | $P_{C_M40_Idle} = P_{ref}$ | 2708 |
| 1b | 2.834 | $P_{C_M40_busy} = 1b - 1a$ | 126 |
| 2a | 3.076 | $P_{fabric_Idle} = 2a - 1a$ | 368 |
| 2b | 3.216 | $P_{fabric_busy} = 2b - 1a$ | 508 |
| 3 | 3.207 | $P_{Alg_avg} = 3 - 1a$ | 499 |

These values are now used to consider the theoretical average using Eq. (7.2), where ‘busy’ subscripts refer to the fabric implementation only and C-application only results from Test 2b and Test 1b respectively, the power while the fabric and C-application are actively contributing to answers. ‘Idle’ refers to Test 1a and Test 2a results for CPU and fabric.

Likewise, ‘ T_{busy} ’ refers to the time during which the algorithm is busy – on the fabric t in range $[0, 13.34]$ ms, and on the Zynq CPU t in range $(13.34, 14.88]$ ms. ‘ T_{idle} ’ refers to the time when the algorithm is not busy – on the Zynq CPU t in range $[0, 13.34]$ ms, and on the fabric t in range $(13.34, 14.88]$ ms. See Table 7.3 for these times.

Eq. (7.2) calculates the theoretical average algorithm power: 506mW, this is close to the tested average algorithm power in Test 3: 499mW. Therefore, the small error proves the consistency *within* the physical test¹¹.

¹¹Naturally this presupposes that the timing results from Table 7.3 are accurate, we have a high level of confidence in these results, since a rigorous timing methodology was followed.

$$\begin{aligned}
P_{\text{Alg_avg_theory}} &= \{(P_{\text{fabric_busy}} \times T_{\text{fabric_busy}}) + (P_{\text{fabric_Idle}} \times T_{\text{fabric_Idle}}) \\
&\quad + (P_{C_M40_busy} \times T_{C_M40_busy}) + (P_{C_M40_Idle} \times T_{C_M40_Idle})\} / T_{\text{total}} \\
&= \{(0.508 \times 0.01334) + (0.368 \times 0.00154) \\
&\quad + (0.126 \times 0.00154) + (0 \times 0.01334)\} / 14.88 \\
&= 506\text{mW}
\end{aligned} \tag{7.2}$$

Therefore, according to Eq. (7.2), one full SPITE calculations energy is:

$$\begin{aligned}
E_{\text{algorithm}} &= P_{\text{Alg_avg_theory}} \times T_{\text{SPITE}} \\
&= 0.506 \times 0.01488 \\
&= 7,54\text{mJ}
\end{aligned} \tag{7.3}$$

and the platforms Energy, taking into account the reference power is

$$\begin{aligned}
E_{\text{platform}} &= E_{\text{algorithm}} + P_{\text{ref}} \times T_{\text{SPITE}} \\
&= 7.54\text{mJ} + 2.708 \times 0.01488 \\
&= 47.8\text{mJ}
\end{aligned} \tag{7.4}$$

Conclusion: Energy per calculation is what defines the power efficiency of an implementation on a specific platform. In order to put the energy consumption of the algorithm in context, a theoretical case study is discussed in Appendix C.9.

Despite the ability to implement frequency scaling (along with a plethora of power saving techniques available on the MicroZed 7020), it was not implemented. The first order design goal was to see how *fast* SPITE can be performed, at a reasonable power. Since the power usage is reasonable, implementing frequency scaling is not required.

Consider the Jetson TK1's *algorithm* power consumption for the 64×64 2D-FFT (see Table 5.2). If we extrapolate similar power requirements for a full SPITE operation, both the CUDA-HF and C-serial-HF calculation, which will run at 0.42 and 0.35W, will outperform the MicroZed's *algorithm* power requirement of 0.506W.

Now consider the best 2D-FFT energy per calculation for the Jetson TK1 *platform*, CUDA-HF. Using the 5.17W offset required for platform evaluation: $5.17\text{W} \times 0.52\text{ms} + 0.21\text{mJ} = 2.9\text{mJ}/\text{calc.}$ The MicroZed 7020's energy per 2D-FFT (assuming that the platform power is constant across the calculation), $3.207\text{W} \times 0.11\text{ms} = 0.35\text{mJ}/\text{calc.}$ Therefore, the MicroZed 7020 platform is 8 times more energy efficient than the Jetson TK1 for this 2D-FFT calculation, illustrating the benefit of a custom FPGA implementation.

Finally, consider the 2D-FFT energy per calculation for the *algorithm*, the Jetson TK1's 0.21mJ vs. the MicroZed 7020's $0.506\text{W} \times 0.11\text{ms} = 0.056\text{mJ}$. The MicroZed's algorithm implementation is 3.75 times more power efficient than the Jetson TK1's.

7.4 Summary

In this chapter it is proven that the implemented SPITE algorithm on the MicroZed 7020 performs desirably. The accuracy is essentially identical to the 64bit python implementation. It performs at a rapid rate of 67.2Hz, and stays within a reasonable power, 3.2W platform and 0.5W algorithm, yielding an energy per SPITE of 47.8mJ and 7.54mJ respectively. These results were compared with results from the Jetson TK1 benchmark from Section 5.2. The 64×64 2D-FFT was shown to be 10 times more energy efficient on the custom MicroZed 7020 implementation than it's Jetson TK1 counterpart. The TK1 chip is used in various mobile, efficient consumer electronics [54], such as tablets and chromebooks, since these have similar low-power requirements as CubeSats. By extension, the MicroZed 7020 is a suitable candidate for use as a CubeSat prototype. Thus the scope's requirements are fulfilled.

Chapter 8

Conclusions and Recommendations

8.1 Overview

SPITE was defined as Sub-Pixel Image Translation Estimation, an algorithm that has to determine the rigid translation between two subsequent images to sub-pixel accuracy. After the problem background was laid out, an in depth investigation was conducted via means of Python software simulations. These simulations used realistic, generated data-sets to determine: a suitable algorithm (NCC with Poly 3x3), the smallest possible size of the required window (64×64) and template (32×32), the type and magnitude of expected error ($E_{periodic}$: error in range $[0.1, 0.15]$ pixel, and E_{fit} : error > 0.15 pixel). The required accuracy of < 0.1 pixel error was shown to be attainable 79% of the time in a ‘real-life’ simulated environment with a worst case mean of 0.084 pixel error in the simultaneous presence noise (SNR= 20) and rotations ($\theta = 0.5^\circ$). The largest absolute error was found in the Monte Carlo simulation, 1.59 pixels.

Thereafter, a hardware investigation was conducted between a Jetson TK1 SoC, and a MicroZed 7020 FPGA SoC, to determine the most suitable candidate to perform SPITE for an embedded nanosatellite platform. The FPGA based SoC was chosen due to its inherent ability¹ to mitigate radiation via orthodox means. A scalable, portable, modular, fast, reliable and autonomous SPITE implementation was fully developed for the MicroZed 7020. Tests were conducted to determine the performance: accuracy is comparable and practically equivalent to the software simulation ($\leq 0.024\%$ of allowable 0.1% pixel error), runtime (14.88ms) and operational power (algorithm 506mW, platform 3.207W). Tests showed that for a 2D-FFT calculation – a dominant calculation of the algorithm – the Microzed 7020 implementation is 30.73 times faster than a typical OBC counterpart, 3.75 times more efficient than the Jetson

¹The ability to reduce radiation effects by including redundancy in the circuit.

TK1's algorithm energy/calc., and 10 times more efficient than the Jetson TK1's platform energy/calc.

The project shows that for the typical operating scenario of a satellite which does not apply motion compensation during capture of successive images, an FPGA prototype solution was developed, with which the inter-frame translation may be estimated to sub-pixel accuracy on-board. This lays the foundation for various on-board image processing techniques which will reduce the downlink bottleneck.

8.1.1 Contributions

The various major and minor contributions are:

Major contributions

1. Python software simulations which model the problem, and characterise SPITE behaviour over a wide range of inputs. This may be altered for further investigations.
2. A SPITE prototype solution on a Xilinx FPGA+CPU capable hardware platform was developed, tested and characterised. Results closely follow those of the software simulations.

Minor contributions

1. A high-level Python benchmark script (Figure 5.5) which determines runtime and instantaneous power of the Jetson TK1 while running a certain application. It functions as a wrapper, thus it is easily customisable and extendible.
2. A UART link between the MicroZed 7020's SPITE implementation and a Python script running on a PC (Section 7.1). It functions as a convenient means of porting test-vectors to the FPGA, and retrieving the SPITE result, thus a true hardware result is obtained. This is significant for testing and validation after radical optimisations, e.g. changing the FFT core to use 16bit instead of 32bit floats.
3. A simulation showed the requirements, and ability of a potential end-point application which increases SNR – image stacking. This coupled with SPITE accuracy metrics *within* certain noisy settings, makes a convincing case for SPITE coupled to image stacking.

8.1.2 Limitations

In hindsight, the following limitations, shortcomings and critiques are enumerated for both the Python simulation, and MicroZed 7020 hardware approaches.

Software limitations

1. Although four sub-images which contain various levels of detail were used from the Stellenbosch image set [1], tests could have made use of a larger, more diverse data-set to better evaluate the SPITE algorithm. The issue of using real-life satellite data is that the precise inter-frame sub-pixel shift is unknown, thus it was easier to generate our own shifts from a higher resolution image.
2. Only two sub-pixel methods were investigated, thus we were ‘forced’ to implement the best method (Poly 3x3). This method happens to be the most computationally expensive of the two options. If more methods were investigated, there may be a method which yields similar, or better results, which is less computationally demanding.

Hardware limitations

1. The desire for a generic, modular, easy to alter design methodology, although easier to debug individual sections, will always take longer to develop, and result in a bulkier implementation.
2. The BRAM-reader block which was created, which is used in $M_{29.30.31}$, M_{16} , $M_{22.23.27}$, $M_{24.25.26}$, and M_{28B} , has the same design flaw as the sub-window loader module. Optimising this will reduce runtime for those blocks, the influence on the whole system’s runtime will depend to what degree the blue branch’s parallelism is exploited – the current critical path of the design.

8.2 Future work

Besides addressing the limitations, the following additional work-items may still be implemented on the FPGA hardware solution, and in the Python software investigation.

8.2.1 Hardware

The FPGA solution adheres closely to the the design driver’s requirements (Section 6.1). However, in order for it to progress from a prototype to a single IP-block which may easily be used as a standalone system, the least squares algorithm (M_{40}) should be implemented on the fabric. Two available choices exist: either implement M_{40} from ground up as was done with the other various math-blocks, or use Vivado-HLS to convert the current C-code to the appropriate RTL. The latter is the simpler, faster to implement option.

With regards to *optimising* the current VHDL solution, Sections 6.6.1 and 6.5.1 discuss appropriate runtime and resource reductions. Ideally *only runtime*

optimisations are required, since Section 4 and Table 7.2 argue that the scale is adequate with $N = 32$, $M = 64$, and that the size of the implementation is small enough. The proposed runtime optimisations look promising, as a potential reduction in runtime of at least $4\times$ is expected².

A proper C-application which makes proper use of the NEON SIMD co-processor shows promise to run the proposed SPITE sub 30ms. The various C-functions will need to be re-worked to enable maximum auto-vectorisation instead of the current OpenMP parallelism, Harnisch [61] mentions how conventional and auto-‘vectorisable’ code often differs significantly.

Furthermore, to progress this from a prototype to a solution better suited for the space-environment, redundancy may be implemented. This may take on various forms, some options are discussed in Section 6.7.

Lastly, from a full-system perspective, a study on the best method to minimise or gate the unwanted fit error as observed in Section 4.6, will be beneficial for reducing the maximum SPITE error. Options such as re-running SPITE on various parts of the same image set, after which results are gated and averaged, seems like a reasonable approach to mitigate the susceptibility to fit error.

8.2.2 Software

A proper study on a simple, and suitable way to determine where in the full image significant information lies, should be conducted. This will then seed *where* in the image the SPITE translation should take place, i.e. an algorithm which chooses an appropriate window from the full image. This is not as simple as it seems, as large amounts of data will need to be processed³. This makes a simple standalone FPGA IP-core solution not viable unless it is able to access the same image multiple times, due to limited BRAM.

Grey scale images were used in this project, essentially a single band intensity. Further work could be done in investigating the effect of applying SPITE on multiple bands, and combining the results.

Further simulations could better quantify the expected gain from image stacking when utilising the current SPITE accuracy. Realistically modelling crucial contributors such as inter-frame rotation and translations, will aid the investigation into image stacking’s utility. A final simulation, which uses real satellite data and determines the inter-frame translation with SPITE, realigns them, and applies image stacking, is one method to convincingly illustrate SPITE’s ability on real data.

The polynomial least squares regression may be replaced with bi-cubic interpolation, and may be investigated to determine sub-pixel results, as Debella-Gilo and Käab [39] report promising results therewith. More research

²This estimation ignores the impact of converting M_{40} from C-application to fabric.

³Either the full image will be operated on, or a sub-sampled version thereof.

could try to model the correlation surfaces maximum, the obvious separation between NCC and the sub-pixel method accommodates swift integration with the FPGA solution.

A pointed investigation into the effect of imager non-uniformities, and the influence of image distortions (due to pitch and roll) on SPITE accuracy is warranted.

Lastly, an investigation into pre-filtering/pre-processing, and its effects on SPITE accuracy, may uncover further improvements for the implementation.

8.3 Conclusion

This project successfully demonstrated a modular IP-core-like implementation which can perform accurate sub-pixel image registration of translated image frames – even in the presence of image noise and slight image rotations. The implementation was suitably fast, features low power consumption and uses moderate FPGA resources. This makes our algorithm a suitable candidate for on-board image processing applications on CubeSats and other small space platforms.

Appendices

Appendix A

Section 1, 2

A.1 Super-resolution

Super-resolution, in specific Super-Resolution Image Reconstruction (SRIR) is the process whereby a set of undersampled consecutive images are combined (resampled) in order to remove the effect of aliasing. Aliasing occurs when the spatial sampling distance ¹ violates the Nyquist rate, the result of this is that high frequency information is muddled with low frequency information. The distinction between SRIR and Super-Resolution Image Restoration is made. The latter tries to infer information beyond the diffraction limit of the optics.

The following steps are required for SRIR [10]:

1. *Acquisition*: In order for SRIR to be successful, the set of images need to have similar scene content, where the difference between two subsequent frames is a sub-pixel shift². Acquisition may be as simple as retrieving images from a moving platform which will have some random jitter, this guarantees a sub-pixel shift. Alternatively sub-pixel shifts can be artificially induced by moving the optical platform. If the precise artificial shifts are known, step 2 may be skipped. Secondly, it is important that observation models accurately describe the imaging system along with its degradations [63]. Lastly, the relationship between output resolution O_{Res} and number of images in set $I_{\#}$ each with a resolution of I_{Res} is $O_{\text{Res}} = I_{\text{Res}}\sqrt{I_{\#}}$ ³. The upper bound of effective gain of image quality is limited to the bandwidth of the optics.
2. *Motion Estimation*⁴: Both integer and sub-pixel shifts between subsequent images in the image set need to be determined accurately in order to

¹This is analogous to sample density or sample spacing. It is the effective

²This shift may also be a integer pixel + sub-pixel shift.

