# Accelerating Software Radio Astronomy FX Correlation with GPU and FPGA Co-processors

Submitted to the Department of Electrical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering
Final Submission (with corrections)

Andrew Woods
University of Cape Town



Supervisor:
Prof. Michael Inggs

Co-Supervisor:
Dr. Alan Langman

October 28, 2010

## Plagiarism Declaration

I know the meaning of plagiarism and declare that all the work in this document, save for that which is properly acknowledged, is my own.

# Abstract

This thesis attempts to accelerate compute intensive sections of a frequency domain radio astronomy correlator using dedicated co-processors. Two co-processor implementations were made independently with one using reconfigurable hardware (Xilinx Virtex 4LX100) and the other uses a graphics processor (Nvidia 9800GT). The objective of a radio astronomy correlator is to compute the complex valued correlation products for each baseline which can be used to reconstruct the sky's radio brightness distribution. Radio astronomy correlators have huge computation demands and this dissertation focuses on the computational aspects of correlation, concentrating on the X-engine stage of the correlator.

Although correlation is an extremely compute intensive process, it does not necessarily require custom hardware. This is especially true for older correlators or VLBI experiments, where the processing and I/O requirements can be satisfied by commodity processors in software. Discrete software co-processors like GPUs and FPGAs are an attractive option to accelerate software correlation, potentially offering better FLOPS/watt and FLOPS/$ performance.

In this dissertation we describe the acceleration of the X-engine stage of a correlator on a CUDA GPU and an FPGA. We compare the co-processors' performance with a CPU software correlator implementation in a range of different benchmarks. Speedups of 7x and 12.5x were achieved on the FPGA and GPU correlator implementations respectively.

Although both implementations achieved speedups and better power utilisation than the CPU implementation, the GPU implementation produced better performance in a shorter development time than the FPGA. The FPGA implementation was hampered by the development tools and the slow PCI-X bus, which is used to communicate with the host. Additionally, the Virtex 4 LX100 FPGA was released two years before the Nvidia G80 GPU and so is more behind the current technologies. However, the FPGA does have an advantage in terms of power efficiency, but power consumption is only a concern for large compute clusters. We found that using GPUs was the better option to accelerate small-scale software X-engine correlation than the Virtex 4 FPGA.

# Contents

# Acknowledgements

I would like to acknowledge a number of people who have helped me considerably during my project:

Firstly I would like to thank my supervisor Prof. Michael Inggs, who I was fortunate enough to have as my undergraduate supervisor. Subsequently he was brave enough to supervise me for my master's study and reckless enough to employ me throughout this last year. His wise and benevolent guidance kept my work on track and always knew when to apply the pressure. Prof. has been a superb mentor and who has allowed me to learn and grow considerably during the past few years under his guidance.

To Dr. Alan Langman, who co-supervised my work and donated much of his time for the technical guidance of this thesis. Alan was always available online, contactable at any hour, to offer excellent advice on technical and other life issues. His deep understanding of technology and excitement and passion for engineering, especially the latest and greatest gadgets, is an inspiration to my career, which I greatly admire.

To Peter "Polar Bear" McMahon, who was always available to give pragmatic and invaluable advice, despite completing his two simultaneous MSc. Degrees. Polar has the maturity and wisdom far beyond his mortal age (but he doesn't sleep, so has lived twice as long :P).

To my sister Keri, who patiently helped me translate my nonsensical language back into English. She always did this without complaint, even though I my requests for help usually came either well into the night or the early hours of the morning.

To my family, Mom, Dad and Kristin, who were always supportive and encouraging, and gladly read through my dissertation, despite it not being in their line of work.

Thanks to KAT/SKA for the funding and allowing me to use their facilities. The entire KAT team were always supportive and interested in my work. Special thanks must go to Alan Langman, Marc Welz, David George, Jasper Horrell and Jason Manley, who went beyond the line of duty to help out.

Thanks go to Dr. Happy Sitole and Dr. Jeff Chen for allowing me to use the CHPC facilities and for the employment over the last year. Special thanks must got o Kevin Colville and Sebastian Wyngaard who were always willing to chat and offer advice. Housed at CHPC, is one of Prof. research groups, the Advanced Computer Engineering (ACE) lab, of which I was fortunate enough to be apart of. Mike Aitken, Andrew van der Byl, Jean-Paul da Conceicao, Jane Hewitson, Ray Hsieh, David Macleod, Arjun Radhakrishnan, Jason Salkinder and Nick Thorne were an awesome bunch of engineers and friends to work with.

Thanks to Prof. Inggs and Dr. Mark Parsons for facilitating the three-month research visit to EPCC at the University of Edinburgh to work on their FPGA compute cluster, "Maxwell". Thanks to Dr. Rob Baxter and James Perry for their guidance on their reconfigurable computer.

We met many amazing people at EPCC, some of which like Mario Antonioletti and Catherine Inglis, we still remain in contact with today.

Thanks to Alan Cantle who went out of his way to accommodate Polar and I for a week at the Nallatech office in Bristol.

Thanks to Adam Deller for his responsive and thorough input to the DiFX correlator, and to Walter Alef and Walter Brisken for suppling me with real world data.

Thanks to Chris Harris from UWA and his invaluable help with this GPU correlator.

To all my mates and girlfriend, who had to put up with my extended writeup and didn't twist my wrist when I declined the pub outings.

# Glossary

**Airy Disc** – The diffraction pattern resulting from a uniformly illuminated circular aperture, has a bright region in the center, known as the Airy disc which together with a series of concentric rings is called the Airy pattern.[1]

**Angular Resolution** The angular resolution of an aperture, is the smallest distance (angular) that two differentiable sources can be recorded.

**Aperture** – an aperture is a hole or an opening through which electromagnetic waves are admitted. [1]

**Arcminute** – A measurement of angle. There are 60 arcminutes in a degree and 60 arcseconds in an arcminute.

**Arithmetic Intensity** – The amount of data reuse. "the ratio of arithmetic operations to memory operations" [2].

**Astrometry** – The measurement of the positions,motions, and magnitudes of stars [3].

**Baseline** – Every antenna pair combination, can be represented as a vector which connects them, called a baseline.

**Block Ram** – On Xilinx FPGAs, block ram is dedicated two-port memory containing several kilobits of data.

**Computational Unit** – The most fundamental part of hardware that can perform arithmetic calculations (eg. FPGA's DSP).

**Control Hazard** – Branch in the pipeline which results in the pipeline stalling (interrupt the flow of the pipeline).

**Correlation Kernel** – see Correlation Matrix.

**Correlation Matrix** – We refer to the all the correlation baseline products for a certain time-slice and frequency as the correlation matrix or correlation kernel.

**Data Hazard** – Data Hazard refers to a situation where we refer to a result that has not yet been calculated. This will often introduce stalling in the processing pipeline [1]. eg. 001: a = b + c; 002: s = a + c;

**Diffraction** – refers to various phenomena associated with wave propagation, such as the bending, spreading and interference of waves passing by an object or aperture that disrupts the wave [1].

**CMAC** - Complex Multiply and Accumulate

**CMP** – Chip multiprocessor. When two or more microprocessors or microprocessor cores are fabricated on a single silicon die. All desktop processors today are chip multiple processors eg. Intel Core 2 Duo.

**CUDA** – Compute Unified Device Architecture created by Nvidia.

**FIFO** – First in First Out queueing system.

**FLOPS** – **FL**oating **P**oint **O**perations **P**er **S**econd.

**FPU** – Floating Point Unit.

**FX** - FX here refers to the order the correlation is performed. FX correlators do a multiplication in the Fourier domain, while XF correlators perform a convolution in the time domain.

**Far Field** – A very far distance from the receiver that even spherical radiation is received as a plane wave.

**GPU** – Graphics Processing Unit.

**GPGPU** – General Purpose on Graphics Processing Units.

**Geodesy** – the branch of mathematics dealing with the shape and area of the earth or large portions of it [3].

**Granularity** The granularity of the parallelism is a description of the smallest chunk of data that can be processed independently. If one were to execute the outer loop of a nested loop on separate processing elements this would be course-grained, likewise if the inner loop was distributed across processors this would be fine-grained.

**HPC** – High performance computing - a class of computing that solves problems requiring large amounts of computation.

**ICs** – Integrated Circuit.

**IPP** – Intel Performance Primitives Library.

**ISA** – Instruction Set Architecture.

**LUT** – Look Up Table; the fundamental memory or building block of FPGAs reconfigurable logic.

**MAC** – Multiply and Accumulate operation.

**MADD** – Multiply and Add operation.

**MUL** – Multiply operation.

**Moore's Law** – or Moore's Curve is the long-term trend in the history of computing hardware, in which the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years, first noted by Intel co-founder Gordon E. Moore [1].

**Processing Elements** – A group of one or more computational units that co-operate to produce an output to a particular algorithm (eg. Groups of DSP to create a correlation engine).

**SIMD** – Single Instruction Multiple Data.

**SIMT** – Nvidia's Cuda architecture that runs thousands of threads on hundreds of processing cores [2].

**SMP** – Shared Memory Processor.

**SSE** – (Intel's) Streaming SIMD Extensions.

**Scalar Processor (SP)** – One of the 8 ALUs on a CUDA GPU's Streaming Multiprocessor.

**Sensitivity** – a mesaure of the performance of a telescope, dish or array often measured in $m^2/K$. This determines how long it takes to observe a source of a particular flux. [4]

**Streaming Multiprocessor (SM)** – The fundamental vector processor on CUDA GPUs. The number of SMs on a CUDA GPU depends on the model and cost.

**Visibility** – Is the Fourier transform of the radio brightness distribution of the sky and is the desired output of a radio astronomy correlator.

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In this thesis we aimed to accelerate compute intensive sections of a software radio astronomy correlator using dedicated co-processors. Two co-processor implementations were developed using reconfigurable hardware (Xilinx Virtex 4LX100) and a graphics processor (Nvidia 9800GT). Radio astronomy telescopes require correlation to perform interferometric operations which allow them to do imaging and other applications. Because radio telescopes operate at high data rates, correlation is an extremely computationally intensive process. In this project we perform the correlation in the frequency domain, which is known as FX correlation. We focus mainly on the engineering aspects of accelerating software with co-processors, although an outline of the astronomy principles behind the correlator will briefly be discussed. In this chapter we provide a brief background to the project, identify the main objectives of the thesis and outline the contents of the rest of the thesis.



## 1.1 Background

Radio astronomy is a branch of observational astronomy that studies astronomical sources detectable in the radio spectrum. Measurement of the radio spectrum is one of the best tools astronomers have to reveal the structure and formation of the Universe. Digital Signal Processing technologies have contributed greatly to the success of radio astronomy and have had a profound influence on how modern-day radio telescopes have evolved.

Traditionally, radio telescopes employed a single large antenna [1], however modern large radio telescopes almost always consist of a number of individual antennas. These smaller dishes can be used together in an interferometric process called *aperture synthesis* - which emulates a larger antenna's response, producing much higher resolution results than could be achieved by a single antenna. Incredibly, an interferometric radio telescope can produce the same angular resolution as an antenna with a diameter of the array's longest baseline. Figure 1.1 shows a simplified two antenna interferometric telescope.

[1] eg. Lovell, GBT, Arecibo [4]

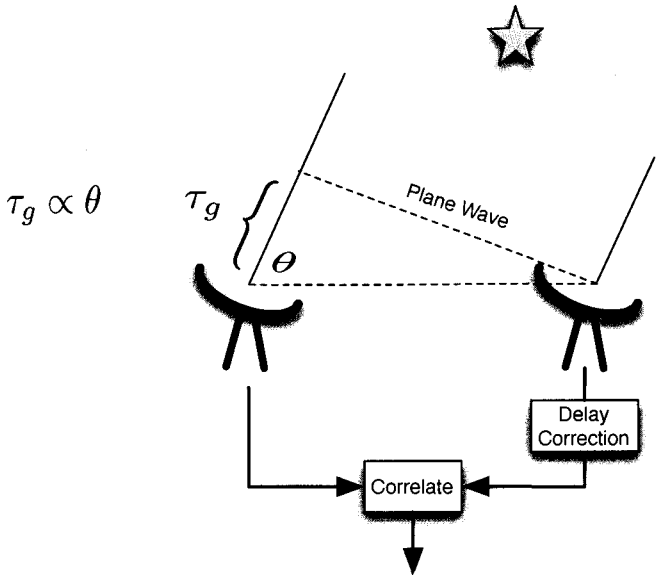



**Figure 1.1** – Diagrammatic Representation of an Interferometric Telescope. The spacing between the antenna introduces a delay $\tau_g$ into the system, which is corrected before correlation.

Interferometric radio telescopes use correlators to compute the cross-correlations of all antenna pair combinations in the array[2]. These complex valued correlation products [3] are used to emulate a larger antenna's response. Each baseline correlation represents a specific spectral response to the brightness function of a larger, synthesised aperture. The Fourier transformation of the brightness distribution of the synthesized aperture can be entirely reconstructed if there are enough baselines to cover the entire spectrum of the brightness distribution[4].

Radio astronomers want a telescope with as many baselines as possible; this number is limited by how many baselines the correlator can process, which is itself dependent on the processing technologies that it is built from. Because of the heavy reliance that radio interferometry has on digital signal processing, a large portion of a radio telescope's budget is spent on the correlator - often requiring custom hardware to maximise performance and power consumption efficiency. The correlator is one of the most computationally expensive operations of the radio astronomy telescope.

## 1.2 Software Correlation

Software correlation uses general purpose compute clusters to perform correlation. Although the latest telescopes require custom hardware, new generations of modern medium-sized general purpose compute clusters can feasibly be used to replace older custom correlator hardware. Software correlation is significantly more accessible and customisable than production hardware correlators. Because of the low cost of commodity clusters, some astronomy institutions are finding it more effective to use software correlation than to support old specialised correlator hardware.

[2]Each antenna pair combination can be represented by a vector which connects the two antennas together from a reference position. This is called a baseline.

[3]known as complex-visibilities

[4]This is an over simplification as there are a number of practical considerations that limit this.

Software correlation is increasingly becoming feasible due to the increased used of other commodity hardware in radio telescopes and the wealth of tools and libraries available in the software environment. Previously, the performance requirements of radio telescopes required almost all custom built hardware[5], but as the performance of commodity hardware has improved and the complexity of building custom hardware has increased, radio telescopes are shifting to use commodity hardware wherever possible. With commodity hardware comes standardised, well-documented interfaces which make software integration easier. The high-level software tools and the availability of optimised libraries allow software a significant reduction in Non Recoverable Engineer (NRE) costs. While software cannot offer the same performance as custom hardware, it is an ideal candidate for small to medium interferometers. [6]

The flexibility of software and the customization of commodity computer clusters opens up an interesting opportunity of employing co-processors to accelerate the demanding sections of correlation. The number of co-processor peripherals available and the high-speed and mature communication interfaces makes software correlation acceleration an exciting and increasingly researched topic and is the focus of this thesis.

The Distributed FX (DiFX) correlator is an example of a popular software correlator implementation [5] and served as an inspiration for the opportunities of software correlation.



## 1.3 Co-processor Software Correlator Acceleration

Although software correlation has many appealing attributes, CPU's architecture is not ideally suited to correlation [6]. New emerging markets, such as gaming and embedded systems, have grown remarkably in recent years, bringing with them their own processing technologies. Graphics processing units (GPUs) and Field Programable Gate Arrays (FPGAs), are ubiquitous in the gaming and embedded markets . These high volume markets have made high performance hardware affordable. Many HPC facilities are adding GPUs and FPGAs as co-processors to their existing CPU cluster infrastructure, which can be used to accelerate suitable applications under the control of the CPU (see Figure 1.2).

Processor architecture is heavily influenced by the applications that it runs. Different architectures use roughly the same number of transistors, but they are employed differently. CPUs dedicate a large proportion of available transistor area to on-board memory, important for desktop computing, but leaving fewer transistors for computational units. In contrast, GPU and FPGA's architecture is much more computation orientated[7]. Figure 1.3 shows a comparison between the peak computational performance of the different architectures.

A correlator's code profile has a close resemblance to game engines and embedded applications, more so than general software - the essence of the correlator's profile is a large number of calculations with relatively little branching in data flow. The similarities that game engines, embedded applications and correlation share, coupled with the growing support of GPUs and FPGAs in the HPC environment, justifies investigating GPUs and/or FPGAs for software correlator acceleration.

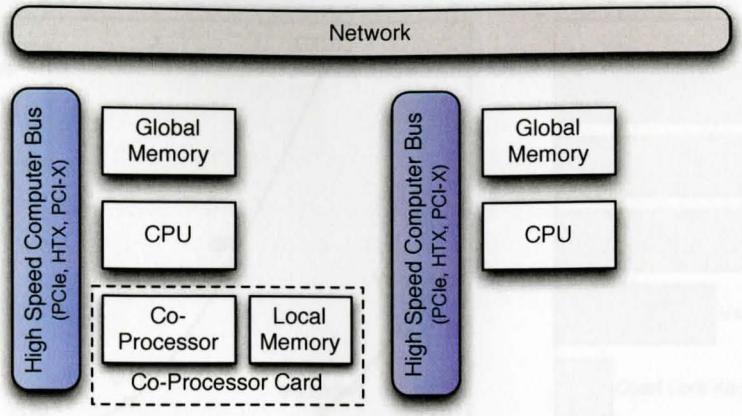



**Figure 1.2** – A network setup with a node that has a co-processor installed. Inspired by McMahon [7]

In this project we implemented two simple correlators, using FPGAs and GPUs independently. In this dissertation we discuss the design, implementation, performance and feasibility of the two co-processor correlators.



[5]this includes not only processors, but network interconnects, memory, storage etc
[6]Software correlation has also been implemented in large scale facilities such as LOFAR.
[7]FPGAs have the flexibility to be either memory or computation oriented.

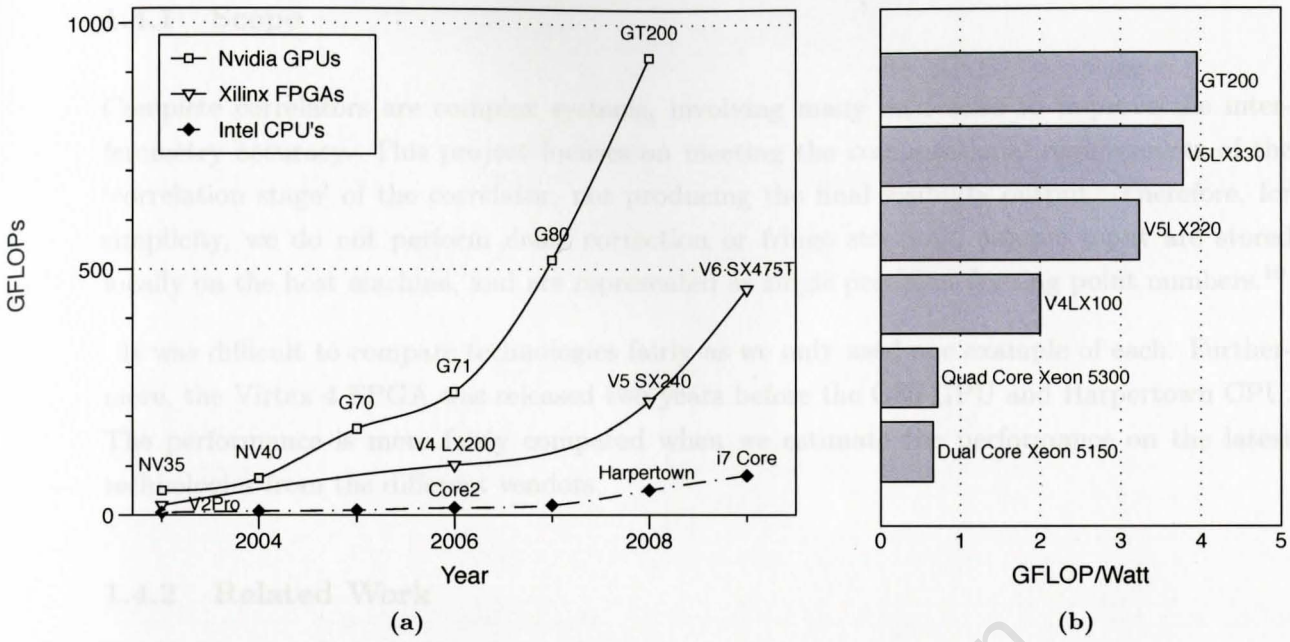



**Figure 1.3** – Comparison between different processing technologies. **(a)** showing the single precision floating point performance [8, 9, 10, 11, 2] and **(b)** showing the GFLOPs/watt of the different architectures [12, 13]. Note however that these are theoretical GFLOP performance figures and real world performance will vary considerably. Due to FPGA's reconfigurable data path, it is typically easier to achieve closer to its peak performance.

## 1.4 Project Objectives and Scope

In this project, we investigate using GPU and FPGA co-processors to accelerate a simple software correlator. The software correlator was created using the popular open-source software correlator, DiFX, as a foundation. The DiFX correlator is a complex software project. While it has thousands of lines of code, the heavy computation is contained in only a few lines. We found that DiFX's large code base complicated performance analysis and verification of our co-processor acceleration. This justified creating a simplified software correlator by preserving the compute intensive sections of the DiFX correlator and removing the rest[8]. The compute intensive sections of DiFX were identified by using Intel's VTUNE Analyser, a profiling tool. As expected, they were the frequency transformation and complex multiplication.

The simplified correlator became the basis of the co-processor design and was used as a performance benchmark. [9].

Specifically our aims were to:

1. present correlator designs for both the GPU and FPGA co-processor platforms.
2. implement the designs on the respective hardware and record the performance results.
3. evaluate the co-processors' performance and compare them with the simplified optimised software correlator implementation.

[8]The simplified correlator focuses on the computationally intensive functions of the correlations while avoiding the smaller intricacies. The more subtle intricacies are important to the accuracy of the correlator, but largely computationally insignificant.

[9]

### 1.4.1 Scope

Complete correlators are complex systems, involving many intricacies to improve the interferometry accuracy. This project focuses on meeting the computational requirements of the 'correlation stage' of the correlator, not producing the final visibility output. Therefore, for simplicity, we do not perform delay correction or fringe stopping, assume input are stored locally on the host machine, and are represented as single precision floating point numbers.[10]

It was difficult to compare technologies fairly as we only used one example of each. Furthermore, the Virtex 4 FPGA was released two years before the G80 GPU and Harpertown CPU. The performance is more fairly compared when we estimate the performance on the latest technologies from the different vendors.

### 1.4.2 Related Work

Similar FX correlation acceleration work has been attempted by the University of Western Australia [14] and Helsinki University of Technology [15] using GPUs and Cell BBE respectively. Both papers have reported encouraging results ranging in between 10-50x speedup over a pure CPU implementation, which justifies our choice to pursue co-processor acceleration.

It must be noted for clarity that the DiFX correlator was only used as a source of inspiration for our correlator implementations. Our implementations are independent and there is no interoperability with the DiFX correlator. However, some design choices were made to potentially allow for DiFX integration - this is discussed in Appendix H



## 1.5 Document Outline

The rest of this dissertation is structured as follows:

*Chapter 2* covers some background radio astronomy, and its importance in the radio interferometry imaging pipeline.

The point of the correlator is to compute the complex valued correlation products for each baseline[11] [16] to form complex visibilities. An FX correlator computes the correlation in the frequency domain, which is broken into two separate stages. Firstly, the FFT is computed for each of the antenna in the array. Secondly, the transformed output of each antenna is multiplied with that of every other antenna[12], and accumulated for a few time steps, as shown in Figure 1.4.

[10] Fixed-point arithmetic would most likely be a better choice for the FPGA implementation, but would require careful consideration on the impact on accuracy, therefore for simplicity single precision floating point arithmetic was used.

[11] Every possible combination of antennas is a baseline

[12] Autocorrelations are also performed

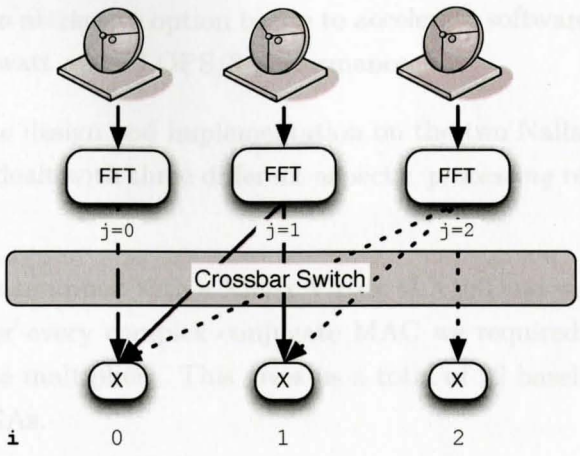



**Figure 1.4** – The correlation operation with 3 antenna, which equates to 6 baselines correlations (including autocorrelation)

The conjugate complex multiplication, performed by the X-engine, is more computationally expensive than the channelisation[13], performed by the F-engine, when using the FFT[14] [15] . Therefore the focus of our co-processor correlator acceleration is only on the conjugate complex multiplication and accumulation stage of the correlation [14, 17].

The number of correlation multiplications is a triangular number - since each antenna needs one less correlation than the previous as shown in Figures 1.4 and 1.5. This triangular progression requires more careful flow control to avoid branches in the pipeline, which is discussed further in the implementation chapters 4 and 5 .

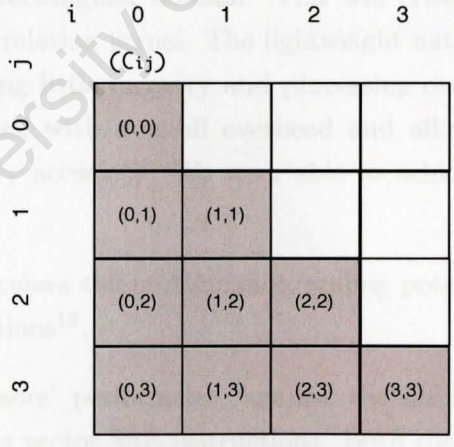



**Figure 1.5** – Showing the resulting triangular number of baseline correlations in a 4 antenna array.

*Chapter 3* looks at the Nvidia GPU and Nallatech FPGA used in this project. Correlation is an extremely compute intensive process but does not necessarily require custom hardware. This is especially true for older correlators or VLBI experiments, where the processing and I/O requirements can be satisfied by commodity processors. Discrete software co-processors like

[13]This is only true above a certain number of dishes in an array - but is almost always the case for modern telescopes which tend to have a substantial number of antenna.

[14]The conjugate complex multiplication is an $O(N^2)$ problem, while the FFT is $O(NlogN)$.

[15]Polyphase filter banks are also sometimes used to do channelisation, which increase the computational requirements of the F-engine

GPUs and FPGAs are an attractive option to use to accelerate software correlation, potentially offering better FLOPS/watt and FLOPS/$ performance.

*Chapter 4* discusses the design and implementation on the two Nallatech H101 FPGAs. The design of the correlator dealt with three different aspects: *processing resources*, *I/O capabilities* and *control*.

Each Nallatech H101 is equipped with a Xilinx Virtex 4LX100 and we were able to implement 88 FPUs per FPGA. For every complex conjugate MAC we required 8 FPUs, which allowed for 11 complex conjugate multipliers. This gives us a total of 22 baseline pipelines using the 2 available Nallatech FPGAs.

We tried three different approaches to describe the correlator's triangular kernel. The naive approach gave a speedup of 4x over the CPU implementation. However, the processing pipeline stalled frequently due to branching[16]. The second implementation removed the stalls and obtained a 5.5x speedup, but required double buffering of input. The final design stems from the second design, but the occasional redundant operation was added to remove the double buffered input, creating a more memory efficient design. The final design was able to eradicate any branches in the pipeline and the pipeline was always fully utilised. This resulted in an overall speedup of 7x over the CPU implementation[17].

*Chapter 5* discusses the GPU correlator design and implementation, which was developed using Nvidia's Compute Unified Device Architecture (CUDA) on a Geforce 9800GT. The GPU CUDA correlator design was based on work done by Harris et al. [14] on GPU correlator acceleration. Harris's idea is to take advantage of CUDA's multiple hardware threads and initialise these threads in a rectangular domain. This will create dormant threads, but also create a simplified square correlation kernel. The lightweight nature of CUDA threads, results in the dormant threads adding little memory and processing overhead. The result is a clean description of a square kernel, with a small overhead and allowing efficient linear memory addressing (coalesced memory accesses). We were able to achieve a 12.5x speedup over the CPU implementation.

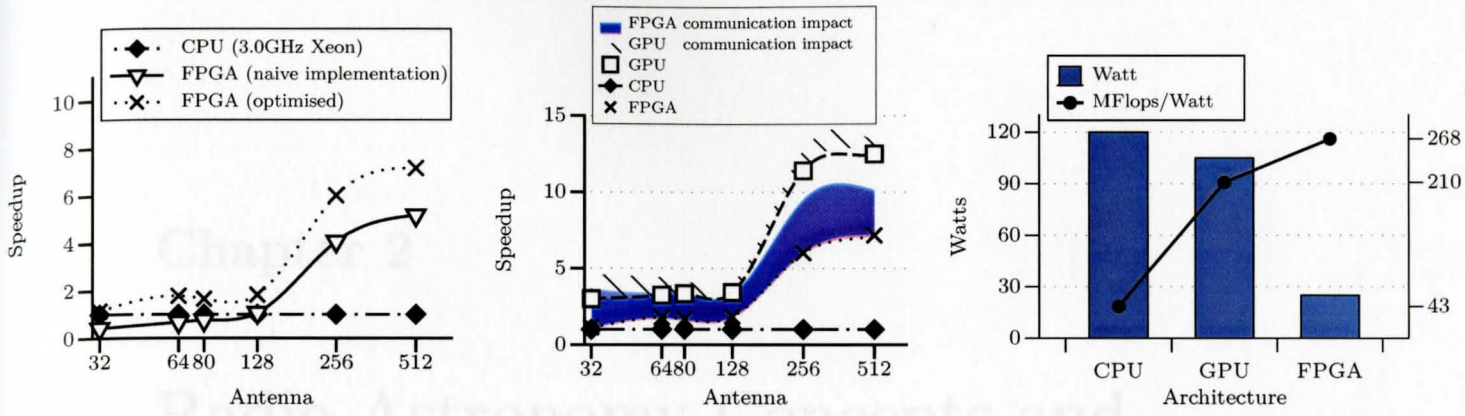
*Chapter 6* presents and discusses the performance, scaling potential and power utilisation of the co-processor implementations[18].

We compare the co-processors' performance against the CPU correlator implementation, which makes use of the CPUs vector SSE instructions. Both correlator implementations were tested on a range of antenna input streams and spectral channels. Speedups of 7x and 12.5x were achieved on the FPGA and GPU correlator implementations respectively. While the GPU delivers consistent performance, the FPGA performs poorly with 64 and fewer antenna streams. Ignoring the time it took to move data from host to co-processor, speedups of 10.5x and 13.5x were achieved on the FPGA and GPU correlator implementations respectively. These results are shown in Figure 1.6.

[16] A branch was taken when a series of baseline correlations had finished with a particular antenna.

[17] The FPGA implementation uses both of the Nallatech boards.

[18] Power utilisation was not measured directly but instead power estimation tools provided by the vendors were used.




(a) Including Bus Transfers

(b) Excluding Bus Transfers

(c) Power Consumption

**Figure 1.6** – Co-processor speed-up vs 3.0GHz Xeon CPU software correlation with 256 spectral channels. **(a)** shows the speedup of two of the FPGA designs **(b)** shows performance results of the best FPGA design and the GPU design. **(b)** also shows the bus overhead on the GPU and FPGA co-processors, where the time spent in I/O is shown in the shaded region. The PCI-X bus had a large impact on the H101's performance, while I/O on the faster PCIe bus on the GPU contributed less to the runtime performance. **(c)** shows the power consumption of the correlator implementations across the differ processing architectures.

Although both implementations achieved speedups and better power utilisation than the CPU implementation, the GPU implementation produced better performance in a shorter development time than the FPGA. The FPGA implementation was hampered by the development tools and the slow PCI-X bus, which is used to communicate with the host[19].[20].

*Chapter 7* discusses possible future work and concludes on the co-processor correlator implementations.



[19]The bus speed is a limitation of the vendor board not inherently of the FPGA.

[20]It should also be noted that the FPGA used in this project is from an older generation of technology, released in 2005, than the GPU and CPU, which were released in 2007.

# Chapter 2

# Radio Astronomy Concepts and Correlation Principles

The objective of a radio astronomy correlator is to compute the complex valued correlation products for each baseline[1] to form complex visibilities [16]. From these complex visibilities the sky's brightness distribution can be reconstructed, which is discussed in detail in Thompson et al.[17] [2]. The correlation operation is where the majority of the compute time is spent, and was the focus for our co-processor acceleration [14].

This dissertation focuses on the computational aspects of FX correlation, not the scientific significance of the result. For a richer treatment of the subject, we advise you to see [17, 18]. In this chapter we very briefly review the background to interferometric telescopes, which will be followed by a more detailed discussion of FX correlators.

We end off the chapter by reviewing related work in the field.



## 2.1 Background

One of the central goals of astronomy is to create a clearer understanding of our Universe. For centuries astronomers have contributed towards this goal by studying the visible objects in the night sky. However, visible light is only a small fraction of the electromagnetic spectrum produced by astronomical objects. In the early 1930's, astronomers discovered that the Universe is full of radio information, which is a key untapped source of information[18]. This discovery helped identify entirely new classes of objects such as pulsars, quasars and radio galaxies. Radio waves have also been used to detect neutral hydrogen, the most abundant element in the Universe. The measurement of neutral hydrogen is one of the best ways to reveal the structure of the Universe[3]. Figure 2.1 is an example of the recording of the radio brightness of the Milky Way, from Hartebeesthoek Radio Astronomy Observatory (HartRAO) by Jonas [19].

[1] Every possible combination of antennas is a baseline

[2] Basically complex visibilities are used to construct the 2D Fourier transformation of the brightness distribution of the observation source.

[3] The spiral structure of the Milky Way was discovered by measuring neutral hydrogen's spectral lines, which occur at around 1.42GHz.

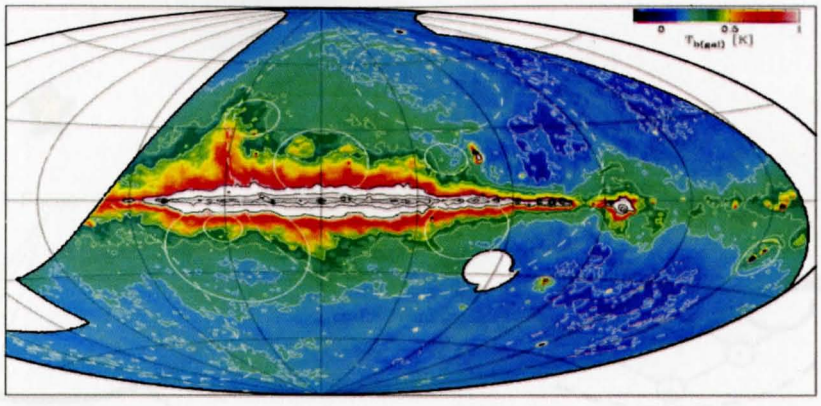


**Figure 2.1** – The Milky Way recorded at HartRAO. Image from J. Jonas [19]

Unfortunately, many of the radio sources of interest are very distant and therefore their signals are extremely weak by the time they reach Earth. Larger receiver antennas provide the resolution and sensitivity needed to detect and accurately record these signals [4], however, building large steerable radio receivers is an extremely expensive undertaking. To address this short coming, a virtual large antenna can be synthesised from an array of smaller ones, using a special type of interferometry, called *aperture synthesis* (see Figure 2.2). Aperture synthesis allows an artificial antenna to be created from the combination of two or more receiver responses, improving what is possible with small antennas. Amazingly, aperture synthesis provides a way for two physically separated antennas to produce the same resolution as a receiver the size of the distance between them![5] Antenna arrays also allow for different antenna configurations for different experiment types.

However, aperture synthesis comes at a large computational expense: the interferometric result required to perform aperture synthesis needs to be computed, typically digitally, in high speed correlation devices. The computational requirements of aperture synthesis grow quadratically with the size of the antenna array. But as the performance of microprocessor technologies have improved and the physical cost of antennas has risen[6], it becomes increasingly cost effective and viable to build large interferometric antenna arrays[7] [8].

[4]Better resolution is acquired by increasing the aperture diameter. Better sensitivity is acquired by increasing the collection area.

[5]However, the synthesised aperture created from the two smaller antennas will have poorer sensitivity and other artifacts.

[6]Typically as a result of steel

[7]Quote the ATA.

[8]Complex correlator but with many small cost effective antennas.