³This is a general value which does not hold true for the *Nonuniform Interpolation Method* when reconstructing

⁴Also commonly referred to as *Image Registration* in literature

recombine these in step 3. As the algorithm makes use of sub-pixel shifts, it is beneficial for the accuracy of such estimates to be reliable [63].

3. *Reconstruction*: The process whereby the initial image set with its successive shift estimations is used to produce a higher-resolution (alias free) output image. A plethora of reconstruction methods exist, and a tangible understanding and overview are out of scope.

A.2 Dynamic voltage and frequency scaling

Dynamic Voltage and Frequency Scaling (DVFS) is the manipulation of operating frequency and voltage in order to reduce power dissipation. Both voltage and frequency are scaled during operation dynamically, increasing frequency under high loads, and decreasing under lower loads. This method is commonly used in both FPGA [64] and SoC [65] designs. To best understand this, a quick review of a CMOS's power consumption⁵ [55]:

$$\begin{aligned} P_{\text{avg}} &= P_{\text{switching}} + P_{\text{short-circuit}} + P_{\text{leakage}} \\ &= \alpha C_L V_{dd}^2 f_{clk} + I_{sc} V_{dd} + I_{leakage} V_{dd} \end{aligned} \quad (\text{A.1})$$

the first term, the term most affected by changing both voltage and frequency, represents the switching component. This is the power dissipated when transitioning from one logic state to another. V_{dd} is the transistors supply voltage, C_L is the load capacitance, f_{clk} the transition frequency and α the node transition activity factor, which is the average number of transitions per clock cycle. In the case of a clock buffer, $\alpha = 1$. $P_{\text{short-circuit}}$ corresponds to the finite short time when there is a short between V_{dd} and ground. This occurs during the rise and fall times during a transition. This value is usually relatively small (9 x smaller than $P_{\text{switching}}$). $P_{\text{switching}}$ and $P_{\text{short-circuit}}$ are bundled under the same term: dynamic power. Lastly, P_{leakage} commonly referred to as static power, is constantly occurring due to the leaking current $I_{leakage}$.

A certain voltage will support a set of frequencies, however, if a higher set of frequencies is required, the voltage first needs to be increased. Thus operating at the highest-allowable frequency for a certain voltage will yield the most optimum result when maximising for throughput at lowest power [66].

⁵Despite being seemingly complex, this is a simplified version of [55].

A.3 Fourier-rotation determination resolution simulation

Consider the polar transform:

$$\begin{aligned} r &= \sqrt{(x - x_c)^2 + (y - y_c)^2} \\ \alpha &= \tan^{-1}\left(\frac{y - y_c}{x - x_c}\right) \end{aligned} \quad (\text{A.2})$$

the way the polar transform is calculated is backwards: the various r and α coordinates are stepped through, the inverse polar transform is then calculated to determine whether such an (x, y) coordinate pair exists, the closest is retrieved and then placed at (r, α) .

$$\begin{aligned} x &= r \cos \alpha \\ y &= r \sin \alpha \end{aligned} \quad (\text{A.3})$$

From this function, it is clear to see that not all pixels in the cartesian image will be mapped to polar coordinate, as is evident in Figure A.1:

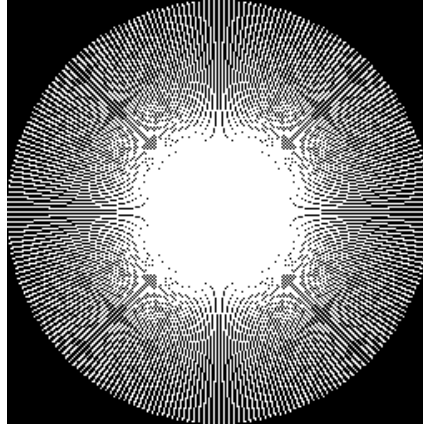


Figure A.1: Pixels effectively mapped from Cartesian plot to polar plot. The successfully mapped plots is represented by the white in the image. This is the result where the original image has resolution of 255x255 and is mapped to a polar plot with resolution 256x256. The effective mapped percentage = 58.5%, this only takes into account pixels within the circle.

A simulation was conducted where: a range of various polar resolutions were used to map various sized Cartesian images. A copy of each mapped image is then rotated a certain angle (using scipy's imrotate, using bilinear interpolation), afterwhich the equation introduced in Section 2.4.2.3:

$$\theta_{rot} = \mathcal{P}\langle |\mathcal{F}\{f(x, y)\}| \rangle \star \mathcal{P}\langle |\mathcal{F}\{g(x, y)\}| \rangle \quad (\text{A.4})$$

is carried out to determine the induced rotation θ_{rot} . In order to determine the accuracy vs number of pixels mapped, the number of cartesian coordinates mapped are tallied to determine:

$$\text{percentage mapped} = \frac{\text{pixels mapped to polar plot}}{\text{pixels in original cartesian plot}} \times 100$$

Figure A.3 shows the various stages of the calculation: first the original and the rotated images, then the magnitude of the Fourier transform, then the polar transform thereof. Figure A.1 shows why percentage mapped is a contributing factor, since a smaller resolution implies less accuracy.

Figure A.2 has the result of this simulation, a variety of angles $[-75.0, -60.0, -45.0, -30.0, -15.0, 0.0, 15.0, 30.0, 45.0, 60.0, 75.0]$ degrees are tested for every polar-cartesian resolution combination. The average error of determining these angles is the mean absolute error. The following polar resolutions are: $[64 \times 64, 128 \times 128, 256 \times 256, 512 \times 512, 1024 \times 1024, 2048 \times 2048]$, and the cartesian resolution are: $[255 \times 255, 511 \times 511, 1023 \times 1023]$. The odd number is due to the algorithm which was implemented which uses the central coordinate as (x_c, y_c) .

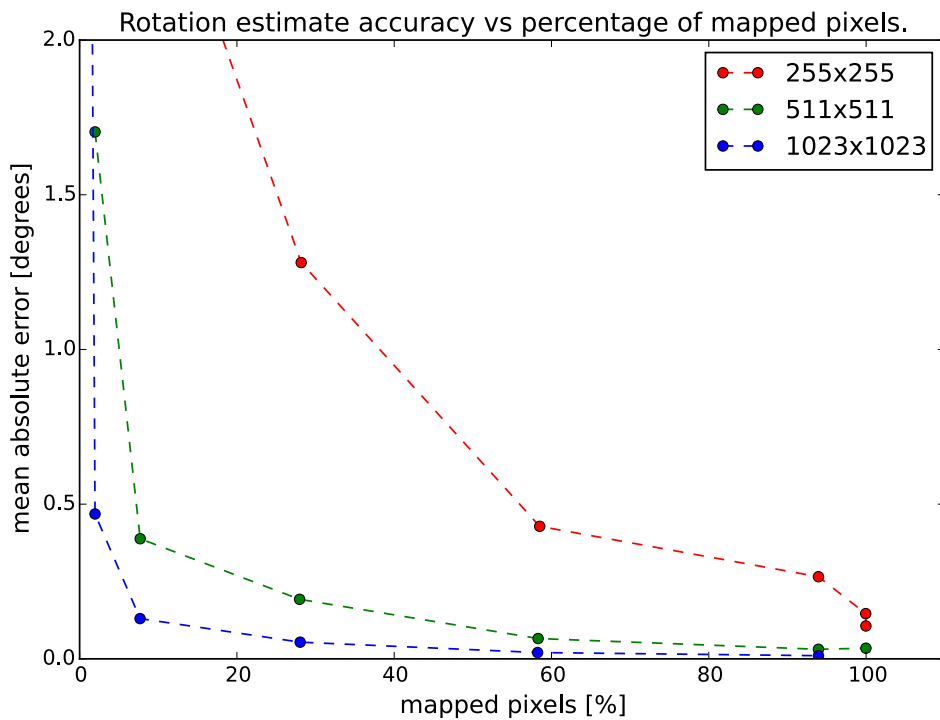


Figure A.2: Accuracy of rotation estimate vs amount of pixels mapped. The accuracy is a mean of rotation estimates. The percentage mapped is a function of the polar resolution. Low percentage relates to lower polar resolution. Points all the way on the right correspond to polar resolution 2048x2048, moving left corresponds to 1024x1024, 512x512 etc.

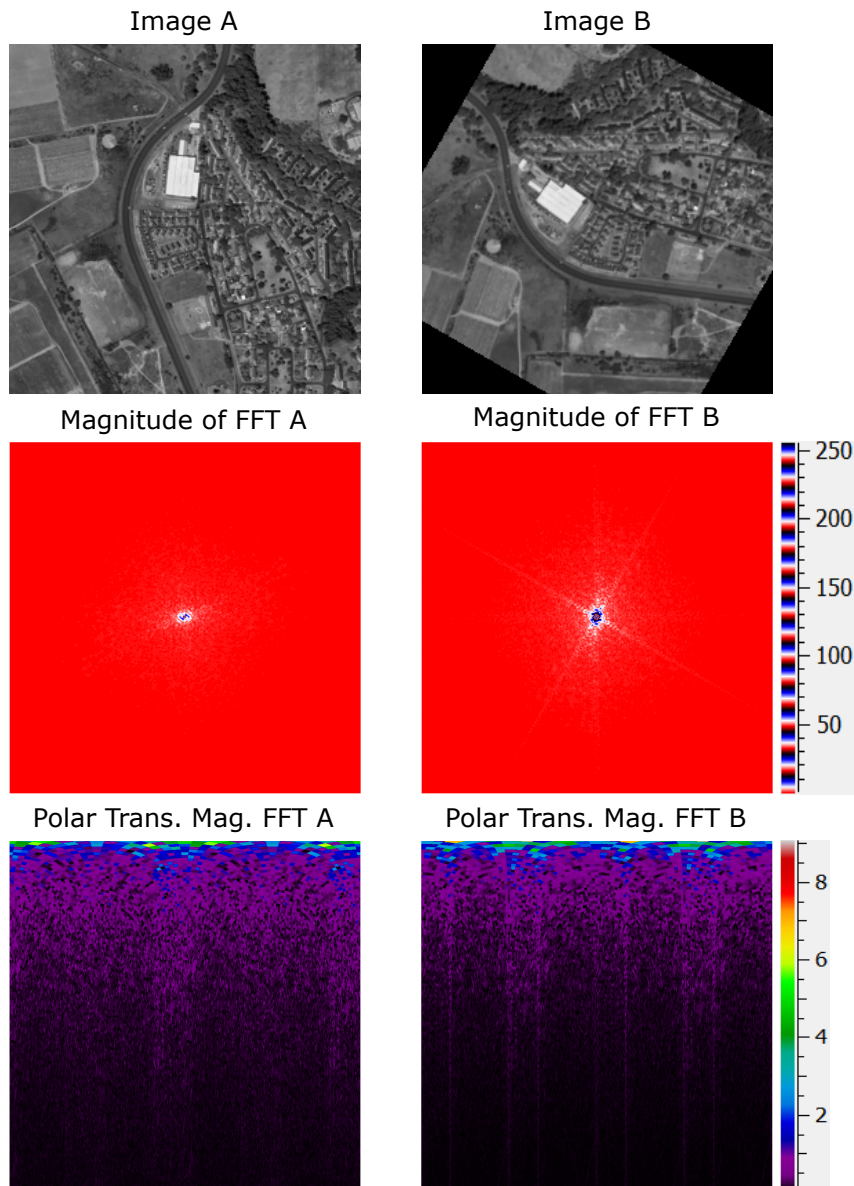


Figure A.3: Various parts of Eq. (A.4), note the translation along the horizontal axis of the polar transform of the FFT's magnitude spectrum. This transformation is documented even clearer in an example by Wilson and Theriot [37]. This corresponds to the rotation in the spatial domain. Note that the FFT's were normalised to a maximum of 255, and 2.5% of the top of the polar transform is removed, hence the polar transforms only range from 0 to 9. This corresponds to the low-frequency component of the FFT, this was done in order to achieve better results for the translation estimation of the polar transform, since important detail stored in the higher frequency is overpowered by the strong low-frequency component. Original is a subsection of [1].

Conclusion: Consider the mean error for a ‘reasonable’ size input image and polar resolution of 255x255 and 512x512 respectively. The mean error is just under 0.5° , thus the error is in the range of rotations we may expect. Thus implementing such an algorithm will detect a larger rotation than is actually present. This method shows promise, but requires much larger data sizes to produce a reasonable accuracy. Such sizes are not feasible to be reconciled with the current SPITE dimensions (discussed thoroughly in Section 4.5).

A.4 Image stacking

The following derivation of SNR gained from image stacking is reproduced from Banks [8]. Given M identical images S :

$$D_i(x, y) = S(x, y) + N_i(x, y) \quad 1 \leq i \leq M \quad (\text{A.5})$$

where $N_i(x, y)$ is random additive noise, $D_i(x, y)$ are the various noisy images, and $S(x, y)$ is the original image. Given that the following conditions are true for $N_i(x, y)$:

zero mean

$$E\{N_i(x, y)\} = 0, \quad (\text{A.6})$$

uncorrelated

$$E\{N_i(x, y)N_j(x, y)\} = E\{N_i(x, y)\}E\{N_j(x, y)\} = 0 \quad (\text{A.7})$$

and a variance of

$$E\{[N_i(x, y) - \bar{N}_i(x, y)]^2\} = E\{N_i^2(x, y)\} = \sigma^2 \quad \text{for all } i \quad (\text{A.8})$$

the resultant averaged image is

$$\bar{D}(x, y) = \frac{1}{M} \sum_{i=1}^M [S_i(x, y) + N_i(x, y)] \quad (\text{A.9})$$

Given the signal to noise *power* ratio:

$$P(x, y) = \frac{S_i^2(x, y)}{\sigma^2} \quad (\text{A.10})$$

then the average signal to noise power ratio is

$$\begin{aligned}
 \bar{P}(x, y) &= \frac{S^2(x, y)}{E\left\{\frac{1}{M} \sum_{i=1}^M N_i(x, y)\right\}^2} \\
 &= \frac{M^2 S^2(x, y)}{E\left\{\sum_{i=1}^M N_i^2(x, y)\right\} + E\left\{\sum \sum_{i \neq j} N_i(x, y) N_j(x, y)\right\}} \\
 &= \frac{M^2 S^2(x, y)}{E\left\{\sum_{i=1}^M N_i^2(x, y)\right\} + \sum \sum_{i \neq j} E\{N_i(x, y)\} E\{N_j(x, y)\}} \\
 &= \frac{M^2 S^2(x, y)}{M \sigma^2} \\
 &= MP(x, y)
 \end{aligned} \tag{A.11}$$

and since the average signal to noise ratio

$$\overline{SNR} = \sqrt{\bar{P}(x, y)} = \sqrt{M} \sqrt{P(x, y)} = \sqrt{M} SNR \tag{A.12}$$

thus the SNR of the averaged image is improved by factor \sqrt{M} .

Appendix B

Section 3, 4, 5

B.1 Least squares regression

Least squares, a regression method which gives an estimate for the parameters in \mathbf{x} , is defined with the following error referred to as the residual:

$$\mathbf{b} = \mathbf{z} - A\mathbf{x} \quad (\text{B.1})$$

the least squares method is interested in minimising the sum of the squares of the residuals' elements, where the sum of the elements squared is

$$e = \sum_{i=1}^n b_i^2 = [b_1 \quad b_2 \quad \dots \quad b_n] \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \mathbf{b}^T \mathbf{b} = (\mathbf{z} - A\mathbf{x})^T (\mathbf{z} - A\mathbf{x})$$

to minimise this cost function e , we differentiate e with respect to \mathbf{x} and set to zero:

$$\frac{\partial e}{\partial \mathbf{x}} = 0 \quad (\text{B.2})$$

which yields [67]

$$A^T A \mathbf{x} = A^T \mathbf{z} \quad (\text{B.3})$$

given at least 6 independent observations, $A^T A$ will be strictly positive definite, satisfying the *sufficient* condition stated in Crassidis et al. [68], this allows $A^T A$ to be inverted:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{z} \quad (\text{B.4})$$

B.2 Linear sub-pixel interpolation

After having performed sub-pixel shifts of magnitudes as fine as 0.2 pixels, finer resolution requirements are handled by linear interpolation. Consider a 0.1

pixel shift requirement (now forget the 5x5 super-pixel for this explanation), imagine you are required to shift a 4×4 image's content 0.1 pixel to the right, and 0.1 pixel down, as in Figure B.1.

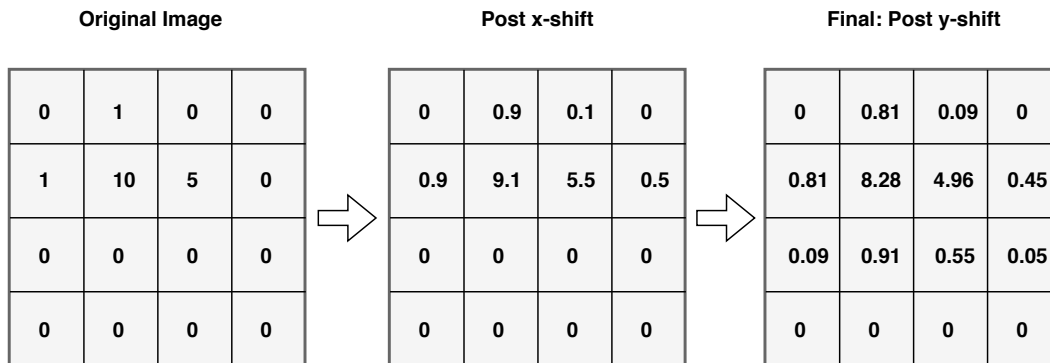


Figure B.1: Explanatory figure: how to linearly interpolate an images content 0.1 pixels to the right, and 0.1 pixels down.

First consider x-shift. Think of the pixels as point values whose values get distributed along the x axes according to the required shift. Thus 0.1 pixel shift relates to 10% loss of value (to the pixel on the right), 90% of the value stays, and 10% of the value of pixel to the left is added. Thus consider 3rd column 2nd row initial value 5. 10% moves to the right, thus 10% of 5 is added to 4th column 2nd row (and subsequently subtracted from initial 5, thus left with 4.5. Consider the value '10' in 2nd column 2nd row, it 'loses' 10% of its value (1) to 3rd column 2nd row, thus final value of 3rd column 2nd row after x-shift: 5.5. Furthermore 0.1 (10%) of 1st column 2nd row gets added to 2nd column 2nd row. Therefore final value of 2nd column 2nd row is : $10 - 1 + 0.1 = 9.1$

Pixel values outside of the grid are considered to have value 0, however all simulations make sure that these borders are not part of test data. After all x-shifts have been completed, the y-shift can start. Precisely the same methodology is followed for y-shift as with x-shift. The only trick is not to *overwrite* the original image while computing post x-shift, and likewise not overwriting post x-shift when calculating post y-shift.

Consider once again the method proposed in Section 4.1. The linear interpolation is applied *after* the pixel increment shifts, where 0.2 in resultant image, implies 1 full pixel shift in initial image. For ex. if a final shift of (0.4; 0.3) is required, the 2000×2000 image is shifted right by 2 pixels, afterwhich it is shifted down by 1 pixel, afterwhich a further 0.5 pixels is shifted down using this linear interpolation method. Only *after both* 'integer' shifts and interpolated shifts are induced, do we group the super-pixel and down-sampled. Thus 'interpolation' is the key-concept here, new information is not created or added, merely a neat way of supplying a continuous sub-pixel shift, allowing testing the SPITE algorithm over a fine granularity.

Figures B.2 and B.3 – which follow the same methodology as Figure 4.5 – show how the linearly interpolated results (Figure B.3) *follow* the uninterpolated results (Figure B.2). This provides confidence that the interpolation technique is valid, since the general trends are followed.

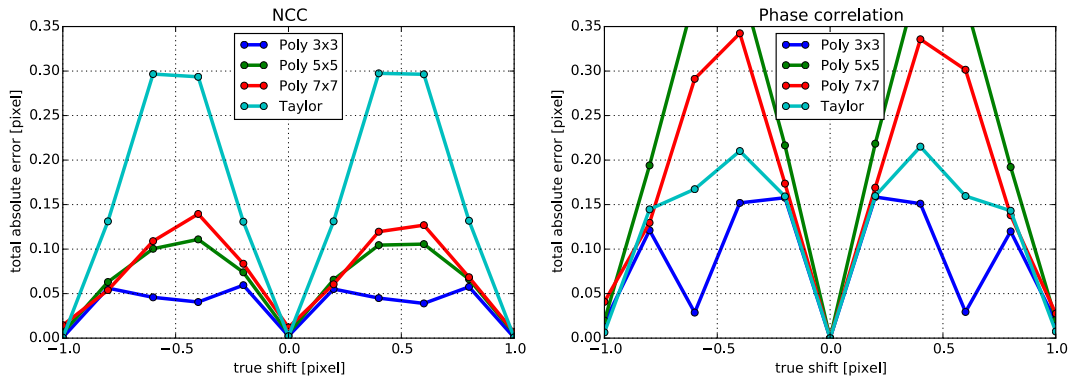


Figure B.2: Mis-registration error, where shift method is bound to 0.2 pixel shift resolution, $x_{\text{shift}} = y_{\text{shift}}$.

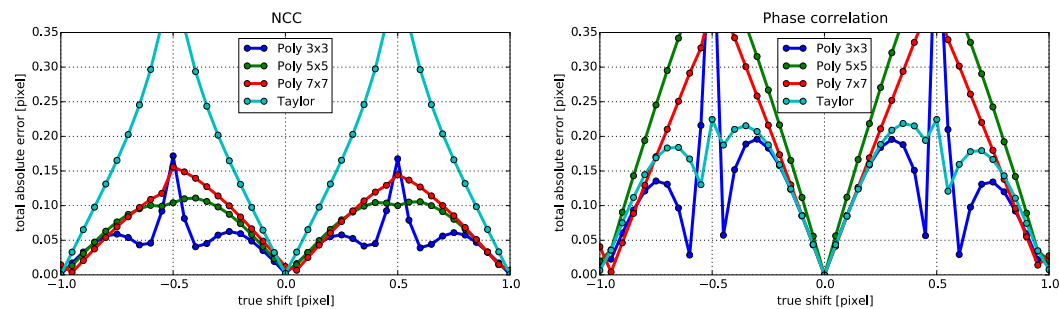


Figure B.3: Mis-registration error, where shift method uses linear interpolation to achieve resolution finer than 0.2 pixel shift (0.05 pixel). This is a precise copy of Figure 4.5, such that $x_{\text{shift}} = y_{\text{shift}}$.

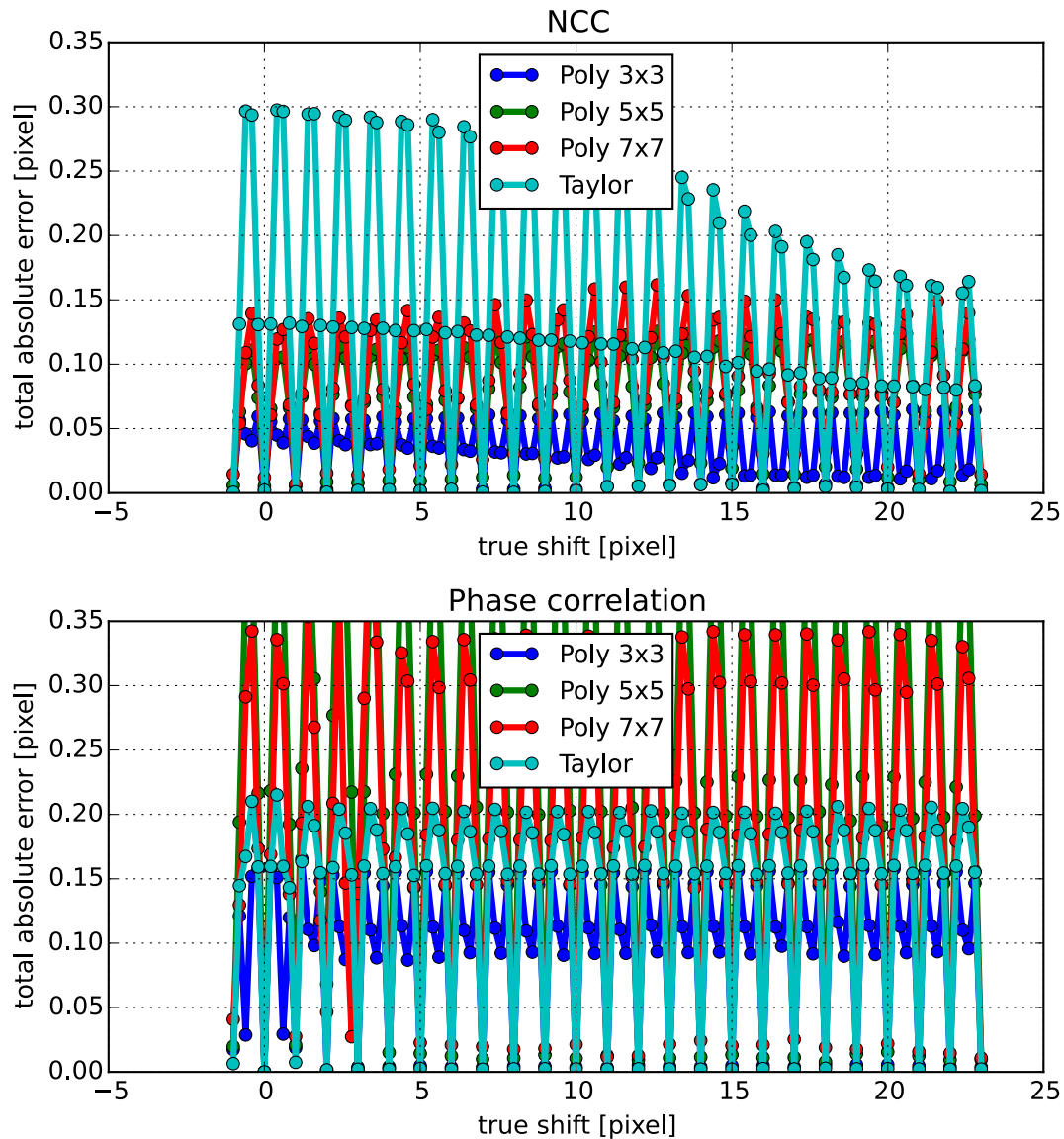


Figure B.4: Mis-registration error, same as Figure 4.4, just a larger range to show that the results hold generally for the image, where $x_{\text{shift}} = y_{\text{shift}}$. Note how Poly 3×3 still generally outperforms Taylor.

B.3 SNR post image stacking

SNR post image stacking was calculated in the following manner:

1. The per-pixel difference between the original and noise-reduced image is taken, these values contribute to a certain spread, standard deviation of this spread is determined.

2. The average pixel value of the original image is divided by the standard deviation calculated in previous step
3. This SNR value is averaged across all four images, so as to work with one value

B.4 Surface fit error

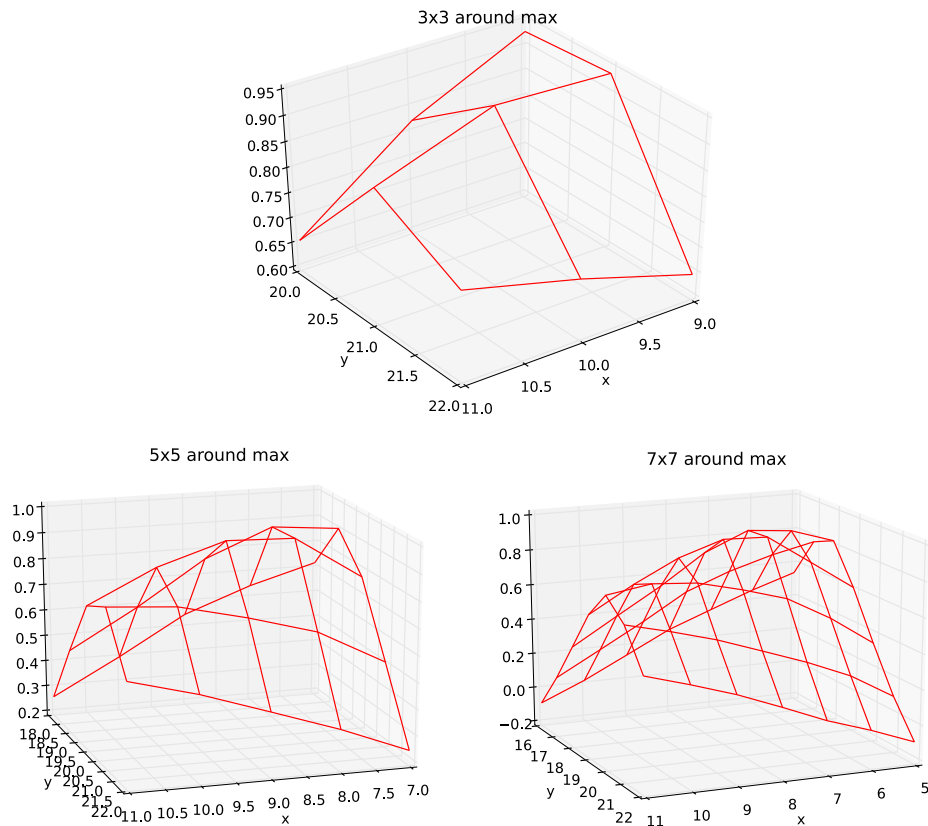


Figure B.5: Various surfaces with varying sizes, for the worst case error of 1.59 for image A.



Figure B.6: Window (left) and template (right) for the translation error of 1.59, clearly not due to lack of features, but possibly a result of over-saturation of image values. Subsections of [1].