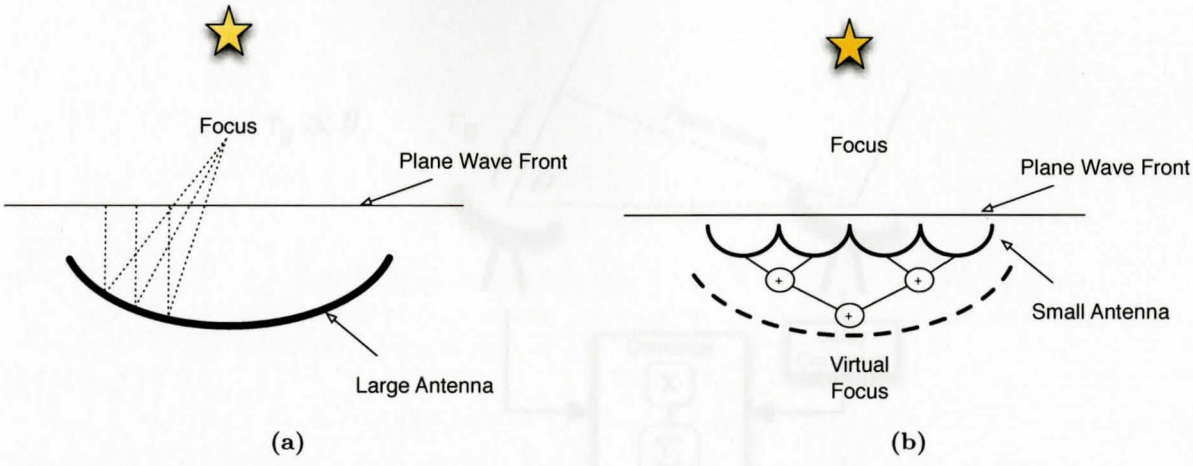



**Figure 2.2** – Diagrammatic representation of (a) a large antenna and its focus point and (b) a virtual large antenna, synthesized from an array of small antennas. Figure adapted from [20].

Aperture synthesis is a complex process and is usually performed in a number of separate stages. Figure 2.3 is a simple example of the decomposition of aperture synthesis (Figure B.5 is a more complete processing overview from van der Merwe and Lord [21]). The objective of this project was to accelerate the correlation operation, but not the entire processing pipeline.

The remainder of this dissertation will focus only on the correlator, but it should be noted that many other operations that occur in a fully working interferometric telescope are not addressed or implemented here. The balance of this chapter will discuss the core functionality of the correlator and the part that was implemented in this dissertation. For a more thorough description of correlation and how it fits into aperture synthesis, refer to Thompson et al. [17], which is a well written and highly recommended reference.

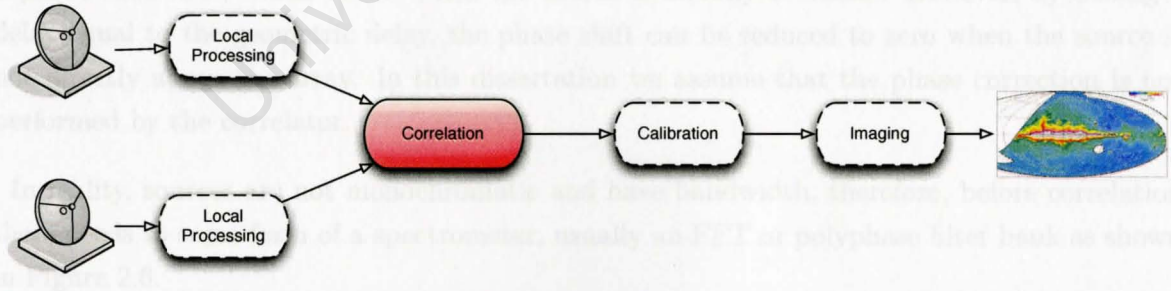



**Figure 2.3** – The radio astronomy processing pipeline from antennas to images, inspired by [22]

## 2.2 Simplified Correlation Operation

In this section we will present a simplified description of correlation and then detail how it is computed digitally.

Figure 2.4 shows a simple two antenna array, where both antennas are observing the same far-field source and for simplicity we will assume that the source is monochromatic. Connected

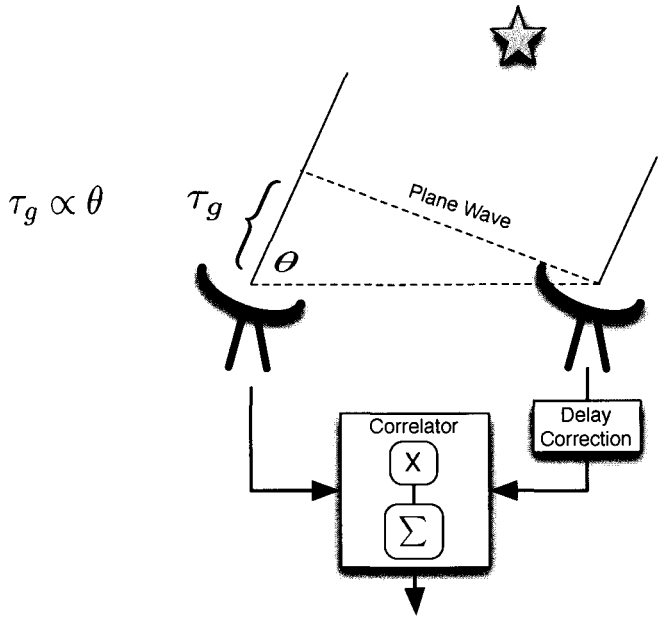



**Figure 2.4** – Diagrammatic Representation of an Interferometric Telescope. The spacing between the antenna introduces a delay $\tau_g$ into the system, which is corrected before correlation.



to the antennas is a correlator, which combines the independent antennas by multiplying [9] the two signals together and integrating for a period of time[10]. The source radiation reaches the antenna as a plane wave as shown, but because of the antennas' geometric spacing the plane wave reaches each antenna at a slightly different time, resulting in a phase shift. These phase shifts reduce the correlation magnitude and cause the incorrect correlation products to be recorded. Figure 2.5 shows the correlation results after integration when observing an object for varying angles from the zenith, which result in different phase shifts. The nulls occur when there is a 90°phase difference. The desired correlation reading is when there is a 0°phase difference, which occurs when the source is directly overhead. However, by adding a delay equal to the geometric delay, the phase shift can be reduced to zero when the source is not directly above the array. In this dissertation we assume that the phase correction is not performed by the correlator.

In reality, sources are not monochromatic and have bandwidth, therefore, before correlation there needs to some form of a spectrometer, usually an FFT or polyphase filter bank as shown in Figure 2.6.

[9] Adding interferometers also exist, which are simpler but often inferior. [23]
[10] Integrating is used to improve SNR and reduce bandwidth

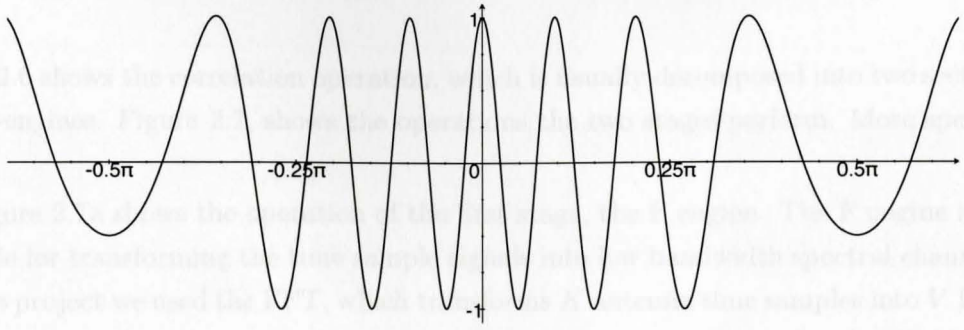



**Figure 2.5** – An example of a fringe produced by an interferometric telescope, when observing a monochromatic point source object, where x is the angle from the zenith of the interferometer.

## 2.3 Computing the Correlation

The correlation dealt with in this project is a *four* dimensional problem - this involves two antenna inputs, $i, j$, in a specified frequency band, $v$, at a discrete time interval $a$ - which we choose to represent as $\mathbf{C}\ [i, j, v, a]$. To design and understand the correlation, it was helpful to visualise the operation graphically. The illustrations in this chapter aid in explaining the FPGA and GPU correlator implementations and will be referred to in later chapters.

### 2.3.1 Computing the Correlation Numerically

In this section we look at how the correlation is computed numerically. For a more fundamental description see Appendix B.2.

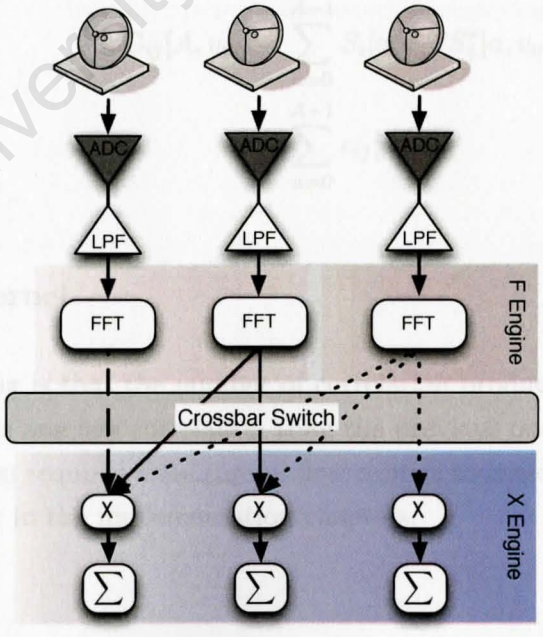



**Figure 2.6** – Correlation operation with arrows showing the input requirements for the different stages. The 3 antennas equate to 6 baselines correlations (including autocorrelation). In large correlators, the F engine channelisation is often performed independently for each antenna and the interconnecting cross-bar switch does the necessary corner turning for the X-engine as described in section 2.4.

Figure 2.6 shows the correlation operation, which is usually decomposed into two sections, the F and X-engines. Figure 2.7, shows the operations the two stages perform. More specifically:

i. Figure 2.7a shows the operation of the first stage, the F engine. The F engine is responsible for transforming the time sample signals into low bandwidth spectral channels[11]. In this project we used the FFT, which transforms $K$ antenna time samples into $V$ frequency channels, $x_k \rightleftharpoons S_k$.

$$S_k[a_n, v] = \sum_{l=0}^{L-1} x_k[l] e^{-i2\pi vl/L} \tag{2.1}$$

ii. (a) Figure 2.7b shows the operation of the first part of the second stage, the X-engine. Here the cross-power spectrums are computed by performing conjugate multiplication. This cross conjugate multiplication is performed with every antenna in the array, but not mixing channels. Equation 2.2, is the conjugate multiplication of antenna '$i$' with antenna '$j$', for the same channel '$v_m$' for a certain time instance '$a_n$'.

$$c_{ij}[a_n, v_m] = S_i[a_n, v_m] S_j^*[a_n, v_m] \tag{2.2}$$

(b) Figure 2.7d shows the operation of the second part of the X-engine, which is the accumulation of ii (a) for a certain period $A$, where $a$ represents the position in the accumulation. The accumulation is used to improve the SNR and lower the output bandwidth.[12]

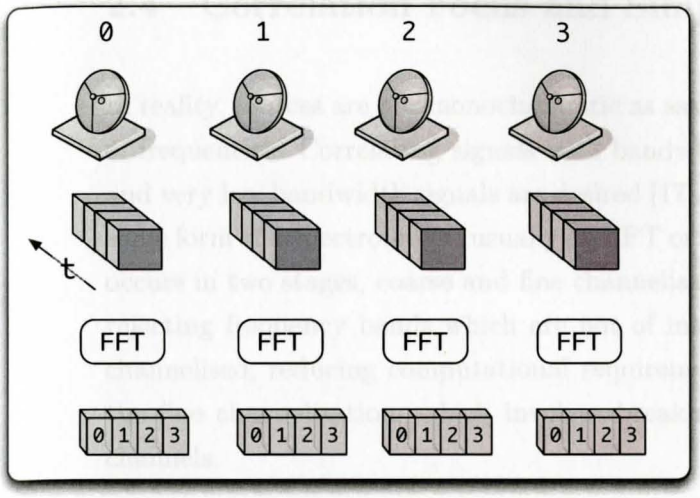
$$\begin{aligned} \mathbf{C}_{ij}[A, v_m] &= \sum_{a=0}^{A-1} S_i[a, v_m] S_j^*[a, v_m] \\ &= \sum_{a=0}^{A-1} c_{ij}[a, v_m] \end{aligned} \tag{2.3}$$

### 2.3.2 Triangular Kernel

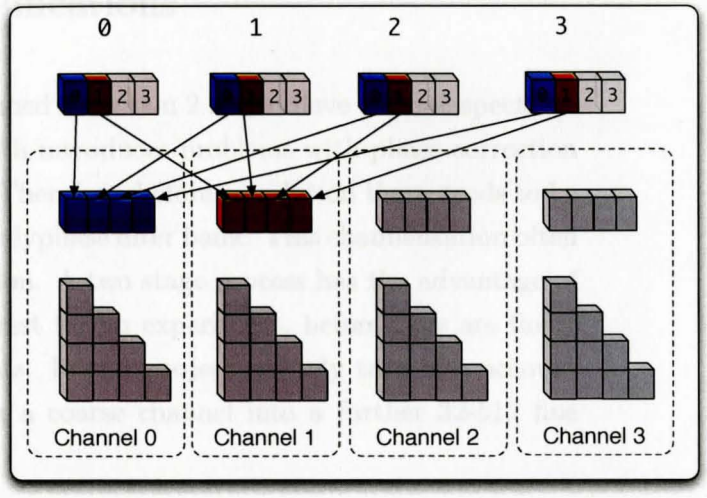
A complexity worth noting is that the number of correlation products is a triangular number - since each antenna needs one less correlation than the previous one, as shown in Figure 2.8. This triangular progression requires more careful flow control to avoid branches in the pipeline, which is discussed further in the implementation chapters.

[11] As close to monochromatic signal as possible

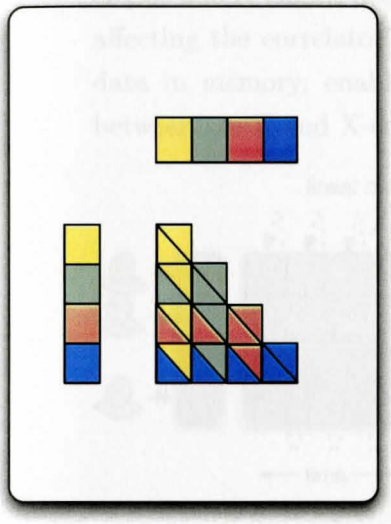
[12] Accumulation does improve the SNR and reduce the output bandwidth of the correlator, but also introduces problems like time-smearing and false-negative detection of transients [17].



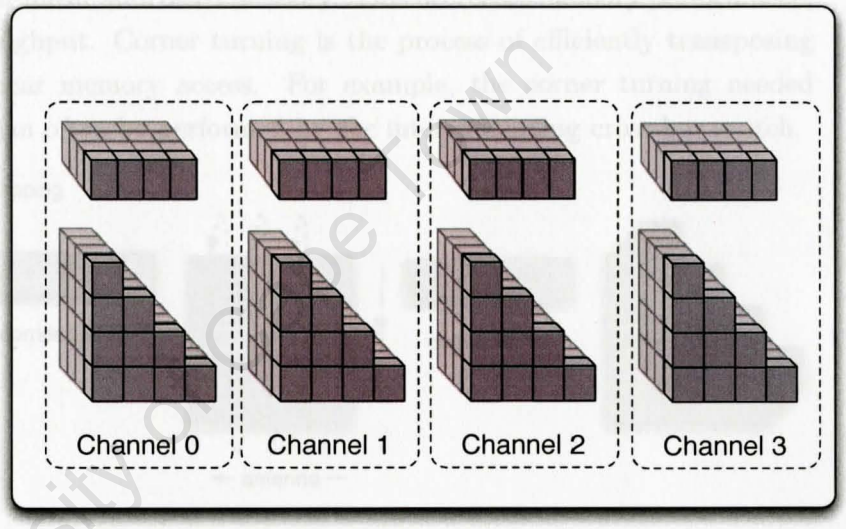

(a) F Engine




(b) X Engine



(c) X Engine computation




(d) Accumulation

**Figure 2.7** – The different stages of the correlator: **(a)** the antenna outputs are transformed into a number of frequency channels by the X-engine. **(b)** all antennas send the same frequency channel to the different X-engines, via a crossbar switch. **(c)** the cross-products are computed. Note that from 4 antennas, 10 cross products are created. More generally for $N_a$ antennas there are $(N_a)(N_a+1)/2$ correlation products. **(d)** the correlation products in **(c)** are accumulated for a certain period before being recorded.

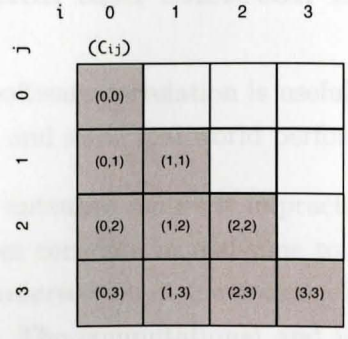



**Figure 2.8** – The resulting triangular number of baseline correlations in a 4 antenna array.

## 2.4 Correlation Focus and Simplifications

In reality, sources are not monochromatic as assumed in section 2.2, and have a broad spectrum of frequencies. Correlating signals with bandwidth introduces problems with phase correction and very low bandwidth signals are desired [17]. Therefore, before correlation there needs to be some form of a spectrometer, usually an FFT or polyphase filter bank. This channelisation often occurs in two stages, coarse and fine channelisation. A two stage process has the advantage of rejecting frequency bands which are not of interest for an experiment, before they are finely channelised, reducing computational requirements. In this project we only take into account the fine channelisation, which involves breaking a coarse channel into a further 32-512 fine channels.

Corner turning is also a consideration in correlation. Figure 2.9 demonstrates that both the F engine and X-engine access non-linear addressed memory. This decreases memory performance, affecting the correlator's throughput. Corner turning is the process of efficiently transposing data in memory, enabling linear memory access. For example, the corner turning needed between the F and X-engine can often be performed by the interconnecting cross-bar switch.

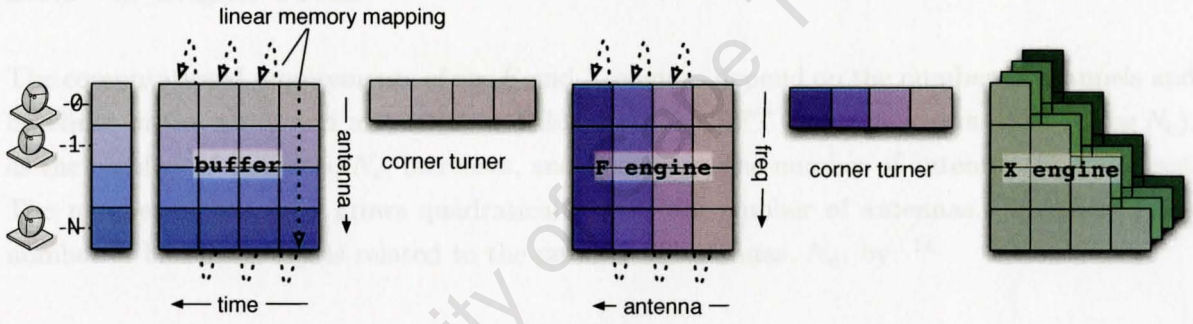



**Figure 2.9** – The non-linear memory accesses by the different stages of the correlator require corner turning to improve memory performance [24].

In this thesis we assumed that data was already optimally ordered in memory and we did not implement any corner turning. However, corner turning is worth mentioning as it can become a major issue in real world correlators - leading to no end of cabling and interconnect issues [4].

## 2.5 Software Correlation and Skeleton Design

In this section we discuss when software correlation is useful, why the X-engine was the focus of our correlator implementation and some real world performance requirements.

The flood of data produced by antennas makes it impractical to store the data and process it later and many radio telescopes correlate in real-time to alleviate this problem. Since the correlator is the joining point or intersection of the antenna feeds, it has the potential of being the bottleneck of the telescope. The computational and networking requirements for large arrays make it justifiable to build custom correlation hardware, usually from FPGA or ASIC devices.

Although the latest telescopes require custom hardware, new generations of modern medium-sized general purpose compute clusters can feasibly be used instead of some older custom correlator hardware. This software correlation is significantly more accessible and customisable than production hardware correlators. Because of the low cost of commodity clusters, some astronomy institutions are finding it more effective to use software correlation than to support old specialised correlator hardware. The flexibility and availability of software can help extend and improve the life of a telescope.

Besides replacing older correlator hardware, another popular domain for software correlation is Very Large Baseline Interferometry (VLBI), because the large antenna separation makes it impractical for online correlation [13]. The recorded antenna data is usually transferred to a central processing point for offline correlation. Off-line correlation has less stringent time requirements than real-time processing, making software a good option.

This project implemented three simple software correlators using an x86 CPU, GPU and FPGAs.

### 2.5.1 X Engine Focus

The computational requirements of the F and X-engines depend on the number of channels and baselines in the array and are listed in Table 2.1. The FFT F engine grows at $O(N_c \log N_c)$, as the number of channels, $N_c$, increases, and linearly as the number of antennas $N_a$ increases. The number of baselines grows quadratically with the number of antennas. Specifically the number of baselines, $N_b$, is related to the number of antennas, $N_a$, by: [14]

$$N_b = \frac{(N_a + 1)(N_a)}{2} \tag{2.4}$$

Therefore the X-engine grows at $O(N_a^2)$ as the number of antennas increases and linearly as the number of channels $N_c$ inceases.



**Table 2.1** – Computation Scaling of F and X-engine

| | **F Engine** | **X Engine** |
|---|---|---|
| **Computation** | $N_c \log N_c \times N_a$ | $\frac{(N_a+1)(N_a)}{2} \times N_c$ |
| **Order** | $O(N \log N)$ | $O(N^2)$ |

There are a number of FFT libraries available for both GPUs and FPGAs, which can be used in our implementation of the F engine. Additionally, in most cases the X-engine is dominant since its computational requirements quickly overtake the F engine as shown in Table 2.1. For this reason the X-engine was the focus of this project and we relied on FFT libraries for the channelisation.

[13] VLBI usually also has lower data rates and fewer antennas in the array

[14] Using the figures as an example, the 2 antenna in Figure 2.4 produce 3 baselines, while the 3 antenna in Figure 2.6 produce 6 baselines, including autocorrelations.

### 2.5.2 Correlator Skeleton Design

The correlation implementation can be divided into two sections - library calls for the F-engine and custom code for the X-engine. This division and the basic operation of the correlator are shown in Figure 2.10.

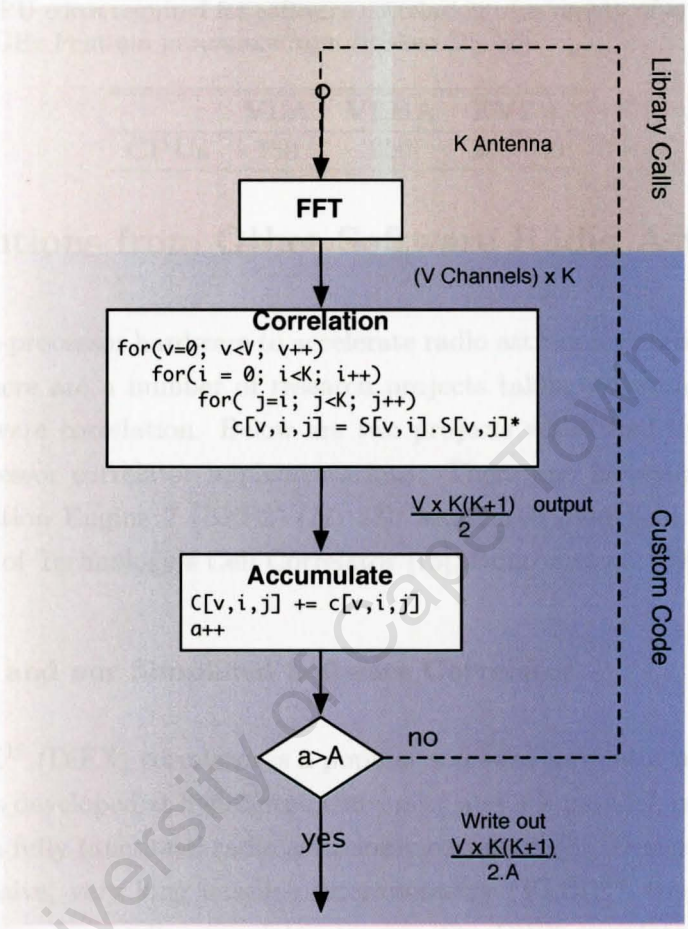



**Figure 2.10** – The division of the correlator into library calls and custom code. The custom code includes the basic operation of the correlator.

### 2.5.3 Real World Correlator Requirements

To demonstrate the large amount of computation necessary for correlation, Table 2.2 shows the performance requirements for the planned meerKAT and SKA correlators taken from the unofficial SKA and KAT requirements (note this excludes the post image processing requirements). Table 2.3 lists the number of CPU cores required to meet the correlator requirements in software. These tables show the high computational burden of correlation.

**Table 2.2** – Performance and data requirements of various planned arrays.

| | **meerKAT** | **SKA**(2017) | **SKA**(2021) |
|---|---|---|---|
| **Antennas** | 80 | 620 | 2400 |
| **Data Rate** (per antenna) | 32 Gbps | 32 Gbps | 32 Gbps |
| **Data Rate** (total) | 3 Tbps | 20 Tbps | 80 Tbps |
| **Processing Requirements** | 52 TeraOps | 3 PetaOps | 47 PetaOps |
| **Completion Date** | 2013 | 2017 | 2021 |

**Table 2.3** – CPU cores required for software correlation of a variety of arrays. Based on 3GHz Pentium processors from Brisken [25, 26].

| | **VLA** | **VLBA** | **EVLA** |
|---|---|---|---|
| **CPUs** | 150 | 250 | 200,000 |

## 2.6 Contributions from Other Software Radio Astronomy

The idea of using co-processor hardware to accelerate radio astronomy correlation is not unique to this project. There are a number of research projects taking advantage of multi-core architectures for software correlation. Below are two projects which had the greatest influence on our two co-processor correlator implementations. There are, however, many others such as, Berkeley Emulation Engine 2 (BEE2) [27, 28], Murchison Widefield Array (MWA) [16], Helsinki University of Technology's Cell Correlator [15], Bunton et al. [29].



### DiFX Correlator and our Simplified Software Correlator

The Distributed FX[15] (DiFX) correlator is a popular software correlator implementation. The DiFX correlator was developed at Swinburne University and is a parallel, open-source, software implementation of a fully functional radio astronomy correlator [5]. Designed to work with the less processor intensive, very long baseline interferometry (VLBI)[16], the DiFX is an attractive correlator solution for smaller correlator arrays. The DiFX correlator has had a positive response in both astronomy and HPC communities, allowing research to be carried out on standard Linux compute clusters, without sharing or endangering production correlators. The National Radio Astronomy Observatory (NRAO) and Max Plank Institute fur Radioastonomie (MPIfR) have adopted the DiFX correlator for the correlation of their Very Long Baseline Array (VLBA) data [30, 31] and have released their own NRAO-DiFX modification [32]. Although the DiFX correlator is not used directly in this project, for reasons explained in Appendix H, it served as an inspiration for the opportunities of software correlation.

We began by using the DiFX correlator as a reference to create a simplified software correlator. The DiFX correlator project is a complex software project, with thousands of lines of code, while the heavy computation is contained in only a few lines. The simplified correlator was an extraction of the compute intensive sections of the code in a new software project. This

[15] FX here refers to how the correlation is performed. FX correlators do a multiplication in the Fourier domain, while XF correlators perform a convolution in the time domain.

[16] VLBI typically uses smaller arrays (<10) with baselines that can span 1000s of kilometers. Since there is relatively small number of data sources, produced at distributed sites it is practical to perform off-line correlation.

simplified correlator became the basis of the co-processor design and was used as a performance benchmark.

The simplified correlator was created to be very minimalistic, performing the correlation operations on raw input data on a single CPU host - ignoring the data unpacking and distributed communication of the DiFX correlator. This allowed the software correlation runtime not to be contaminated with other unrelated operations, which were not the focus.

We borrowed the DiFX correlator's approach to using Intel's Performance Primitive's (IPP) libraries to perform the correlation operations on the pre-correlated data. The IPP contains optimised libraries that take advantage of modern x86 processor's Streaming SIMD Extensions (SSE). The IPP libraries were used to implement the FFT channelisation and the complex MAC.

See the attached DVD for the source code and more details on the software correlator implementation.

### GPU Correlator

Chris Harris was, at the time of development, working on GPU acceleration of software correlation [14]. Our GPU implementation borrows ideas from Harris' GPU correlator design and is discussed in more detail in Chapter 5.



## 2.7 Conclusion

Radio astronomy correlation is a vital and very computationally intensive part of a synthetic aperture array, often requiring custom hardware to maximise performance. However software correlation is a much more accessible platform, making it appealing for correlator prototyping and replacing older hardware correlators.

In the next Chapter we look at the Nvidia GPU and Nallatech FPGA co-processors which were used to accelerate software correlation.

# Chapter 3

# Software Co-Processor Acceleration

Correlation is an extremely compute intensive process but does not necessarily require custom hardware. This is especially true for older correlators or VLBI experiments, where the processing and I/O requirements can be satisfied by commodity processors.

Discrete software co-processors like GPUs and FPGAs are an attractive option to accelerate software correlation, potentially offering better FLOPS/watt and FLOPS/$ performance. These different technologies bring with them their own unique architecture, development tools and environment. These differences need to be addressed and understood when developing the software correlator. This chapter looks at the Nvidia GPU and Nallatech FPGA used in this project and their respective development tools.



## 3.1 Code Acceleration

The limited speed of serial processing is inadequate to perform radio astronomy correlation in a reasonable amount of time. Fortunately the correlation workload is *embarrassingly parallel* and can be easily and logically divided up amongst multiple sequential processors and processed in parallel. Software correlators, such as the DiFX correlator, use compute clusters to accelerate the correlation in this manner. Radio astronomy correlators exhibit a class of parallelism called *data-parallelism*, which maps well to FPGAs and GPU architecture.

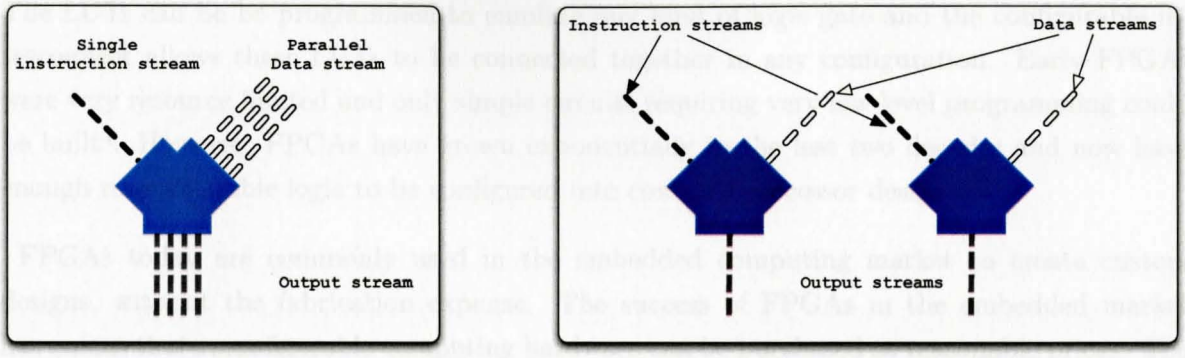
An example of data-parallelism is a code loop, when the same operation is performed across an array of data. If each loop iteration is independent, the order in which each iteration is computed is not important, making them suitable for parallel computation.[1]

Many scientific computing applications display a large amount of data-parallelism, but it is rarely present in desktop applications and therefore commodity microprocessors are not designed to exploit data-parallelism.[2] However, SIMD co-processors can be added via computer expansion buses such as PCIe and PCI-X to improve a system's SIMD capabilities. Capable SIMD processors, such as GPUs and FPGAs, have grown remarkably in performance and

[1] Another name for data-parallelism is in fact loop-parallelism.

[2] CPU manufacturers have shown a moderate interest in exploiting data-parallelism and have added some limited SIMD hardware. The current SSE SIMD instructions are limited to 128 bit vectors, inadequate for serious number crunching.

programability and are becoming an attractive option to be used in scientific applications. Figure 3.1 shows a data-parallel application being processed on scalar processors and a SIMD processor.




**(a)** SIMD Processing **(b)** Scalar Processing

**Figure 3.1** – The above figure shows parallel computation either on **(a)** a vector processor or **(b)** the data-parallelism being exploited by multiple scalar processors. However **(b)** requires an instruction stream for each scalar processor and synchronisation of data. Inspired by Arstechnica [33]

Figure 3.2 **(a)** shows a typical software application with a processing hot-spot, which could be suitable for co-processor acceleration. In Figure 3.2 **(b)** is the same application with the hot-spot computed on the co-processor, however there is now a host-device communication overhead which must be taken into consideration. With processor capabilities running ahead of memory and inter-processor communication speeds, it is important that hot-spots have a high arithmetic intensity or a high FLOP/Byte ratio to minimise the impact of the communication overhead [34].

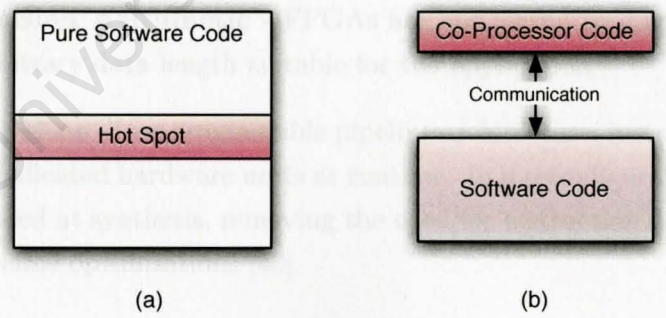



**Figure 3.2** – (a) original software design (b) co-processor accelerated software with communication overhead

The co-processors used in this project and their respective development environments are introduced below.

## 3.2 Reconfigurable Computing (RC)

*Reconfigurable computing* is a category of computing that makes use of special-purpose hardware that allows the programmer to adapt the hardware to better suit a specific computational

problem. This flexibility potentially lets the user create an architecture which makes efficient use of the processing resources. FPGAs are a type of reconfigurable hardware which allow any kind of operation and interconnection to be created and are a popular technolgy used in reconfigurable computers [35]. FPGAs consist of an array of LUTs[3] and a configurable interconnect. The LUTs can be be programmed to emulate any kind of logic gate and the configurable interconnect allows these LUTs to be connected together in any configuration. Early FPGAs were very resource limited and only simple circuits requiring very low level programming could be built[4]. However, FPGAs have grown exponentially in the last two decades and now have enough reconfigurable logic to be configured into complex processor designs.

FPGAs today are commonly used in the embedded computing market to create custom designs, without the fabrication expense. The success of FPGAs in the embedded market has meant that reconfigurable computing hardware can be purchased at reasonable prices. The reconfigurable computing industry has successfully implemented a number of HPC applications, such as image processing [36], data mining [37] and bioinformatics [38] [5].

### 3.2.1 Advantages of RC

From a processing perspective FPGAs have relatively weak floating point arithmetic when compared to GPUs and only offer around one fifth of the theoretical performance as shown in Figure 1.3a. However, reaching GPU's peak performance is only possible when making full utilisation of its processing pipeline[6], which is rarely achieved. Because of FPGA's flexibility, a custom pipeline can be created for a particular application, implementing only the functional units needed, allowing them to get closer to their theoretical peak. Some unique FPGA optimisations allow for:

- **Variable Precision Arithmetic** - FPGAs are not locked to a specific data type and can use any arbitrary data length suitable for the application.
- **Optimised Pipeline** - In a programable pipeline architecture, instructions are unpacked and issued by dedicated hardware units at runtime. In a reconfigurable pipeline, the data path is determined at synthesis, removing the need for instruction decoding and allowing application pipeline optimisations [40].
- **On-chip Communication** - On-chip FPGA Block RAM and distributed memory can be connected in any configuration, allowing very low latency and high bandwidth on-chip communication.

Despite these advantages, a custom pipeline creates an extra layer of programming complexity to FPGA computing. This is partly being addressed by new programing languages for FPGA reconfigurable computing.

[3] Look up tables (LUTs)
[4] This was typically done in Register Transfer Languages (RTLs)
[5] Via [39]
[6] Peak performance figures are calculated assuming all ALUs on the GPU are performing MADD and MUL operations per clock cycle

### 3.2.2 Programming FPGAs

Much of FPGAs' potential performance advantage comes from the ability to create highly parallel compute architectures. Unless this parallelism is realised in the hardware, it is very unlikely that any speed-ups will be achieved due to FPGAs' slow clock rate, typically in the low 100MHz.

FPGAs are typically programmed using hardware descriptive languages (HDL). Programming FPGAs effectively requires a thorough understanding of hardware design concepts such as pipelining and dealing with different clock domains [41].

FPGAs' programmable logic density has grown to a point where many believe it would be more practical to use high level languages (HLLs). FPGA HLLs attempt to hide many of the underlying hardware concepts, which an HDL developer is responsible for[7]. HLLs provide an abstraction to these concepts and are compiled into HDLs before synthesis. FPGA HLLs aim to reduce hardware design times as well as appealing to a larger audience, including software developers unfamiliar with hardware concepts.

#### HLL for FPGAs

The majority of FPGA HLLs are based on a subset of ANSI C syntax. ANSI C does not explicitly allow the programmer to identify parallelism in the algorithm and the FPGA HLLs differ in their approach of how to express this parallelism. We investigated three different FPGA HLLs: Impulse-C, Mitrion-C and Dime-C. Impulse-C has compiler directives to hint to the compiler the area of code and type of parallelisation that should be implemented. Mitrion-C diverges from the ANSI C standard quite significantly and looks like C but is really a functional language, very different from the traditional procedural C. Dime-C does not require any explicit modification to identify parallelism, but this requires code to be written in a way that the compiler recognises the parallelism.