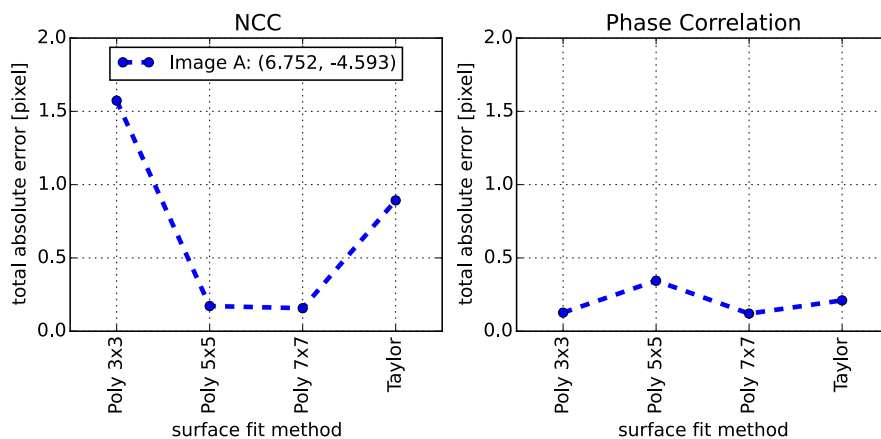


Figure B.7: SPITE errors where correlation and surface fit methods are altered for image A's translation where the maximum error occurs. Clearly Poly 3×3 yields the worst error for this case.

B.5 Noisy sectioned images

A note on the use of the Stellenbosch image: 'This thesis has modified this data for use from its original source, www.capetown.gov.za, the official website of the City of Cape Town. The City of Cape Town makes no claims as to the content, accuracy, timeliness, or completeness of any of the data provided at this site. The data provided at this site is subject to change at any time. It is understood that the data provided at this site is being used at one's own risk.'
Usefull links:

- <https://citymaps.capetown.gov.za/EGISViewer>
- <http://www.ngi.gov.za/index.php/what-we-do/aerial-photography-and-imagery>

It is a subsection of the full Stellenbosch image .

These images, all subsections of the Stellenbosch image [1], can only be appreciated in their proper context: zoom in till the image starts pixelating. If the pdf reader smoothes the pixels, try a different reader.

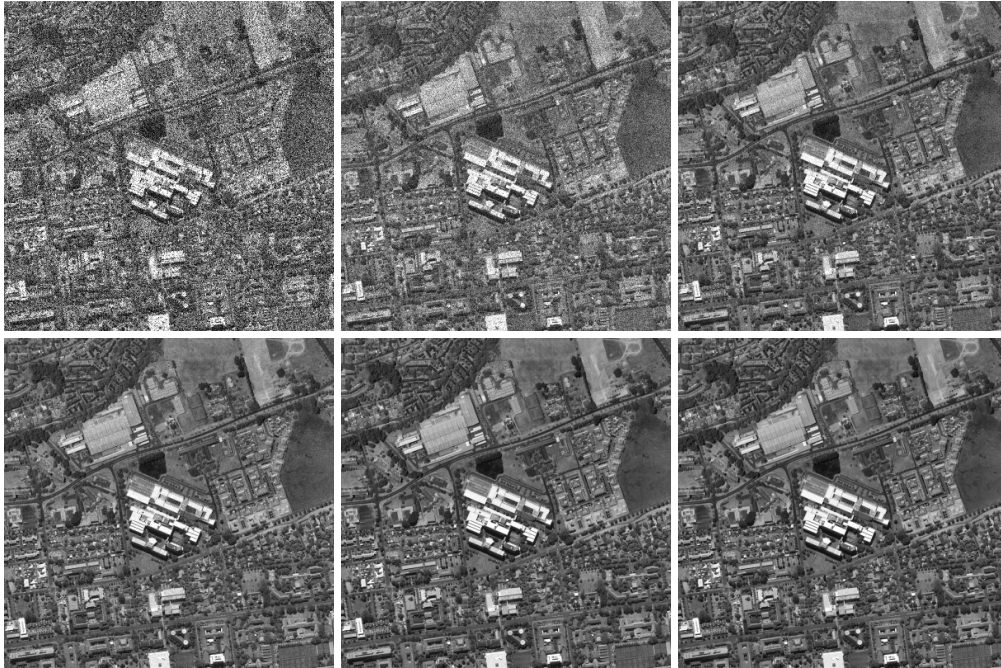


Figure B.8: Image A with the varying SNR's: 2.5, 5, 10, 20, 50, ground truth. Subsection of [1].



Figure B.9: Image B with the varying SNR's: 2.5, 5, 10, 20, 50, ground truth. Subsection of [1].

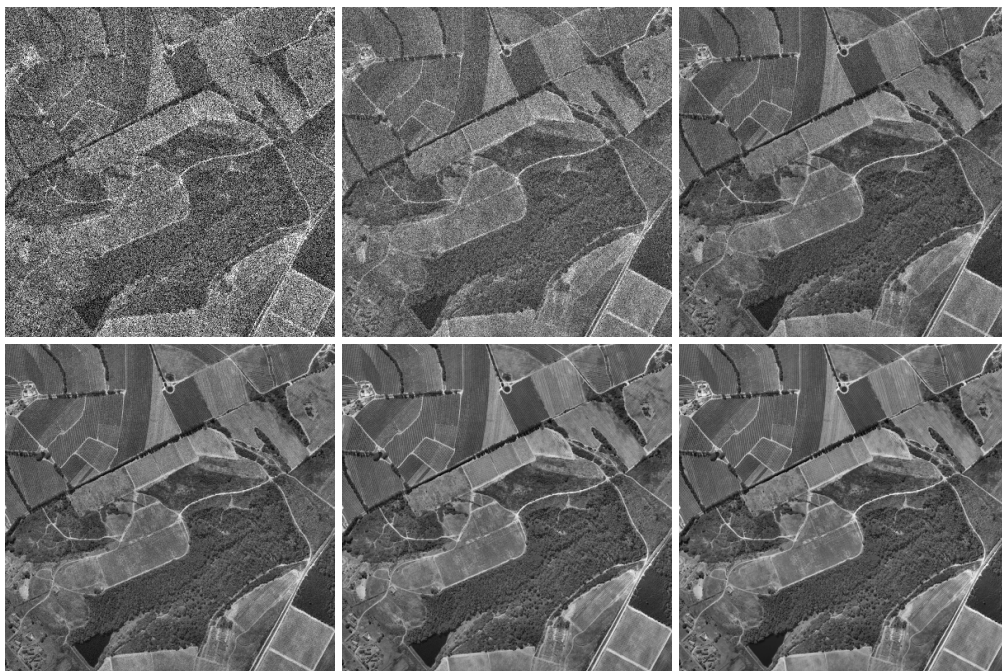


Figure B.10: Image C with the varying SNR's: 2.5, 5, 10, 20, 50, ground truth. Subsection of [1].

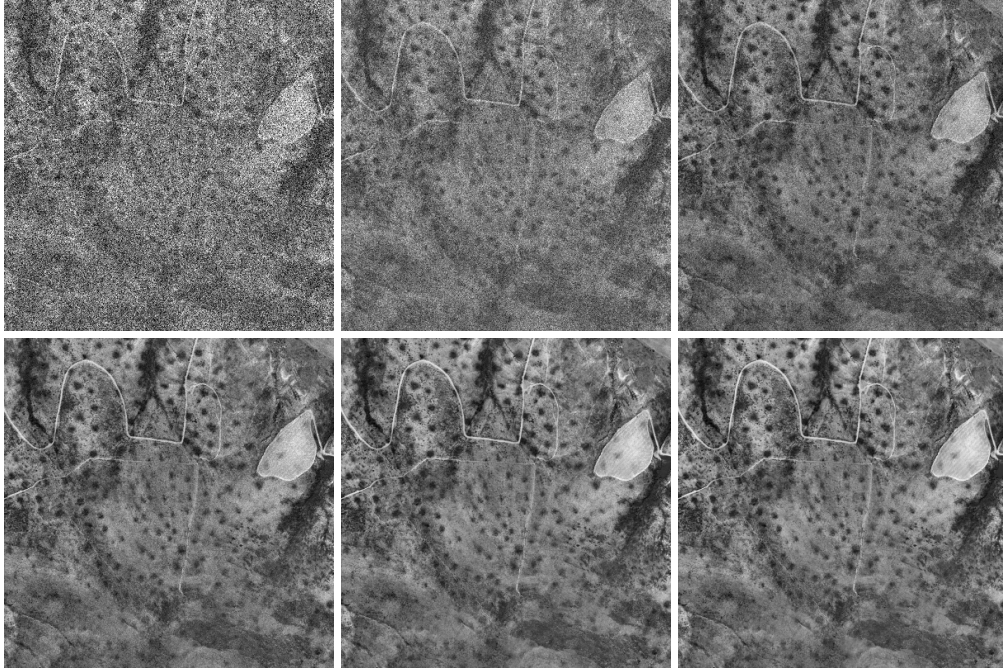


Figure B.11: Image D with the varying SNR's: 2.5, 5, 10, 20, 50, ground truth. Subsection of [1].

B.6 Counters and base 2^n

Why are image processing methods, and specifically embedded systems sold-out on having images and data sizes with dimensions $2^n \times 2^m$? And why is zero indexing the standard?

Consider the binary counting system: where 2 bits provide the following numbers: 0, 1, 2, 3 which are represented in bit form as $00_2, 01_2, 10_2, 11_2$. This is a range of $0 \rightarrow 2^n - 1$, but consisting of 2^n unique numbers, where n is the number of bits used. Therefore, it is more efficient to count from $0 \rightarrow 3$ using 2 bits as opposed to counting from $1 \rightarrow 4$ using 3 bits. Furthermore, when counting 4 unique locations, from $0 \rightarrow 3$, the user does not need to clear the counter after reaching 11_2 , as the next increment will overflow the counter and automatically set it back to 00_2 .

When organising data along x, y coordinates, it is useful to keep both x and y dimensions as 2^n , such that data is used most optimally. Consider an image consisting of dimensions 4×4 , where each pixel has an address, consisting of its column appended to its row. If counting through the image pixels row-by-row, after incrementing from 0011_2 , the last element of the first row, the next address 0100_2 will refer to the next rows data point. This shows the power and simplicity of sticking to 2^n .

Lastly the FFT implementations, all want their data in 2^n , this has to do with the butterfly computation optimisation, this forces most image processing

techniques to use image dimensions of $2^n \times 2^m$.

B.7 Benchmark implementation

Power measurement methodology, compiler directives and code snippets are provided.

B.7.1 Instantaneous power:

The power measured and plotted in Figure 5.6 is the instantaneous SoC power, and not the full board power.

To determine the C-serial HF, C-parallel HF and CUDA HF platform power, the voltage across a $5\text{m}\Omega$ sense resistor – RC511 – was measured. Now the board draw current can be determined by Ohms law, and consequently the board power since the board power supply is known, 12V. Using this method, we found that the fluctuation of board idle power is less than 2.5% across five tests for all configurations.

Furthermore, a sanity check was conducted for the SoC power with C-parallel HF for 2D-FFT 128×128 . Board *idle* and *runtime* power was determined, a 2.4W change in board power from 6.48 to 8.88W. This is similar to the difference between idle and runtime *SoC* power 0.25 and 2.35W. This validates the SoC power measurement method, slight discrepancies may be attributed to the inconsistent, slow power measurements, non-linear behaviour of the power source¹, and measurement inadequacies.

For safeties sake, all tests were conducted *with* the cooling fan connected. All tests were conducted with: a usb mouse and keyboard connected via a USB-hub, along with the HDMI output connected to a screen, so as to simplify the interface and get rapid results. Clearly the board power may be reduced by disconnecting such devices.

B.7.2 C-serial, C-parallel

In order to compile the most optimal C-code for the platform, the following compiler directives were provided: ‘ gcc -mfloat-abi=hard -mfpu=neon-vfpv4 -mcpu=cortex-a15 -fopenmp -O3 ’. Both C-serial and C-parallel use the same source code, however NUM_THREADS changes from 1 to 4. Timing is achieved by using, OpenMP’s ‘omp_get_wtime()’ within a for-loop for added resolution. The 1D-FFT code from [60] was wrapped appropriately to determine a 2D-FFT. A snippet of code which shows the timing and wrapping of 1D-FFT code:

```
1 float *actualoutreal = linear_reals(num_of_cols*num_of_rows); //
   Input data, output written over input
```

¹In order for an increase of 1W in a certain subsystem, due to non-idealities and losses, the total input power be greater than 1W to attain the required subsystem power increase.

```

2 float *actualoutimag = linear_reals(num_of_cols*num_of_rows); //
   Input data, output written over input
3 float *buff_real = malloc(num_of_rows * sizeof(float) *
   NUMTHREADS);
4 float *buff_imag = malloc(num_of_rows * sizeof(float) *
   NUMTHREADS);
5
6 //Timing function math
7
8 omp_set_num_threads(NUMTHREADS);
9 t1_math = omp_get_wtime();
10 //For-loop which ensures a proper timing resolution
11 for(res = 0 ; res < RESOLUTION ; res++){
12 //First Dimension
13 #pragma omp parallel for
14 for(i = 0; i < num_of_rows ; i++){
15     Fft_transform( &actualoutreal[i*num_of_cols], &actualoutimag[
   i*num_of_cols], num_of_cols);
16 }
17
18 //Second dimension
19 omp_set_num_threads(NUMTHREADS);
20 #pragma omp parallel for
21 for(i = 0; i < num_of_cols ; i++){
22     //Create intermediate buffer
23     int ID = omp_get_thread_num();
24     int k;
25     for(k = 0 ; k < num_of_rows ; k++){
26         buff_real[k + ID*num_of_cols] = actualoutreal[i + k*
   num_of_rows];
27         buff_imag[k + ID*num_of_cols] = actualoutimag[i + k*
   num_of_rows];
28         //printf("%f ", buff_real[k]);
29     }
30     Fft_transform( &buff_real[ID*num_of_cols], &buff_imag[ID*
   num_of_cols], num_of_rows);
31     //update with intermediate buffer values
32     for(k = 0 ; k < num_of_rows ; k++){
33         actualoutreal[i + k* num_of_rows] = buff_real[k + ID*
   num_of_cols];
34         actualoutimag[i + k* num_of_rows] = buff_imag[k + ID*
   num_of_cols];
35     }
36 }
37 }
38 t2_math = omp_get_wtime();

```

B.7.3 CUDA

Nvidia's 'nvcc' compiler was for the CUDA code. CudaEvents, in conjunction with cudaEventRecord(), cudaDeviceSynchronize() and cudaEventSynchro-

nize() are used to ensure an accurate runtime within a RESOLUTION for-loop. cudaDeviceSynchronize() ensures that the full cufftExecC2C() function is completed before re-running it.

```
1 cudaEventRecord(start, 0);
2 for(i = 0; i < RESOLUTION; i++){
3     cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
4     cudaDeviceSynchronize();
5 }
6 cudaEventRecord(stop, 0);
7 cudaEventSynchronize(stop);
```

Note that time required for memory allocation is timed but not considered part of `t_math`. This is a reasonable assumption since the manner in which `t_math` is determined, already hinders it. It removes any ability to stream calculations consistently. Furthermore, the GK20A microarchitecture has one dedicated copy machine, this means, data transfers and calculations can be pipelined. This feature is also not utilised, implying that CUDA runtime answers are worst case, and have scope for improvement.