The deviation of Mitrion-C from ANSI-C, made it less accessible than Dime-C and Impulse-C and for this reason we did not consider Mitrion-C seriously as an easy to use option. We chose to use Dime-C over Impulse-C since, at the time of the FPGA correlator development, not all the memory interfaces were accessible using Impulse-C on our Nallatech FPGAs. However, it must be noted that Impulse-C and Mitrion-C offer more polished and refined development tools and environment than Dime-C

All FPGA development in this project used Dime-C and in the next section we discuss Dime-C and its development environment.

### 3.2.3 Dime-C and its Development Environment

Dime-C is a C-to-HDL language created by Nallatech. Dime-C converts ANSI-C into HDL, which is compiled to program the FPGA. However, there are a few omissions from the standard ANSI-C language - notably pointers and recursion [42].

[7]FPGAs perform best when an algorithm is described in a parallel and pipelined manner. Keeping track of pipeline timing is a laborious and error prone task which is exacerbated with the growing size of FPGA designs.

### Writing ANSI-C for hardware synthesis

Although Dime-C syntax looks like ANSI-C, the semantics are quite different to ordinary C. An ANSI-C software program is written to control a processor, while a Dime-C is written to create one. Because FPGAs have no fixed structure, Dime-C is used to describe a custom datapath and the operation units required. This minimises the percentage of the processor dedicated to control, freeing up resources for other processing. Only operational units required are implemented, as shown in Figure 3.3. Knowing the structure of the code and the types of computation at compile time, allows the Dime-C compiler to create a customised architecture for the application.

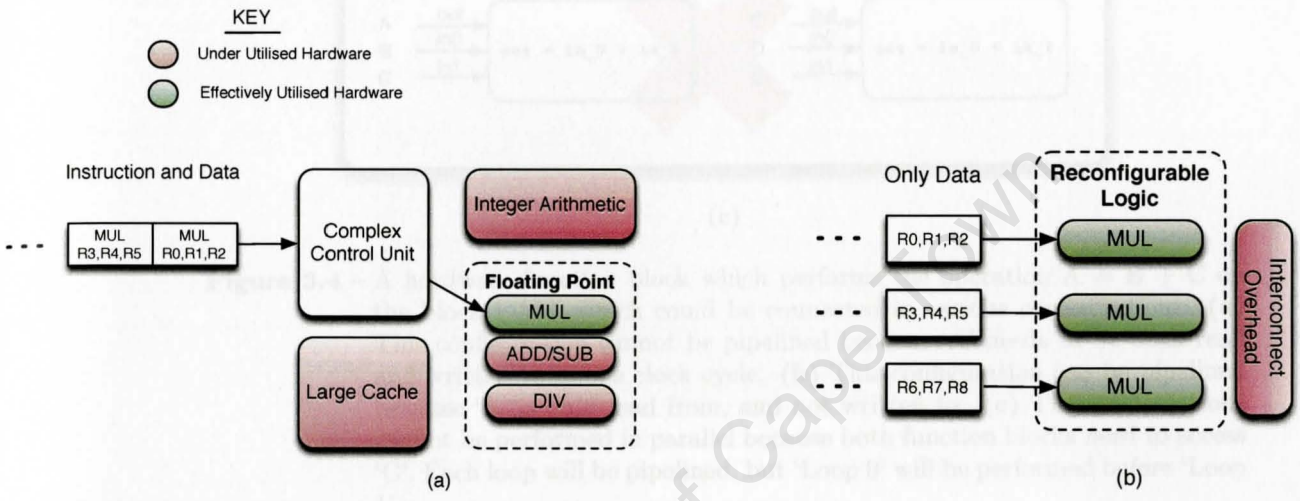



**Figure 3.3** – Transistor utilisation in **(a)** a microprocessor, and **(b)** an FPGA

To get the most from Dime-C, it is important to structure the code in a way that allows the compiler to easily identify areas that can be parallelised. The Dime-C compiler attempts to get a performance speedup by identifying loops in the application that can be pipelined and execute these loops in parallel. The amount of parallelism and speedup possible depends on data dependancies and is restricted by the limitations of the underlying hardware. Data arrays are mapped to block RAM and cannot be accessed more than once[8] per clock cycle to avoid data dependancies which can create problems for parallel execution as shown in Figure 3.4a [42]. The function blocks in Figure 3.4c cannot be performed in parallel because both function blocks need to access 'C'. Each loop will be pipelined, but 'Loop 0' will be performed before 'Loop 1'. This shows that it is important not to re-use variables unnecessarily, even if doing so requires duplication of data.

[8]The block RAM on the Virtex 4 FPGAs used is dual-ported, but only one port is connected to the FPGA device, while the other port is used to access the BRAM from the host.

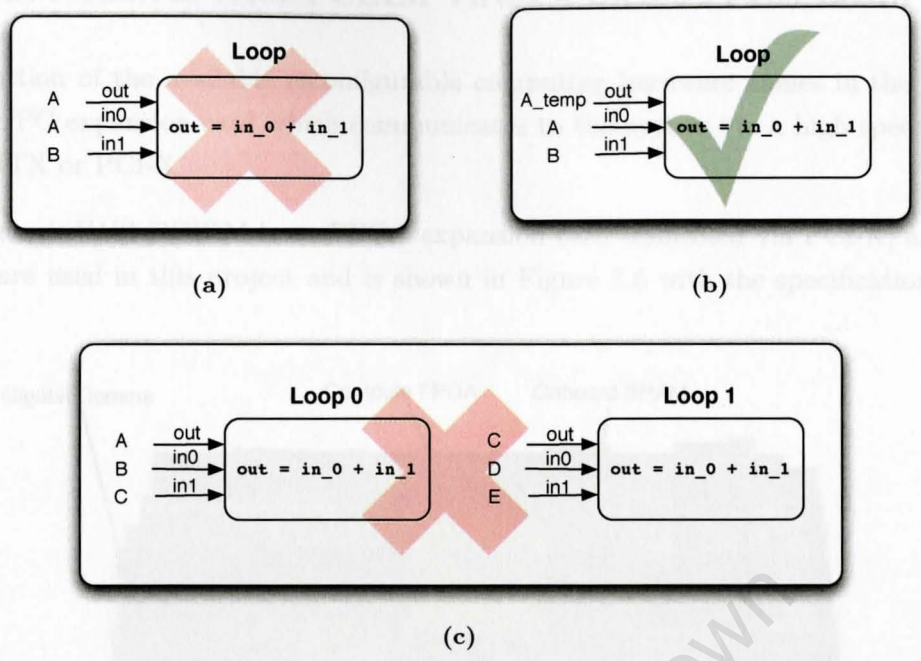



**Figure 3.4** – A hardware function block which performs the operation A = B + C on the block RAM, which could be connected in various configurations: **(a)** This configuration cannot be pipelined because 'A' needs to be both read and written to in one clock cycle. **(b)** This configuration can be pipelined because 'A' is only read from, and not written to. **(c)** These calculations cannot be performed in parallel because both function blocks need to access 'C'. Each loop will be pipelined, but 'Loop 0' will be performed before 'Loop 1'.

Nested loops are problematic to Dime-C. In the case of nested loops, only the innermost loop will be pipelined and stalls will be encountered on each outer loop iteration, making it preferable to convert nested loops into a single fused/coalesced loop if possible. This nested loop problem was encountered in the correlator implementation and is discussed in Chapter 4.

When writing Dime-C programs, the user must be aware that all code is synthesised into hardware, consuming logic. For example, conditional statements require a different data path for each unique branch, as shown in Figure 3.5, so it is expensive to accommodate the exceptions to the main data path.

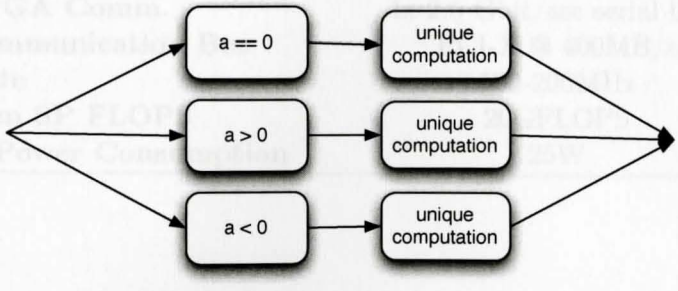



**Figure 3.5** – A conditional statement synthesised into a hardware block.

### 3.2.4 The Nallatech H101-PCIXM Virtex 4 LX100 FPGA Board

A large portion of the available reconfigurable computing hardware comes in the form of an accelerator PC expansion card which communicates to the system via a high speed bus, such as PCIe, HTX or PCI-X.

The Nallatech H101-PCIXM is an FPGA expansion card connected via PCI-X, and was the RC hardware used in this project and is shown in Figure 3.6 with the specifications listed in Table 3.1.

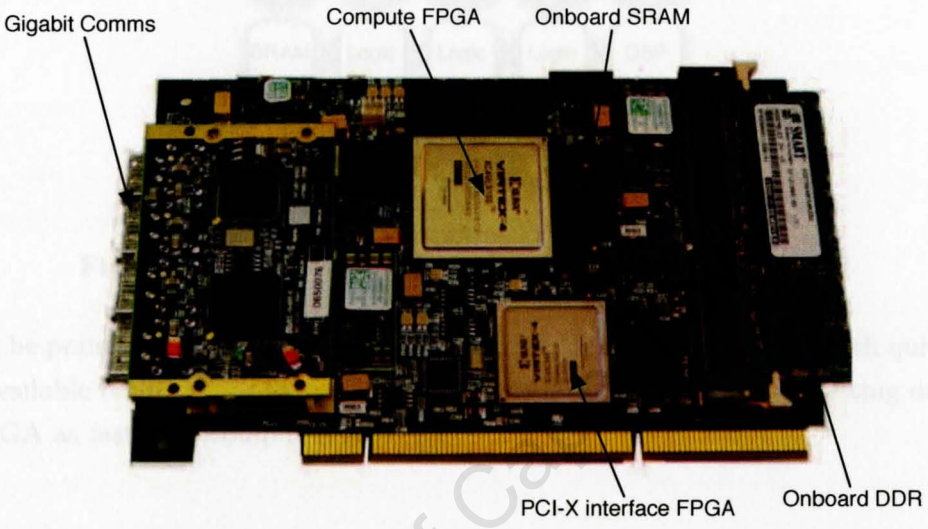



**Figure 3.6** – The Nallatech H101-PCIXM [43]

**Table 3.1** – Nallatech H101-PCIXM Specifications [43].

| | |
|---|---|
| **Processor Type** | Virtex-4 LX100 |
| **Block Ram** | 240 x 18Kbits |
| **DSPs** | 96 |
| **Slices** | 49,152 |
| **Internal Memory** | 0.5MB @ 0.5 TBytes/sec bandwidth |
| **External Memory** | 16MB DDR-II SRAM @ 6.4GB/sec |
| | 512MB DDR2 SDRAM @ 3.2GB/sec |
| **Inter FPGA Comm.** | 4x 2.5 Gbit/sec serial links |
| **Host Communication Bus** | PCI-X @ 400MB/s |
| **Clock rate** | 100-200MHz |
| **Maximum SP FLOPS** | 20GFLOPS |
| **Typical Power Consumption** | 25W |

The FPGA used in the Nallatech H101 is a Xilinx Virtex 4 LX100. FPGAs consist of reconfigurable logic, hardware DSPs and Block RAM as shown in Figure 3.7. The number of these resources depends on the FPGA family and model. The Virtex 4 LX100 used in the Nallatech H101s is a mid range FPGA from Xilinx's 90nm generation [44]. Newer 40nm Virtex 6's [45]

have considerably more resources. This shouldn't affect the fundamental design of our FPGA correlator, but would enable us to process more baselines in parallel.

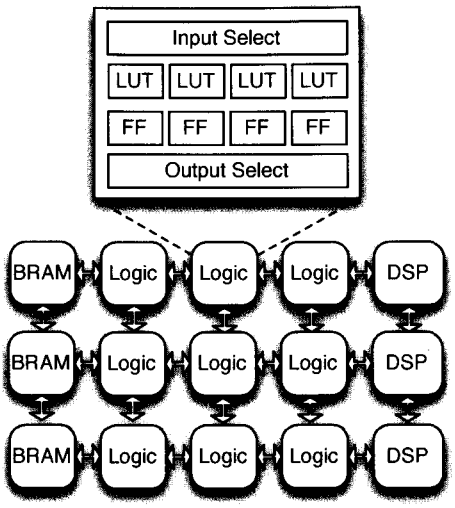



**Figure 3.7** – FPGA Architecture. Inspired by Thomas et al. [46]

It should be pointed out that the H101 has a hierarchal memory structure with quite extreme drops in available bandwidth as shown in Table 3.1. This made it difficult getting data on and off the FPGA as fast as it computed it.



## 3.3 General Purpose Graphics Processing

The video gaming industry has seen substantial growth in recent years and is estimated to be worth $9.5 billion in the U.S. alone [47]. This competitive industry relies largely on visual presentation, which is reliant on the rate that GPUs can compute the video frames. The pixels in a static graphics frame are largely independent and are processed in parallel by multiple graphic pipelines that exist in a single *graphics processing unit* (GPU) [48]. Unlike CPUs which target a variety of application types, the GPU processing is very specific, therefore the GPU's architecture is designed specifically for graphics. Graphics requires a lot of processing and very little complex control, similar to the requirements of correlation. GPUs provide a lot more computational performance than the equivalent CPU generation [49] (see Figure 1.3a).

In 2006, Nvidia, a graphics card manufacturer, released their Compute Unified Device Architecture (CUDA), enabling one to program their graphics pipeline in a standard software environment. This has allowed GPUs to be used in computational applications other than for graphics[9]. Many HPC and graphics algorithms share similar traits and types of computational requirements, which has allowed GPUs to be successfully used in linear algebra [50], database operations [51], $k$-means [52], AES encryption [53] and $n$-body simulations [54] [10].

[9] GPU's have been used to do general purpose processing since GPUs began offering programable shaders in the early 2000's, via 3rd party development tools, such as Brooke. However, these type of tools were a hack to use the graphics pipeline to do other processing. Not until the CUDA GPUs has the GPU hardware been slightly modified to accommodate general purpose processing, allowing for a more refined interface.

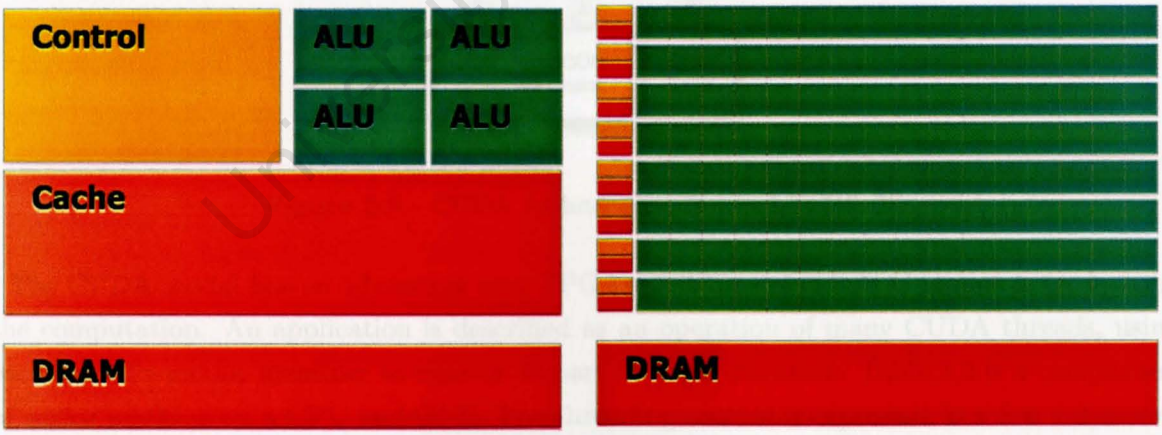
[10] Via [39]

### 3.3.1 Advantages of GPUs

- **Commodity Price** - The ubiquitous success of GPUs has made them affordable high performance hardware. In recent years GPUs have been producing peak performance in an order of magnitude greater than CPUs of the same generation.
- **Large development community** - GPUs have been embraced by the HPC and other general purpose computing domains and there is a large repository of available libraries, tutorials and forums.
- **Backward compatibility and future support** - Nvidia is a financially healthy company, with a clear intent to support the CUDA architecture in the future. Together with the advent of multi-vendor GPGPU OpenCL API, this creates confidence that an investment into GPU software will be supported in future.

### 3.3.2 Programming GPUs

GPUs are large programable parallel processors that can be programmed in a similar way to a CPU [14], however, the large caches and control logic found in CPUs, is either significantly reduced or absent. GPUs instead use the majority of the chip die area to implement ALUs and thus have far greater computational peak performance than CPUs. This reduction in control and on-board memory means that algorithms relying on fast random memory accesses or complex control branches will perform poorly on a GPU - but applications that execute in a predictable instruction flow can achieve much greater throughput.




(a) CPU (b) GPU

**Figure 3.8** – Comparison of transistor expenditure in CPUs and GPUs. Taken directly from CUDA guide [2].

### 3.3.3 CUDA Architecture and its Development Environment

Figure 3.8 shows how transistor space is used on a CPU vs. GPU. CUDA is both a programming library and the GPU architecture created by Nvidia to utilise their GPUs for general purpose

processing. CUDA is not so much a new architecture but more of a re-branding of the GPU architecture, presenting a more suitable API for traditional software developers[11].

Figure 3.9 shows the CUDA GPU Architecture. The fundamental computational unit of the CUDA architecture is a Scalar Processor (SP) which executes CUDA threads. Eight of these SPs, together with a small shared cache are grouped to form a Streaming Multiprocessor (SM). SMs are an analogy to the architecture of SMP multi-core CPUs[12]. The difference is that SMs have a much smaller cache and a single control unit. A single SM administers the scheduling and control for all the SPs (SM behaves very much like a vector unit and schedules vector instructions of length 32, called a warp [55]). If all threads perform the same operation, this operation can be computed in parallel, if not the threads will be serialised. Likewise, if each thread requests linear global memory access, this can be done in a single request, if not, this needs to be serialised. In parallel programs these types of non-divergent operations and memory access patterns are common and GPUs take advantage of this by having one control unit for multiple threads, leaving more transistors for computation.

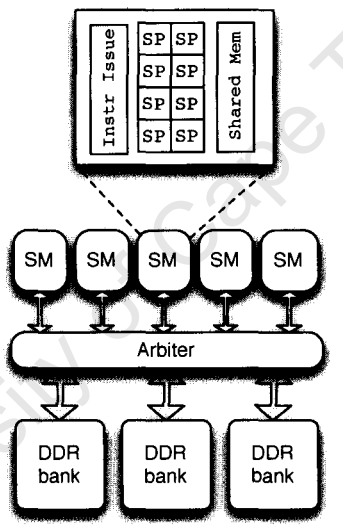



**Figure 3.9** – CUDA Architecture. Inspired by [46, 2].

The CUDA model has an advantage over FPGAs as it uses standard C language to describe the computation. An application is described as an operation of many CUDA threads, using each thread's unique identifier to express its part in the application. Table 3.2 is a comparison of vector addition on a CPU and GPU. The threading control is expressed in a few extensions to the C language. For a more detailed description of the Nvidia CUDA architecture, see the CUDA Programming guide [2].

The number of SMs on a CUDA GPU depends on the model, with entry level GPUs having 1 SM and high-end GPUs having 30[13].

[11] For example the fundamental computation unit in CUDA is the Scalar Processor (SP) which executes a CUDA thread - while a graphics programmer refers to shader processors, which executes a shader programs.
[12] This is a big abstraction
[13] 1 and 30 SM refer to Nvidia 8300 and Nvidia GTX280 GPUs respectively

**Table 3.2** – Comparison of vector addition on a CPU and GPU. The CPU code uses an incrementing loop variable as the index to the array. CUDA code instantiates 'N' threads and uses the unique thread ID as the index. This type of linear addressing works very well on GPUs.

| CPU Code | CUDA Code |
|---|---|
| `for(int i=0; i<N; i++)`<br>`  C[i] = A[i] + B[i];` | `C[thread_id] = A[thread_id] + B[thread_id];` |

### 3.3.4 Zotac 9800 GT GPU Board

In this project we used the Zotac 9800 GT GPU Card (shown in Figure 3.10) for the correlator implementation, the specifications are shown in Table 3.3. and Palatino looks like this.

**Figure 3.10** – Nvidia 9800GT Reference Board [56]

**Table 3.3** – Nvidia 9800GT Specifications [56, 2].

| | |
|---|---|
| **Processor** | 9800 GT GPU (G92) |
| | 112 SPs (14 MPs) @ 1.5GHz |
| **Internal Memory** | 8192 32bit Registers/MP |
| | 16KB Shared Memory/MP |
| **Onboard Memory** | 512MB GDDR3@ 57.6GB/sec |
| **Memory interface** | 256bit |
| **Host Communication Bus** | 16 lane PCI-E 2.0 @ 8GB/s |
| **Maximum SP FLOPS** | 504 GFLOPS |
| **Maximum Power Consumption** | 105W |

## 3.4 Conclusion

Discrete software co-processors like GPUs and FPGAs are an attractive option to accelerate software correlation, potentially offering better FLOPS/watt and FLOPS/$ performance. There are a number of software tools for both technologies, that are designed specifically to accommodate software developers, removing the need for much of the domain specific knowledge to access these technologies. Additionally, FPGA and GPU are both technologies that have had a huge market penetration and have seen sustained growth. This is promising for future support of these technologies, with improved development environments and performance.

Having presented the FPGA and GPU hardware, the next chapter looks at the FPGA X-engine correlator design and implementation on the two Nallatech H101 FPGAs.

# Chapter 4

# FPGA Implementation of Correlator X Engine

This chapter discusses the FPGA implementation of the correlator X-engine. The correlation dealt with in this project is a *four* dimensional problem - this involves two antenna inputs, $i, j$, in a specified frequency band, $v$, at a discrete time interval $a$. This gives us three degrees of parallelism: baseline, time and frequency. In our FPGA implementation, time parallelism was exploited, resulting in an FPGA X-engine correlator which computes eleven time slices simultaneously. We begin with the design of the basic processing element (PE), which is used to compute the baseline correlations for a particular time slice, which is replicated eleven times to create the correlator engine. The final FPGA correlator achieved a speedup up to 7x over a 3.0GHz Xeon CPU.

The FPGA correlator development dealt with three different aspects: *processing resources*, *I/O capabilities* and *control*. These points are discussed below. The FPGA correlator went through an evolutionary process of three different designs. Though they have different approaches to the control of the correlation engine, they share the same processing and I/O design aspects. The three implementations and their divergence in control are discussed after we describe the processing element and I/O design.



## 4.1 Correlation Engine - Creating the pipeline

### 4.1.1 System Overview

In this chapter, we will discuss our final FPGA correlator design, as shown in Figure 4.1. Here we have a hybrid system with the CPU acting as the F-engine and the two FPGAs performing the X-engine operation. The uncorrelated data is read from disk by the CPU performing the F-engine. The output of the F-engine is passed via the PCI-X bus and is divided between the two FPGAs by even and odd channels. The correlated result is then sent back via the PCI-X communication bus and stored on disk.

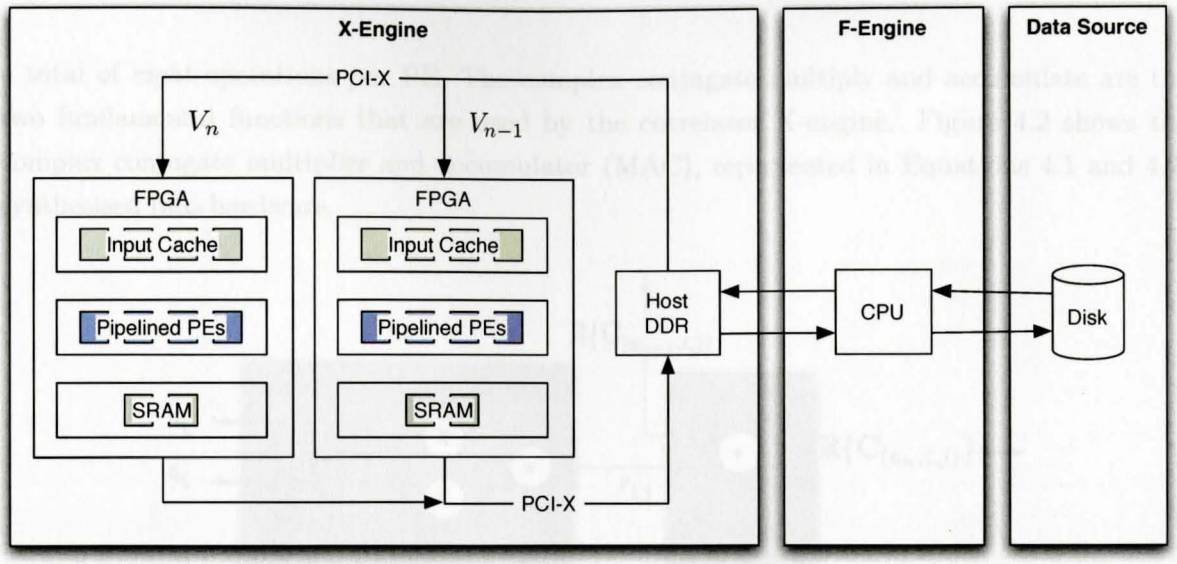



**Figure 4.1** – The FPGA correlator system design.

### 4.1.2 Single Correlator Engine

In this section we present the basic correlator processing element (PE) which computes all the baselines for a certain spectral channel, $v_m$ and time slice, $a_n$. This result is then accumulated for a period before being sent back to the host CPU. This process is repeated for each spectral channel and time slice. Since the input to the correlator is complex valued, the correlator needs to deal with both real and imaginary data.

The basic PE performs the complex conjugate multiplication, which can be simplified to *four* multiplications and *two* additions/subtractions, as shown below:

$$\begin{aligned} S_i[a_n, v_m]S_j^*[a_n, v_m] &= (p_i + \mathrm{j}q_i)(p_j + \mathrm{j}q_j)^* \\ &= (p_i + \mathrm{j}q_i)(p_j - \mathrm{j}q_j) \\ &= \underbrace{(p_i p_j + q_i q_j)}_{P_{a_n,ij}} + \mathrm{j}\underbrace{(q_i p_j - p_i q_j)}_{Q_{a_n,ij}} \end{aligned} \tag{4.1}$$

The result shown in Equation 4.1 is the cross correlation products for a certain baseline, time slice and frequency, and must be accumulated for 'A' time slices, $\mathbf{C}_{ij}[A, v_m]$ :

$$\mathbf{C}_{ij}[A, v_m] = \sum_{a=0}^{A-1} S_i[a, v_m]S_j^*[a, v_m] \tag{4.2}$$

We therefore require *two* more additions for both the real and imaginary parts of $\mathbf{C}_{ij}[A-1, v_m]$, where $\mathbf{C}_{ij}[A-1, v_m]$ represents the running total from the previous time slice $a_{n-1}$. The output to the correlator at time slice $a_n$ is therefore $\mathbf{C}_{ij}[A, v_m]$ as shown:

$$\begin{aligned} \mathbf{C}_{ij}[A, v_m] = \quad & \mathbb{R}\{\mathbf{C}_{ij}[A-1, v_m]\} && + P_{a_n,ij} \\ + \ \mathrm{j} \ & \left(\Im_{\mathrm{m}}\{\mathbf{C}_{ij}[A-1, v_m]\} && + Q_{a_n,ij}\right) \end{aligned} \tag{4.3}$$

This gives us a total of *four* multiplications and *four* addition/subtraction operations, giving

a total of *eight* operations per PE. The complex conjugate multiply and accumulate are the two fundamental functions that are used by the correlator X-engine. Figure 4.2 shows the complex conjugate multiplier and accumulator (MAC), represented in Equations 4.1 and 4.2, synthesised into hardware.

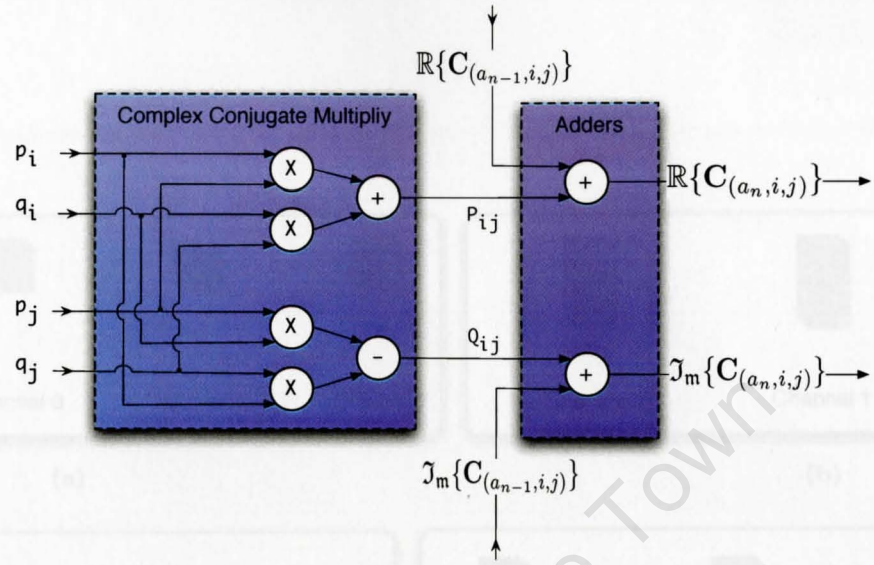



**Figure 4.2** – The basic correlator processing element, which is used to build the correlation X-engine. This computes a correlation product and accumulation for a certain time slice and frequency.

### 4.1.3 Parallel Correlator Engine and Reducing Memory Accesses

The basic PE presented above computes a single complex conjugate MAC per clock cycle[1]. A single PE will compute the entire triangular correlation matrix[2] for a certain time-slice and frequency channel in $N_b$ clock cycles [3]. There are multiple copies of the PE, each computing its own correlation matrix in parallel[4]. Multiple PEs can be connected to exploit the parallelism in either frequency or in time[5]. In frequency parallelism, correlation matrices for different frequency channels are computed independently and concurrently, and each PE computes multiple correlation matrices for different time-slices, as shown in Figure 4.3. In time parallelism, correlation matrices for different time-slices are computed independently and concurrently, and each PE computes multiple correlation matrices for different time-slices, as shown in Figure 4.4. Notice that at stage **(d)** in Figures 4.3 and 4.4, both methods have reached the same point. It should also be noted that the number of PEs is usually less than the number of frequency channels or time steps in the correlation, so the above process has to be repeated. Figure 4.5 is pseudo code describing the different orders of computing the correlation.

[1]Since the PE is pipelined, calculating a CMAC in one clock cycle does not reduce the clockspeed.

[2]We refer to the all the correlation baseline products for a certain time-slice and frequency as the correlation matrix or correlation kernel.

[3]The computation in $N_b$ clock cycles is assuming that memory is already in block RAM, therefore one clock cycle away. If this is not the case, there will be additional overhead. Memory considerations are discussed in

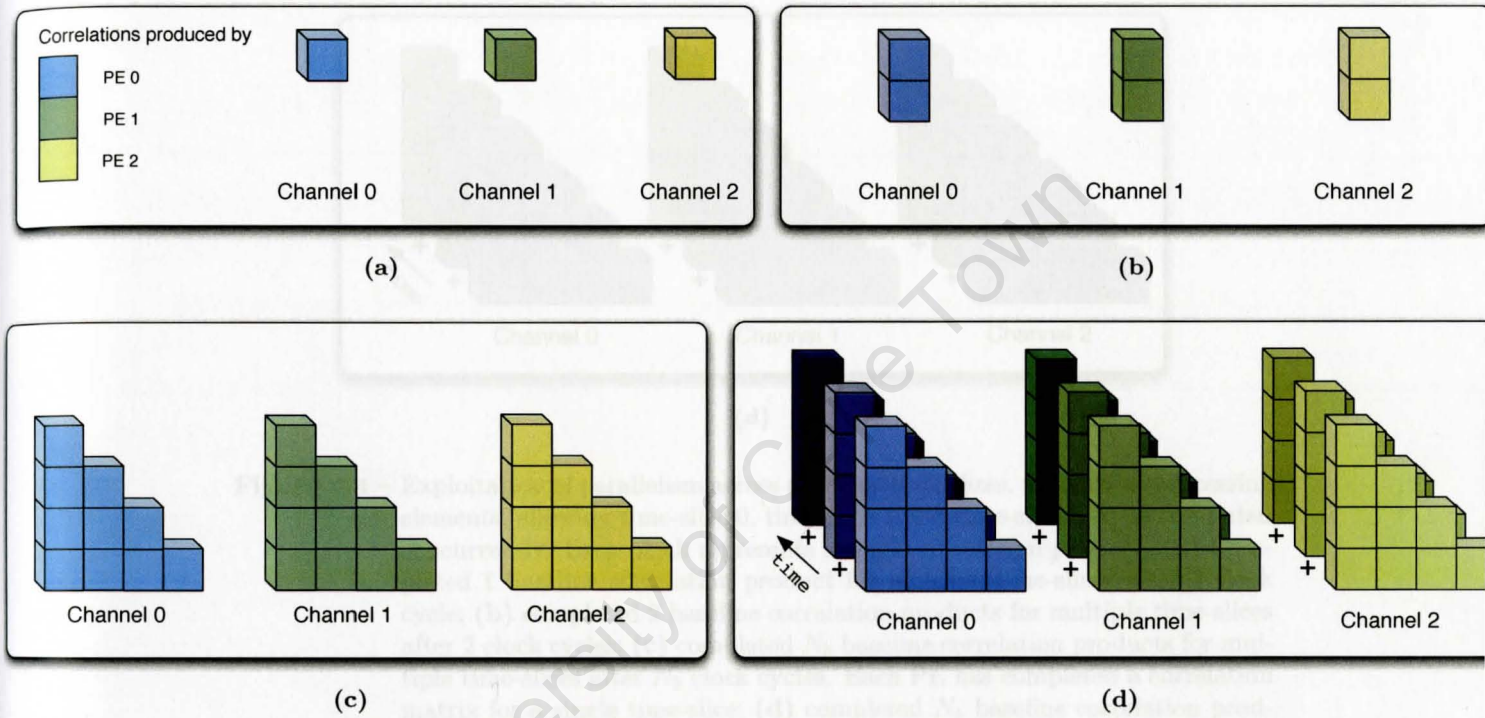



**Figure 4.3** – Exploitation of parallelism across different *frequencies*, using three processing elements, allowing channel 0, channel 1 and channel 2 to be computed concurrently. Each block represents a single correlation product. **(a)** Completed 1 baseline correlation product for multiple channels after 1 clock cycle; **(b)** completed 2 baseline correlation products for multiple channels after 2 clock cycles; **(c)** completed $N_b$ baseline correlation products for multiple channels after $N_b$ clock cycles. Each PE has completed a correlation matrix for a single channel; **(d)** completed $N_b$ baseline correlation products for multiple channels and 3 time slices, after $3N_b$ clock cycles.

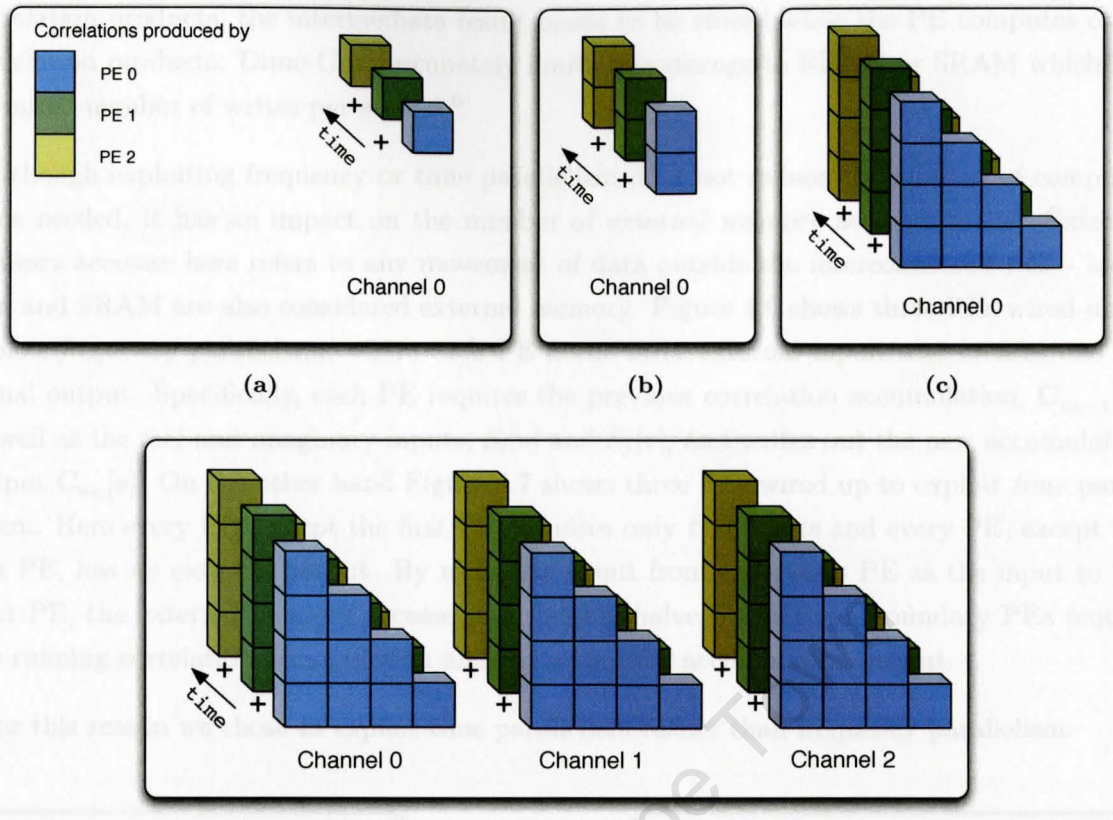



**Figure 4.4** – Exploitation of parallelism across different *time slices*, using three processing elements, allowing time-slice 0, time-slice 1 and time-slice 2 to be computed concurrently. Each block represents a single correlation product. **(a)** Completed 1 baseline correlation product for multiple time-slices after 1 clock cycle; **(b)** completed 2 baseline correlation products for multiple time-slices after 2 clock cycles; **(c)** completed $N_b$ baseline correlation products for multiple time-slices after $N_b$ clock cycles. Each PE has completed a correlation matrix for a single time-slice; **(d)** completed $N_b$ baseline correlation products for multiple time-slices and 3 frequency channel, after $3N_b$ clock cycles.

```
while (observing)
  for m = 0 to accumulation_length
    for n = 0 to num_channels
      compute_correlation_matrix
  send_result_to_host
```

(a)

```
while (observing)
  for n = 0 to num_channels
    for m = 0 to accumulation_length
      compute_correlation_matrix
  send_result_to_host
```

(b)

**Figure 4.5** – **(a)** Pseudo code for computing the correlation matrix for all frequency channels and then accumulating across time-slices, as shown in Figure 4.3. **(b)** Pseudo code for computing the correlation matrix for the full accumulation length and then for all frequency channels, as shown in Figure 4.4.