Appendix C

Section 6, 7, 8

C.1 Resources

| Math Block | 256x256 | Kbits $M = 256, N = 128$ | BRAMS (36Kbit) 128x128 | BRAMS (36Kbit) 64x64 | BRAMS (36Kbit) |
|--------------|--------------------------------------|---|--|--|----------------|
| Window | 8bit MxM | $(M \times M \times 8) / 1024 = 512$ | 15 | 4 | 1 |
| Template | 8bit NxN | $(N \times N \times 8) / 1024 = 128$ | 4 | 1 | 0.5 |
| 14 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 15 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 16 | 1x 32bit float image NxN | $(N \times N \times 32) / 1024 = 512$ | 15 $N \times N \times 32 / 1024 = 128$ | 4 $N \times N \times 32 / 1024 = 32$ | 1 |
| 17 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 18 | | | 0 | 0 | 0 |
| 19 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 20 | | | | | |
| 21 | | | | | |
| 22 | 1x64 bit float image + 2x buffers | $(65536 \times 64) / 1024 + 1 = 4097$ | 115 $16384 \times 64 / 1024 + 1 = 1025$ | 30 $4096 \times 64 / 1024 + 1 = 257$ | 9 |
| 23 | | | 0 | 0 | 0 |
| 24 | 2xinput buffers, 32bit x2 xMxM | $(65536 \times 2 \times 32) / 1024 + 1 = 4097$ | 115 $16384 \times 64 / 1024 + 1 = 1025$ | 30 $4096 \times 64 / 1024 + 1 = 257$ | 9 |
| 25 | | | 0 | 0 | 0 |
| 26 | | | 0 | 0 | 0 |
| 27 | | | 0 | 0 | 0 |
| 28B | 1x input buffer, 32bit xMxM | $(65536 \times 32) / 1024 + 0.5 = 2048 + 0.5$ | 58 $(16384 \times 32) / 1024 + 0.5 = 513$ | 16 $(4096 \times 32) / 1024 + 0.5 = 129$ | 5 |
| 29 | 1x24 bit int image with MxM elements | $65536 \times 24 / 1024 + 1 = 1537$ | 44 $16384 \times 24 / 1024 + 1 = 385$ | 12 $4096 \times 24 / 1024 + 1 = 97$ | 4 |
| 30 | | | 0 | 0 | 0 |
| 31 | | | | | |
| 32 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 33 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 34 | | | 0 | 0 | 0 |
| 35 | | | 0 | 0 | 0 |
| 36 | | | 0 | 0 | 0 |
| 37 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 38 | 1xinput buffer | $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 $(512 \times 32) / 1024 = 16$ | 0.5 |
| 39 | 1xinput buffer, 32bit x NxN | $32 \times 128 \times 128 / 1024 + 0.5 = 512 + 0.5$ | 16 $32 \times 64 \times 64 / 1024 + 0.5 = 128 + 0.5$ | 5 $32 \times 32 \times 32 / 1024 + 0.5 = 32 + 0.5$ | 2 |
| 40 | | | 0 | 0 | 0 |
| Total | | | 386 | 106 | 35.5 |

Figure C.1: Various BRAM resource requirements for different sizes of M , where $N = M/2$.

C.2 FSM-Design methodology

As mentioned in the design methodology, a simple, easy to understand, easy to debug, deterministic behaviour is desired for the FSMs in this project. Although these are not necessarily the most optimal, they make up with easy customisability. The following transition flow diagram Figure C.2 will be referenced in this discussion, the VHDL code is included at the end of this discussion.

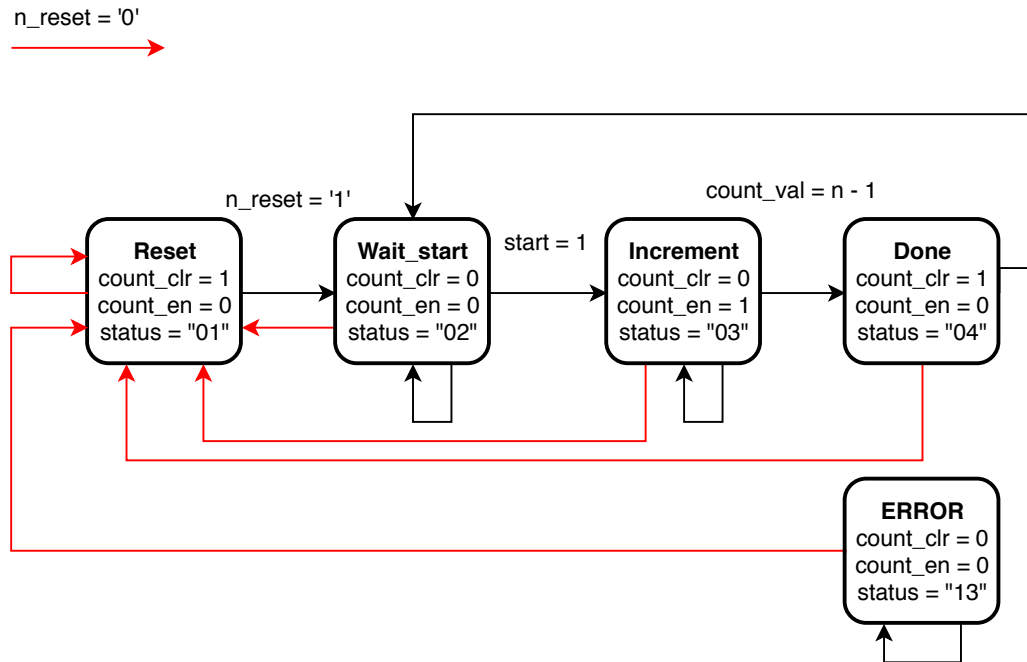


Figure C.2: Flow diagram of a simple 'count up to n' FSM. Note: each state can enter reset.

First is the ability to easily understand the FSM code. This is enabled by having a certain guaranteed output per state. Thus if a FSM is known to be in a certain state, it will have certain outputs, instead of having a range of conditional outputs within a single state. Similarly, since the output signals are updated *in* the next_state logic, the control signals are synchronised with the state i.e. the 'counter_clr' signal will be asserted at the same time, synchronously as the Reset state is entered.

Each state has an 8bit status register, which is updated along with the state value. Counters are also all external, this separates counting logic from the FSM. This allows easy counter observation during HW debug, as it is easy to set up a probe on any output port with an ILA in post-synthesis mode. This does have the downside of having to manually change the counters widths when scaling up.

The FSMs tend to follow a flow diagram approach, this makes it easy to follow and design with pen and paper, and thus step through the logic easily.

Lastly, deterministic behaviour is desired. Ability for output signals to be glitchy is removed by making use of dedicated control registers which are updated in every state transition. Each bit within the control register is mapped to a certain output pin. The glitches occur if a certain output pin is mapped to a certain status register. How will this induce glitchy signals? Since each state is mapped to a certain bit pattern, when transitioning between status A and B, since all bits do not toggle simultaneously, a transition state C may

be entered for a brief moment causing state C specified control signals to be enabled for a short time. This may cause havoc with asynchronous logic, thus the extra logic to ensure that does not occur.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity FSM_example is
  generic(counter_width : integer := 12);
port (
  clk          : in std_logic;
  --control inputs
  n_reset     : in std_logic;
  start       : in std_logic;
  --value to which we count
  n           : in std_logic_vector(counter_width -1 downto 0);
  --input from an external counter
  count_val   : in std_logic_vector(counter_width -1 downto 0);
  --the status for debugging
  status      : out std_logic_vector(7 downto 0);
  --Control outputs
  count_en    : out std_logic;
  count_clr   : out std_logic
);
end FSM_example;

architecture count_up_to_n of FSM_example is

Type state_types is (Reset , wait_start , increment , done, ERROR);
signal current.STATE : state_types;
constant num_contrl_sigs : integer := 2;
signal current_CTRL    : STD_LOGIC_VECTOR(num_contrl_sigs-1 downto 0);
signal current.STATUS  : std_logic_vector(7 downto 0);
-- Map output control signals to current_CTRL
-- 1 count_en   MSB
-- 0 count_clr  LSB

--state specific CTRL registers
constant Reset_CTRL      :
  std_logic_vector(num_contrl_sigs-1 downto 0) := "01";
constant wait_start_CTRL:
  std_logic_vector(num_contrl_sigs-1 downto 0) := "00";
constant increment_CTRL :
  std_logic_vector(num_contrl_sigs-1 downto 0) := "10";
constant Done_CTRL      :
  std_logic_vector(num_contrl_sigs-1 downto 0) := "01";
constant ERROR_CTRL     :
  std_logic_vector(num_contrl_sigs-1 downto 0) := "00";

--state specific STATUS register
constant reset_STATUS:   STD_LOGIC_VECTOR (7 downto 0):= x"01";
constant wait_start_STATUS:STD_LOGIC_VECTOR (7 downto 0):= x"02";

```

```

constant increment_STATUS: STD.LOGIC_VECTOR (7 downto 0):= x"03";
constant done_STATUS:      STD.LOGIC_VECTOR (7 downto 0):= x"04";
constant ERROR_STATUS:    STD.LOGIC_VECTOR (7 downto 0):= x"13";

begin
  process (clk)
  begin

    if (clk 'Event and clk = '1') then
      if n_reset = '0' then
        current_state <= Reset;
        current_CTRL  <= Reset_CTRL;
        current_status <= Reset_STATUS;
      else
        case current_state is
          when Reset =>
            if n_reset = '1' then
              current_state <= wait_start;
              current_CTRL  <= wait_start_CTRL;
              current_STATUS<= wait_start_STATUS;
            else
              current_state <= current_state;
              current_CTRL  <= current_CTRL;
              current_STATUS<= current_STATUS;
            end if;

          when wait_start =>
            if start = '1' then
              current_state <= increment;
              current_CTRL  <= increment_CTRL;
              current_STATUS<= increment_STATUS;
            else
              current_state <= current_state;
              current_CTRL  <= current_CTRL;
              current_STATUS<= current_STATUS;
            end if;

          when increment =>
            if unsigned(count_val) = unsigned(n) -1 then
              current_state <= done;
              current_CTRL  <= done_CTRL;
              current_STATUS<= done_STATUS;
            else
              current_state <= current_state;
              current_CTRL  <= current_CTRL;
              current_STATUS<= current_STATUS;
            end if;

          when done =>
            current_state <= wait_start;
            current_CTRL  <= wait_start_CTRL;
            current_STATUS<= wait_start_STATUS;

```



```

when ERROR =>
    current_state <= current_state;
    current_CTRL <= current_CTRL;
    current_STATUS<= current_STATUS;

when others =>
    current_state <= ERROR;
    current_CTRL <= ERROR_CTRL;
    current_STATUS<= ERROR_STATUS;

end case;
end if;
end if;
end process;

--Map signals to output ports
status <= current_STATUS;
count_en <= current_CTRL (1);
count_clr <= current_CTRL (0);

end count_up_to_n;

```

C.3 FPGA development steps and processes

A quick introduction to FPGA development process, from conception to completion.

VHDL is a hardware description language, which means low-level logic blocks are described. As an example in C-code you would say: `uint32 x = a + b`, whereas in VHDL you would code a 32bit unsigned adder circuit, where the output `x`, is the result of the two inputs `a` and `b`. These values, `x`, `a` and `b` are physical 32bit wide busses, which need to be routed to the next/previous appropriate logic unit. This process will usually run on a clock, which the user provides. Thus resets, clearing, clock signals and data moving is in the hands of the designer.

Such low-level manipulation may be daunting, yet many IP-cores are available which do high-level mathematical functions such as FFT, floating-point mathematical operations etc. These are easily instantiated and manipulated in a block diagram, giving the designer low-level control of a high-level mathematical function.

Simulation is the first method of verifying the logic's functionality, this usually makes use of wave-diagrams and test-benches. Up till this point the code is generic to any platform. *Constraints* limit the design in various aspects e.g.: at what clock speed the design will perform, on which platform (which

FPGA family and chip, what development board, if any is used), lastly which signals will be mapped to physical output pins.

After verifying the logic, *synthesis* takes the logic (Register-Transfer Level (RTL)) and transforms it to actual gate-level logic units present on the specific FPGA. It can give an initial timing estimate, as each gate has specific properties.

After synthesis, a *post-synthesis timing simulation* can be undergone, to double check that the logic (with its various timing delays) still functions as required. Using a good test-bench aides this process, thus time invested in developing a good one is not wasted.

Next, the *implementation* run is undergone. This consists of placing the physical logic units (gates) on the physical device, and then routing the signals between the gates. Now it becomes apparent that implementation runs for denser designs take longer and will usually result in longer routing delays. This drives down the maximum possible operating frequency of the design.

A *post-implementation timing simulation*, similar to the post-synthesis timing simulation, may now be performed to validate the functioning of implemented logic.

Lastly, this design is translated into code which the FPGA understands: *bitstream*. This is generated and then flashed onto the FPGA. Now the *HW debugging* may take place. Most development boards, make use of some sort of JTAG interface, by which the HW may be probed. Logic units such as Integrated Logic Analysers (ILA's) and Virtual Inputs/Outputs (VIO's) are added to the design early on, knowing that they will be used for HW validation and debugging purposes. This means that if a design takes up 80% of resources, only 20% of resources are available for debugging, a designer needs to be aware of this.

Many modern FPGA's have a CPU-hard coded onto the same chip as the FPGA. In the case where one is not available, a soft-core processor may (usually) be instantiated using the FPGA fabric. Once instantiated, they are part of the logic, Xilinx's Vivado conveniently separates the coding processes, using the Software Development Kit (SDK), an eclipse based development environment, to create and compile C-code for the processor. This means HW debugging may be somewhat more complex when making use of a processor.

C.4 2D-FFT block diagram

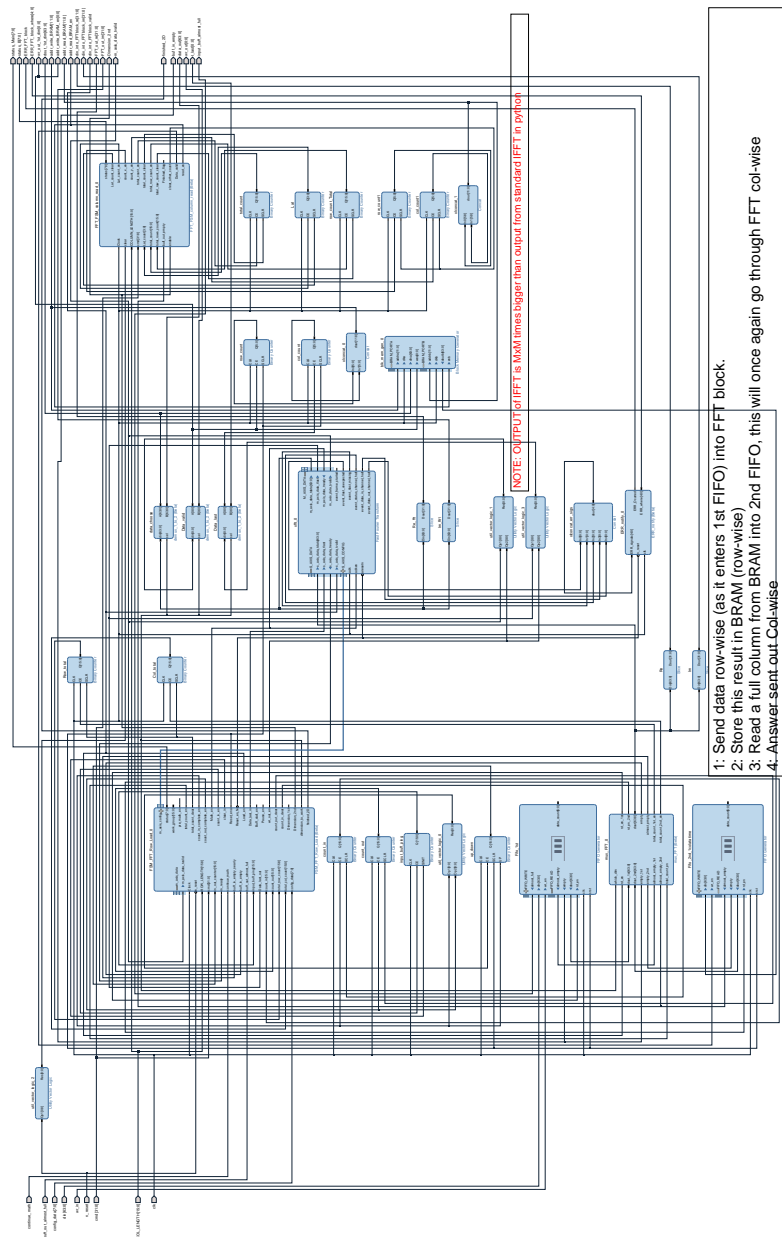


Figure C.3: 2D-FFT block diagram, shown to display various aspects of a more complex math-block. Note the various counters/address lines, BRAMs, FIFOs etc. which need to be adjusted when scaling up.

C.5 Polynomial fit C-code

The functions which are used for the polynomial least squares regression on Zynq CPU: `getCofactor()`, `determinantOfMatrix()`, `adjoint()`, `inverse_mat()`, `transpose()` and `mult_mat()`, source from [69][70].

```

1 #include "math.h"
2
3 ///---Poly fitting Math---///
4 void getCofactor(int size, double mat[size][size], double temp[
   size][size], int p, int q, int n)
5 {
6 int i = 0, j = 0;
7
8 // Looping for each element of the matrix
9 for (int row = 0; row < n; row++)
10 {
11 for (int col = 0; col < n; col++)
12 {
13 // Copying into temporary matrix only those element
14 // which are not in given row and column
15 if (row != p && col != q)
16 {
17 temp[i][j++] = mat[row][col];
18
19 // Row is filled , so increase row index and
20 // reset col index
21 if (j == n - 1)
22 {
23 j = 0;
24 i++;
25 }
26 }
27 }
28 }
29 }
30
31 /* Recursive function for finding determinant of matrix.
32 n is current dimension of mat[ ][ ]. */
33 double determinantOfMatrix(int size, double mat[size][size], int n
   )
34 {
35 double D = 0; // Initialize result
36
37 // Base case : if matrix contains single element
38 if (n == 1)
39 return mat[0][0];
40
41 double temp[size][size]; // To store cofactors
42
43 double sign = 1; // To store sign multiplier
44
45 // Iterate for each element of first row

```

```

46  for (int f = 0; f < n; f++)
47  {
48      // Getting Cofactor of mat[0][f]
49      getCofactor(size, mat, temp, 0, f, n);
50      D += sign * mat[0][f] * determinantOfMatrix(size, temp, n - 1)
51      ;
52      // terms are to be added with alternate sign
53      sign = -sign;
54  }
55
56  return D;
57 }
58
59 // Function to get adjoint of A[N][N] in adj[N][N].
60 void adjoint(int size, double A[size][size], double adj[size][size]
61             )
62 {
63     if (size == 1)
64     {
65         adj[0][0] = 1;
66         return;
67     }
68     // temp is used to store cofactors of A[][]
69     int sign = 1;
70     double temp[size][size];
71
72     for (int i=0; i<size; i++)
73     {
74         for (int j=0; j<size; j++)
75         {
76             // Get cofactor of A[i][j]
77             getCofactor(size, A, temp, i, j, size);
78
79             // sign of adj[j][i] positive if sum of row
80             // and column indexes is even.
81             sign = ((i+j)%2==0)? 1: -1;
82
83             // Interchanging rows and columns to get the
84             // transpose of the cofactor matrix
85             adj[j][i] = (sign)*(determinantOfMatrix(size, temp, size -1))
86             ;
87         }
88     }
89
90 // Function to calculate and store inverse, returns false if
91 // matrix is singular
92 void inverse_mat(int size, double A[size][size], double inverse[
93                 size][size])

```

```

94 // Find determinant of A[][]
95 double det = determinantOfMatrix(size , A, size);
96 if (det == 0.0)
97 {
98     //print( "Singular matrix, can't find its inverse");
99 }
100
101 // Find adjoint
102 double adj[size][size];
103 adjoint(size , A, adj);
104
105 // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
106 for (int i=0; i<size; i++)
107     for (int j=0; j<size; j++)
108         inverse[i][j] = adj[i][j]/det;
109 }
110
111 void transpose(int r , int c , double original[r][c] , double
112     trans[c][r] )
113 {
114     for(int i=0; i<r; ++i)// was r
115         for(int j=0; j<c; ++j)//was c
116             {
117                 trans[j][i] = original[i][j];
118             }
119 }
120 //Number of columns in A must equal number of rows in B
121 void mult_mat(int r1, int c1, int c2 , double A[r1][c1] , double B
122     [c1][c2] , double result[r1][c2] )
123 {
124     float sum = 0.0;
125     for(int i=0; i<r1; i++){
126         for(int j=0; j<c2; j++){
127             for(int k=0; k<c1; k++)
128                 {
129                     sum = sum + A[i][k]*B[k][j];
130                 }
131             result[i][j] = sum;
132             sum = 0.0;
133         }
134     }

```

C.6 C-equivalent math-blocks

C.6.1 M_{28B}

```

1 //FFTshift (a.k.a M28B): do np.fft.fftshift() equivalent
2 /* N: dimension of initial. Assume width = height */

```

```

3 void fftshift(int N, float initial[N][N], float shifted[N][N]){
4     int i, k;
5     for(i = 0; i < N ; i++){
6         for (k = 0 ; k < N ; k++){
7             if(k<N/2 && i<N/2)//1st quadrant
8                 shifted[i][k] = initial[i + N/2][k + N/2];
9             if(k>=N/2 && i<N/2)//2nd Quadrant
10                shifted[i][k] = initial[i + N/2][k - N/2];
11            if(k<N/2 && i>=N/2)//3rd quadrant
12                shifted[i][k] = initial[i - N/2][k + N/2];
13            if(k>=N/2 && i>=N/2)//4th Quadrant
14                shifted[i][k] = initial[i - N/2][k - N/2];
15        }
16    }
17 }

```

C.6.2 $M_{24.25.26}$

```

1 //Consider the 2D as one long 1D array
2 void complex_mult_post_FFT(float real_1 [], float imag_1 [], float
   real_2 [] , float imag_2 [] , float res_real [], float res_imag
   [], int n){
3     int i;
4     for (i = 0; i < n ; i ++){
5         imag_1[i] = -imag_1[i]; //conjugate
6         res_real[i] = (real_1[i] * real_2[i]) - (imag_1[i] * imag_2[i]
   );
7         res_imag[i] = (imag_1[i] * real_2[i]) + (real_1[i] * imag_2[i]
   );
8     }
9 }

```

C.6.3 $M_{32.33.34.35.36}$

```

1 //std_dev (a.k.a. M32.33.34.35.36)
2 /* Std-deviation which has a constant mean, this mimics the fact
   that a SAT will be used to determine the mean, which will have
   a small footprint on runtime */
3 float fake_std_dev_2D(int HEIGHT, float arr[HEIGHT][HEIGHT]){
4     int i, k;
5     float fake_mean = 5; //see comment above function name
6     float sum = 0;
7     float std_dev ;
8
9
10    //Determine pix value minus mean with sum
11    for(i = 0; i < HEIGHT ; i++){
12        for(k = 0; k < HEIGHT ; k++){
13            sum = sum + (arr[i][k] - fake_mean) * (arr[i][k] - fake_mean
   );
14        }
15    }
16    sum = sum/(HEIGHT*HEIGHT);

```

```

17 std_dev = (float)sqrt((float)sum);
18 return std_dev;
19 }

```

C.6.4 $M_{22_23_27}$

This FFT code is available at [60]:

```

1 /* Copyright (c) 2017 Project Nayuki. (MIT License)
2 * https://www.nayuki.io/page/free-small-fft-in-multiple-languages
3 */
4 bool Fft_transformRadix2(float real[], float imag[], size_t n) {
5     // Length variables
6     bool status = false;
7     int levels = 0; // Compute levels = floor(log2(n))
8     for (size_t temp = n; temp > 1U; temp >>= 1)
9         levels++;
10    if ((size_t)1U << levels != n)
11        return false; // n is not a power of 2
12
13    // Trigonometric tables
14    if (SIZE_MAX / sizeof(float) < n / 2)
15        return false;
16    size_t size = (n / 2) * sizeof(float);
17    float *cos_table = malloc(size);
18    float *sin_table = malloc(size);
19    if (cos_table == NULL || sin_table == NULL)
20        goto cleanup;
21    for (size_t i = 0; i < n / 2; i++) {
22        cos_table[i] = cos(2 * M_PI * i / n);
23        sin_table[i] = sin(2 * M_PI * i / n);
24    }
25
26    // Bit-reversed addressing permutation
27    for (size_t i = 0; i < n; i++) {
28        size_t j = reverse_bits(i, levels);
29        if (j > i) {
30            float temp = real[i];
31            real[i] = real[j];
32            real[j] = temp;
33            temp = imag[i];
34            imag[i] = imag[j];
35            imag[j] = temp;
36        }
37    }
38
39    // Cooley-Tukey decimation-in-time radix-2 FFT
40    for (size_t size = 2; size <= n; size *= 2) {
41        size_t halfsize = size / 2;
42        size_t tablestep = n / size;
43        for (size_t i = 0; i < n; i += size) {
44            for (size_t j = i, k = 0; j < i + halfsize; j++, k +=
tablestep) {

```



```

45     size_t l = j + halfsize;
46     float tpre = real[l] * cos_table[k] + imag[l] *
sin_table[k];
47     float tpim = -real[l] * sin_table[k] + imag[l] *
cos_table[k];
48     real[l] = real[j] - tpre;
49     imag[l] = imag[j] - tpim;
50     real[j] += tpre;
51     imag[j] += tpim;
52 }
53 }
54 if (size == n) // Prevent overflow in 'size *= 2'
55 break;
56 }
57 status = true;
58
59 cleanup:
60 free(cos_table);
61 free(sin_table);
62 return status;
63 }
64
65 static size_t reverse_bits(size_t x, int n) {
66     size_t result = 0;
67     for (int i = 0; i < n; i++, x >>= 1)
68         result = (result << 1) | (x & 1U);
69     return result;
70 }

```

C.7 Timing C-code

In order to determine how many clock cycles a C-function takes, a timer is used: ‘XTime_GetTime()’¹, it gets the time from the ‘Global Timer Register’. This register increments its value every second CPU clock cycle.

All functions timed, use the following compile flags and directives: “-Wall -O3 -c -fmessage-length=0 -MT”\$@” -mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard ” where:

- ‘-O3’ ensures the most optimised code according to the compilers ability is compiled
- ‘-mfloat-abi=hard’ along with ‘-mfpu=vfpv3’ ensures that the relevant floating math instructions are generated, such that the relevant floating point register set are selected²

¹Function in `xtime.l.c`, part of the board-support package.

²The data is forwarded to the relevant FPU by writing to their register, different versions (vfpv3) of such FPU’s have different number and type of registers, due to different functionality.

- note the lack of debug directives, thus no debug overhead is added, that's why the UART link is required, to read out the answer.
- note the lack of profiling, as this actually adds overhead to the CPU

The general timing of functions uses the following methodology: unless stated otherwise, make use of a 10,000 iteration for loop, to reduce the overhead of the timer. Furthermore, the test is repeated five times, and the *average* of these runs is documented as the final value. Unless otherwise stated the variance from the 5 runs is negligible. All these runs are performed on the hardware itself, using the 'Launch on Hardware (GDB)' option. Finally, since no debug code is added, the value is sent out via UART to the SDK terminal:

```

1  XTime_GetTime(&clkStart)
2  for ( i = 0 ; i < resolution ; i ++){
3      function (test_paramater);
4  }
5  XTime_GetTime(&clkEnd)
6  cpu_clk_cycles = 2*(clkEnd - clkStart);
7  //A UART send to terminal
8  xil_printf("Output took %lld clock cycles.\n", cpu_clk_cycles);
9

```

C.8 FSM-streaming ability

C.8.1 Standard streaming operation

The standard FSM-Streaming functionality is explained in terms of an example, aided by a waveform diagram (Figure C.4) with its block diagram layout (Figure C.5), lastly the functionality is described by the state transition diagram in Figure C.6.

Consider a typical math unit where the WORK_GROUP size is 16, and every data-element in the WORK_GROUP is multiplied by N^2 . Let FSM-Streaming already be waiting for data, in the 'buff_stall' status. Data is now streamed into the input buffer at a throughput rate of 1 data-element every 3 clock-cycles, a typical data-rate in this project. A slightly longer delay is induced between the 2nd last and last data-element to show the independence from various data rates. In this example the 'output_buffer_nearly_full' is toggled for a short time to mimic a full output buffer scenario, causing FSM-Streaming to go into a 'buffer-stall' state.

The following is shown in Figure C.4:

1. 'Data_last_in' signal, which is initially asserted by FSM-Streaming, propagating through the Math-Units input to its output – 'Data_last_out'. This signal notifies FSM-Streaming that the full WORK_GROUP has been operated upon.

2. The input buffer status awareness (FIFO Input), making use of ‘empty’ and ‘almost_empty’ signals. One would think only either ‘empty’ or ‘almost_empty’ signal is required. This is not the case since using FWFT buffers. FSM-Streaming can only respond to any signal on the next clock cycle, thus only making use of ‘empty’ signal would read a non-value from the buffer, thus the requirement for the ‘almost_empty’ signal. Furthermore the ‘empty’ signal is required for the very last data-point in the Input Buffer, where ‘almost_empty = 1’ and ‘empty = 0’.
3. Output buffer awareness, see FIFO Output’s ‘buff_out_almost_full’ signal, forcing FSM-Streaming to go into ‘buff-stall’ status, once again the ‘almost_full’ signal is used due to the nature of FSM-Streaming.
4. FSM-Streaming’s counter, ‘input_buff_ping’ keeping count of the data-elements which have been read into the math unit, starts counting at 1. This is due to the FWFT nature, and the way that logic is implemented. Note how ‘current_state’ transitions to ‘Tail’ after ‘input_buff_ping’ equals the ‘WORK_GROUP’ value of 16.
5. FIFO Input’s ‘data_count’, showing that the full data-set (size of WORK_GROUP) was operated on.

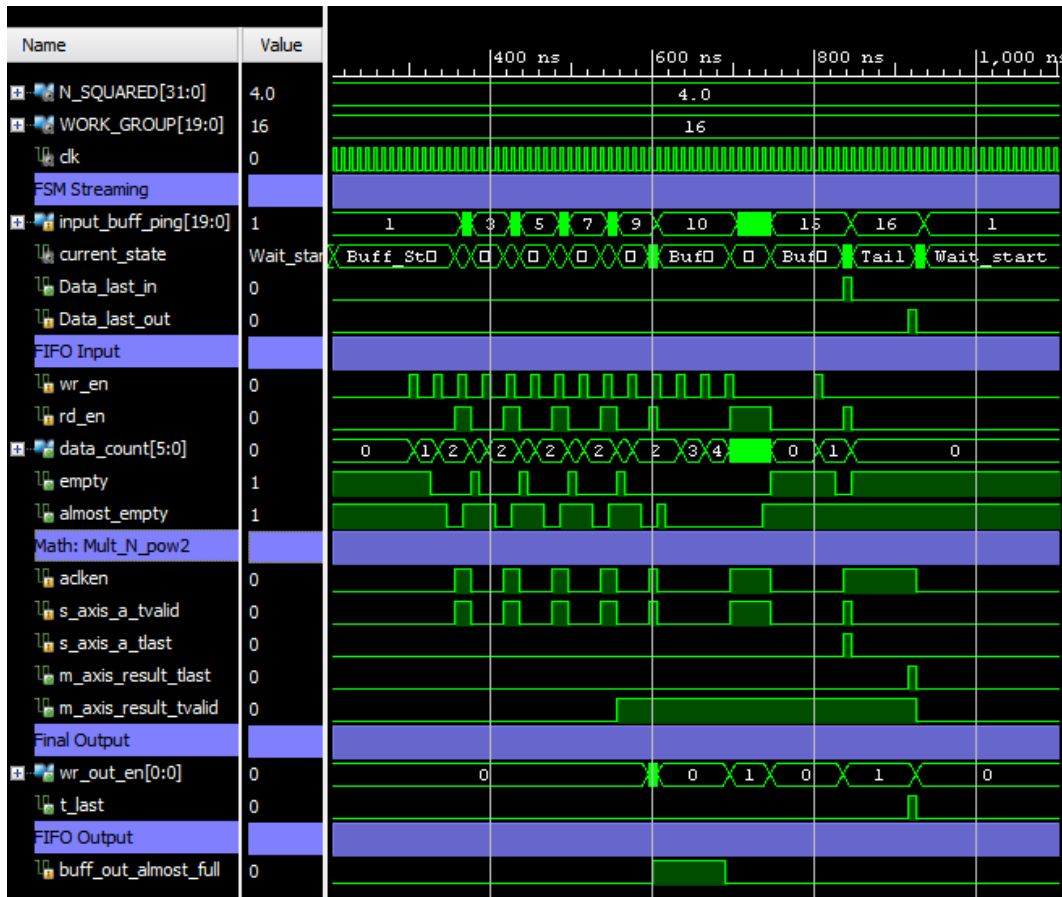


Figure C.4: Standard FSM-Streaming operation via wave diagram.

Figure C.5 is the typical layout out of a streaming math-block. Lastly Figure C.6 shows the state-transition-diagram of FSM-Streaming. The complexity arises from the various input-data-rates which need to be accommodated for, along with the output/input buffer status checking.

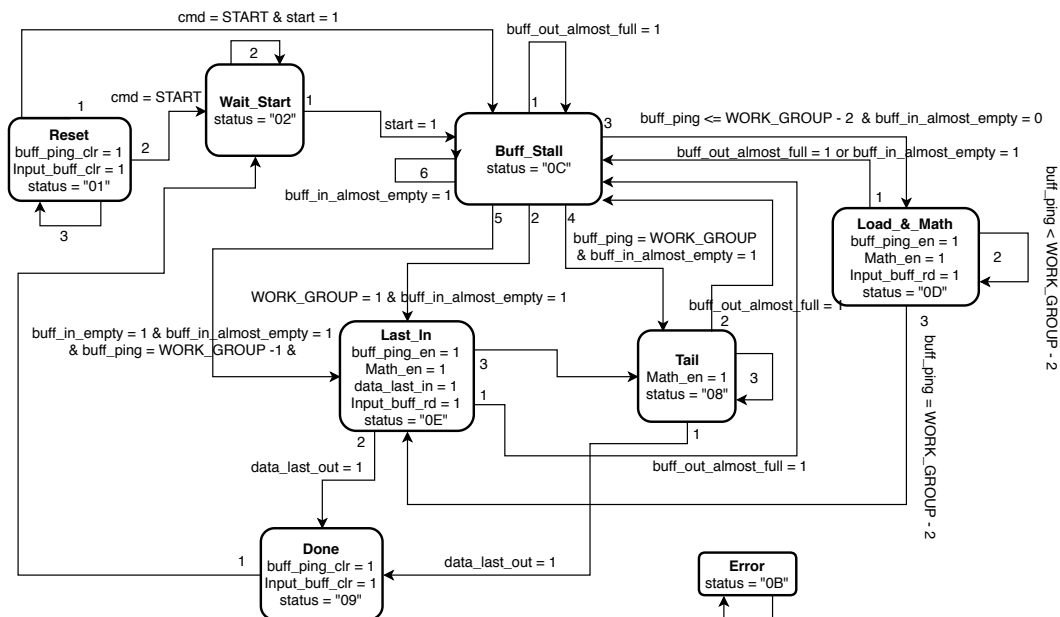


Figure C.6: The FSM-Streaming state transition diagram: all states can enter ‘Reset’ (first priority) via any state. The numbering indicates order of priority (lower has higher priority). Erroneous states transition to ‘Error’ state. Output control signals are zero unless otherwise stated in specific state.

C.8.2 Versatility: float-fix-float

At the time of development, the initial design was put together on the MicroZed 7010 board. It has roughly half the amount of fabric resources which the 7020 has. This meant, for HW debugging to be effective, the full design had to fit onto the board, and for this to happen some optimisations had to take place.

Consider the math-block $M_{17,18}$, from a mathematical point of view, it makes sense to do its accumulation in fixed point, as you do not lose resolution since we accumulate pixels where the over-all pixel mean has been subtracted and squared ($M_{14,15}$). Furthermore, the reduction in clock cycles is (from 23 for floating point at max-frequency to 2 for fix-point at max frequency, a reduction of 21 clock-cycles). Lastly, from a resource perspective, a 48bit unsigned accumulator uses $1 \times$ DSP48 (instead of 5 for floating point).

Consider the standard streaming math-block Figure C.5, now examine Figure C.7 which has extra IP blocks integrated to allow for fix-point. Namely Xilinx’s float-to-fix and fix-to-float IP-cores. They are highlighted as green, furthermore some simple logic needs to be added since Xilinx’s fix-point math is not implemented in the same streaming protocol as the floating point IP-cores (these are in red).

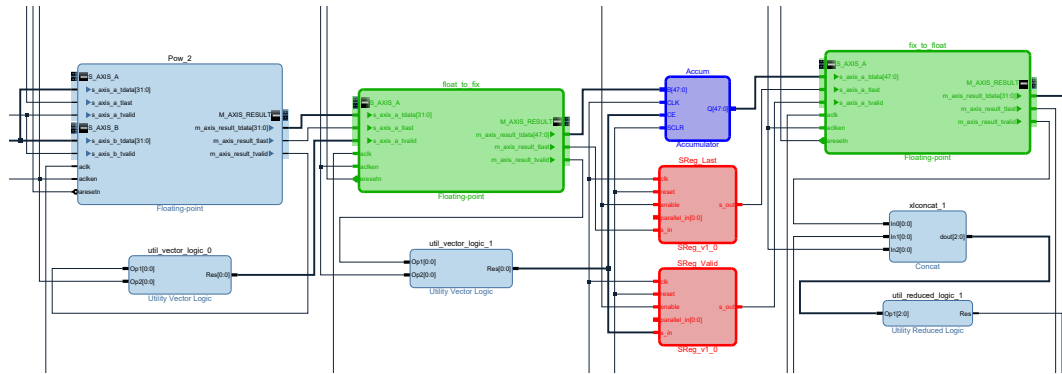


Figure C.7: The normal FSM-streaming math-block with slight adjustments. Standard floating-point math-unit ‘Pow_2’ on the left (light blue), ‘float_to_fix’ data converter on the left (green), fix-point accumulator (dark blue), extra logic to facilitate ‘data_valid’ and ‘data_last’ control signals (red), lastly converting back to floating-point (green on the right).

Consider the waveform diagram C.8, it shows how SReg_Valid and SReg_Last (red signals) incorporate the appropriate logic for the ‘last’ and ‘valid’ signals. Besides this logic, not much else is required to incorporate the fix-to-float logic.



Figure C.8: $M_{17.18}$ waveform showing how simple it is to add logic to enable certain fix-point math operations.

C.8.3 Versatility: lower throughput

Consider $M_{19.20.21}$, its WORKGROUP size is 1, therefore a very-low throughput math-block may be implemented since the Grey branch, which $M_{19.20.21}$ forms part of is by no means the bottleneck in the system. Therefore, the maximum resource savings are desired for lower throughput. Thankfully, Xilinx's floating point IP-core has a AXI-blocking-mode, this adds a 't_ready' signal, so that the previous math-unit knows whether or not it may send its data out. This in tandem with a 'cycles/operation' option, which reduces the throughput in favour of less resource usage, is utilised to create a low-throughput low-resource math-block as in Figure C.9 and evidenced in Figure C.10 waveform diagram.

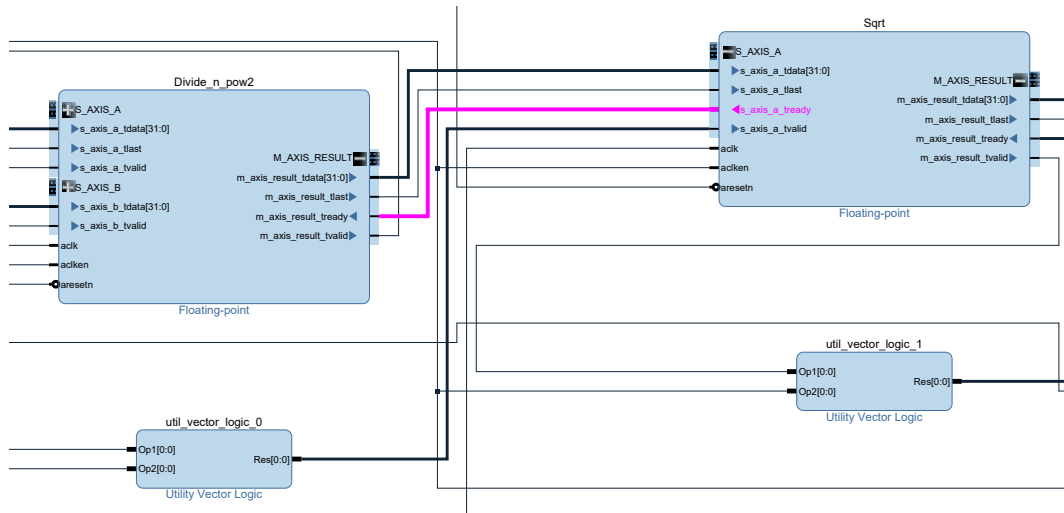


Figure C.9: $M_{19_20_21}$ block diagram showing how simple it is to swap out high-throughput non-blocking math IP-cores for low-throughput ones to reduce resource usage. Note the pink 't_ready' signal required for the blocking streaming protocol.

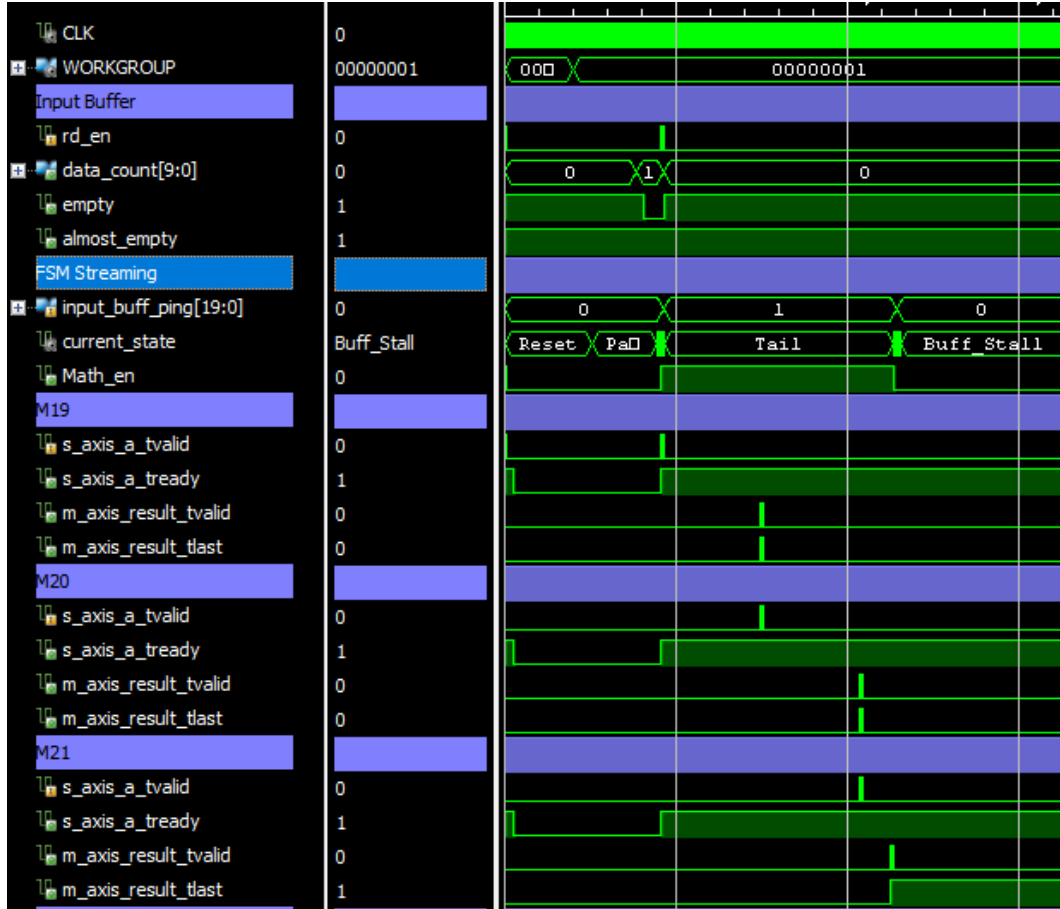


Figure C.10: $M_{19,20,21}$ waveform, note how FSM Streaming is in ‘Tail’ state significantly long (in comparison to other waveforms) due to the low-throughput, and how the logic of ‘t.last’ progresses through the various math-units from 19 to 21.

C.9 Theoretical case study: SPITE energy

Consider a theoretical scenario. A CubeSat with a 40Whr battery [71] (E_{battery}), which is to be discharged by 40% ($\sigma_{\text{Discharge_factor}} = 0.4$), using a power system with 65% efficiency ($\sigma_{\text{power_system_factor}} = 0.65$), with a energy³ budget where 1% is allocated to SPITE ($\sigma_{\text{SPITE_factor}} = 0.01$). This means, that the following number of SPITE calculations may be performed ($\#_{\text{SPITE}}$) in such a scenario:

$$\#_{\text{SPITE}} = \frac{E_{\text{battery}} \times 3600 \sigma_{\text{Discharge_factor}} \sigma_{\text{power_system_factor}} \sigma_{\text{SPITE_factor}}}{E_{\text{Operation}}}$$

³Usually referred to as power budget, however working with energy instead simplifies the problem, as a power percentage assumes that it must be able to run constantly at a certain budget.

where $E_{\text{Operation}}$ is either $E_{\text{algorithm}}$ or E_{platform} .

Since the amount of energy available to perform SPITE is 374J, this results in the following number SPITE operations which may be calculated, shown in Table C.1:

Table C.1: Theoretical study: the number of SPITE operations which are possible, if run consecutively for a certain energy, as laid out in case study.

| Operation | #SPITES | Consecutive time[s] | Energy [J] |
|------------------|----------------|----------------------------|-------------------|
| Algorithm | 49671 | 739 | 374 |
| Platform | 7827 | 116 | 374 |

On one battery cycle, if only the IP is to be added to a pre-existing MicroZed 7020 satellite, the algorithm could run consistently for over 12 minutes⁴, where if the MicroZed 7020 is integrated into the satellite, it could run for just under two minutes.

⁴Clearly this assumes that the a similar platform power is to be expected as we determined in our tests.

References

- [1] City of cape town, egisviewer. <https://citymaps.capetown.gov.za/EGISViewer>, (accessed 10.06.2017).
- [2] *Zynq-7000 SoC*. Xilinx, July 2018. V1.12.2.
- [3] *Data Sheet Nvidia Tegra K1 Series Processors with Kepler Mobile GPU for Embedded Applications*. Nvidia Corporation, February 2015. Ver. 2.
- [4] Lüdemann, J., Barnard, A. and Malan, F.D.: Sub-pixel image registration on an embedded satellite platform. In: *69th International Astronautical Congress*. 2018.
- [5] National Academies of Sciences, E. and Medicine: *Achieving Science with CubeSats: Thinking Inside the Box*. The National Academies Press, Washington, DC, 2016. ISBN 978-0-309-44263-3.
- [6] Malan, D.F., Steyn, H.W., Wiid, K., Burger, H. and Visagie, L.: The development of “nsight-1” – earth observation and science in 2u. In: *Proceedings of the 31st Annual AIAA/USU Conference on Small Satellites*. 2017.
- [7] *Gecko Imager*. Space Advisory Company. <http://www.spaceadvisory.com/products/payloads/gecko-imager/>, (accessed 25.8.2018).
- [8] Banks, S.P.: *Signal Processing, Image Processing and Pattern Recognition*, chap. Picture Enhancement, Restoration and Analysis. Prentice Hall International (UK) Ltd, 1990. ISBN 0-13-812587-2.