Each correlation product (represented as a block in Figures 4.3 and 4.4) needs to be accumulated to the correlation product in the next time step. Since each PE is responsible for many

section 4.2

[4]Where the number of PEs depends on the size of the FPGA used

[5]There can of course be a hybrid where time and frequency parallelism are both exploited, but this is not considered here.

correlation products, the intermediate result needs to be stored while the PE computes other correlation products. Dime-C unfortunately limits this storage to BRAM or SRAM which has a limited number of writes per second.[6]

Although exploiting frequency or time parallelism does not reduce the number of computations needed, it has an impact on the number of external memory accesses made. External memory accesses here refers to any movement of data outside the interconnected PEs - block ram and SRAM are also considered external memory. Figure 4.6 shows three PEs wired up to exploit *frequency* parallelism, where each PE needs *three* external inputs and creates *one* external output. Specifically, each PE requires the previous correlation accumulation, $\mathbf{C}_{a_{n-1}}[v]$, as well as the real and imaginary inputs, $S_i[v]$ and $S_j[v]$, and writes out the new accumulation output,$\mathbf{C}_{a_n}[v]$. On the other hand Figure 4.7 shows three PEs wired up to exploit *time* parallelism. Here every PE, except the first PE, requires only *two* inputs and every PE, except the last PE, has *no* external output. By using the result from a previous PE as the input to the next PE, the external memory accesses are roughly halved. Only two boundary PEs require the running correlation accumulation and write the new accumulation output.

For this reason we chose to exploit time parallelism rather than frequency parallelism.

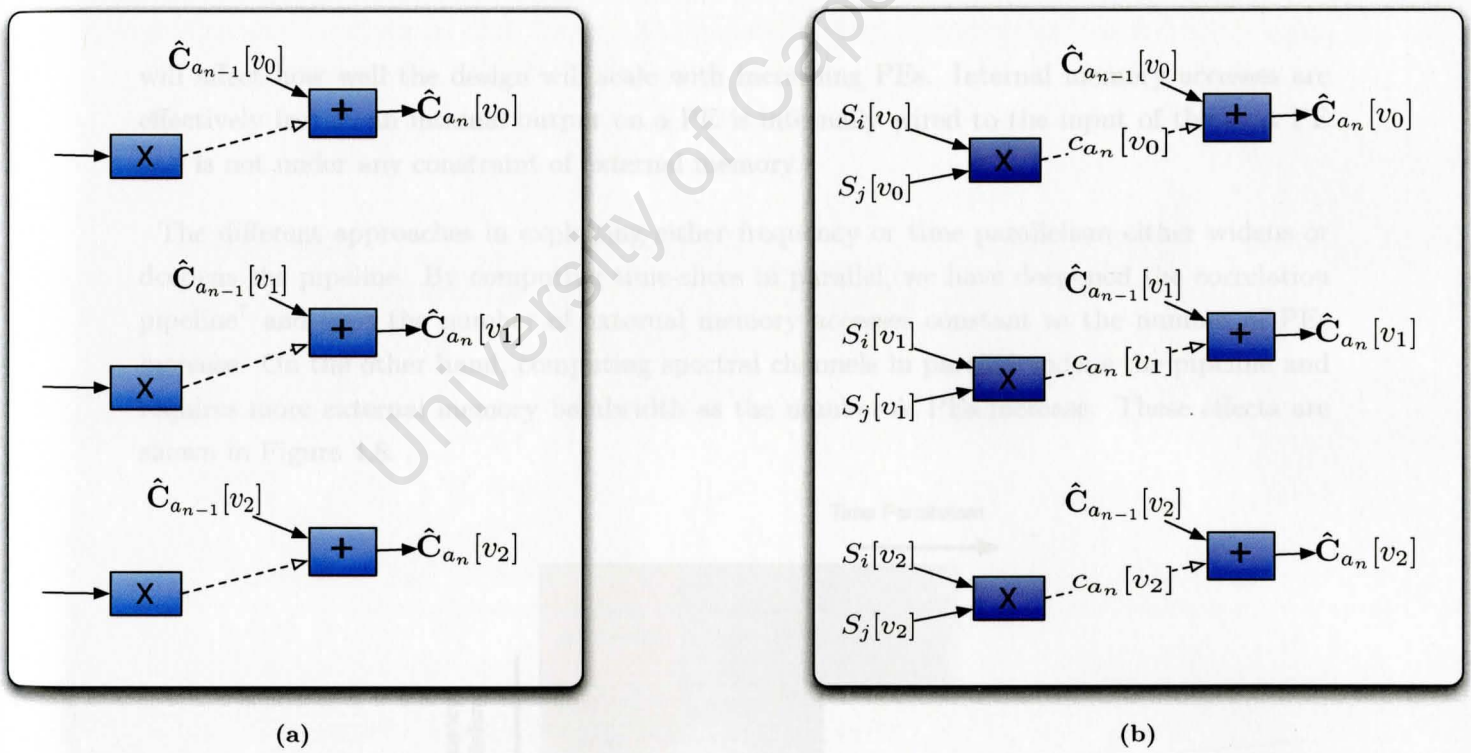



**Figure 4.6** – Correlation X-engine computing multiple *channels* simultaneously. External communications are shown as solid lines and internal communications are shown as dashed lines. **(a)** a simplified diagram not showing all inputs and intermediate outputs which are shown in **(b)**.

Reducing the number of external memory accesses is important as there is a limitation on the number of accesses that can be made per clock cycle (discussed in next section). This

[6]Note this is a limitation of the Dime-C compiler rather than the hardware - ideally each correlation product would be stored in its own register.

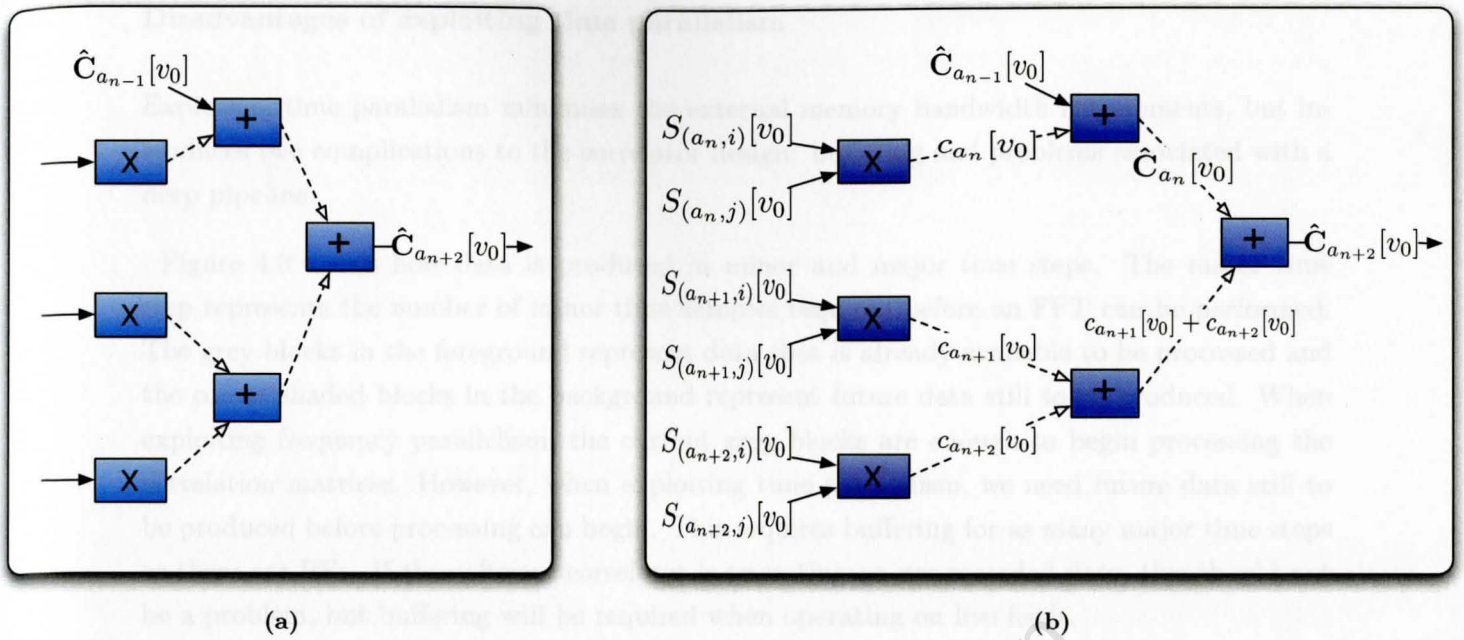



**Figure 4.7** – Correlation X-engine computing multiple *time-slices* simultaneously. External communications are shown as solid lines and internal communications are shown as dashed lines. **(a)** a simplified diagram not showing all inputs and intermediate outputs which are shown in **(b)**.

will affect how well the design will scale with increasing PEs. Internal memory accesses are effectively free as an internal output on a PE is internally wired to the input of the next PE and is not under any constraint of external memory.

The different approaches in exploiting either frequency or time parallelism either widens or deepens the pipeline. By computing time-slices in parallel, we have deepened the correlation pipeline[7] and kept the number of external memory accesses constant as the number of PEs increase. On the other hand, computing spectral channels in parallel widens the pipeline and requires more external memory bandwidth as the number of PEs increase. These effects are shown in Figure 4.8.

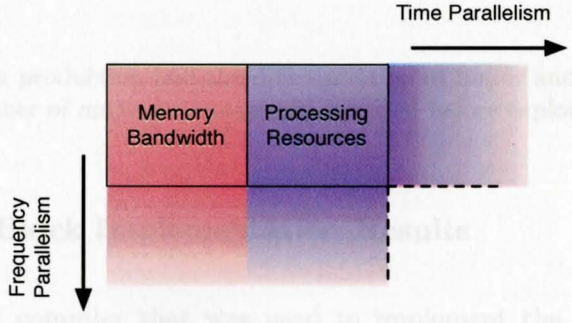



**Figure 4.8** – Deepening the pipeline requires more processing resources, while widening the pipeline requires both more processing and external memory bandwidth.

[7] Creating a systolic array

### Disadvantages of exploiting time parallelism

Exploiting time parallelism minimises the external memory bandwidth requirements, but introduces two complications to the correlator design: buffering and problems associated with a deep pipeline.

Figure 4.9 shows how data is produced in minor and major time steps. The major time step represents the number of minor time samples required before an FFT can be performed. The grey blocks in the foreground represent data that is already available to be processed and the colour shaded blocks in the background represent future data still to be produced. When exploiting frequency parallelism, the current grey blocks are enough to begin processing the correlation matrices. However, when exploiting time parallelism, we need future data still to be produced before processing can begin. This requires buffering for as many major time steps as there are PEs. If the software correlator is operating on pre-recorded data, this should not be a problem, but buffering will be required when operating on live feeds.

The second complication is that by exploiting time parallelism, a deep pipeline is created. A deep pipeline does not affect the throughput, but increases the pipeline latency. This increased latency becomes a problem when control hazards, caused from branches, are introduced into the pipeline. When a branch occurs, the entire pipeline needs to be flushed before the next computation can begin. The larger the pipeline latency, the larger the branching penalty. Removing these control hazards is dealt with in 4.3.

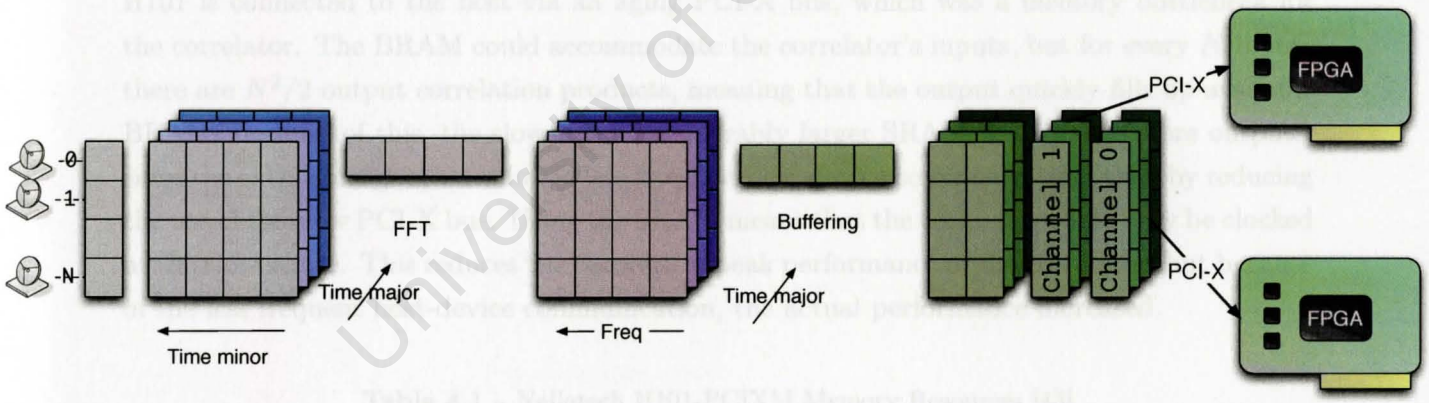



**Figure 4.9** – Data production and the differentiation of major and minor time steps. A number of major time steps are required before exploiting time parallelism.

### 4.1.4 Correlator Block Implementation Results

The Nallatech Dime-C compiler that was used to implement the correlator, only supports traditional data types and does not have any native support for fixed-point arithmetic. For this reason, all data storage and arithmetic in the correlator uses 32bit floating point numbers[8]. Fixed-point arithmetic would most likely allow better utilisation of the FPGA hardware, but Dime-C doesn't have any native fixed point support, so the conversion would have to be done manually. Additionally Dime-C uses the Xilinx Core Generator for floating point arithmetic,

[8] All data was complex and separate float arrays were used for the real and imaginary numbers.

which conforms to the IEEE-754 standard [42], making it more convenient to validate the output with the CPU correlator's output. The conversion to fixed point arithmetic falls outside the scope of the project and is left for future work.

The Virtex 4 LX100 FPGA has enough resources to synthesise 11 PEs, using floating point arithmetic. We had two Nallatech cards at our disposal, giving us a total of 22 PEs. This meant we could compute 22 correlation products every clock cycle. Each PE consisted of 8 FPUs and the FPGA was clocked at 100MHz, resulting in a theoretical peak performance of 17.6 GFLOPs.

## 4.2 I/O Management - Feeding the pipeline

Supplying the processing engines with an uninterrupted flow of data is the ultimate goal of parallel computation, as starvation causes under utilised resources. Data needs to be shipped to the FPGA co-processor as efficiently as possible and stored in the most suitable memory type and location to provide enough memory bandwidth for all PEs.

The H101 has both external SRAM and internal Block RAM memory banks as shown in Table 4.1. Both types of memory are arranged into banks and each bank has a limited number of accesses per clock cycle[9]. The Block RAM allowed our correlation design to be clocked at 155MHz while the SRAM can only be clocked at a slower rate of 100MHz. The Nallatech H101 is connected to the host via an aging PCI-X bus, which was a memory bottleneck for the correlator. The BRAM could accommodate the correlator's inputs, but for every $N$ inputs there are $N^2/2$ output correlation products, meaning that the output quickly fills up available BRAM. Because of this, the slower but considerably larger SRAM was used to store outputs, since the extra storage space allowed less frequent host-device communication, thereby reducing the use of the slow PCI-X bus. Using the SRAM meant that the correlator could only be clocked at the slower rate. This reduces the theoretical peak performance of the correlator, but because of the less frequent host-device communication, the actual performance increased.



**Table 4.1** – Nallatech H101-PCIXM Memory Resources [43]

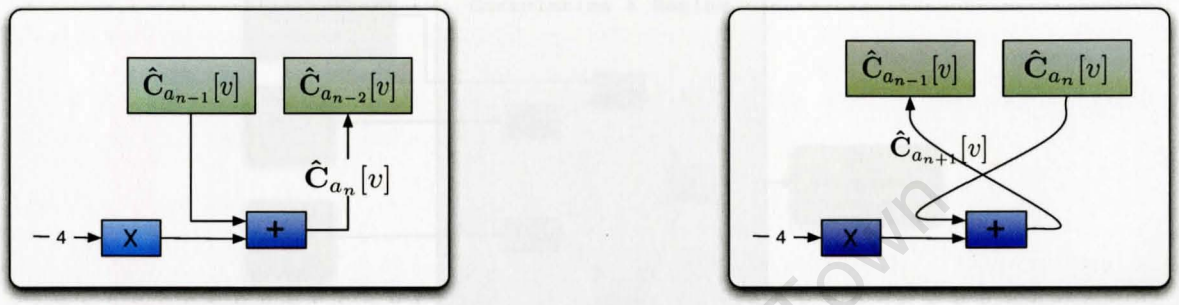
| | **Static RAM** | **Block RAM** |
|---|---|---|
| **Banks** | 4 | 240 |
| **Size/Bank** | 4MB | 16kbits |
| **Total Size** | 16MB | 480KB |
| **Bank Accesses/Clock Cycle** | 2 | 1 |
| **Clock Rate** | 100MHz | 70 - 250MHz |

Using FIFO buffers allows for asynchronous data transfers to the FPGA, which should minimise the host-device communication overhead. Surprisingly, asynchronous data transfers faired

[9] Block ram has one read/write operation per bank and SRAM can perform two read/write operations per bank.

worse than synchronous data transfers. This shouldn't be the case, but investigating the cause of this inefficiency was left for future work.

Double buffering[10] was introduced to arrays requiring more than one access per clock cycle. In all three correlator implementations, the correlation output needed an intermediate buffer to support the two accesses per clock cycle - this is shown in Figure 4.10. A similar double buffering scheme was introduced for the correlation input in one of the designs presented in section 4.3.




**(a)** clock tick t **(b)** clock tick t+1

**Figure 4.10** – Double Buffering of the output.

### 4.2.1 Memory Use in the Correlation Engine

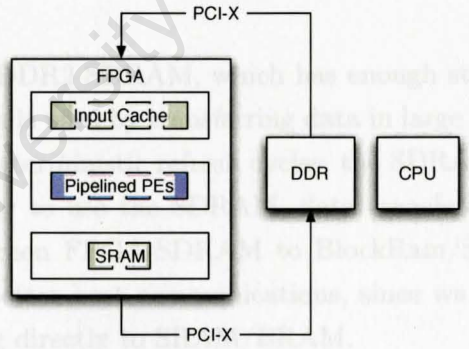



**Figure 4.11** – Memory arrangement of the correlator X-engine.

Figure 4.11 shows the memory arrangement for the correlation engine. The input data was stored in internal cache, composed from BRAM banks. Each cache bank can be accessed once per clock cycle. The Dime-C compiler will only create pipelined processing elements if this is not violated. Unless the PEs are pipelined, the performance is poor and therefore it is crucial to double buffer the input in multiple cache banks. Figure 4.12 shows the data flow of a pipelined and non-pipelined Dime-C processing block. See Appendix C.2 for more details on pipeline and parallel execution on FPGAs. Figure 4.13 shows the correlation engine presented in Figure 4.7 connected to the respective memory interfaces.

[10]BRAM is dual ported, but because it provides input to the correlator, one port is connected to the host and the other to the FPGA

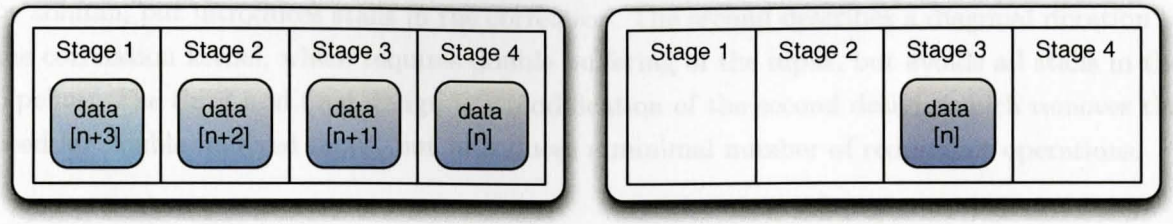



**Figure 4.12** – A 4 stage Dime-C processing block which has been fully pipelined in **(a)** and serialized **(b)**. **(a)** would have four times the throughput of **(b)**

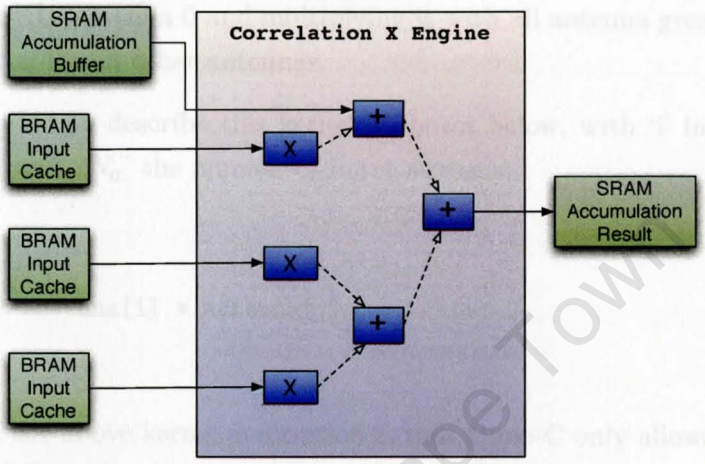



**Figure 4.13** – Correlation X-engine and its external memory interfaces.

### 4.2.2 Dynamic RAM

The H101 also has 512MB DDR2 SDRAM, which has enough storage to dramatically reduce the frequency the PCI-X bus is used by transferring data in large chunks at a rate of 400MB/s. However, because of the indeterministic refresh cycles, the SDRAM cannot be used for DIME-C pipelined access. In order to use the SDRAM, data transfers happen in two steps: PCI-X to FPGA-SDRAM and then FPGA-SDRAM to BlockRam/SRAM. This added overhead outweighed the benefit of better host communications, since we were achieving data rates of about 300MB/s transferring directly to SRAM/BRAM.

## 4.3 Control - Keeping the Pipeline Full

The FPGA's correlator X-engine's performance is reliant on computing the correlation in parallel using multiple PEs, which has been discussed in Section 4.1 and Section 4.2. This section looks at maintaining the data flow to the pipeline of a particular PE. Each PE computes the correlation products for all baselines of a particular time-slice and frequency channel. Branching in the data flow introduces pipeline stalls, which must be avoided if possible. The penalties for pipeline stalls are particularly severe because the correlation engine is deeply pipelined.

In this section we present three correlator designs, each with a different description of the data flow. The first implementation has a more natural way of describing the correlation, column

by column, but introduces stalls in the correlator. The second describes a diagonal iteration of the correlation kernel, which requires double buffering of the input, but avoids all stalls in the pipeline. The third and final design is a modification of the second design, which removes the need for double buffered input, but introduces a minimal number of redundant operations.

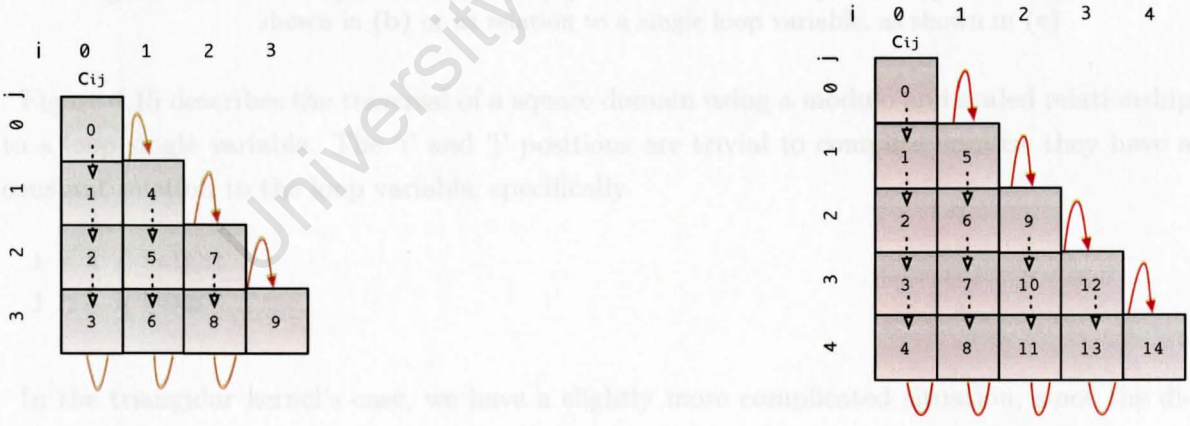
### 4.3.1 Design 1: Nested Loop

The original nested loop implementation of the triangular correlator kernel, as described in Section 2.2, describes the correlation in an intuitive way, computing the kernel column by column. ie starting with antenna 0 and multiplying it with all antenna greater than and equal to itself and repeating for all other antennas.

The pseudo-code used to describe this kernel is shown below, with 'i' indexing the column antenna, 'j' the row and ''$N_a$' the number of input streams:

```
for i = 0 to Na
   for j = i to Na
       c[i,j] += antenna[i] * antenna[j]
```

The problem with the above kernel description is that Dime-C only allows for the innermost loop to be pipelined. Therefore for each column, the pipeline stalls, introducing $N_a \times L$ bubbles in the pipeline, where $L$ is the pipeline latency. Figure 4.14 shows the kernel operations and branches when there are 4 and 5 antennas in the array.




**(a)** 4 antennas **(b)** 5 antennas

**Figure 4.14** – Computation of the correlation with a nested loop PE. The stalls in the pipeline are shown with red arrows. There will always be a pipeline latency L even with no branches, but in **(a)** there is an additional 3.L stalls, giving us a total of 10 + 3L + L clock cycles and in **(b)** an additional 4.L stalls, giving us a total of 15 + 4L + L clock cycles. These pipeline stall penalties could be avoided by using a different design.

With this nested loop description, it takes $\frac{N_a(N_a+1)}{2}$ cycles to compute the baselines and $N_a.L$ cycles overhead for the pipeline stalls[11]. We also have to transfer the data across the the PCI-X bus which we will denote as $\mathcal{T}$. The total number of cycles taken is shown in Equation 4.4.

[11] This is again the baselines for a specific time slice and spectral channel.

$$Cycles = \left(\frac{N_a.(N_a+1)}{2} + N_a.L\right) + \mathcal{T} \tag{4.4}$$

### 4.3.2 Design 2: Single Loop with Double Buffering

The previous implementation involved two loop variables to describe the triangular shape of the correlation operations. The problem with this description is that *only* the inner loop can be pipelined, affecting the performance of the correlator. What we want is a one dimensional description of the correlators kernel which can be done by flattening or coalescing the nested loop into a single loop. Describing the correlator as a single loop will result in a fully pipelined solution, reaching close to peak performance.

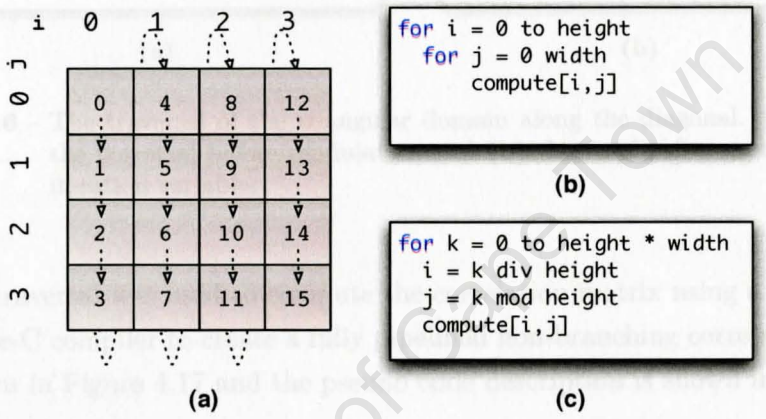



**Figure 4.15** – The square domain in **(a)** can be traversed by two loops variables, as shown in **(b)** or as relation to a single loop variable, as shown in **(c)**

Figure 4.15 describes the traversal of a square domain using a modulo and scaled relationship to a loop single variable. The 'i' and 'j' positions are trivial to compute because they have a constant relation to the loop variable, specifically

```
i = k / height
j = k % height
```

In the triangular kernel's case, we have a slightly more complicated situation, since the dimensions of the domain are not constant. In order to flatten the two loops into a single loop, we need to relate a common loop variable to 'i' and 'j'. The solution we used was to iterate down the diagonal of the triangular domain. By using modulo arithmetic, the diagonal length was constant. From this diagonal constant we could derive 'i' and 'j' from a *single* loop variable. The diagonal iteration of the triangular domain is shown in Figure 4.16.

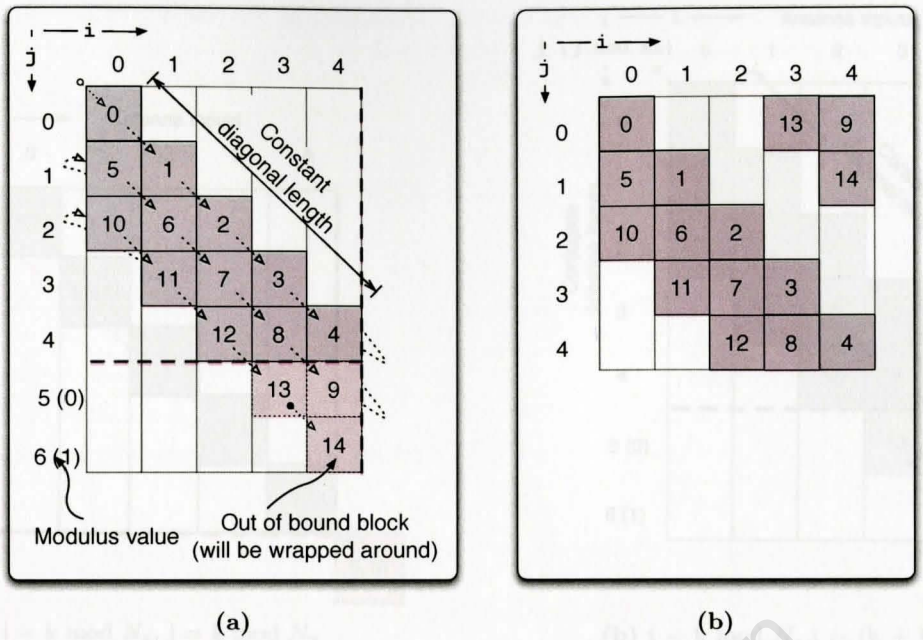
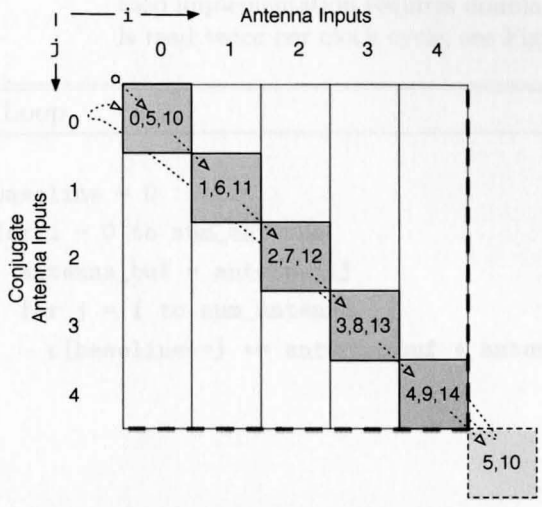
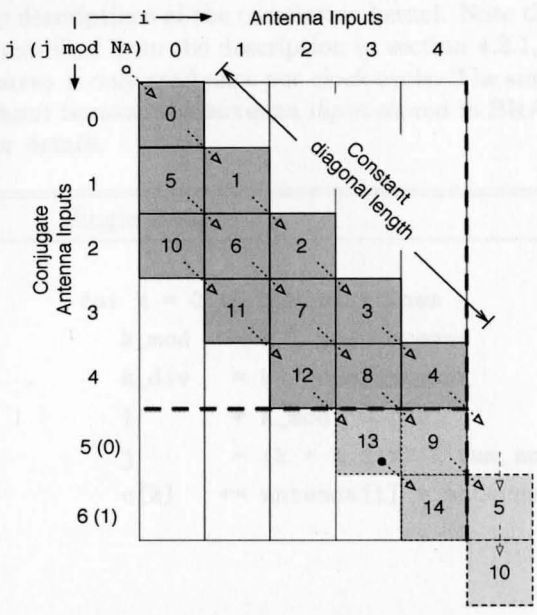



**Figure 4.16** – The traversal of the triangular domain along the diagonal. **(a)** showing the traversal before modulation and **(b)** the result after modulating the iteration variable.

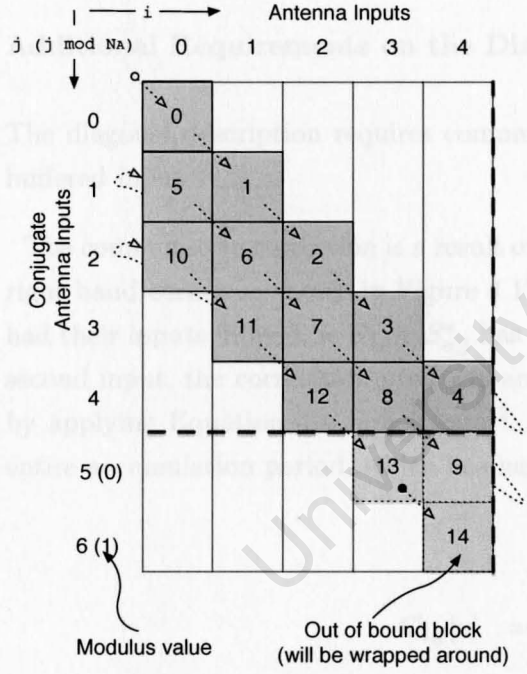
This diagonal traversal was used to compute the correlation matrix using a single loop. This allowed the Dime-C compiler to create a fully pipelined non-branching correlation engine. The traversal is shown in Figure 4.17 and the pseudo code description is shown in Table 4.2.



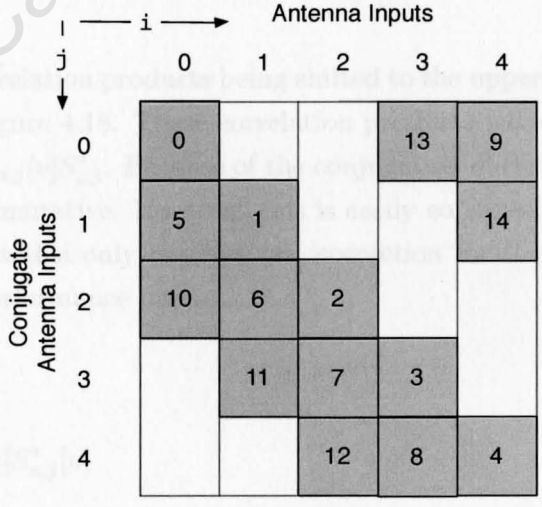

**(a)** i = k mod $N_a$, j = k mod $N_a$




**(b)** i = k mod $N_a$, j = (k + k/$N_a$) mod $N_a$




**(c)**




**(d)** 15 + L

**Figure 4.17** – In this figure we illustrate the single loop correlation engine behaviour. Each block represents a baseline corresponding to antenna 'i' and 'j'. The number on the block records the value of the incrementing variable 'k' at particular values of 'i' and 'j'. The unshaded blocks and dashed borders show which blocks will be 'wrapped around' using modulo arithmetic. **(a)** is the result if we increment down the diagonal and modulate on the dashed borders, which will result in repetition of the main diagonal. Instead what we need is to increment 'j' twice on multiples of $N_a$ as shown in **(b)**. This extra incremental results in the kernel we want as shown in **(c)** before modulo along the 'j' axis and in **(d)** after the 'j' axis modulo. More examples are shown in Appendix E.1.