- [9] Jain, P.M. and Shandliya, V.K.: A review paper on various approaches for image mosaicing. *International Journal of Computational Engineering Research*, vol. 3, no. 4, pp. 106–109, 2013.
- [10] S. Susan Young, Ronald G. Driggers, Eddie L. Jacobs: *Signal Processing and Performance Analysis for Imaging Systems*, chap. Super-Resolution. Artech House. ISBN 978-1-59693-287-6.
- [11] Gonzales, R.C. and Woods, R.E.: *Digital Image Processing*. 3rd edn. Pearson Prentice Hall, 2010. ISBN 0-13-234563-3.
- [12] James R. Wertz, Wiley J. Larson: *Space Mission Analysis and Design*, chap. Thermal. 3rd edn. Microcosm Press and Kluwer Academic Publishers, 1999. ISBN 1-881883-20-8.

- [13] Lee, J., Kim, E. and Shin, K.G.: Design and management of satellite power systems. In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pp. 97–106. IEEE, 2013.
- [14] Sensitivity of ccd cameras. Tech. Rep., Oxford Instruments. <http://www.andor.com/learning-academy/sensitivity-of-ccd-cameras-key-factors-to-consider>, (accessed 10.04.2018).
- [15] Barnard, A.: Space radiation: Environment and effects on electronic system. University Lecture, 2016.
- [16] James R. Wertz, David F. Everett, Jeffery J. Puschell: *Space Mission Engineering: The New SMAD*, chap. The Space Environment. Microcosm Press, 2011. ISBN 978-1-881-883-15-9.
- [17] Wirthlin, M.: High-reliability fpga-based systems: space, high-energy physics, and beyond. *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, 2015.
- [18] Azambuja, J.R., Nazar, G., Rech, P., Carro, L., Kastensmidt, F.L., Fairbanks, T. and Quinn, H.: Evaluating neutron induced see in sram-based fpga protected by hardware-and software-based fault tolerant techniques. *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4243–4250, 2013.
- [19] single-event-latch-up (sel). <https://www.jedec.org/standards-documents/dictionary/terms/single-event-latch-sel>, (accessed 17.11.2018).
- [20] *Call for a sustainable future in space*. European Space Agency, 2017. http://www.esa.int/Our_Activities/Operations/Space_Debris/Call_for_a_sustainable_future_in_space, (accessed 10.4.2018).
- [21] Aggarwal, P.K.: Dynamic (vibration) testing: Design-certification of aerospace system. 2010.
- [22] de Segovia, J.: Physics of outgassing. Tech. Rep., Instituto de Fisica Aplicada, Madrid, Spain, 2000.
- [23] Miria M. Finckenor, Kim K. de Groh: A researcher’s guide to: International space station, space environmental effects. Tech. Rep., NASA, 2015. Chapter: Aspects of the Space Environment.
- [24] Ramo, S., Whinnery, J.R. and van Duzer, T.: 3rd edn. ISBN 3257227892.
- [25] Wolf, M.: *Computers as components*. 3rd edn. Morgan Kaufman, Waltham, MA, 2012.
- [26] Zitova, B., Flusser, J. and Šroubek, F.: Image registration: a survey and recent advances. In: *Proc. of the 12th IEEE Int. Conf. on Image Processing (ICIP 2005)*. 2005.
- [27] Xiong, Z. and Zhang, Y.: A critical review of image registration methods. *International Journal of Image and Data Fusion*, vol. 1, no. 2, pp. 137–158, 2010.

- [28] Kupfer, B., Netanyahu, N.S. and Shimshoni, I.: An efficient sift-based mode-seeking algorithm for sub-pixel registration of remotely sensed images. *IEEE Geoscience and Remote Sensing Letters*, vol. 12, no. 2, pp. 379–383, 2015.
- [29] Durgam, U.K., Paul, S. and Pati, U.C.: Surf based matching for sar image registration. In: *Electrical, Electronics and Computer Science (SCEECS), 2016 IEEE Students' Conference on*, pp. 1–5. IEEE, 2016.
- [30] Jia, X., Guo, K., Wang, W., Wang, Y. and Yang, H.: Sri-surf: A better surf powered by scaled-ram interpolator on fpga. In: *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pp. 1–8. IEEE, 2016.
- [31] Vourvoulakis, J., Kalomiros, J. and Lygouras, J.: Fully pipelined fpga-based architecture for real-time sift extraction. *Microprocessors and Microsystems*, vol. 40, pp. 53–73, 2016.
- [32] Szelinski, R.: *Computer Vision: Algorithms and Applications*. Springer, 2010.
- [33] De Castro, E. and Morandi, C.: Registration of translated and rotated images using finite fourier transforms. *IEEE Transactions on pattern analysis and machine intelligence*, , no. 5, pp. 700–703, 1987.
- [34] Tian, Q. and Huhns, M.N.: Algorithms for subpixel registration. *Computer Vision, Graphics, and Image Processing*, vol. 35, no. 2, pp. 220–233, 1986.
- [35] Lathi, B.P. and Ding, Z.: *Modern Digital and Analog Communication Systems*. International 4th edn. Oxford University Press, 2010. ISBN 978-0-19-538493-2.
- [36] Erturk, S.: Translation, rotation and scale stabilisation of image sequences. *Electronics Letters*, vol. 39, no. 17, pp. 1245–1246, 2003.
- [37] Wilson, C.A. and Theriot, J.A.: A correlation-based approach to calculate rotation and translation of moving cells. *IEEE Transactions on Image Processing*, vol. 15, no. 7, pp. 1939–1951, 2006.
- [38] Lewis, J.P.: Fast normalized cross-correlation. Tech. Rep., Industrial Light and Magic, 1995.
- [39] Debella-Gilo, M. and Käab, A.: Sub-pixel precision image matching for measuring surface displacements on mass movements using normalized cross-correlation. *Remote Sensing of Environment*, vol. 115, no. 1, pp. 130–142, 2011.
- [40] Rais, M.: *Fast and accurate image registration. Applications to on-board satellite imaging*. Ph.D. thesis, Université Paris-Saclay, 2016.
- [41] Albinet, M., Camarero, R., Isnard, M., Poulet, C. and Perret, J.: Improving multispectral satellite image compression using onboard subpixel registration. In: *Satellite Data Compression, Communications, and Processing IX*, vol. 8871, p. 887106. International Society for Optics and Photonics, 2013.

- [42] Mobasherya, M. and Dastfard, M.: Separation of ikonos sensor's electronic noise from atmospheric induced effects. *ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, , no. 3, pp. 117–121, 2013.
- [43] Coherent noise. Tech. Rep., U.S. Department of the Interior. <https://landsat.usgs.gov/coherent-noise>, (accessed 03.06.2017).
- [44] Topic 5: Noise in images. Tech. Rep., School of Physics, University of Edinburgh, 2007. <https://www2.ph.ed.ac.uk/~wjh/teaching/dia/documents/noise.pdf>, (accessed 03.06.2017).
- [45] McFee, C.: An introduction to ccd operation. <http://www.mssl.ucl.ac.uk/wwwdetector/ccdgroup/opttheory/ccdoperation.html>, (accessed 01.06.2017).
- [46] Keep the noise down. low noise: An integral part of high-performance ccd (hccd) camera systems. Tech. Rep., Photometrics, 2010. <https://www.photometrics.com/resources/technotes/pdfs/snr>, (accessed 01.06.2017).
- [47] Dunlap, J.C.: Characterization and modeling of nonlinear dark current in digital imagers. 2014.
- [48] Peebles, P.Z.: *Probability, Random Variables and Random Signal Principles*. 4th edn. McGraw-Hill, 2001. ISBN 1-290-07-118181-4.
- [49] Walter schottky. <https://www.famousscientists.org/walter-schottky>, (accessed 06.06.2017).
- [50] Welsch, B.: image-registration docs. <https://image-registration.readthedocs.io/en/latest/>, (accessed 20.03.2017).
- [51] Welsch, B.: Cross-correlation taylor, 2006. http://solarmuri.ssl.berkeley.edu/~welsch/public/software/cross_cor_taylor.pro, (accessed 20.03.2017).
- [52] Proakis, J.G. and Manolakis, D.K.: *Digital Signal Processing*. International 4th edn. Pearson, 2014. ISBN 1-292-02573-5.
- [53] Crow, F.C.: Summed-area tables for texture mapping. In: *ACM SIGGRAPH computer graphics*, vol. 18, pp. 207–212. ACM, 1984.
- [54] Nvidia corporation: Tegra mobile devices. <https://www.nvidia.com/object/tegra-phones-tablets.html>, (accessed 10.9.2018).
- [55] Chandrakasan, A.P. and Brodersen, R.W.: Minimizing power consumption in digital cmos circuits. *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.
- [56] Milluzzi, A. and George, A.: Exploration of tmr fault masking with persistent threads on tegra gpu socs. In: *Aerospace Conference, 2017 IEEE*, pp. 1–7. IEEE, 2017.
- [57] Vijay, M. and Mittal, R.: Algorithm-based fault tolerance: a review. *Microprocessors and Microsystems*, vol. 21, no. 3, pp. 151–161, 1997.

- [58] *Fast Fourier Transform v9.0, LogiCORE IP Product Guide*. Xilinx, October 2017.
- [59] Berg, M.: Fpga design strategies for the space radiation environment. Radiation Effects and Analysis Group, NASA Goddard Space Flight Center Muniz Engineering and Technologies.
- [60] Project nayuki. (mit license): Free fft and convolution. <https://www.nayuki.io/page/free-small-fft-in-multiple-languages>, (accessed 18.05.2018).
- [61] Harnish, M.: Using your c compiler to exploit neon advanced simd. Tech. Rep., Duolos, 2010.
- [62] *Current Sensing Circuit Concepts and Fundamentals*. Microchip, July 2010. DS01332B.
- [63] Borman, S. and Stevenson, R.: Spatial resolution enhancement of low-resolution image sequences-a comprehensive review with directions for future research. *Lab. Image and Signal Analysis, University of Notre Dame, Tech. Rep*, 1998.
- [64] Nunez-Yanez, J. and Beldachi, A.: Run-time power and performance scaling with cpu-fpga hybrids. In: *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pp. 55–60. IEEE, 2014.
- [65] Chen, Y.-L., Chang, M.-F., Liang, W.-Y. and Lee, C.-H.: Performance and energy efficient dynamic voltage and frequency scaling scheme for multicore embedded system. In: *Consumer Electronics (ICCE), 2016 IEEE International Conference on*, pp. 58–59. IEEE, 2016.
- [66] Attia, K.M., El-Hosseini, M.A. and Ali, H.A.: Dynamic power management techniques in multi-core architectures: A survey study. *Ain Shams Engineering Journal*, 2015.
- [67] Gelb, A.: *Applied optimal estimation*. MIT press, 1974.
- [68] Crassidis, J.L. and Junkins, J.: *Optimal Estimation of Dynamic Systems*. Chapman and Hall/CRC, 2004. ISBN 1-58488-391-X.
- [69] GeeksfoGeeks: Determinant of a matrix, . <https://www.geeksforgeeks.org/determinant-of-a-matrix/>, (accessed 20.09.2017).
- [70] GeeksfoGeeks: Adjoint and inverse of a matrix, . <https://www.geeksforgeeks.org/adjoint-inverse-matrix/>, (accessed 20.09.2017).
- [71] Clyde space: 40whr cubesat battery. <https://www.clyde.space/products/49-40whr-cubesat-battery>, (accessed 08.07.2018).