**Table 4.2** – A comparison of the nested loop and single loop descriptions of the correlation kernel. Note that the nested loop pseudo code has been slightly modified from the description in section 4.3.1, to include an antenna buffer so that the `antenna` array is only read once per clock cycle. The single loop implementation requires double buffered input because the `antenna` input stored in BRAM is read twice per clock cycle, see Figure 4.17 for details.

| Nested Loop | Single Loop |
|---|---|
| `baseline = 0`<br>`for i = 0 to num_antenna`<br>`  antenna_buf = antenna[i]`<br>`  for j = i to num_antenna`<br>`  c[baseline++] += antenna_buf * antenna[j]` | `for k = 0 to num_baselines`<br>`  k_mod = k % num_antenna`<br>`  k_div = k / num_antenna`<br>`  i = k_mod`<br>`  j = (k + k_div) % num_antenna`<br>`  c[k] += antenna[i] * antenna[j]` |



### Additional Requirements on the Diagonal Description

The diagonal description requires commutative correction, more complex control and double buffered input.

The commutative correction is a result of some correlation products being shifted to the upper right hand corner as shown in Figure 4.17d and Figure 4.18. These correlation products have had their inputs flipped, ie $S_{a,i}[v]S^*_{a,j}$ has become $S_{a,j}[v]S^*_{a,i}$. Because of the conjugation of the second input, the correlation products are *not* commutative. However, this is easily corrected by applying Equation 4.5 in software[12]. Note that this only requires one correction for the entire accumulation period, which has negligible performance impact.

$$\begin{aligned} \mathbf{C}_{i,j}[v] &= \sum_{a=0}^{A-1} S_{a,i}[v]S^*_{a,j}[v] \\ &= \left( \sum_{a=0}^{A-1} S^*_{a,i}[v]S_{a,j}[v] \right)^* \end{aligned} \tag{4.5}$$

In this implementation, more complex control is needed, as the single loop requires modulo and division arithmetic which would have performance implications on a microprocessor, since this would typically take more than a single cycle to compute. Fortunately, using FPGAs, complex control only results in more logic utilisation and can still be computed in a single cycle and so adds no major overhead.

Double buffering was required since the single loop description loads a new 'i' *and* 'j' value every clock cycle, because of its diagonal iteration. Double buffering provides the means to

[12] See Appendix G.2 for commutative derivation details.

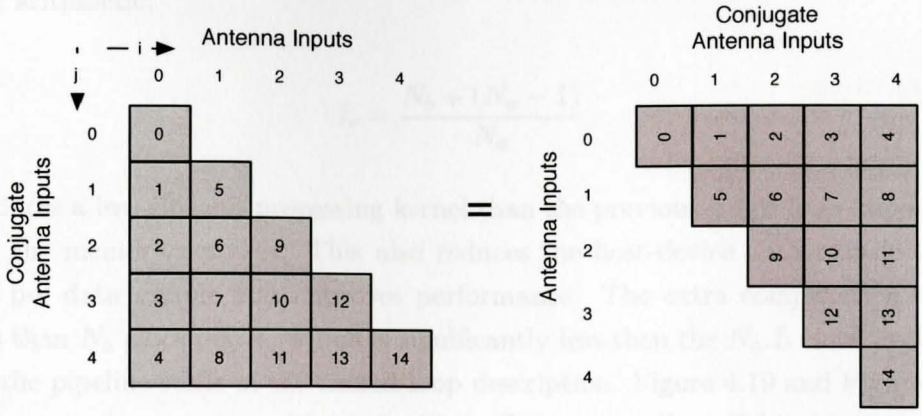



**Figure 4.18** – This figure shows that the correlation kernel can be computed as either antenna[i] × antenna[j] or antenna[j] × antenna[i], as long as commutative correlation in Equation 4.5 is applied on the mirrored outputs.

access the same memory cache twice per clock cycle. However, the repercussions are halving the available input cache and increasing the host-device data transfers to fill the extra buffer. Removing the double buffer is addressed in the next design.

#### Performance and Final Design

The performance in clock cycles of the single loop implementation can be described as:

$$cycles = \frac{N_a(N_a+1)}{2} + L + 2.\mathcal{T}, \tag{4.6}$$

where $N_a$ is the number of antennas, $L$ the pipeline latency and $\mathcal{T}$ the host-device transfer delay. Therefore there are $N_a.(L-1)$ fewer pipeline stalls than there are in the nested loop implementation. However, in this implementation, there are twice as many host-device transfers to fill the double buffered inputs.

### 4.3.3 Design 3: Single loop without double buffered input

The single loop implementation discussed above in section 4.3.2 requires double buffering of the input, which halves the already limited BRAM and increases the host-device communication.

Removing the double buffering can be accomplished if some redundant operations are added. By traversing down a fixed size column, we avoid the varying length columns of the nested loop implementation and only require a single loop variable. We can also remove the double buffering requirement in the previous single loop implementation, as we only need to load a new 'j' value each clock cycle, with the 'i' value being copied from the 'j' value at the start of each new column.

The length of the fixed column, $l_c$, is the smallest multiple of antenna $N_a$ that includes all the baselines $N_b$:

$$l_c = \lceil \frac{N_b}{N_a} \rceil$$

in integer arithmetic:

$$l_c = \frac{N_b + (N_a - 1)}{N_a}$$

This produces a less efficient processing kernel than the previous single loop implementation, but halves the memory accesses. This also reduces the host-device data transfers, increases operations per data sample and improves performance. The extra computation overhead is always less than $N_a$ clock cycles, which is significantly less then the $N_a.L$ clock cycle overhead caused by the pipeline stalls in the nested loop description. Figure 4.19 and Figure 4.20 show the single loop implementation without double buffering operation. Table 4.3 is a pseudo code comparison of the two single loop implementations.

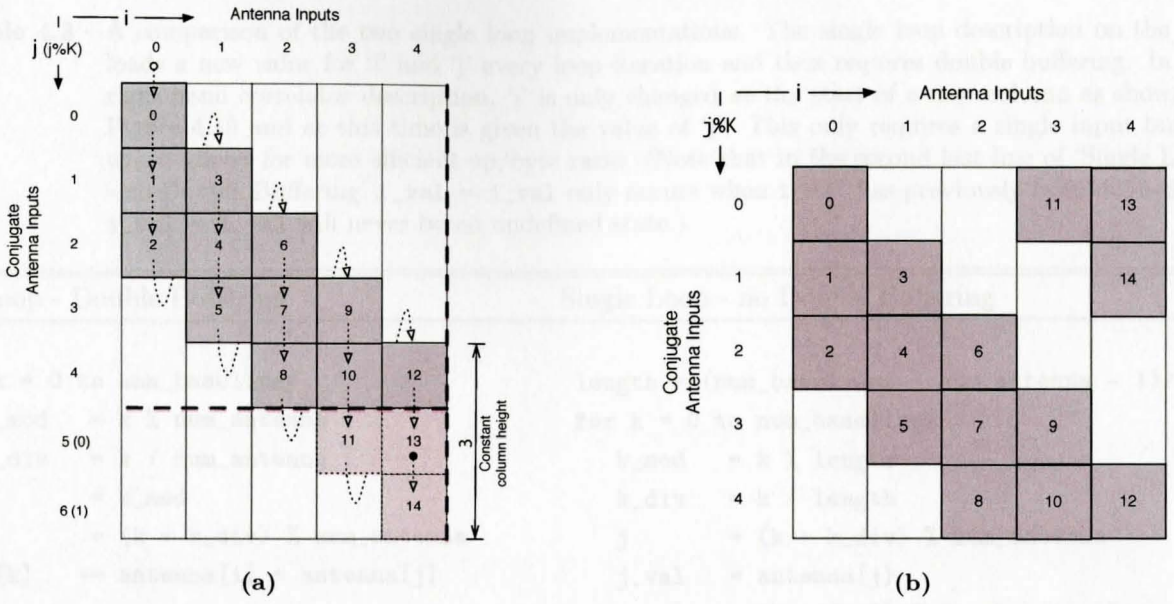



**Figure 4.19** – Computing the correlation matrix using the single loop without requiring double buffered input when $N_a = 5$. Here $l_c = \lceil \frac{N_b}{N_a} \rceil = \lceil \frac{15}{5} \rceil = 3$. This requires $l_c.N_a + L = 15 + L$ clock cycles.

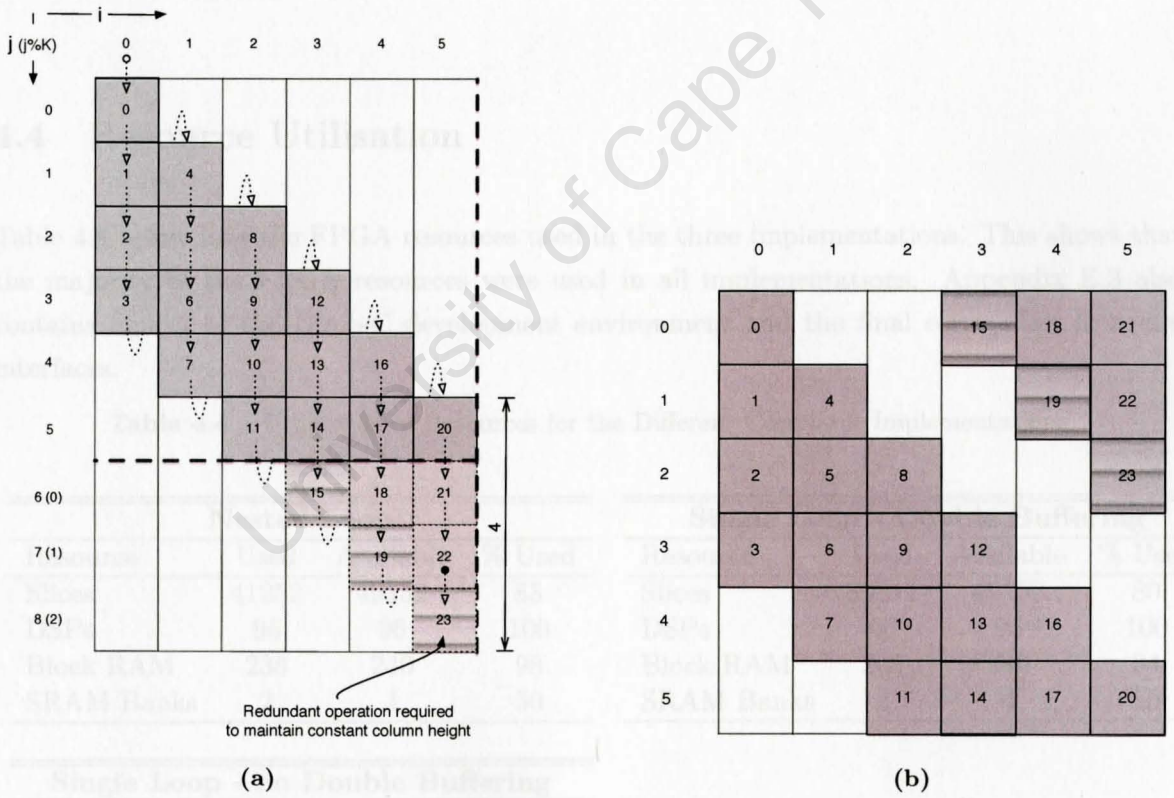



**Figure 4.20** – Computing the correlation matrix using the single loop without requiring double buffered input when $N_a = 6$. Here $l_c = \lceil \frac{N_b}{N_a} \rceil = \lceil \frac{21}{6} \rceil = 4$. This requires $l_c.N_a + L = 24 + L$ clock cycles. The redundant operations are shown as striped blocks. In this example there are 3 redundant outputs.

**Table 4.3** – A comparison of the two single loop implementations. The single loop description on the left loads a new value for 'i' and 'j' every loop iteration and thus requires double buffering. In the right hand correlator description, 'i' is only changed at the start of a new column as shown in Figure 4.19 and at this time is given the value of 'j'. This only requires a single input buffer, which allows for more efficient op/byte ratio. (Note that in the second last line of 'Single Loop - no Double Buffering' `i_val = i_val` only occurs when `i_val` has previously been defined, so `i_val = i_val` will never be an undefined state.)

| Single Loop - Double Buffering | Single Loop - no Double Buffering |
|---|---|
| `for k = 0 to num_baselines`<br>`  k_mod = k % num_antenna`<br>`  k_div = k / num_antenna`<br>`  i = k_mod`<br>`  j = (k + k_div) % num_antenna`<br>`  c[k] += antenna[i] * antenna[j]` | `length = (num_baselines + num_antenna - 1)/num_antenna`<br>`for k = 0 to num_baselines`<br>`  k_mod = k % length`<br>`  k_div = k / length`<br>`  j = (k + k_div) % num_antenna`<br>`  j_val = antenna[j]`<br>`  i_val = (k_mod==0) ? j_val : i_val;`<br>`  c[k] += i_val * j_val` |



## 4.4 Resource Utilisation

Table 4.4 below lists the FPGA resources used in the three implementations. This shows that the majority of the FPGA resources were used in all implementations. Appendix E.3 also contains figures of the Dime-C development environment and the final correlation firmware interfaces.

**Table 4.4** – Utilisation of Resources for the Different Correlator Implementations

| **Nested Loop** | | | |
|---|---|---|---|
| Resource | Used | Available | % Used |
| Slices | 41252 | 49152 | 83 |
| DSPs | 96 | 96 | 100 |
| Block RAM | 236 | 240 | 98 |
| SRAM Banks | 2 | 4 | 50 |

| **Single Loop - Double Buffering** | | | |
|---|---|---|---|
| Resource | Used | Available | % Used |
| Slices | 39812 | 49152 | 80 |
| DSPs | 96 | 96 | 100 |
| Block RAM | 203 | 240 | 84 |
| SRAM Banks | 2 | 4 | 50 |

| **Single Loop - no Double Buffering** | | | |
|---|---|---|---|
| Resource | Used | Available | % Used |
| Slices | 45075 | 49152 | 91 |
| DSPs | 90 | 96 | 93 |
| Block RAM | 191 | 240 | 79 |
| SRAM Banks | 2 | 4 | 50 |

## 4.5 Conclusion

The single loop implementation of the X-engine, without double buffering, managed to achieve a 7x speedup over the single threaded 3.0GHz Xeon Harpertown implementation. The X-engine design utilised the majority of the available resources on the FPGA, as shown in Table 4.4, meaning our X-engine has grown to the capacity of the Virtex 4LX100 without under utilising resources. In addition, all the pipeline hazards were removed. These two factors resulted in a satisfactory optimised implementation. In Chapter 6, we discuss and elaborate on the performance of the FPGA X-engine.

Having presented the FPGA X-engine in detail, we discuss the GPU implementation in the next chapter.

# Chapter 5

# GPU Correlator Implementation

In this chapter, we discuss the GPU correlator design and implementation. The GPU CUDA correlator design was based on work done by Harris et al. [14]. Harris's idea is to take advantage of CUDA's multiple hardware threads and initialise a square domain of threads, ignoring the triangular shaped correlation kernel. This will create dormant threads, but also create a simplified square correlation kernel. The lightweight nature of CUDA threads results in the dormant threads adding little memory and processing overhead. The outcome is a clean description of a square kernel, with a small overhead, and efficient linear memory addressing (coalesced memory accesses). We were able to achieve a 12.5x speedup over the CPU implementation.



## 5.1 Design

### 5.1.1 System Overview

As with the FPGA correlator chapter, we begin by presenting a system overview of the GPU correlator. Figure 5.1 shows a hybrid system of an F-engine CPU and the GPU performing the X-engine operations. The uncorrelated data from disk is processed by the CPU which performs the FFT channelisation. The output of the CPU's F-engine is passed to the GPU via the PCIe bus. The work is divided between the GPU's Streaming Multiprocessors(SM) where each SM performs part of the correlation. The result is fed back and stored on disk.

### 5.1.2 Design Considerations

Figure 5.2 shows how Nvidia's CUDA GPU hardware is comprised of a number of vector processes, called Streaming Multiprocessors (SMs), which execute a program called a block. Since the number of SMs varies between generations and models, a CUDA application is typically written with far more blocks than SMs. Each block will then typically be responsible for a small portion of the entire application. In our case, each block calculated a baseline for all frequencies and time.

Each SM is composed of 8 scalar processors (SPs), which are the processing elements which actually execute CUDA block programs. Each block program consists of up to 512 threads,

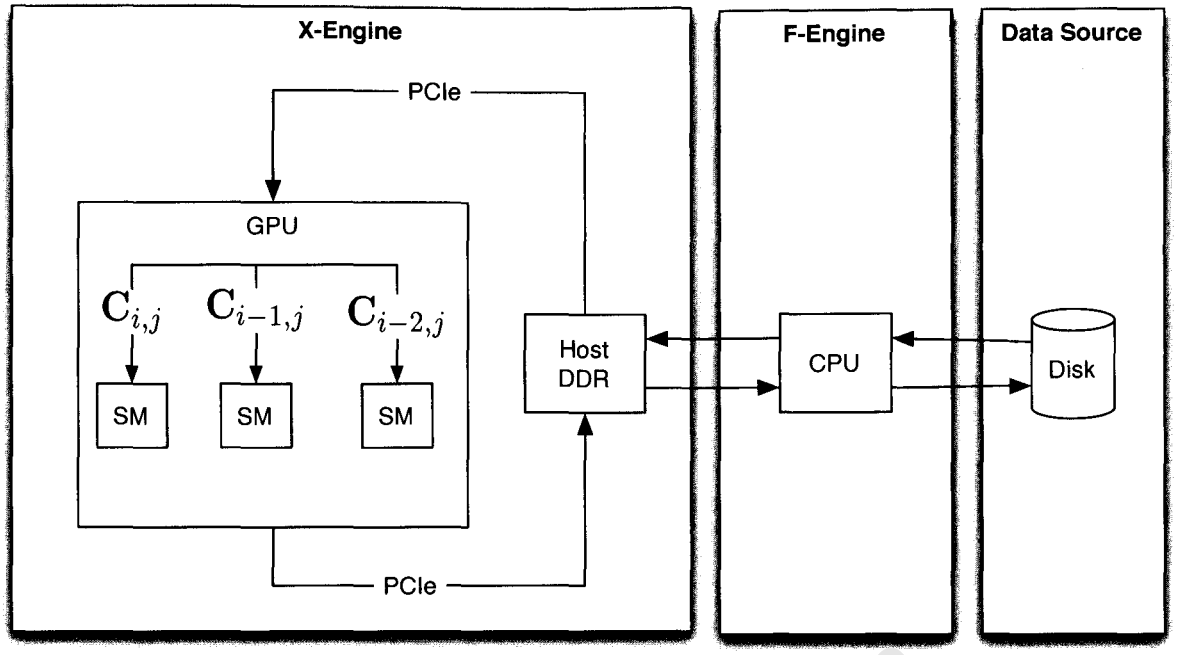



**Figure 5.1** – The GPU correlator system design.

with each thread describing the operation each SP must execute [2]. Although there are 8 SPs per SM, there is only one instruction issue unit [55]. For each clock cycle, each SP has a choice to perform the issued operation or not to. If all SPs are performing the same operation in SIMD fashion, they operate on 8 data locations in parallel, however, if they need to perform different operations, their execution is serialised. Therefore it is important that groups of 32 threads, called a warp, within the block program are performing the same instruction on their unique data [1].

The SMs are all connected to global memory via a common memory bus. Linear access to global memory greatly improves data throughput, so in addition to blocks executing in SIMD fashion, it is important to access sequential groups of data. This required antenna data to be packaged in the order the SMs will read to ensure high memory throughput [2].

### 5.1.3 X-Engine Design

With the design considerations mentioned above, the GPU X-engine needs to divide the computation of the correlation kernel into blocks, which can run independently. Each block needs to perform its section of work by accessing linear memory addresses to ensure coalesced memory access.

Figure 5.3 (a) shows the approach suggested by Harris [14], which we used to implement our correlator X-engine. Here, each baseline was allocated to a separate block of code. Each thread in the block is responsible for the correlation of a specific frequency channel within that baseline as shown in Figure 5.3 (b). Therefore we are exploiting frequency and baseline parallelism.

[1] The reason that a warp is 32 and not 8 is presumably to simplify thread scheduling and to allow for the number of SPs per SM to grow in future generations.

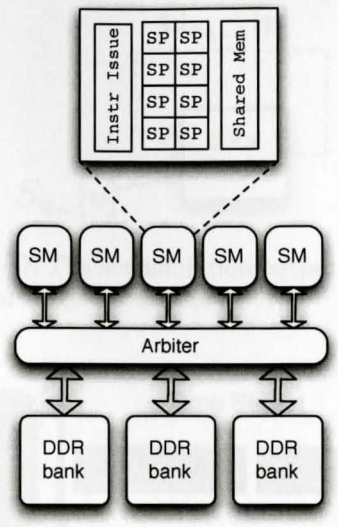
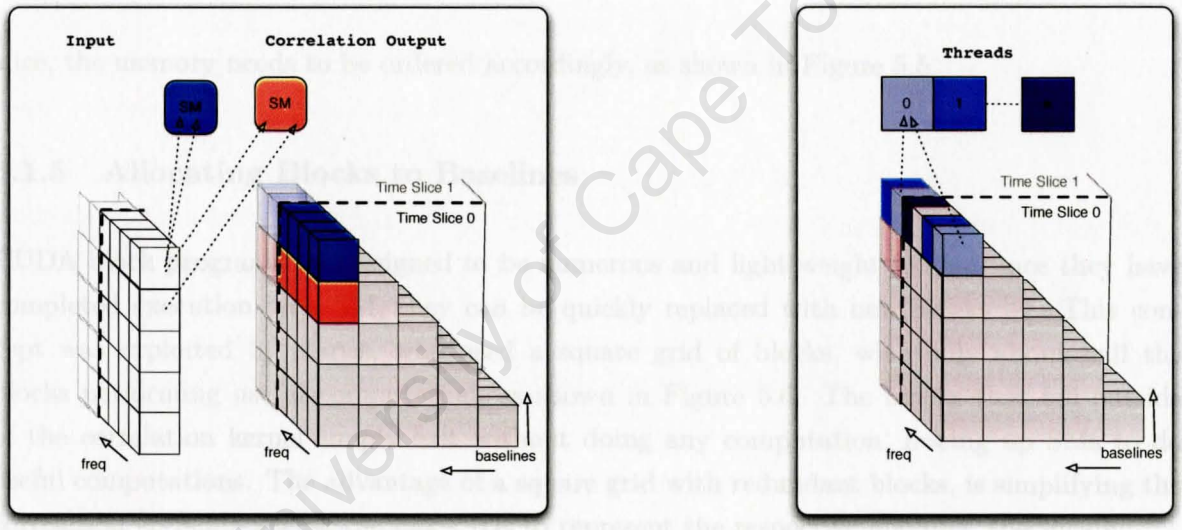



**Figure 5.2** – CUDA Architecture. Inspired by Thomas et. al. [46, 2].




**(a)** Block Operation

**(b)** Thread Operation

**Figure 5.3** – GPU X-engine computation. In **(a)** each block is responsible for a single baseline for all frequencies and time slices. **(b)** shows that each thread in a block is only responsible for a single frequency, but for all time slices.

Exploiting frequency parallelism does not require more global memory accesses than time parallelism as in the case of the FPGA implementation, since the intermediate accumulated result can be stored in a buffer. Since each thread is only ever responsible for one frequency channel, it does not need to write out the accumulation result until completion, as shown in Figure 5.4.

### 5.1.4 Memory Ordering

To ensure coalesced memory accesses, we need to store the correlation input in a linear fashion. Since each subsequent thread in a block is accessing a subsequent frequency for a specific time

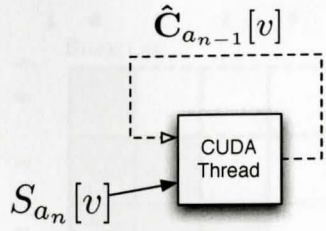



**Figure 5.4** – CUDA thread I/O. The accumulation output is stored in the thread register and isn't written to global memory.

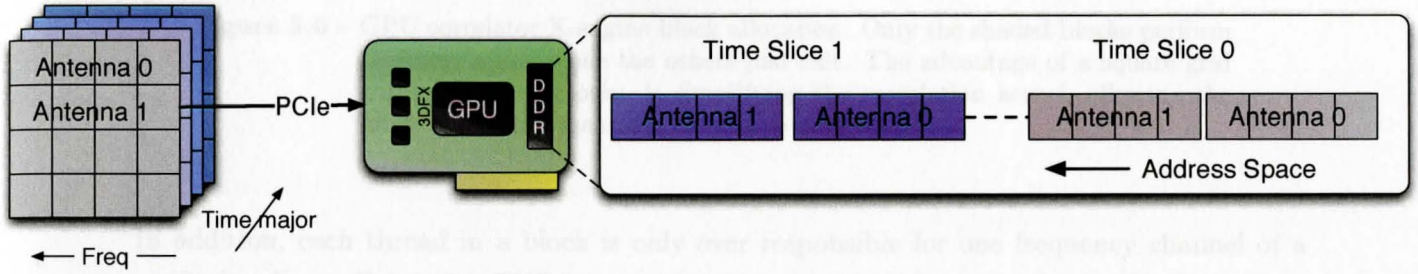



**Figure 5.5** – GPU Memory Management

slice, the memory needs to be ordered accordingly, as shown in Figure 5.5.

### 5.1.5 Allocating Blocks to Baselines

CUDA block programs are designed to be numerous and light weight so that once they have completed execution on a SM, they can be quickly replaced with new blocks [2]. This concept was exploited by Harris, who used a square grid of blocks, with only about half the blocks performing useful computation, as shown in Figure 5.6. The blocks that fall outside of the correlation kernel simply exit without doing any computation, freeing up SMs to do useful computations. The advantage of a square grid with redundant blocks, is simplifying the correlation kernel, allowing the block IDs to represent the respective antenna, specifically:

```
//blocks part of the correlation kernel
if BlockID_i <= BlockID_j
   corr += antenna[BlockID_i] * antenna[BlockID_j]

//blocks outside the correlation kernel
else
   terminate
```

### 5.1.6 Limitations of Design

The current GPU implementation allocates one block per baseline. Current Nvidia GPUs have between 2 and 30 SMs, therefore if there are fewer baselines than SMs on a GPU, the GPU is not being fully utilised. This is only a problem for small array experiments.

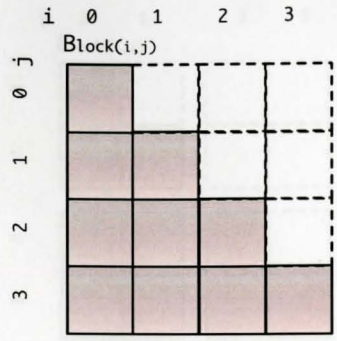



**Figure 5.6** – GPU correlator X-engine block allocation. Only the shaded blocks perform the correlation, while the others just exit. The advantage of a square grid with redundant blocks, is simplifying the correlation kernel, allowing the block IDs to represent the respective antenna.

In addition, each thread in a block is only ever responsible for one frequency channel of a specific baseline. Currently, CUDA supports a maximum of 512 threads per block and the correlator implementation can therefore only compute correlations with 512 or less frequency channels. This could easily be a problem that would limit certain correlation experiments. However, it should be relatively straightforward to expand a threads responsibilities to more than a single frequency. This is left for possible future work and was not addressed in this dissertation.

Nvidia state that Cuda can theoretically support up to $2^{16}$ blocks [2]. Since each baseline is computed in a block, this means that the GPU correlator can compute up to $2^{16}$ baselines - however we never tested this limit.

## 5.2 Implementation on Nvidia Geforce 9800GT

The Nvidia Geforce 9800 GT[2] (G92) that was used in this project has 14 SMs, each containing 8 SPs. Therefore 112 correlation products are computed simultaneously (8 different frequencies within the 14 baselines). Since the different SMs on a GPU act independently to compute different baselines, the same design should scale to a larger GPU or a GPU cluster with more SMs. Memory bandwidth is always a potential bottleneck, but according to specifications, newer GPUs' memory bandwidth has scaled with their compute capabilities (Nvidia GTX280) [13].

## 5.3 Optimisation

Harris also suggests other approaches to computing the correlation matrix, including a group parallel approach as shown in Figure 5.7. In this design, a thread's responsibility is extended to more than one baseline. This reduces the global memory access required, since many of the baselines computed by a thread have the same 'i' and 'j' antenna input. Note, however, redundant threads are still used to describe the triangular correlation kernel.

[2]The Geforce 9800GT is based on the same architecture as the Geforce 8800GT, both based on the G92 Nvidia architecture.

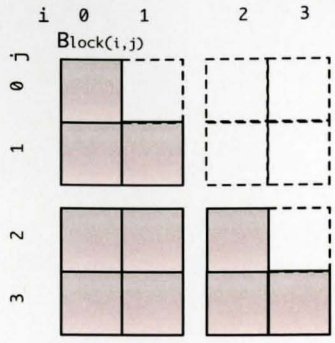



**Figure 5.7** – The group parallel approach suggested by Harris. In this example each thread is responsible for 4 baselines. The redundant baseline allocations are the hollow blocks.

Although CUDA threads contain far less context than CPU threads, there is still some overhead to thread creation, scheduling and context switching. Because of these overheads, the redundant thread blocks suggested by Harris [14] should have some performance impact. The MWA GPU correlator [16] also borrowed ideas from Harris, but removed the redundant blocks, presumably with some performance increase.

Besides these two optimisations, careful tuning of the CUDA code, using information reported by the CUDA profiler and other 3rd party applications can make a substantial increase in SP occupancy and memory access performance [3].

Neither of the two optimisations were implemented, nor did major code tuning take place. The reason for this is that the GPU correlator mainly served as a means to benchmark and justify the FPGA correlator.

## 5.4 Conclusions

The X-engine GPU implementation achieved a 12.5x speedup over the single threaded 3.0GHz Xeon Harpertown implementation. This speedup has been achieved with relatively little programming effort compared to the FPGA implementation. This demonstrates the suitability of GPU architecture to X-engine correlation. In the next chapter, we will discuss and evaluate the Nallatech H101s and Nvidia CUDA GPUs for radio astronomy correlation.

[3] PTX assembly code and Decuda help provide useful insight into a CUDA program's performance profile [55].

# Chapter 6

# Performance Results and Discussion

This chapter presents and discusses the performance, scaling potential and power utilisation of the co-processor implementations[1].

We compare the co-processors' performance against the CPU correlator implementation, which makes use of the CPUs vector SSE instructions. Both correlator implementations were tested on a range of antenna input streams and spectral channels. Speedups of 7x and 12.5x were achieved on the FPGA and GPU correlator implementations respectively. While the GPU delivers consistent performance, the FPGA performs poorly with 64 and fewer antenna streams. Ignoring the time it took to move data from host to co-processor, speedups of 10.5x and 13.5x were achieved on the FPGA and GPU correlator implementations respectively.

Although both implementations achieved speedups and better power utilisation than the CPU implementation, the GPU implementation produced better performance in a shorter development time than the FPGA. The FPGA implementation was hampered by the development tools and the slow PCI-X bus, which is used to communicate with the host[2].

We begin this chapter by presenting a variety of performance results from our correlator implementations. This is followed by an evaluation of the co-processor implementations and a performance comparison with other existing correlators. We end the chapter by concluding with the results of our correlator implementations and discuss the areas where they succeeded and areas which still require work.



## 6.1 Benchmark Environment and Method

In this section we describe the testing environment in which the correlator results were obtained.

### 6.1.1 Runtime Measurement

Benchmark runtimes include the total time or wall time, which includes the overhead of transferring the input and receiving the output from the co-processors, as shown in Figure 6.1. To

[1]Power utilisation was not measured directly but instead power estimation tools provided by the vendors were used.

[2]The bus speed is a limitation of the vendor board not inherently of the FPGA.

get high resolution timing, Intel Performance Primitive Libraries were used [57]. All transfers were done synchronously, although asynchronous transfers could hide some of the data transfer latency, which is left for future work.

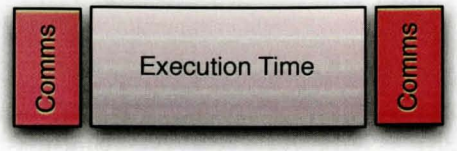



**Figure 6.1** – Typical Execution Time Contribution

### 6.1.2 Correlator Input

All input to the correlator was synthetic, single polarisation, complex-valued data, represented in floating point format. Extensions to real world data and dual polarisations can be extended as future work. Table 6.1 summarises the correlator input details[3].

**Table 6.1** – Benchmark Experiment Configuration

| Accumulation period | Polarisation | Sample Representation |
|---|---|---|
| 1000 time-slices | single | complex 64bit floating point (2×32bit floats) |



### 6.1.3 Validation

The outputs of the two co-processors, as well as the optimised CPU correlator were compared with each other. Float rounding errors were considered and a small variation in output was allowed, typically $10^{-6}$. Although the Nvidia 9800GT does not adhere to IEEE-754 spec, the output never deviated outside of our allowable error range. See Appendix D for more details on output validation.

### 6.1.4 Benchmark Platforms

Table 6.2 shows the platforms used to run performance benchmarks for the three correlator implementations.

[3] Auto-correlations were calculated in all experiments.

**Table 6.2** – Benchmark System Configurations

| | CPU | GPU | FPGA |
|---|---|---|---|
| **Processor** | Intel Xeon Harpertown X5450 | Nvidia Geforce 9800GT | Xilinx Virtex 4LX100 |
| **Clock Rate** | 3.0GHz | 1.5GHz | 100MHz |
| **Manufacturer** | Dell | Zotac | Nallatech H101 |
| **No. of Processors** | 1 | 1 | 2 |
| **Cores per Processor** | 4 | 14 | 1 |
| **No. Cores Used** | 1 | 14 | 1 |
| **Maximum SP FLOPS** | 48 GFLOPS | 504 GFLOPS | 20GFLOPS |
| **Avg. Power Usage** | 120W | 105W | 25W |
| **Host Machine** | Dell Xeon | Dell Core 2 Duo | Dell Xeon |
| **Host OS** | Ubuntu 8.04 x64 | Ubuntu 8.10 x86 | CentOS 5.2 x64 |

### 6.1.5 Notes on Benchmarks

The Nallatech H101 host machine was populated with *two* H101s which our FPGA correlator implementation took advantage of. The workload was then divided by frequency and split between the two cards. Therefore, if there are $v$ frequency channels, FPGA card *one* calculates channels 1 to $\frac{v}{2}$, while FPGA card *two* calculates channels $\frac{v}{2}+1$ to $v$.

The CPU implementation takes advantage of SSE vector instructions, but is a single threaded application only executing on a single core. In Section 6.2.2 we normalise the performance results to give a fairer comparison.

### 6.1.6 Arithmetic Intensity

An important concept for co-processor acceleration is arithmetic intensity. Arithmetic intensity is the ratio of arithmetic operations to memory operations [2]. The FPGA and GPU co-processors have better computational performance than the CPU, but data needs to be transferred to and from the co-processor, which is an additional overhead that doesn't apply to CPU correlator. Correlation experiments with a high computational density re-use the same data in a number of different calculations, reducing the percentage of time spent in host-device communication.

In Chapter 2.5.1 we discussed the computational requirements of the X-engine and saw how the computation scaled linearly with frequency channels, '$N_c$' and quadratically with antennas, '$N_a$'. Table 6.3 looks at the computation and communication requirements of the X-engine:

**Table 6.3** – Computation vs communication as the number of antennas and frequency channels increase.

| | Computation | Communication | Arithmetic Intensity $\frac{Comp.}{Comms.}$ |
|---|---|---|---|
| **Antennas** | $\frac{N_a(N_a+1)}{2}$ | $N_a$ | $\frac{(N_a+1)}{2}$ |
| **Frequency Channels** | $N_c$ | $N_c$ | 1 |

In this table we expect to see better co-processor performance for experiments with a large number of antennas, while the number of frequency channels should have little effect on performance.

## 6.2 Final Implementation Benchmark Results

To help us evaluate the performance of our correlator implementations, we present a variety of results testing different aspects of performance. More specifically:

i. GFLOPS

ii. Bandwidth per antenna stream

iii. Clock cycles required

iv. Speedup vs CPU

We also look at other aspects of our correlation implementations, including:

i. Host-device communication

ii. FPGA implementation comparison

iii. Power and performance ratios

iv. Detailed analysis of the speedup

v. Performance normalisation

vi. FFT performance



### 6.2.1 General Performance Results

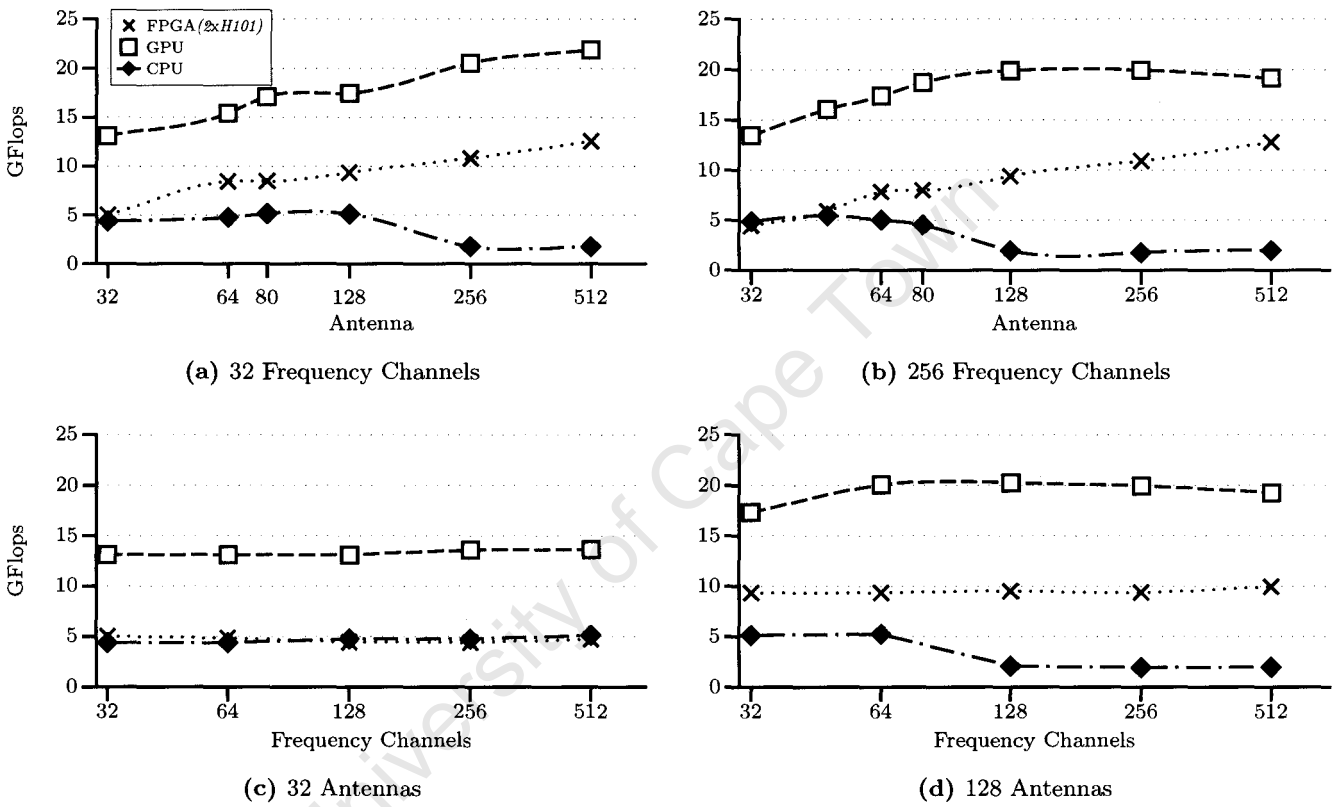
In this section, we look at four important performance criteria which demonstrate the overall performance of the correlator implementations. The next section will investigate more specific performance criteria.

The figures in this section are formatted such that the top row, graphs **(a)** and **(b)**, are the results obtained when running the correlation experiment with a fixed number of frequency channels, while the bottom row, graphs **(c)** and **(d)**, show the results of running the correlation experiment with a fixed number of antennas.

### Performance in GFLOPS (GFLOP/sec)

This section explores the effect the number of frequency channels and antennas have on the GFLOPS of the correlator implementations. The results report how fast the correlator implementations can perform off-line correlation.

The GFLOPS were calculated as follows: for '$N_b$' baselines, '$N_c$' frequency channels and '$A$' time-steps, the correlator performs $N_b.N_c.A$/runtime complex MAC per second. With 8 FLOPS per complex MAC, the correlator's performance is $8.N_b.N_c.A$/runtime FLOPS, which in terms of antennas is $8\frac{N_a(N_a+1)}{2}N_c.A$/runtime FLOPS.




(a) 32 Frequency Channels

(b) 256 Frequency Channels

(c) 32 Antennas

(d) 128 Antennas

**Figure 6.2** – GFLOPS obtained on the correlator implementations.

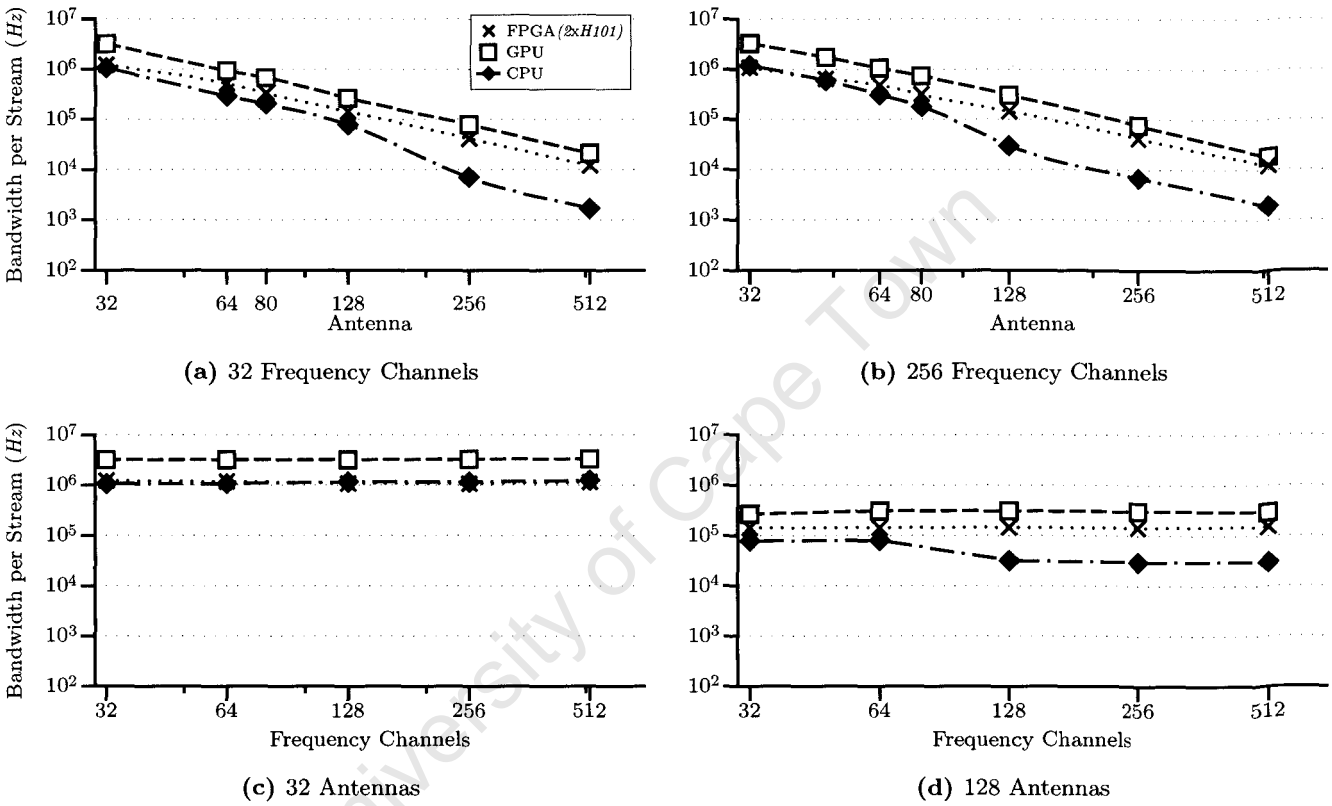
Figure 6.2 graphs the performance of the three correlator implementations, measured in GFLOPS. The GPU outperforms the other implementations by a wide margin. Both the GPU and FPGA's performance improve as the number of antennas increase, which increases the compute intensity and decreases the percentage of time spent in device-host communication as discussed in section 6.1.6. However, the FPGA's performance improves more significantly as the number of antenna inputs increases and comes closer to matching the GPU's performance. The greater impact that the increased arithmetic intensity has on the FPGA's performance suggests that the FPGA has a greater communication overhead than the GPU. Increasing the number of frequency channels in the experiment has little effect on the correlator's performance, since it doesn't affect the computation to data transfer ratio.

There exists a knee in the CPU performance for all graphs, except in **(c)**. This is likely to be attributed to the cacheing effect when the correlation dataset for a specific time slice exceeds

the Xeon's 3MB L2 cache per core. **(c)** has the smallest dataset and never more than the CPU's 3MB cache is required, explaining the absence of the knee.

### Real-Time Bandwidth per Antenna

This section explores the effect the number of frequency channels and antennas have on the bandwidth per antenna on the correlator implementations. The results report the maximum bandwidth that can be processed with a live data feed. Obviously, higher antenna bandwidths can be processed offline - but couldn't be performed in real-time.




**(a)** 32 Frequency Channels

**(b)** 256 Frequency Channels

**(c)** 32 Antennas

**(d)** 128 Antennas

**Figure 6.3** – Real-Time Bandwidth per Antenna

Figure 6.3 graphs the effect the number of frequency channels and antennas have on the real-time bandwidth per antenna, '$B$', for the correlator implementations[4][5]. Each correlator implementation is capable of computing roughly a constant number of CMAC/s [6]. The number of CMAC/s required for a single polarisation is $B.N_b$, therefore as $N_b$ grows exponentially with $N_a$, we see an exponential drop in $B$ as shown in **(a)** and **(b)**.

In **(c)** and **(d)** $N_b$ is constant, so $B$ is also constant. This shows that the number of antennas has little effect on the bandwidth, except for the CPU in **(d)**, which has a drop in performance due to cacheing effect mentioned in Figure 6.2.

[4]Note Figure 6.3 plots the *log* of bandwidth of antennas.

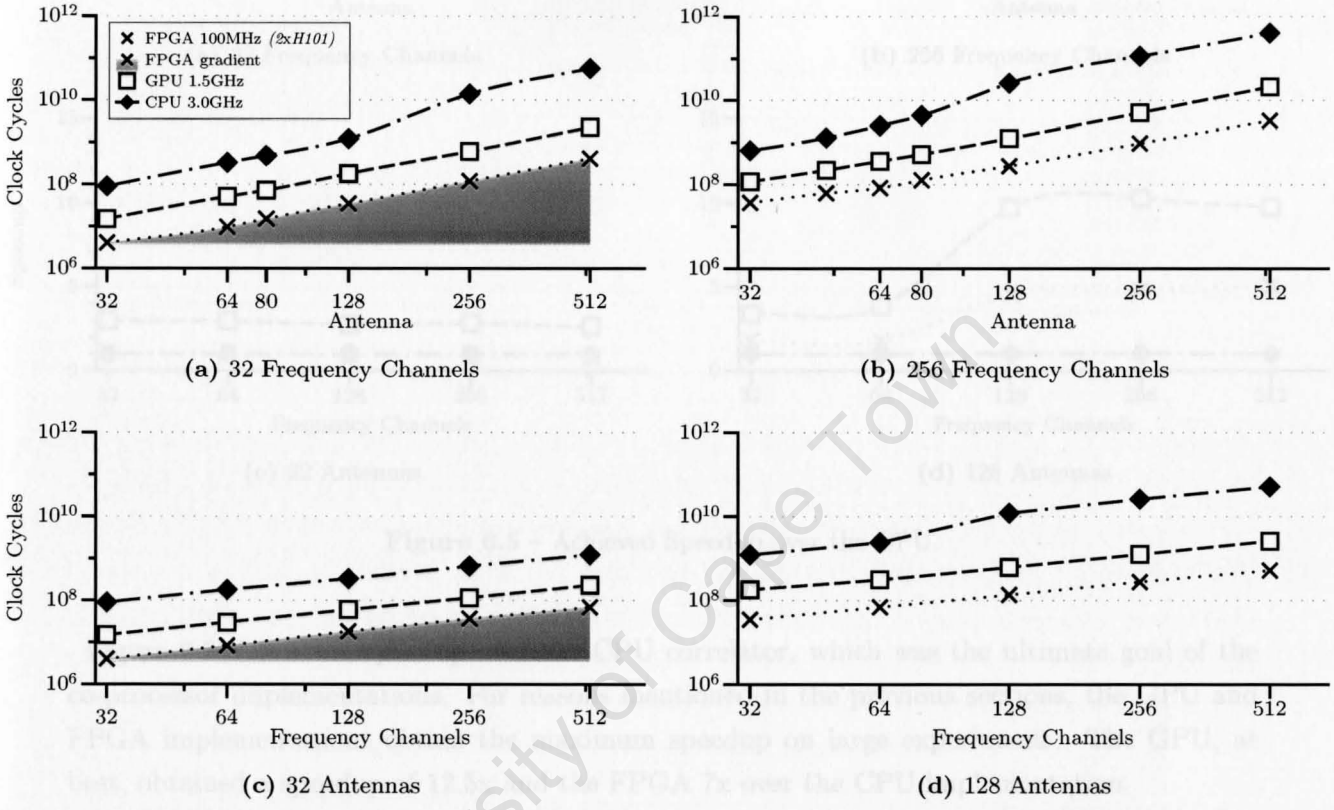
[5]The antenna input was assumed to be in analytic representation, therefore sampling occurred at half the Nyquist rate.

[6]Each correlator implementation is only capable of computing *roughly* a constant number of CMAC/s, there is performance variation as shown in Figure 6.2.

### Total Number of Clock Cycles Required

This section explores the effect the number of frequency channels and antennas have on the number of clock cycles required to compute the correlation for 1000 time-slices.

The number of clock cycles required to compute the various correlation experiments was calculated by runtime × clock-rate.




**(a)** 32 Frequency Channels

**(b)** 256 Frequency Channels

**(c)** 32 Antennas

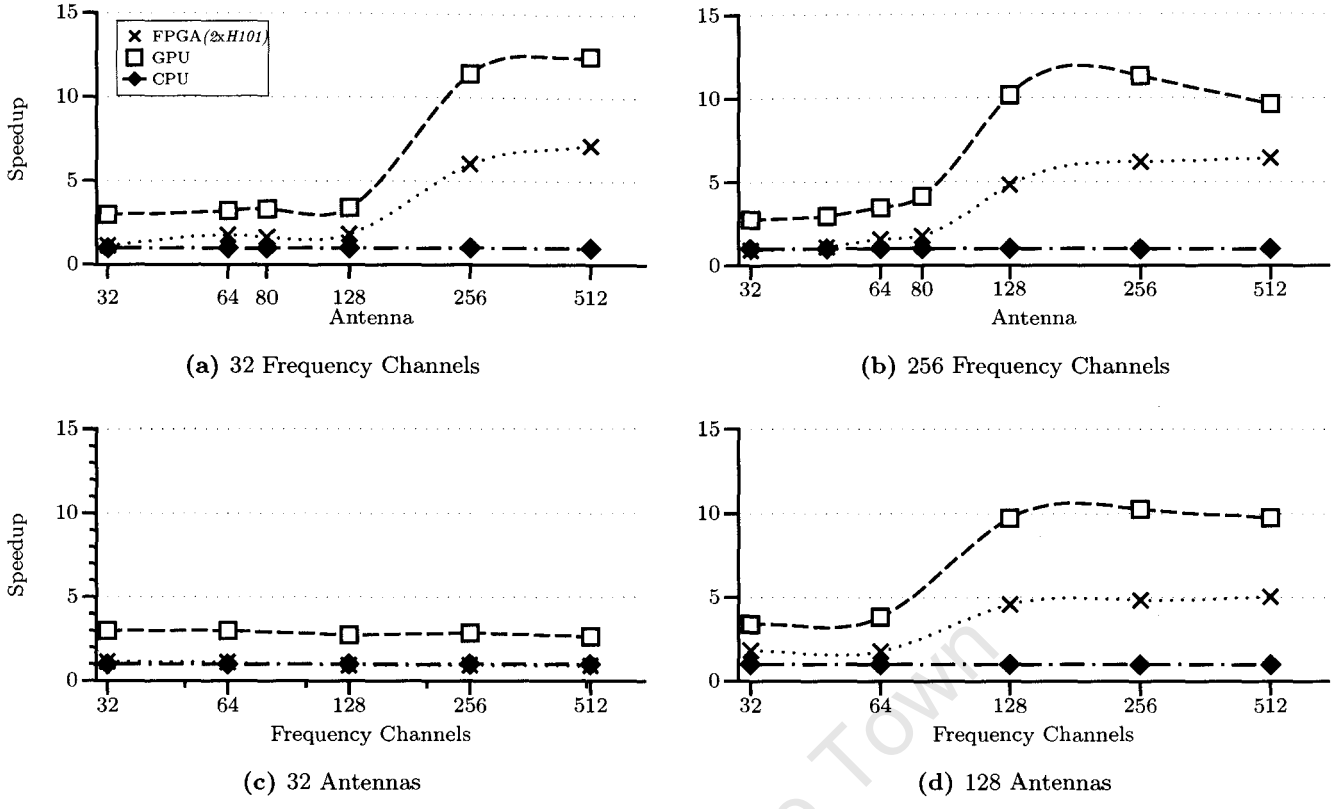
**(d)** 128 Antennas

**Figure 6.4** – Clock Cycles Required

Figure 6.4 graphs the number of clock cycles taken to compute the correlation. Larger experiments have more cross products to compute, with the number of cycles required increasing $O(N)$ with the number of frequency channels and $O(N^2)$ with the number of antennas. The different scaling of required cycles is reflected in the steeper gradient in **(a)**. The FPGA requires roughly an order of magnitude less cycles than the GPU, which in turn requires an order of magnitude less than the CPU. Clock cycles can be loosely translated into power consumption, and so this experiment roughly demonstrates the different power requirements across different architectures, with the FPGA offering the best power efficiency. Power consumption is discussed in more detail in section 6.2.2.

### Achieved Speedup

This section explores the effect the number of frequency channels and antennas have on the speedup of the correlator implementations.




**(a)** 32 Frequency Channels

**(b)** 256 Frequency Channels

**(c)** 32 Antennas

**(d)** 128 Antennas

**Figure 6.5** – Achieved Speedup over the CPU.

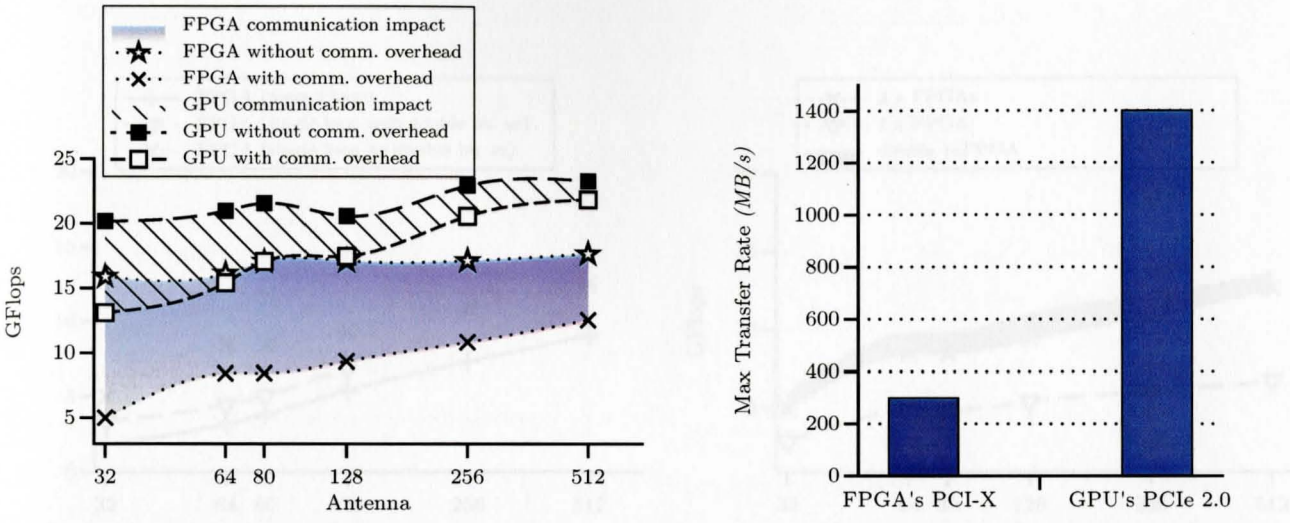
Figure 6.5 shows the speedup over the CPU correlator, which was the ultimate goal of the co-processor implementations. For reasons mentioned in the previous sections, the GPU and FPGA implementations obtain the maximum speedup on large experiments. The GPU, at best, obtained a speedup of 12.5x and the FPGA 7x over the CPU implementation.

In order to create a more detailed picture of the correlator's profile, further experiments were conducted. Theses are discussed in the next section.

### 6.2.2 Specific and Detailed Benchmarks

In this section, we present benchmarks which demonstrate specific aspects of the correlators' performance. We will investigate: host-device communication overhead; the performance of the different FPGA correlator designs; power and performance ratios; detailed analysis of the speedup, performance and bandwidth results on the correlator implementations; FFT performance; and result normalisation.

**Host-Device Transfer Impact on Performance**




**(a)** Transfer Impact on Performance

**(b)** Expansion Bus Performance

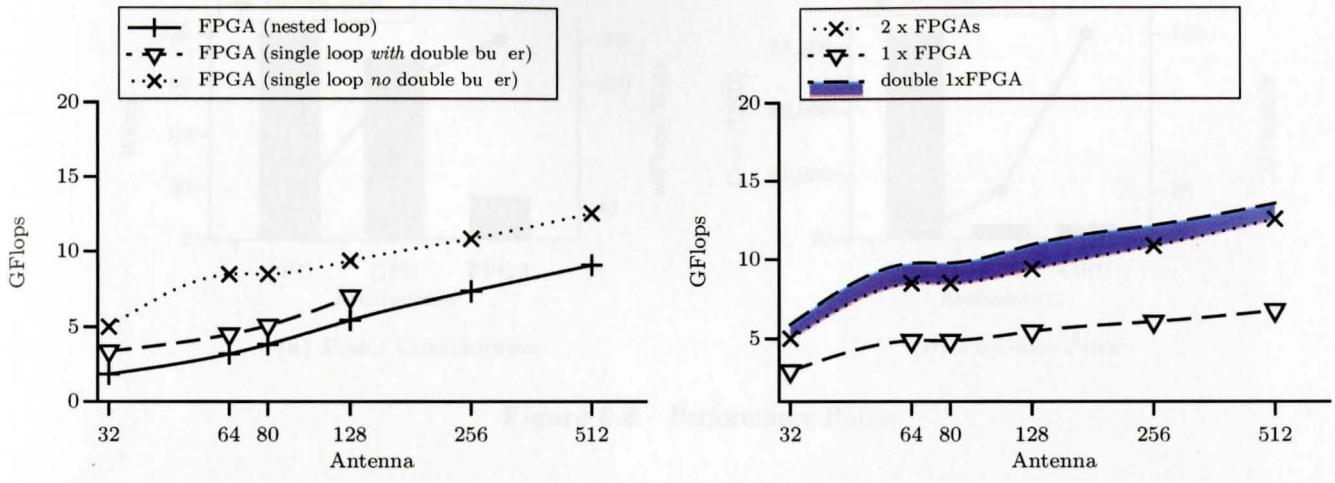
**Figure 6.6** – Host-Device Transfer Impact

Host-device communication overhead can impact the performance of accelerator cards quite significantly. The normal operating process is to transfer data to the FPGA or GPU, do the cross correlation and then transfer the data back to the CPU (as shown in Figure 6.1). To measure the transfer times, we ran two sets of timed experiments. Firstly, timing the entire time to compute cross-correlation and secondly, the time taken to compute the cross-correlations once all the relevant data had been loaded on the co-processors. These times benchmarks the host-device bus performance, not the accelerator chip-architecture itself.

Figure 6.6 **(a)** shows the performance impact that the host to device transfer have on the correlator implementations. The blue shading and the striped pattern represent the performance lost due to host-device communication for the FPGA and GPU respectively. The larger size of the blue shaded region compared to the striped pattern demonstrates the poor performance of the FPGA's PCI-X bus. Figure 6.6 **(a)** demonstrates that the same performance can be achieved for correlation experiments with a small number of antenna, if host-device communication overheads are ignored. FPGA is affected by the host-device communication bottleneck more significantly due to the slower PCI-X bus as shown in 6.6 **(b)** and therefore has the greatest improvement as the computation-communication ratio increases. The CPU correlator does not feature since it has zero transfer overhead.

Figure 6.6 (b) details the difference in transfer rates achieved across the expansion bus on the FPGA and GPU. Clearly, the FPGA's expansion bus performs much worse than the GPU's.

It is interesting that both buses perform quite significantly under spec, with the PCI-X being advertised as having a maximum transfer rate of 1GB/s and the 8xlane PCIe 2.0 advertised as having a maximum transfer rate of 8GB/s. From these benchmarks, it is unclear whether the host system or the device was the cause of the worse performance. In the next section, we add a second FPGA to the PCI-X bus and find that the overall bandwidth across the PCI-X bus increases, indicating the FPGA's bus performance is the cause for the bottleneck.




**(a)** FPGA Implementations

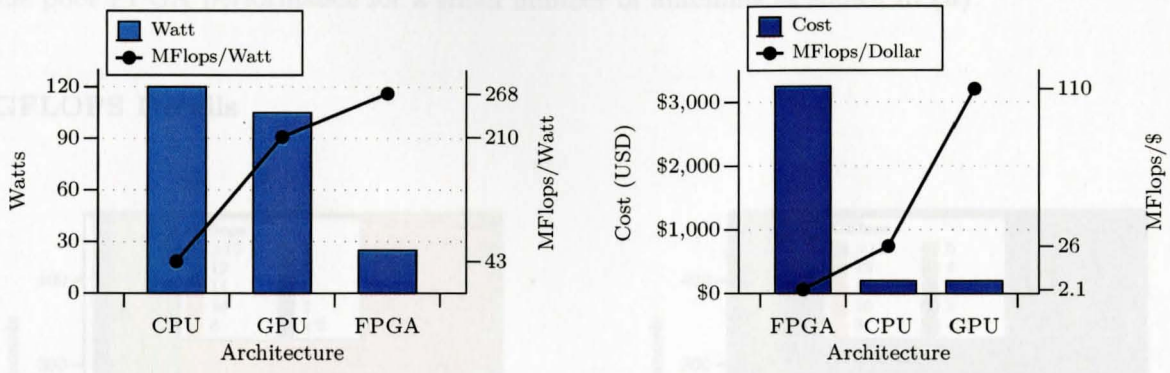
**(b)** FPGA Scaling

**Figure 6.7** – FPGA Implementation Comparison

Figure 6.7 **(a)** shows the performance of the three different FPGA implementations as discussed in Chapter 4. The single loop with double buffering could only run smaller experiments due to its larger memory requirements. The final FPGA design performed about 50% faster than the original nested loop implementation.

Figure 6.7 **(b)** shows the performance scaling with the number of FPGAs used in the implementation. The performance using a single H101, using both H101s, and the linear scaling in performance with two H101s. The blue shaded region is the difference between linear scaling and the actual performance achieved when using two H101s. Note that we achieved close to linear speedup when using the two FPGAs, indicating that the host's PCI-X bus is able to scale well with the two expansion cards. By adding an extra FPGA card, we have doubled the required data throughput on the host PCI-X bus, but not on each device. The linear increase in speed seems to indicate that the FPGA's PCI-X performance is the main cause for the sub spec performance presented in the previous section. This, however, does not mean that the PCI-X bus delivers enough bandwidth to the FPGA correlator, rather the host-device inefficiencies in Figure 6.6 are not related to populating two FPGAs in a single host.

### Power and Performance Ratios



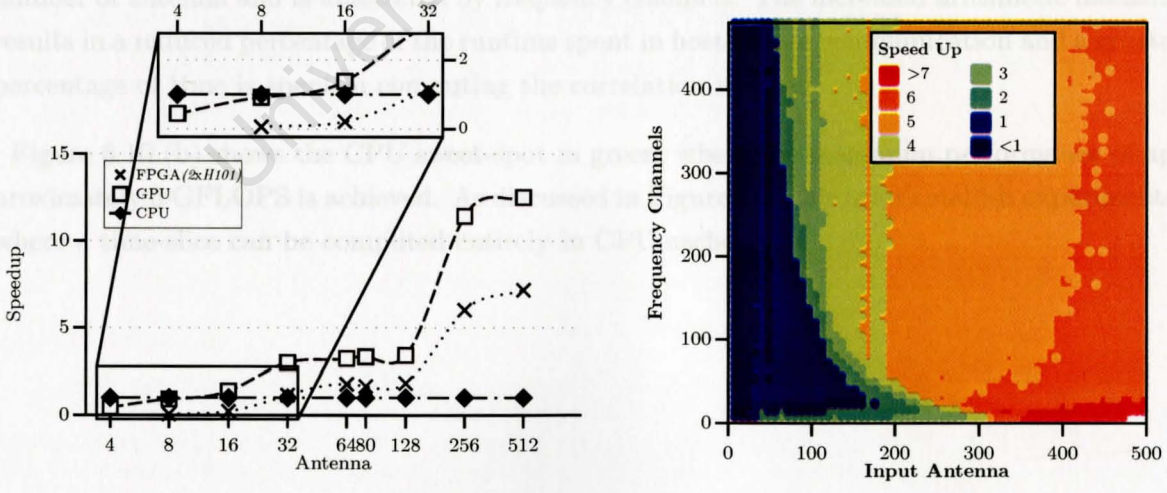

**(a)** Power Consumption

**(b)** Purchase Price

**Figure 6.8** – Performance Ratios

Figure 6.8 **(a)** shows the peak power consumption for the three architectures and the power efficiency in MFLOPS/Watt. The GPU and CPU have similar power requirements but the GPUs superior performance results in a higher Flop/Watt ratio. The FPGA is the architecture which offers the best Flop/Watt performance, but is also by far the most expensive as seen in Figure 6.8 **(b)**. However, the price of FPGAs vary considerably depending on the quantity, model and manufacturer. The price listed was based on the cost to equip our lab with two Nallatech H101s. Note that these figures are excluding the cost and power consumption of the host systems for the FPGA and GPU [7].

### Speedup Details




**(a)** 32 Frequency Channels

**(b)** FPGA speedup vs. CPU

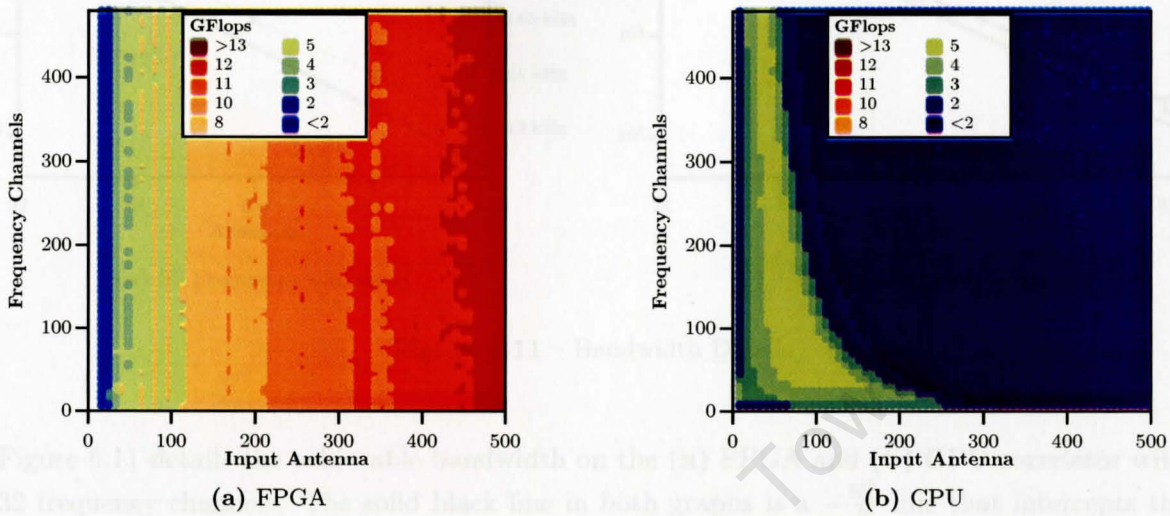
**Figure 6.9** – Speedup Details

Performance figures for experiments with less than 32 antennas were not shown because of the poor co-processor performance, as shown in Figure 6.9 **(a)**.

[7] Power utilisation was not measured directly but instead power estimation tools and data sheets provided by the vendors were used [43] [56] [58].

Figure 6.9 **(b)** is a 2D speedup graph, with the x-axis representing the number of antennas, the y-axis the number of frequency channels and the colour the speedup achieved. **(b)** reiterates the poor FPGA performance for a small number of antennas as shown in **(a)**.

### GFLOPS Details



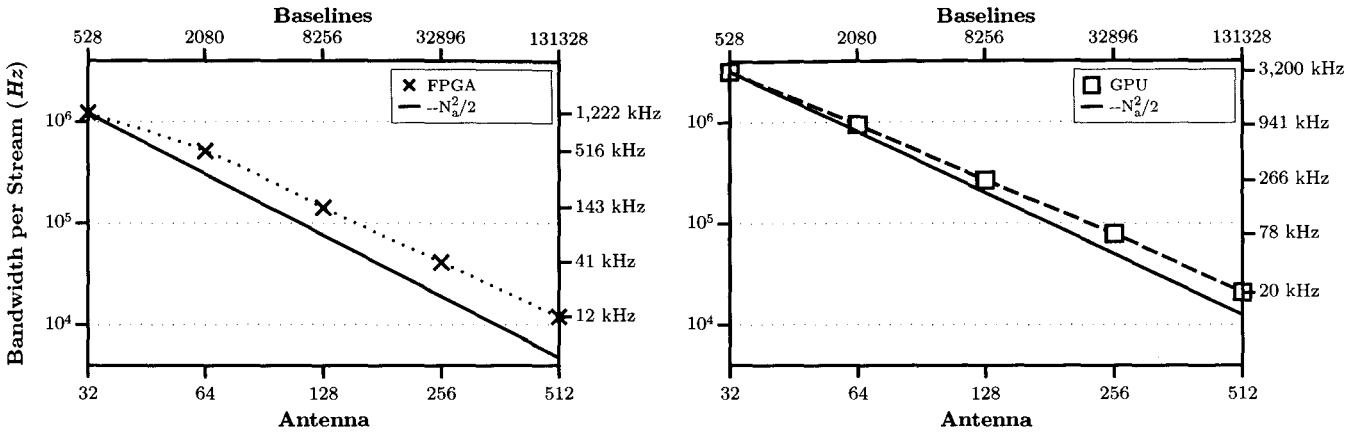

(a) FPGA (b) CPU

**Figure 6.10** – GFLOPS Details

Figure 6.10 (a) and (b) are 2D colour graphs for a varying number of frequency channels and antennas. Figure 6.10 (a) shows the GFLOPS achieved on the 2 Nallatech H101s, which illustrates that the FPGA's performance is dependent on the number of antennas, not the number of frequency channels. This is because the arithmetic intensity increases with the number of antenna and is unaffected by frequency channels. The increased arithmetic intensity results in a reduced percentage of the runtime spent in host-device communication and a greater percentage of time is spent in computing the correlation matrix.

Figure 6.10 (b) shows the CPU sweet-spot in green, where the maximum performance of approximately 5 GFLOPS is achieved. As discussed in Figure 6.2, this is for smallish experiments, where a time-slice can be computed entirely in CPU cache.

### Bandwidth Details




**(a)** 32 Frequency Channels

**(b)** 32 Frequency Channels

**Figure 6.11** – Bandwidth Details

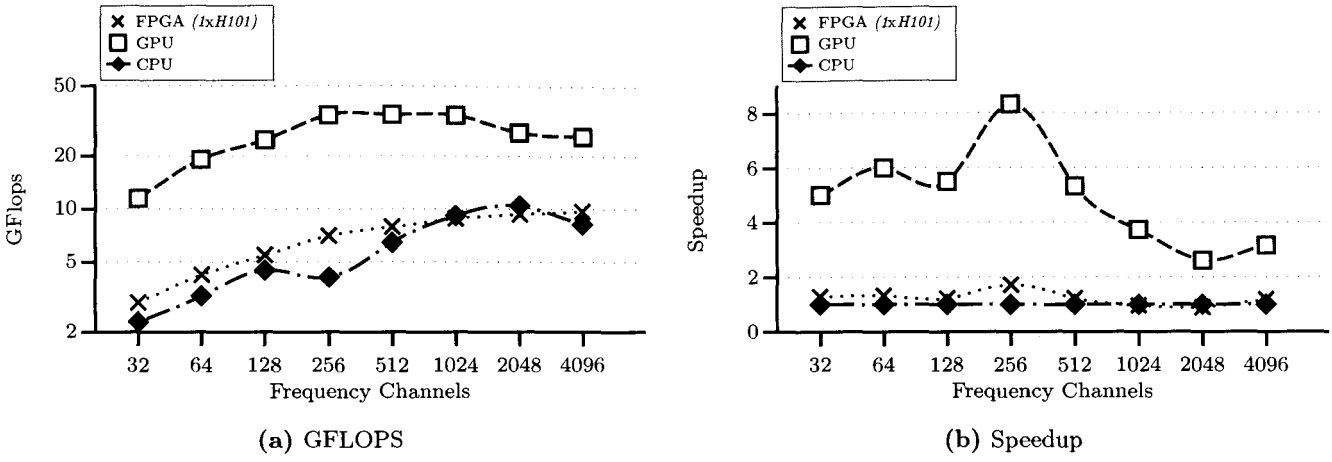
Figure 6.11 details the achievable bandwidth on the **(a)** FPGA and **(b)** GPU correlator with 32 frequency channels. The solid black line in both graphs is a $-\frac{N_a^2}{2}$ line that intercepts the bandwidth achievable with 32 antenna. This line shows the theoretical drop in bandwidth as the number of baselines increase. The reason that the correlator implementations perform above the line is because the correlator's GFLOPS performance increases with larger array sizes as shown and discussed in Figure 6.2.

### FFT performance

The F-engine channelisation was performed by an FFT, using vendor specific libraries as discussed in Chapters 2.3.1 and 2.5.2. Since these libraries were developed independently and the X-engine dominates the computational requirements of the correlator, as discussed in Chapter 2.5.1, there has so far been little mention of the FFT F-engine. However, the performance of the F-engine must also be taken into consideration for software correlation acceleration. Figure 6.12 presents the GFLOPS[8] performance and speedup of the three architectures, CPU, GPU and FPGA using the vendor libraries Intel Performance Primitives (IPP) Library v5.3.1, Nallatech Single Core FFT [60][9] and CUFFT v2.0 [2] respectively.

[8]FLOPS was calculated using: $5N\log_2(N)/\text{time}$ to compute fft [59] .

[9]Nallatech have two FFT libraries: single butterfly and 11 butterflys. We could only get the single butterfly version to produce the correct output. To compensate we divided the single butterfly FFT runtime by 11. This is a reasonably accurate estimation, since the multiple butterfly version has 11 times the computational hardware, and no additional communication overhead.




**(a)** GFLOPS

**(b)** Speedup

**Figure 6.12** – FFT Details - **(a)** Reports the GFLOPS achieved on the three architectures using the vendor FFT libraries. **(b)** Reports the speedup over the CPU of the other architectures.

Figure 6.12 (a) reports the GFLOPS achieved on the three architectures using the vendor FFT libraries. The graph shows that the 9800GT GPU far outpaces both the CPU and the H101 FPGA, while the CPU and FPGA are closely match in performance[10]. Figure 6.12 (b) shows the speedup of the other architectures over the CPU. As for the X-engine, the GPU is the clearly performs best, with up to an 8x speedup[11].

Both the FFT and correlation have similar processing requirement profiles, therefore if we assume that the vendor FFT libraries are optimised, they provide a rough estimate for what we could hope to achieve from an optimised X-engine using the different architectures. The CPU in Figure 6.12 (a) achieved approximately 10 GFLOPS, while the CPU X-engine achieved approximately 5 GFLOPS, which demonstrates that the CPU correlator X-engine implementation could potentially be further optimised. This is also true for our GPU implementation, which achieves approximately 35 GFLOPS in the FFT benchmark but 23 GFLOPS in our X-Engine correlation (excluding communication). On the other hand, using a single H101 achieves around 9.5 GFLOPS when running the FFT and our H101 X-engine achieves around 9 GFLOPS using a single FPGA. This suggests an optimised X-engine design.

[10]Note that Demorest [61] achieved similar performance when benchmarking the CUDA FFT library.

[11]These benchmarks were performed using only 1xFPGA, not both H101s as in the previous results. Additionally, no host-device communication overhead was considered in these performance results.

### Result Normalisation (4 CPU Cores)

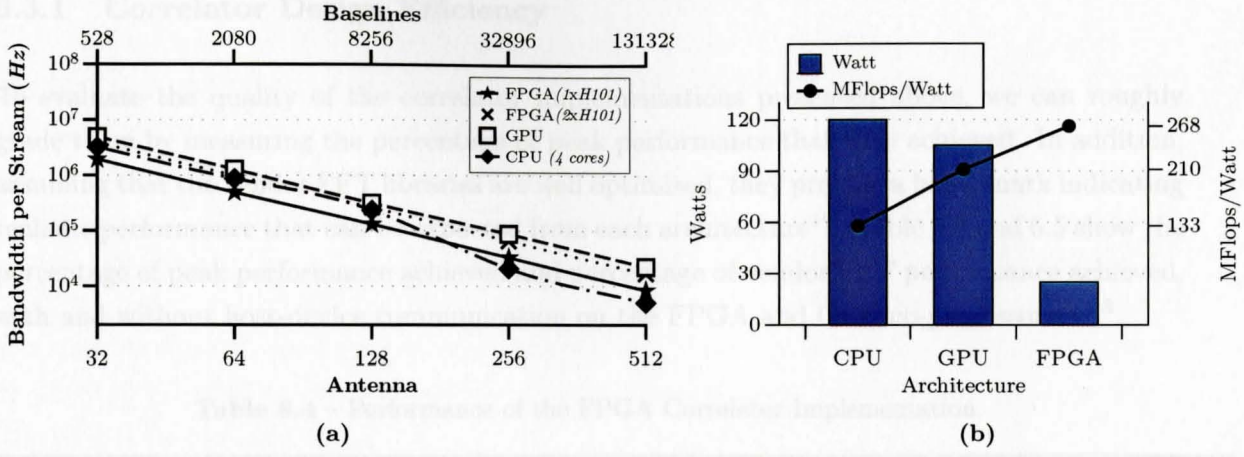



**Figure 6.13** – Normalised Performance Results – **(a)** Reports the normalised bandwidth per stream using 32 frequency channels. **(b)** Reports the normalised power performance ratios.

The CPU correlator implementation used the SSE vector instructions via Intel's IPP library, which makes use of the Harpertown Xeon's SIMD capabilities. However, this was a single threaded application, only utilising one of the four CPU cores, which causes the CPU performance to be understated. On the other hand, our FPGA implementation used two FPGAs, which causes the FPGA performance to be inflated. The results in Figure 6.13 are normalised to show a *fully* utilised *single* processor[12].

The normalised results paint a different picture compared to the previous results. The co-processors lose the clear advantage over the CPU implementation for some of the benchmarks. However, other factors like architecture generation should be considered for an unbiased comparison. Since the Virtex 4 FPGA is an older generation of technology compared to the more recent G80 Nvidia GPU and Intel Harpertown CPU.

### Result Normalisation (Improving CPU Cacheing)

The knee seen in the CPU performance is probably due to caching effects, as discussed before. Cacheing effects can be significantly reduced if careful consideration is taken[13]. Therefore, the knee in the CPU's performance cannot be exclusively architecture related and a better correlator implementation would probably avoid this. Additionally, if all four cores were used, the knee would appear later, since there would be more cache available to the whole CPU.

## 6.3 Discussion of Benchmarks

In this section, we analyse the benchmark results above and conclude with the performance results of our correlator implementations.

[12]The CPU performance was an estimate, calculated as 3.5x the single threaded implementation. The 0.5x speedup difference is allocated to overhead.

[13]Eg BLAST with Matrix operations

### 6.3.1 Correlator Design Efficiency

To evaluate the quality of the correlator implementations presented above, we can roughly grade them by measuring the percentage of peak performance that they achieved. In addition, assuming that the vendor FFT libraries are well optimised, they provide a benchmark indicating realistic performance that can be expected from each architecture[14]. Table 6.4 and 6.5 show the percentage of peak performance achieved and percentage of vendor FFT performance achieved, with and without host-device communication on the FPGA and GPU co-processors[15] [16].

**Table 6.4** – Performance of the FPGA Correlator Implementation.

| | **Including Host-device Communication** | **Excluding Host-device Communication** |
|---|---|---|
| **Performance** (GFlops) | 12.5 | 17.2 |
| **Percentage of Peak Peformance** | 65% | 90% |
| **Vendor FFT Performance** (GFlops) | - | 18 |
| **Percentage of FFT Performance** | - | 95% |

**Table 6.5** – Performance of the GPU Correlator Implementation.

| | **Including Host-device Communication** | **Excluding Host-device Communication** |
|---|---|---|
| **Performance** (GFlops) | 22 | 23.5 |
| **Percentage of Peak Peformance** | 6.5% | 7% |
| **Vendor FFT Performance** (GFlops) | - | 35 |
| **Percentage of FFT Performance** | - | 67% |



**Table 6.6** – GPU Correlator Implementation Profile.

| **SM Occupancy** | **Coalesced Memory Access** | **Warp Serialisation** |
|---|---|---|
| 67% | 100% | 0% |

The FPGA correlator implementation delivers performance closely comparable to that of the vendor FFT, which indicates that the FPGA implementation is reasonably well optimised.

[14]FFT has a similar processing profile to the correlator so we can expect similar performance.

[15]Together the two Virtex 4LX100 FPGAs could deliver a peak performance of 19.2 GFLOPS. Each FPGA could implement 96 FPUs in Dime-C, giving us a total of 192 FPUs with both the H101s. The cards were clocked at 100MHz (The clock was limited to 100MHz because of the SRAM) and therefore produce a peak performance of 19.2 GFlops.

[16]The Geforce 9800GT with its 112 SP clocked at 1.5GHz could deliver 336 GFLOPS at peak performance. Each SP can perform a MADD and MUL per clock cycle, but only the MADD operation is useful in our case. Therefore, 336GFLOPS was quoted instead of 504GFLOPS.

On the other hand, the GPU fairs slightly worse in terms of percentage of peak performance reached, meaning there is room for code optimisation. Table 6.6 is a summary of the CUDA correlator profile, which indicates that our GPU correlator is achieving linear memory access and that each warp is executing the same branch of code. However, SM occupancy could be improved by thinning register usage, which allows for more active warps to run simultaneously. If more warps are scheduled, higher memory latency can be tolerated before performance deteriorates. Although the GPU correlator is less efficient than the FPGA implementation, there was significantly less development effort invested in it and the GPU optimisations mentioned in Chapter 5.3, would be a good starting point to improve the efficiency if the GPU correlator development was continued.

### 6.3.2 Estimated Scaling with Future Hardware Generations

The technologies used in this thesis are no longer cutting edge. As technologies follow Moore's Law, older generations' performance is quickly dwarfed by the new architecture models. As a continuation of the discussion on performance normalisation in section 6.2.2, we attempt to project a fair comparison between the different correlator implementations by estimating the performance of our correlator implementations on the latest hardware.

To estimate performance on current technologies, we use a straight forward method of comparing the specifications of the hardware used in this thesis and that of current hardware generations. This simplistic approach overlooks some implementation factors that would be involved in porting our correlator implementations to future hardware, but produces a rough estimate of what could be achieved. Table 6.7 shows the peak performance difference of the different processing technologies and Figure 6.14 graphs the performance of our correlator implementations, assuming this theoretical difference can be translated into real world performance.



**Table 6.7** – Processor Performance Growth

| | **Xilinx Virtex 4 LX100** | **Xilinx Virtex 6 SX475T** | **Resource Growth** |
|---|---|---|---|
| **DSPs** | 96 | 2,016 | 21x |
| **Logic Cells** | 110,592 | 476,160 | 4.3x |
| **BRAM** (Kbits) | 4,320 | 38,304 | 9x |
| **Release Date** | 2005 | 2009 | - |

FPGA Resource Growth [8, 10]

| | **Nvidia Geforce 9800GT (G92)** | **Nvidia Geforce GTX285 (GT200)** | **Performance Growth** |
|---|---|---|---|
| **Theoretical Peak** | 504 | 1,063 | 2.1x |
| **Release Date** | 2007 | 2008 | - |

GPU Performance Growth [56, 13]

| | Intel Xeon X5450 Harpertown | Intel Xeon W5580 Nehalem | Performance Growth |
|---|---|---|---|
| **SPEC CPU2006** | 26.3 | 37.3 | 1.4 |
| **Release Date** | 2007 | 2009 | - |

CPU Performance Growth [62]

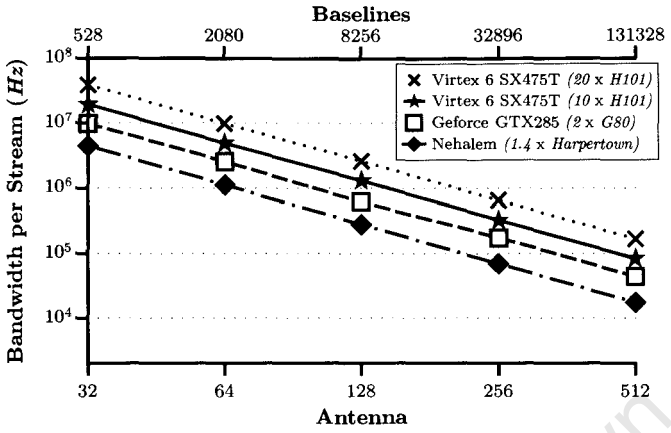



**Figure 6.14** – Performance scaling with future hardware generations.

In the above Figure 6.14, we assume that our correlators' performance scales linearly with the change in peak of newer technologies.

When comparing the scaled correlators' performances, our FPGA implementation performs by far the best, offering 20x the Nallatech H101's performance. The bigger jump in performance the FPGA experienced over other architectures can be contributed to two aspects. Firstly, the Virtex 4 is four years older than other latest corresponding technology, while the CPU is two years older and the GPU is only one year older. Secondly, the Virtex 4LX100 is mid-range in the Xilinx LX family. Characteristics of the LX family include large numbers of logic cells, but only few hardwired DSPs - the DSPs are important for computationally intensive applications like correlation and were the limiting factor in our correlator implementation. These factors account for the 20x growth in DSPs and our 20x estimate FPGA correlator performance.

However, the 20x estimation only considers DSP resources, while other resources such as logic cells have seen less growth. Although the logic cells were not the resource limitation, a 20x sized H101[17] would need significantly more interfaces and control logic, requiring logic cells. A more conservative estimate of performance growth would likely be 10x the H101, which is also shown in Figure 6.14, which would still deliver better performance than the CPU and GPU correlator implementations.

Note that we have not considered external I/O concerns. A larger correlation element would need larger I/O capabilities, which would likely need multiple high speed connections such as 10GbE or PCIe. The bottleneck of getting data into the correlator has not been considered in this performance scaling.

Although Figure 6.14 is a simplistic and idealised view of the scaling of our correlator implementations, it shows that the age and family choice of the FPGA contributes to its relatively poor performance when compared to the GPU.

[17]These performance figures are 20x a *single* H101.

### 6.3.3 Result Conclusions

The following is a summary of the above performance results.

The GPU correlator implementation offered the best performance of up to a 12.5x speedup over the CPU, as well as the best FLOP/$ ratio. The FPGA implementation, while being faster than the CPU, is only roughly half the speed of the GPU and is 30x the cost. The FPGA does however, offer better FLOP/Watt performance. When comparing the correlators' GFLOPS performance with the vendor FFT libraries, we see that the FPGA correlator achieves similar performance, indicating that it is well optimised. In comparison, the GPU correlator achieves 2/3 of the vendor FFT performance, indicating that it is moderately optimised with room to grow.

When the results are normalised to estimate the performance on all four cores on the CPU, the GPU is at best 4x faster and the FPGA is 3x faster[18] - making the co-processor correlators less appealing than they previously appeared. However, if we look at the advancements in FPGA,GPU and CPU technologies and apply the same scaling to our correlator implementations, we find that we may expect up to 30x and 6x the performance of the CPU with the FPGA and GPU respectively[19]. This highlights that the age and family choice of the FPGA contributes to its relatively poor performance when compared to the GPU.

Both correlators suffer from host-device communication overhead, which is reduced when the arithmetic intensity increases. However, the FPGA's performance is affected more considerably due to its slower PCI-X bus.

## 6.4 Comparison with Other Correlators

Besides looking at performance and correlator efficiencies, a good benchmark is to compare the performance of our correlators with other correlators. Unfortunately, this is extremely difficult to do accurately. Many correlators report the bandwidth that can be processed for a certain sized array and how many processing nodes are used. However, the functions and capabilities[20] performed by the correlator vary. Some correlators only report performance of the F-engine, X-engine, data transfers and marshaling all as a single figure, while others report each section separately. Some correlators, such as the CASPER project, are designed to be hardware independent and report the number of X-engines required for a particular antenna array. However, the number of processing nodes needed to implement the CASPER X-engines will depend on the implementation platform.

Another large consideration is the correlator interconnect. Large correlators are almost always built from separate processing nodes and because of this, the interconnect design and capabilities influence the scaling of the correlator design considerably [63, 64]. Generally, benchmarks for single node correlators do not consider the interconnect and packetisation involved in scaling up correlation, such as in this dissertation.

[18]The FPGA correlator is 3x faster than the 4 core CPU implementation when using 2xH101s and 1.5x faster when using only a single H101.

[19]The speedups are best case scenarios, and these are probably not achievable in practice.

[20]Capabilities include ADC sample size, whether dual or single polarisation is used, etc.

A further consideration not taken into account is the correlator's power consumption. Powering large correlators is expensive, especially in remote locations, so power efficiency is very important in production correlators. However, the power requirements of the different correlators are not reported here.

Taking these factors into consideration, a comparison of different correlators is shown in Table 6.8.

**Table 6.8** – Performance of Other Correlators

| | Antenna | Polarization | Bandwidth | Processor | Correlation Nodes | Bandwidth per Node |
|---|---|---|---|---|---|---|
| **DiFX** | 20 | dual | 64MHz | Pentium 4 | 300 | 0.2MHz |
| **GMRT** | 32 | dual | 32MHz | Quad-core Itanium | 16 | 2MHz |
| **UWA**(1)(2) | 32 | single | 90MHz | Nvidia 8800GTS | 1 | 90MHz |
| **MWA**(1)(2) | 32 | dual | 3.7MHz | Nvidia Tesla C1060 | 1 | 3.7MHz |
| **CASPER**(1) | 32 | dual | 250MHz | ROACH V5SX95 | 4 | 62.5MHz |

(1) Does not include the cost to Fourier transform the data.
(2) Does not include the cost of host-device communication.

Source: DiFX [5]; GMRT [65, 66], UWA [14, 67, 63], MWA [16, 68], CASPER [63, 64].

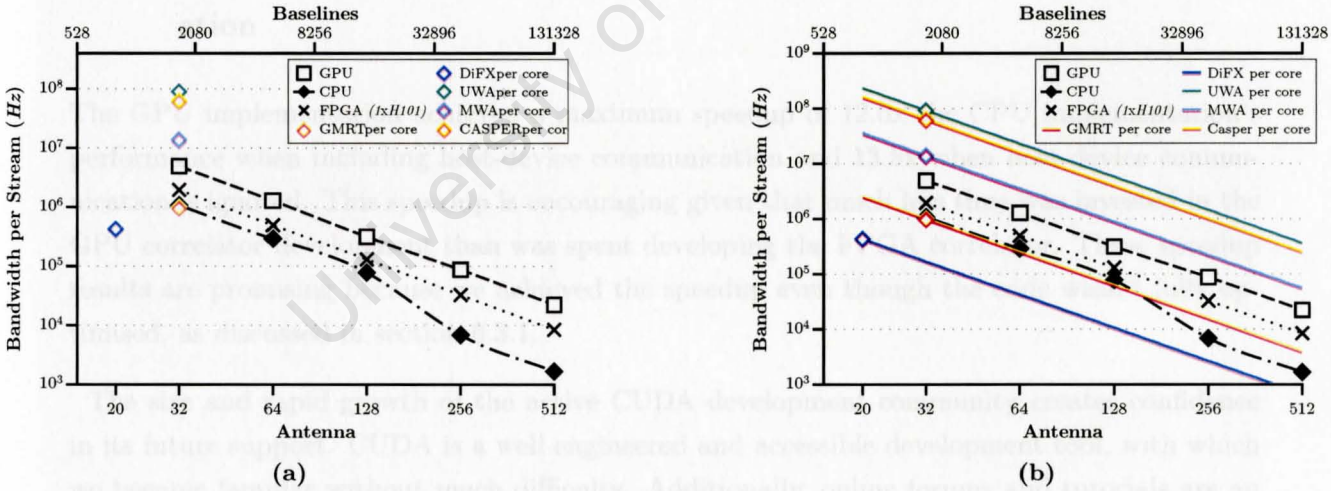



**Figure 6.15** – Performance Comparison of Various Correlators

Figure 6.15 (a) plots the points results quoted in Table 6.8, while Figure 6.15 (b) interpolates the performance for different numbers of antennas by assuming a quadratic decrease in bandwidth as the number of antennas increases.

Our software correlator performs better than the other two software correlators, the DiFX and the GMRT software correlators, until the cacheing effect influences our CPU correlator's performance. However, the DiFX and the GMRT software correlators are performing all correlator functions and are distributed nodes, with the interconnect overhead included in the results, while neither of these factors are included in our results.

Both the MWA and UWA[21] GPU correlators perform better than our GPU correlator implementation. The MWA uses a more capable Tesla GPU, which accounts for some of their performance gain. In addition, we have known inefficiencies in our implementation. The UWA seems to have unrealistically high performance results using a single 8800 GTS GPU, but our interpretation of their results may be incorrect and we advise you to see Chris Harris's paper [14]. The GPU correlators only include the X-Engine performance results and do not consider scaling to multiple nodes.

The CASPER FPGA correlator performs considerably better than our implementation. This is due to a much more capable FPGA and some clever correlator design.

Again, it's very difficult to compare different correlators' performances, but Figure 6.15 shows that we are producing realistic performance results. However, a true performance comparison would need a far more detailed analysis than what is presented in this section.

## 6.5 Conclusions on the Co-processor Correlator Implementations

In this section we discuss the merits of the co-processor correlator implementations and their suitability for simple correlator X-engine acceleration.

### 6.5.1 Evaluation of Nvidia CUDA GPUs for Software Correlation Acceleration

The GPU implementation achieved a maximum speedup of 12.5x the CPU implementation's performance when including host-device communication and 13.5x when host-device communication is ignored. This speedup is encouraging given that much less time was invested in the GPU correlator development than was spent developing the FPGA correlator. These speedup results are promising because we achieved the speedup even though the code wasn't fully optimised, as discussed in section 6.3.1.

The size and rapid growth of the active CUDA development community creates confidence in its future support. CUDA is a well engineered and accessible development tool, with which we became familiar without much difficulty. Additionally, online forums and tutorials are an invaluable resource for CUDA development, which is sorely missed from Dime-C.

### 6.5.2 Evaluation of Nallatech H101 for Software Correlation Acceleration

The FPGA implementation achieved a maximum speedup of 7x the CPU implementation's performance when including host-device communication and 10.5x when host-device communication is ignored. Despite the speedup, the development effort and hardware costs do not justify using the Nallatech H101 for our simple software correlation acceleration. However,

[21] Made by Chris Harris from the University of Australia (UWA).

the smaller power requirements are attractive for large clusters and newer FPGA generations offering far more processing resources than the Virtex 4LX100s.

The development of the FPGA correlator used Dime-C, a C-to-HDL development environment from Nallatech, as introduced in Chapter 3.2. Dime-C succeeds in providing a more familiar environment for software developers than using HDLs, where pipelining and parallel execution are not automated. This gives FPGA application development a jump starts. However, the disadvantage is that Dime-C does not have the active user community, mature development environment, documentation and existing library development that traditional HDLs have. These factors became increasingly important as the FPGA correlator development progressed, where more detailed information and examples could help highlight certain aspects and behaviour of Dime-C.

Many of the problems encountered with the FPGA implementations, could be rectified if there were more flexibility in the Dime-C compiler. The need to write intermediate accumulation results back to external memory creates a memory bottleneck, and ultimately impacts the performance (as mentioned in Section 4.1.3). In a pure VHDL/Verilog correlator, these results could be stored in internal registers, which would greatly alleviate the memory bandwidth problem.

As mentioned above, the cost of FPGA accelerator cards vary, but they are generally expensive in comparison to GPU and CPU architecture. However, the power and cooling requirements of large CPU clusters are much higher than that of FPGA clusters, offsetting the initial higher FPGA purchase price. Nevertheless, the running expense is generally only a consideration for large computing clusters, but we are only concerned with small scale correlation.

It should also be noted that the Virtex 4 FPGA is an older generation of technology, released in 2005, while the G80 Nvidia GPU and Intel Harpertown CPU were released in 2007. Additionally, the old parallel PCI-X interface is considerably slower than the PCIe interface on the GPU. Newer Virtex 6 FPGAs offer 20x the resources than the Virtex 4LX100 used in this project, which should translate into a significant performance increase. Furthermore, floating point arithmetic[22] was used for the FPGA correlator and the number of FPGA processing elements and memory throughput could be increased by using fixed point arithmetic and fewer bits per sample.

Arithmetic precision is an area that is important to mention as it can have significant performance impact. We used 32bit floating point arithmetic as a matter of convenience to compare accuracy across the different correlator implementations and to allow compatibility to the DiFX correlator[23]. However 32bit floating point precision requires about three times the FPGA resources and double the bandwidth of 16bit fixed point arithmetic. Additionally radio astronomy correlation can be implemented with much lower precision than 32bits without sacrificing the accuracy of the result. For these reasons, production correlators usually use 16 bit or 8 bit fix point arithmetic. Due to time limitations lower precision solutions were not investigated, but could significantly improve the FPGA's performance and are worth considering.

Considerable development effort was spent optimising the FPGA correlator kernel and we were able to achieve 90% of the peak performance. Much less time was invested in the GPU

[22]The choice of using floating point arithmetic was mainly due to convenience.
[23]This was the original intension of the implementation.

development and it already outperforms the FPGA implementation, while not being fully optimised.

All these factors justify using GPUs to accelerate small-scale software X-engine correlation.

# Chapter 7

# Conclusion and Future Work

This chapter discusses possible future work and finally concludes with the co-processor correlator implementations.



## 7.1 Future Work

The correlator development in this thesis concentrated on the X-engine, but both the FPGA and GPU have shown to have good FFT performance and are commonly used to accelerate the F-engine. Vendor FFT libraries already exist for these platforms, therefore there should not be major development effort to integrate the libraries into our correlator. In Addition, we could use polyphase filter banks for F-engine channelisation as they are commonly used to provide less spectral leakage than FFTs. The polyphase filter bank development is another possible avenue for future work.

The FPGA implementation could be improved by using smaller sample sizes and fixed-point arithmetic, replacing the 32 bit floating point data representation. Adding asynchronous transfers will help combat the slow PCI-X interface on the Nallatech H101s. It would also be interesting to measure our FPGA correlator's performance on current generation technologies, such as Xilinx's Virtex 6 with a PCIe 2.0 interface. This should theoretically provide roughly 20x more computation and significantly better host-device memory bandwidth.

The GPU implementation is currently not fully optimised and there are still opportunities to implement some of the optimisations mentioned in Chapter 5.3. Using GPU development tools[1] to more thoroughly profile the GPU execution, we could identify further optimisations. However, there are currently more complete GPU correlators available, such as the MWA GPU correlator, which would serve as a better starting platform for future correlation development.

## 7.2 Conclusion

Both the co-processor correlators have successfully achieved speedups over the CPU correlator, are more power efficient, and in the case of the GPU, provide more performance/$. The

[1]The GPU tools we refer to include: tools available from Nvidia (Profiler and PTX assembly code) and from other 3rd parties (eg. decuda).

increased compute density of the co-processor correlators mean that fewer processing nodes are needed, bringing down other infrastructure cost, such as space and network interconnect requirements.

Although both the GPU and FPGA correlator implementations do offer better performance over the CPU, the GPU correlator development was considerably less time-consuming and the hardware more affordable. However, the FPGA implementation does offer better power utilisation, which does bring down the running costs if large correlator implementations are needed. In conclusion, GPUs do offer an inviting platform for software correlation acceleration but it is difficult to justify the H101 for correlation acceleration for small to medium compute clusters.

# Appendix A

# Source Code and Project Directory

Please find the DVD attached to this dissertation. All source code and related files to this MSc can be found on the DVD.

# Appendix B

# Astronomy Background

## B.1 Angular Resolution

Angular resolution describes the angular distance between two point sources that can be differentiated by an aperture. Because of the diffraction effect, an antenna beam has side lobes, which are sensitive to sources outside the main antenna beam, limiting resolution, as shown in Figure B.1.

When a planar electromagnetic wave enters an aperture, the electromagnetic wave is distorted in what is called a diffraction pattern. Therefore a finite sized aperture cannot correctly record the radio brightness without some distortion of the original signal, as shown in Figure B.1.

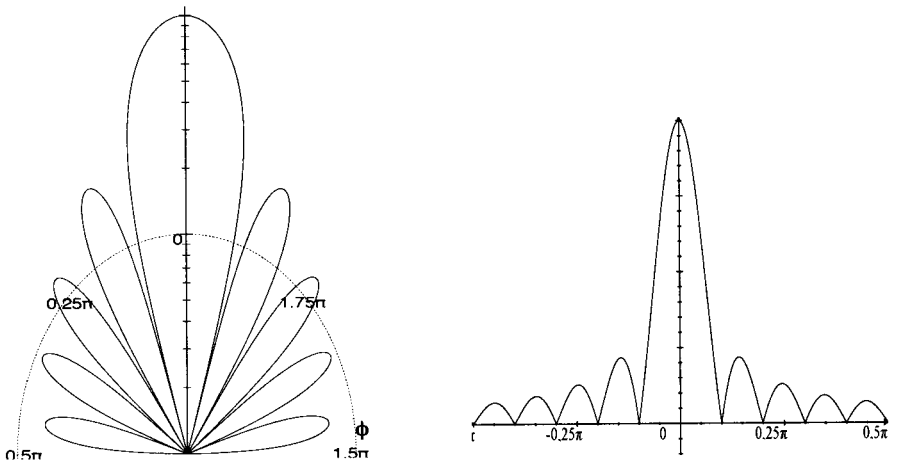
**(a)** **(b)**

**Figure B.1** – **(a)** The original point source. **(b)** The diffracted recording of a point source through a finite sized aperture

The diffraction distortion is due to the interaction of the original EM wave with the edges of a finite sized aperture, which creates the fringe pattern of destructive and constructive interference. Diffraction effects all types of EM waves when entering an aperture, but is more severe for longer wavelengths. The diffraction fringe in Figure B.2 can be described as a function of $sinc(\theta)$, where $\theta$ is the angular offset from the pointing direction of the aperture. The distance to the first zero of the diffraction pattern of a circular aperture is given by Equation B.1 [69]:

$$sin(\theta) = 1.22\frac{\lambda}{D}, \tag{B.1}$$

where $\lambda$ is the wavelength of the EM wave and $D$ the diameter of the aperture.




(a) Log Polar Plot (b) Linear Plot

**Figure B.2** – Response to an aperture at a given angular offset from the pointing direction. In this example the angular resolution is $\pi/10$

If two objects are closer than the first minima, in Figure B.2 this is $\frac{\pi}{10}$, for a particular aperture, they cannot be distinguished. Therefore the first minima, determines the resolving capabilities of an aperture and is called the angular resolution, see Figure B.3. The angular resolution, represented in the right-hand side of Equation B.1, depends on both wavelength and aperture diameter. As a consequence of dealing with radio waves, which have a long wave length, radio astronomy requires large telescopes in order to improve the resolution and produce detailed radio brightness readings.[1] [2] [3]



[1] An example of diffraction, is a television or computer monitor - which consists of many individual pixels that cannot be resolved by the human eye at a distance and appear as single picture.

[2] The dimensions of a single radio aperture needed to meet the angular resolution requirements are extremely impractical. For example, to achieve the same angular resolution as the naked human eye, a radio antenna's aperture observing a source at $1.4GHz$ must be 750m in diameter. [70]

[3] By knowing the impulse response of an aperture, a closer reconstruction of the original source can be made by performing a deconvolution.

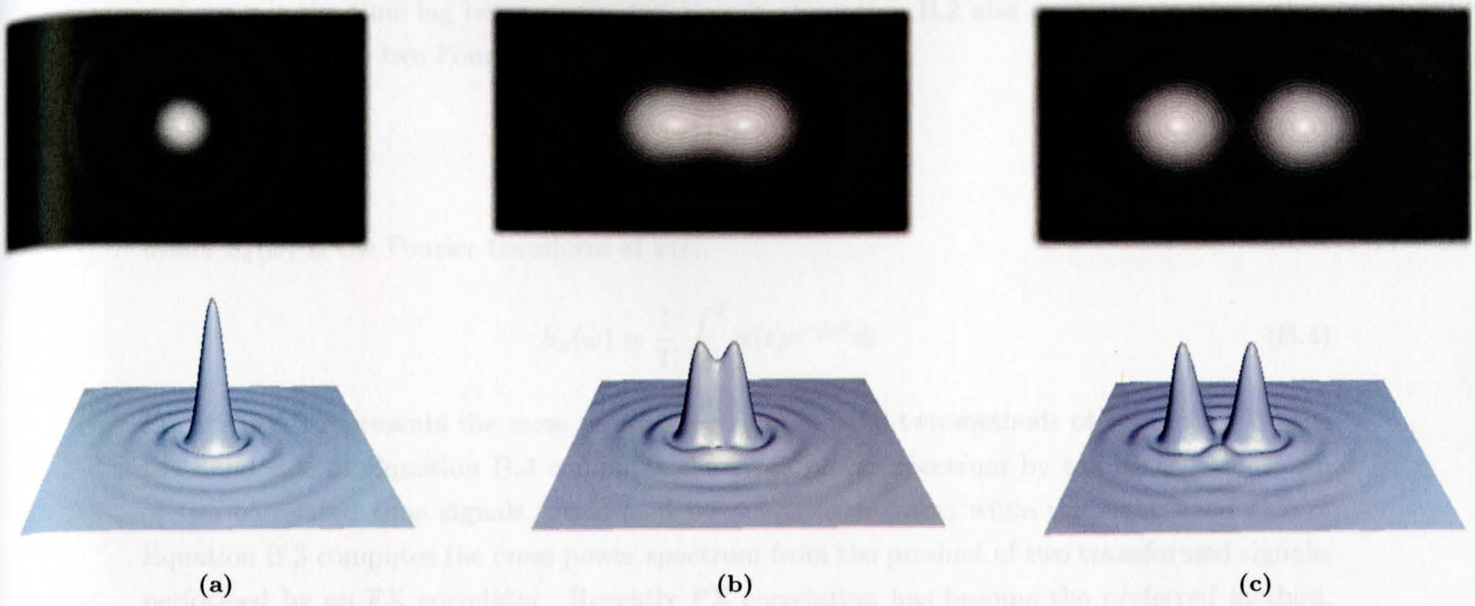



**Figure B.3** – The diffraction response of a circular aperture to a distant point source. Instead of detecting a single point, a broad band is detected with concentric rings, forming an airy disc. (a) two unresolved point sources (b) two just resolved point sources and (c) two completely resolved point sources. Figure inspired by [69]

## B.2 Correlation

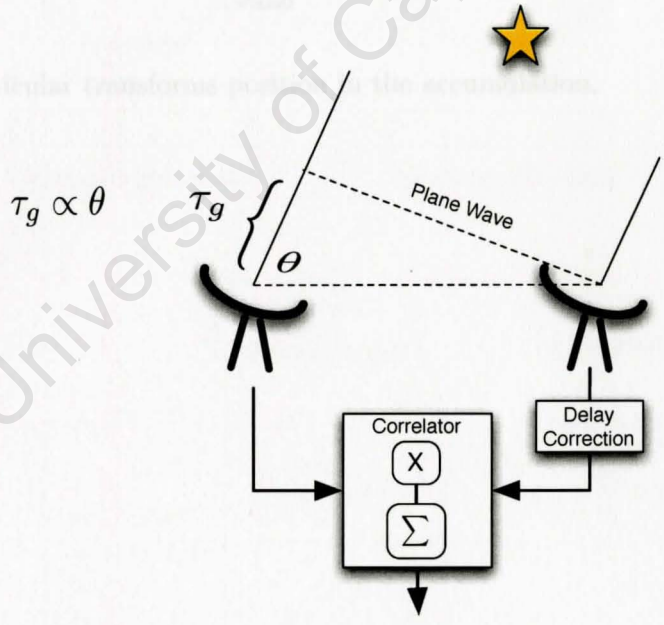



**Figure B.4** – Diagrammatic Representation of an Interferometric Telescope. The spacing between the antenna introduces a delay $\tau_g$ into the system, which is corrected before correlation.

In Figure B.4 we have two antennas, both pointing at the same source and producing two continuous voltage signals, which we will call $f(t)$ and $g(t)$. The cross-correlation function, $R_{fg}(\tau)$ can be defined directly as [71]:

$$R_{fg}(\tau) = \lim_{T\to\infty} \frac{1}{T} \int_0^T f(t)g^*(t-\tau)dt, \tag{B.2}$$

where $\tau$ is the time lag between the two signals. Equation B.2 also could be represented as the product of the two Fourier transformed inputs,

$$\underbrace{\mathscr{F}\{R_{fg}(\tau)\}}_{XF} = \underbrace{S_f(\omega)S_g^*(\omega)}_{FX} \tag{B.3}$$

where $S_x(\omega)$ is the Fourier transform of $x(t)$

$$S_x(\omega) = \frac{1}{T}\int_0^T x(t)e^{-j\omega t}dt \tag{B.4}$$

Equation B.3 represents the cross power spectrum and the two methods of computing it: the left hand side of Equation B.3 computes the cross power spectrum by taking the transform of two correlated time signals, performed by an XF correlator, while the right hand side of Equation B.3 computes the cross power spectrum from the product of two transformed signals, performed by an FX correlator. Recently FX correlation has become the preferred method, as when there a large number of baselines FX correlators require less computation than XF correlators - and FX correlation was the method implemented in this dissertation.

Typically after the cross power spectrum has been computed, it is integrated for a period $\tau_{int}$ to reduce bandwidth and storage requirements and improve SNR, as shown in Equation B.5[4]:

$$C_{x,y}(\omega) = \int_{a=0}^{\tau_{int}/\tau} S_{a,f}(\omega)S_{a,g}^*(\omega) \tag{B.5}$$

where $a$ is the particular transforms position in the accumulation.



[4]where $\tau_{int} > \tau$.

## B.3 KAT Correlator Prototype

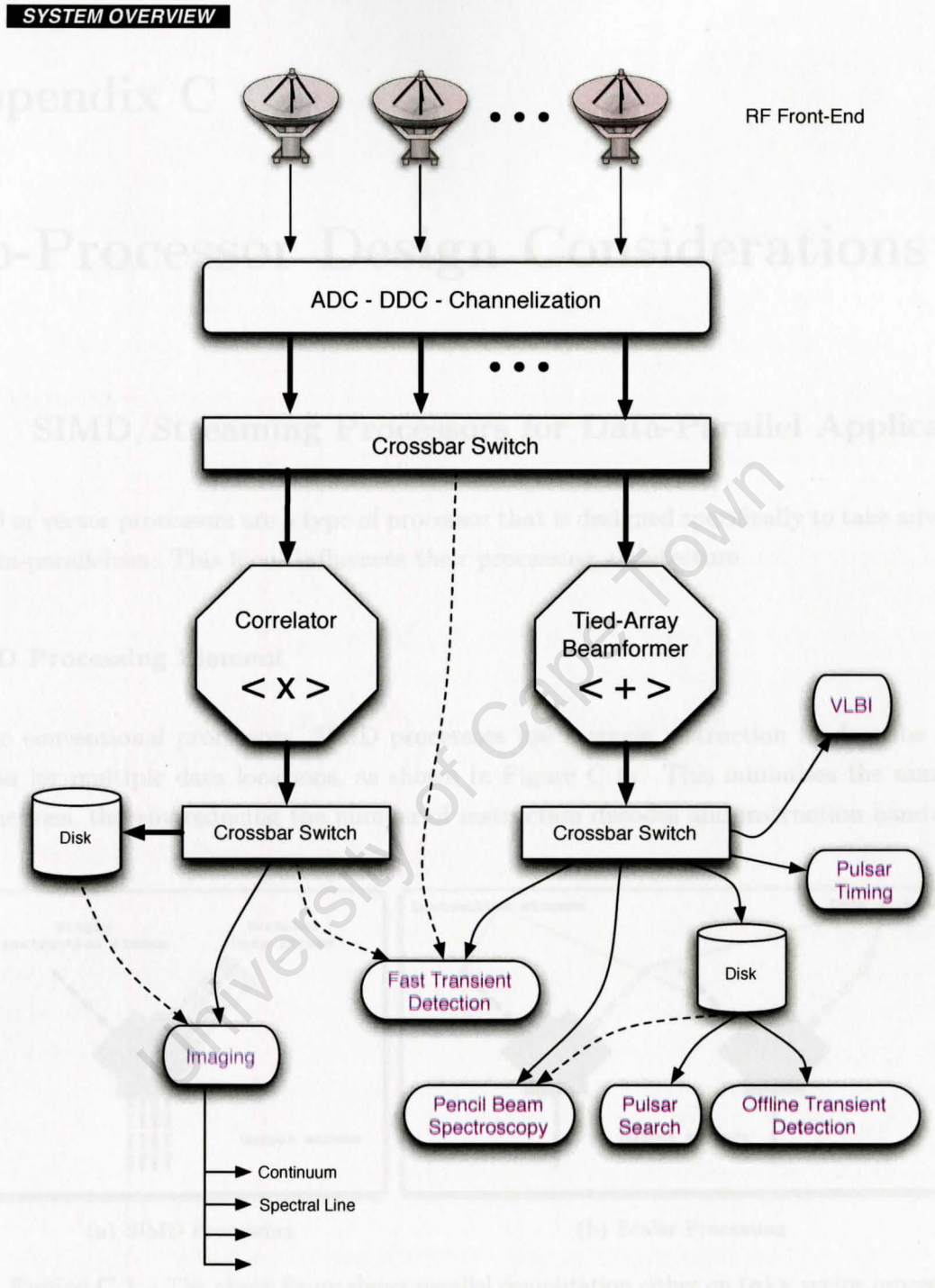



**Figure B.5** – Radio Astronomy Processing Pipeline, courtesy of Lord and van der Merwe [21]

# Appendix C

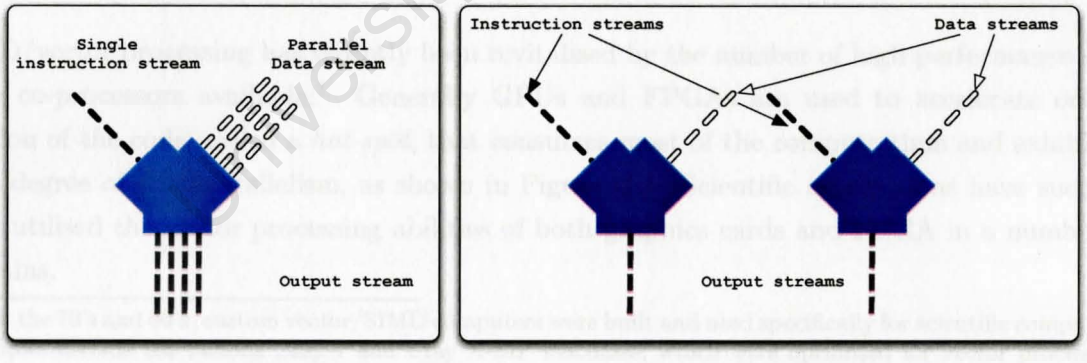
# Co-Processor Design Considerations

## C.1 SIMD/Streaming Processors for Data-Parallel Application

SIMD or vector processors are a type of processor that is designed specifically to take advantage of data-parallelism. This focus influences their processing architecture.

### SIMD Processing Element

Unlike conventional processors, SIMD processors use a single instruction to describe an operation for multiple data locations, as shown in Figure C.1a. This minimises the number of instructions, thereby reducing the number of instruction decodes and instruction bandwidth.



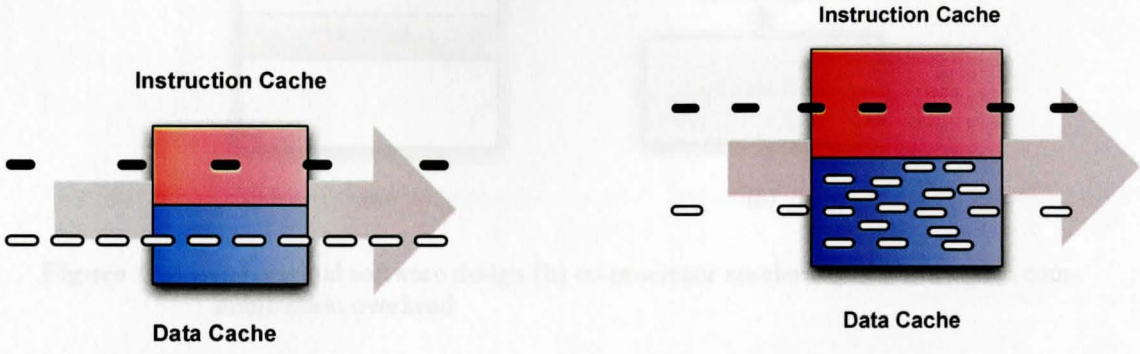

**(a)** SIMD Processing **(b)** Scalar Processing

**Figure C.1** – The above figure shows parallel computation either on **(a)** a vector processor or **(b)** the data-parallelism being exploited by multiple scalar processors. However **(b)** requires an instruction stream for each scalar processor and synchronisation of data. Inspired by Arstechnica [33]

### SIMD Memory Architecture

Desktop applications are generally I/O centric, requiring fast random access to different parts of program memory. Because processor performance has grown faster than off-chip memory access speeds, CPUs are forced to hide latencies by using large on-chip caches and more complex

prefetching techniques. Figure C.2b shows a typical program flow of a desktop application and the need for large data caching.




(a) Streaming Applications

(b) Static Applications

**Figure C.2** – The different data flows of **(a)** a streaming application on a SIMD processor, with little need for caches and **(b)** a desktop application with large cache to provide low memory latency. Inspired by Arstechnica [33].

Data-parallel applications, are processing intensive and perform repetitive operations on a predictable flow of data. Since there is little data reuse, cache size has little effect on performance and repetitive operations mean that there is little need for out of order processing. Because of this, most of the processor die is used to make many simple computation units that lack the complexity and cache of modern microprocessor design. Figure C.2a shows a typical program flow of a streaming application and the need for only small data cache.

### C.1.1 SIMD Co-Processors in HPC

SIMD/vector processing has recently been revitalised by the number of high performance software co-processors available.[1] Generally GPUs and FPGAs are used to accelerate only a portion of the code, called a *hot-spot*, that consumes most of the compute time and exhibits a high degree of data-parallelism, as shown in Figure C.3. Scientific applications have successfully utilised the vector processing abilities of both graphics cards and FPGA in a number of domains.

[1] In the 70's and 80's, custom vector/SIMD computers were built and used specifically for scientific computing. Examples include the famous Cray-1 and Cray X-MP machines, which were optimised for vector processing. However, in the 90's, with the success of the personal computer, and the increasing cost and complexity of semiconductor fabrication, custom vector processors could not compete with the now commodity desktop microprocessors. Today, most scientific computers are built or derived from processor technology originally intended for other computing domains like personal or transactional computing.

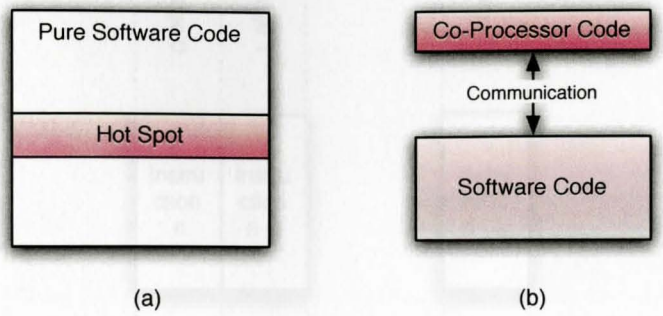



**Figure C.3** – (a) original software design (b) co-processor accelerated software with communication overhead

## C.2 Deep and Wide Parallelism

High level languages for FPGAs hide many of the complexities of FPGA development and can create parallel pipelined processing engines. However, the user still needs to write HLL code in a way that can be parallelised by the Dime-C compiler. The types of parallelism and the restrictions are presented below:

### Pipelining (Deep or Temperal Parallelism) also Systolic Array

Pipelining is an important concept to microprocessors and this is no different for RC [72, 73]. Pipelining allows instructions to be issued before the previous instruction has been completed.

Typically instuctions take more than one clock cycle to be computed and the amount of time it takes is often refered to as the instruction latency, L (measured in clock cycles). In an unpipelined execution unit running a program with $N$ instructions, it would take $L \times N$ cycles to complete [72]. However in a pipelined execution unit, the same program would only take $L + N$ cycles [2] [3].

Figure C.4 shows an 'L' staged pipeline engine computing 'n' instructions. Building pipelined execution units is a key concept for RC. Pipelining coupled with parallel computation is what creates speedups.

### Simultaneous Execution (Wide or spacial Parallelism)

Apart from pipelining, it is important to identify where instructions can be executed in parallel. For the correlator this happens in two cases: when the same instruction is executed on different independent data (SIMD); and when a single output is created from a series of simple instructions in a reduction operation.

[2] A cycle is the time to complete a single stage of a pipeline, which might not necessarily be equivalent to one clock tick.

[3] ignoring all pipelined hazards

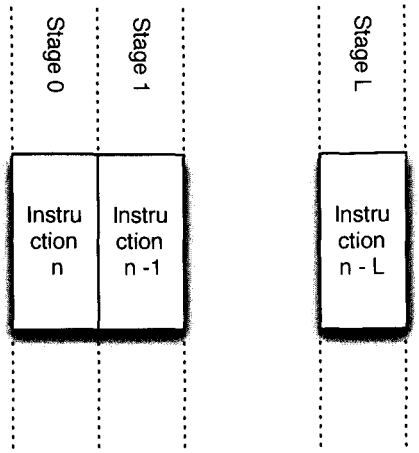



**Figure C.4** – A processing pipeline with 'L' stages. If no pipeline hazards occur, 'n' instructions can be computed in n+L clock cycles.

### C.2.1 SIMD Execution

The first case is the classical SIMD (Single Instruction Multiple Data) case. Here we have the same instruction applied to an array of independent data. For example:

```
for(i=0;i<100;i++)
   A[i] = B[i] + C[i];
```

In the above example we are free to compute each element of array $A$ in parallel, since each operation is independent. Now we can divide the work between the different processing elements. This type of parallelism is fundamental to SSE, GPUs and FPGAs. Thus in a pipelined processor, with $P$ different processing elements, our program is able to execute in $\frac{L+N}{P}$ cycles. Figure C.5 shows two pipelined engines computing in SIMD fashion.

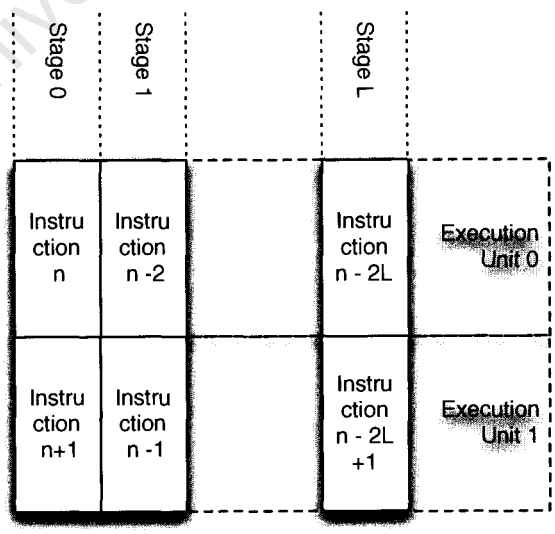



**Figure C.5** – 2 pipelined engines computing interleaved instruction. In a true SIMD processor, only one instruction would describe the operation of both processing elements.

### C.2.2 Reduction

In the second case of parallel instructions, is identifying output that is computed from a series of instructions. Again, an example of this is:

```
for(i=0;i<100;i++)
   A[i] = (B[i] + C[i]) + (D[i] - E[i]);
```

Above a complex expression involves three separate operations. On a traditional microprocessor, to calculate '$A$', the expression must be decomposed into three simple operations, and take three passes through the pipeline before the result can be computed, ie:

```
for(i=0;i<100;i++) {
    temp_reg0 = B[i] + C[i];
    temp_reg1 = D[i] - E[i];
    A[i]      = temp_reg0 + temp_reg1;
    }
```

However in this case, since the FPGA has a reconfigurable pipeline, it is not limited to computing a single operation per pass, as a microprocessor is. Therefore, the above can be computed in a single pass through a custom pipeline. Complex expressions as shown above, with '$N$' operations can be decomposed into $\log_2 N$ stages. Thus, in the best case scenario, a pipelined engine, with decomposed operations and '$P$' processing elements, can is able to execute a program in $\log_2 (N + L)/P$ cycles.

Figure C.6 shows how 3 additions can be performed in two stages.

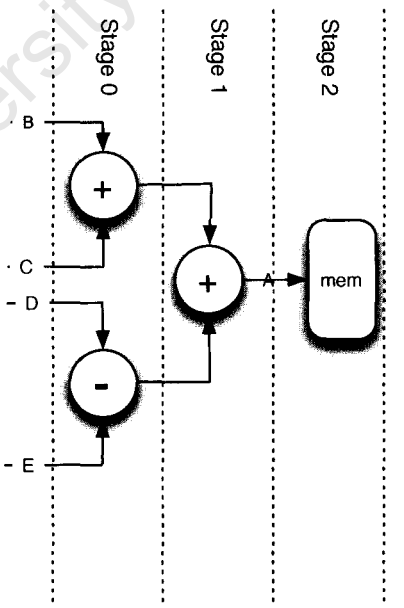



**Figure C.6** – 3 adders are used in a reduction operation to compute A = B + C + D + E. By computing B + C and D + E independently and in parallel, 'A' can be computed in only two stages. In general '$N$' elements can be computed in '$\log_2 N$' stages.

## C.3 Memory and I/O Limitations in GPUs and FPGAs

In section C.2, we describe the ideal case of parallelism, or the extent we aim for. However it comes to implementation, we run into problems which limit the extent of parallelism that we can achieve.

Memory technologies have improved at a slower rate than processor technologies, and building a computationally dense multi-core processor exaggerates this problem. Commonly a computational unit waists cycles waiting for data and actual application performance can be significantly less than theoretical performance.[4] The ideal is to create an application that runs as close to theoretical peak performance as possible. What limits this is often the memory constraint of an architecture. Apart from the speed and size limitations the following memory two issues surfaced during our correlator implementation:

### Addressing Multiple Global Memory Addresses

Different execution units operate on different memory locations simultaneously. This involves moving data from external memory into the processing elements. Multiple accesses puts a huge burden on memory, greatly increasing the bandwidth needed. Both GPUs and FPGAs address this issue differently:

*Coalesced Access (GPUs)*: Ideally we would like to be able for each PE to address any location in global memory independently of other PE. Unfortuantley this would require that each PE has a separate address and data bus, which would be unreasonably expensive. Instead, as a compromise, GPUs are able to fetch 16 adjacent memory locations per memory access, requiring only a single address location and a larger data bus. [5] The different PEs appear to the user as separate threads and these threads are grouped together in groups called warps [2]. If warps access sequential memory addresses, the GPU coalesces the memory requests into a single linear memory accesses and we get much better memory performance.

*Memory Striping (FPGA)*: The Xillix FPGA that was used had 240 of individually configurable block rams available. The block rams can be stringed together to create a single addressable memory space, which would be ideal. Unfortunately in this configuration, only one memory address can be accessed per clock, which is not sufficient. Instead of one large address space, the block ram can be configured in many separate and independent memory banks, with each bank addressable per clock. This provides the bandwidth desired, but requires the user manually separate data into the respective banks. This is known as memory striping as shown in Figure C.7.

### Communication Bus Speed

The GPU and FPGA are both connected to the host machine via a communication expansion bus. The communication bus is the co-processors interface to the host machine, which holds

[4] The theoretical performance of different architectures is shown in Figure 1.3a.

[5] Different Nvidia GPUs have different sized busses. The smaller buses found in the low end cards would need to make multiple fetches from memory

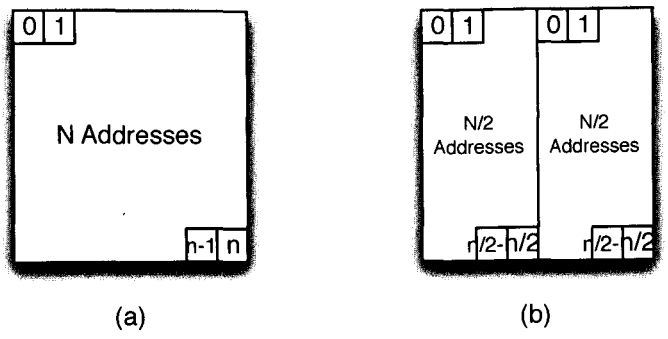



**Figure C.7** – Striped Memory

the data for processing. So before computation can begin, there is the overhead of transfering data from host to co-processor. The speed of the bus is important to minimize the overhead. The busses used by each co-processor were:

*PCI-X (FPGA)*: The Nallatech FPGA uses a PCI e**X**tended interface to communicate to the host. PCI-X is a revision to the popular PCI bus. Like PCI, PCI-X is a parallel bus, but supports double the clock rate. The Nallatech FPGA was able to achieve data rates in the region of 400MB/s in half-duplex mode and 100MB/s full duplex mode [6]. These data rates are relatively slow by today's standards and the limitations caused by the bus influenced the FPGA correlator's performance.

*PCIe (GPU)*: The Nvidia GPU uses a PCI Express bus, the successor to PCI-X. The serial PCIe bus is able to achieve much higher data rates than PCI-X and we were able to get transfers in the region of 1.4GB/s in both full and half duplex.



[6] Theoretical data rates according to the PCI-X spec are 1064MB/s

# Appendix D

# Testing

This section describes the testing procedure. The two objectives of the testing were to verify that our correlation algorithm was valid and record the data precision of the various architectures. Secondly since the G80 CUDA GPU is not 100% IEEE754 floating point compliant, we set out to measure the difference between the CPU and GPU correlator implementations[1].



## D.1 Output Validation

Comparing our different correlator implementations does not validate the correlation output as it will not detect if the correlation algorithm implemented is correct. We validated the CPU correlator implementation in two tests:

i. compare the power spectral-density output produced by our CPU correlator and simulated in Python.

ii. ensure the power spectral-density function computed by our correlator implementations, produces the same result as the Fourier transform of the autocorrelation (Wiener-Khinchin theorem) as shown in Equation D.1.

$$\mathscr{F}\{R_f(\tau)\} = S_f(\omega) \tag{D.1}$$

## D.2 Data Precision Impact

The G80 CUDA GPU is not 100% IEEE754 floating point compliant and to measure the impact we compared the correlation of two random noise signals on the GPU and CPU. Table D.1 compares the results of the cross-product spectrum with different number of spectral points and accumulation length. All random signals were generated from the same initial seed.

[1] All testing was performed on synthetic data.

**Table D.1** – CPU vs. GPU output

| FFT length | Accumulation Period | Average Correlation Output | Normalised Average Error | $\sigma$ | Normalised $\sigma$ |
|---|---|---|---|---|---|
| 32 | 10 | 6.92 | 8.89e-8 | 6.64e-7 | 8.79e-8 |
| | 100 | 68.17 | 2.42e-7 | 2.02e-5 | 2.47e-7 |
| | 1,000 | 663.43 | 7.32e-7 | 5.75e-4 | 8.44e-7 |
| | 10,000 | 6667.28 | 1.78e-6 | 1.52e-2 | 2.80e-6 |
| 256 | 10 | 6.77 | 7.66e-8 | 5.95e-7 | 9.59e-8 |
| | 100 | 66.42 | 2.10e-7 | 1.64e-5 | 2.96e-7 |
| | 1,000 | 666.45 | 6.87e-7 | 5.63e-4 | 8.67e-7 |
| | 10,000 | 6666.05 | 2.23e-6 | 1.86e-2 | 2.28e-6 |

Table 6.8 shows that there is very little difference in the GPU and CPU output. The normalised standard deviation value grows in proportion to the average correlation output, likely due to the fact that more of the mantisa is required to represent the integer part of large numbers and limits the accuracy of the fractional part. However, even at worst case, the error is small enough to not raise any concern.

The Dime-C uses IEEE754 floating point representation, so the differences between the CPU and FPGA were only related to float round errors.

(aprox 1.5pgs. [ex. pics])

i. Correlation FPGA and GPU output agrees with:

- mathematical description of sinusoidal correlation
- pre and post correlation power conservation (autocorrelation)
- Testing Data
  - Synthetic
  - Real from PED data

ii. Data Precision Impact

- Associative effect on output
- Nallatech Floating point arithmetic is IEEE compliant
- G80 GPUs used are not true IEEE compliant, effects analysed.

iii. additional assumptions:

(a) Assumed single polarisation.

(b) not IQ data **see polar pg 38** - Assume real data input, with no DC offset.

(c) Total band is divided up into a number of subband frequencies which is then divided up into channels - just assume one global freqency block.

(d) Basically this is arranged so that all data is pre-formatted - no need for corner turning in FFT - in the ideal format inorder to test the correlator, this is not a complete correlation design - but will try reference to articles where the specific simplifications are dealt with.

(e) Assuming the data is all sampled with the same global clock, so phase information is coherent across all inputs.

(f) All input is arrange by a number of time samples for a certain antenna before FFT. ie the FFT input has been corner turned already to allow for linear memory addressing.

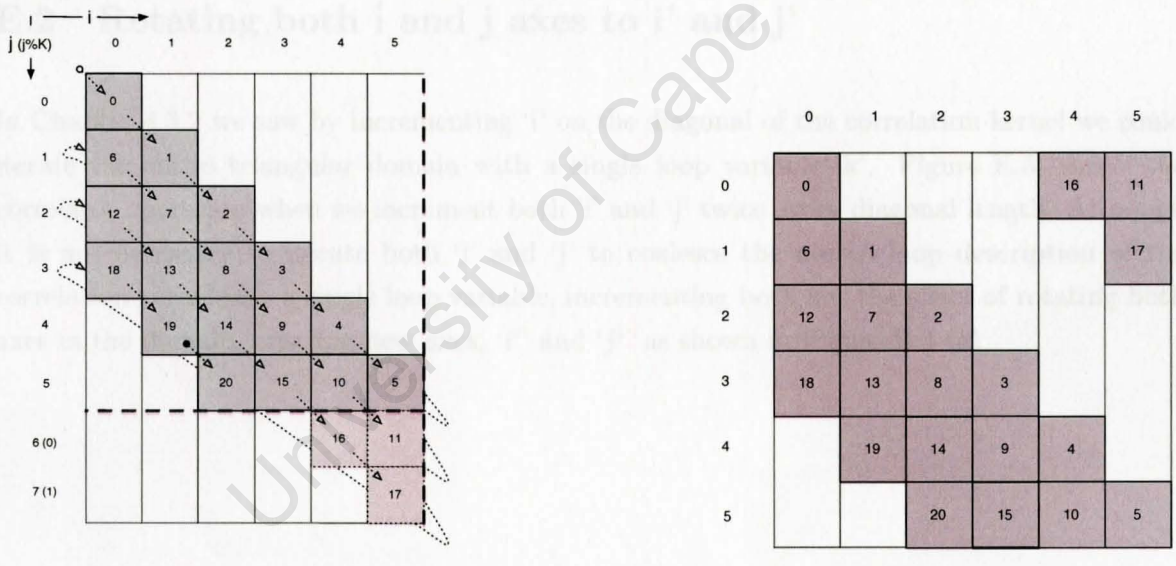
(g) input to F engine - real data, input to X engine complex

# Appendix E

# Correlation on FPGAs

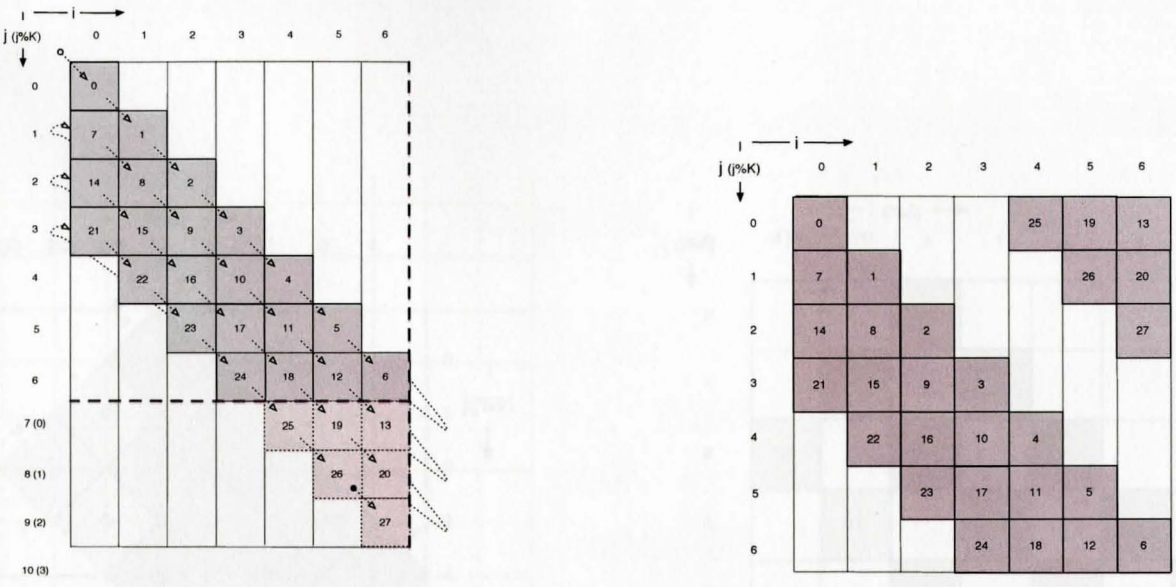
## E.1 FPGA correlation examples

Figures E.1 and E.2 are more examples of the single loop with double buffered input referenced from Chapter 4.




**(a)** **(b)** 21 + L

**Figure E.1** – Example Single loop diagonal width 6 and $K = 6$

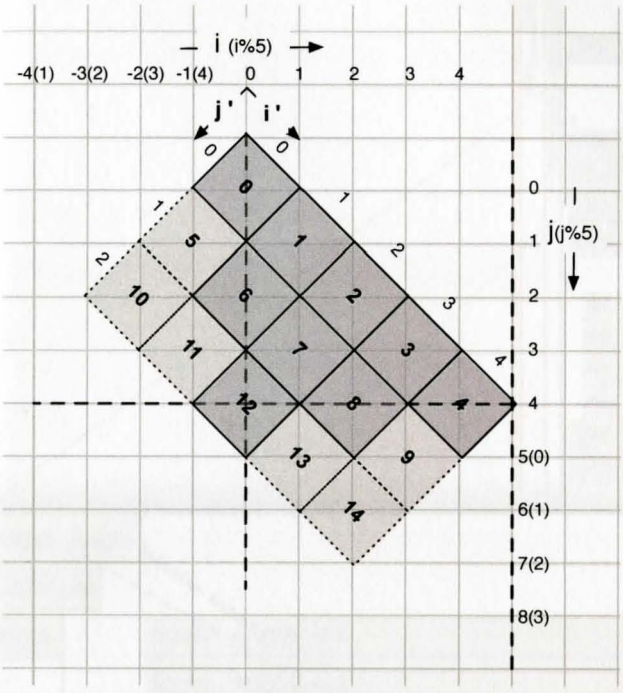



**(a)** **(b)** 28 + L

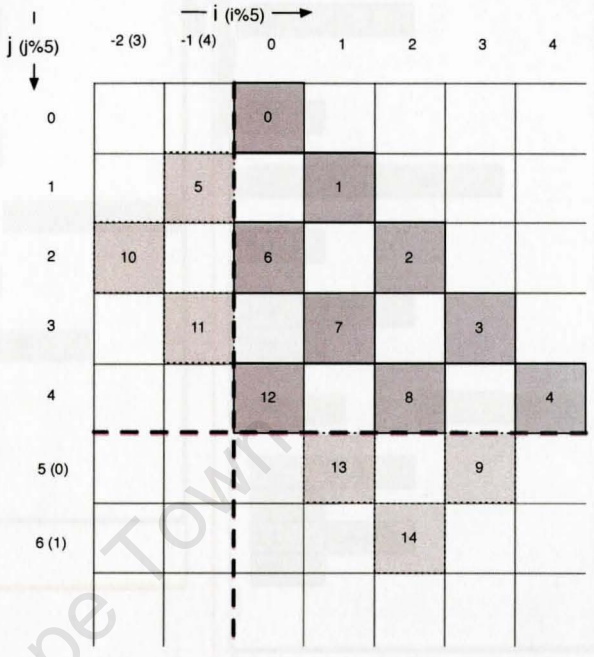
**Figure E.2** – Example Single loop diagonal width 7 and $K = 7$

## E.2 Rotating both i and j axes to i' and j'

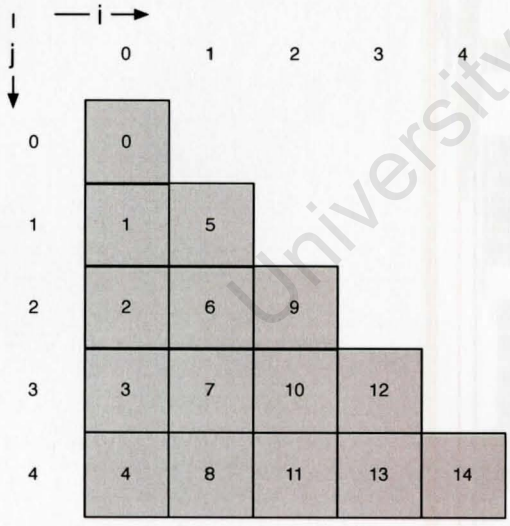
In Chapter 4.3.2 we saw by incrementing 'i' on the diagonal of the correlation kernel we could iterate the entire triangular domain with a single loop variable 'k'. Figure E.3, shows the correlator operation when we increment both 'i' and 'j' twice every diagonal length. Although it is not necessary to iterate both 'i' and 'j' to coalesce the nested loop description of the correlation kernel into a single loop variable, incrementing both has the effect of rotating both axes in the domain, creating new axes, '$i'$' and '$j'$' as shown in Figure E.3 (a).



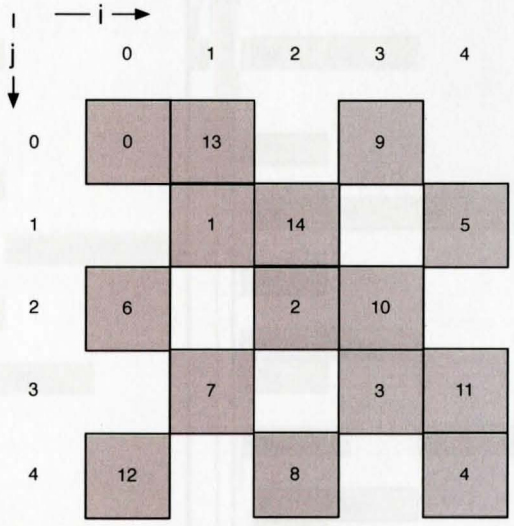

(a) diagonal increment on rotated axis no mod




(b) diagonal increment on normal axis, no mod




(c) column increment




(d) result with mod

**Figure E.3** – Rotation of both '$i$' and '$j$' axes by incrementing on both '$i$' and '$j$' on the diagonal to create rotated axes '$i$'' and '$j$''. This is one such method to describe the triangular correlation kernel with a single loop variable '$k$', from which '$i$' and '$j$' can be derived. The value of the incrementing '$k$' is drawn inside each block. See Chapter 4.3.2

## E.3 Implementation Pictures

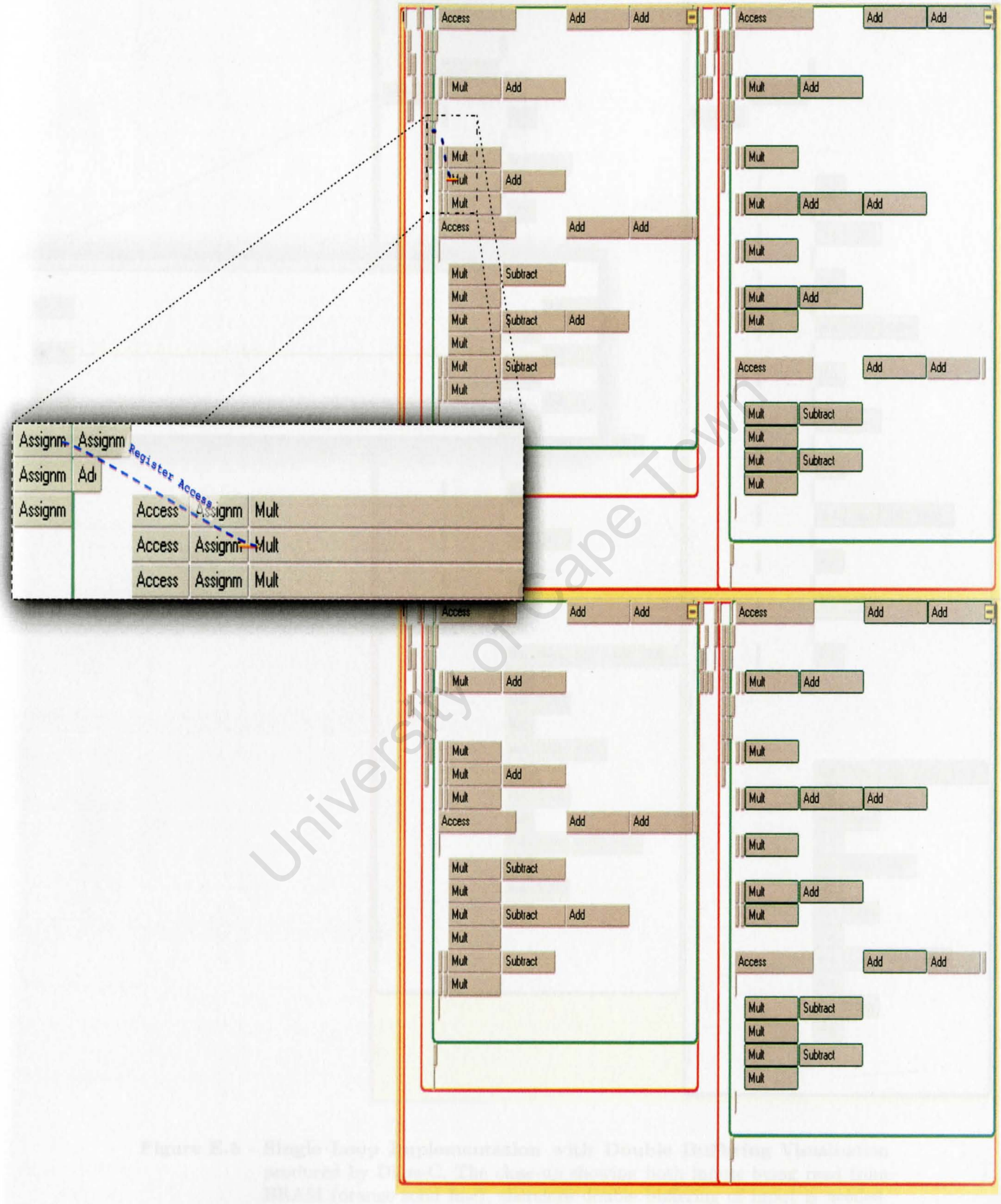



**Figure E.4 – Nested Loop Implementation** - Visualisation produced by Dime-C. The close-up showing one of the correlator inputs being read from a register (dotted line) and the other from BRAM (orange solid line), so no double buffering of input is needed. See Chapter 4 for details.

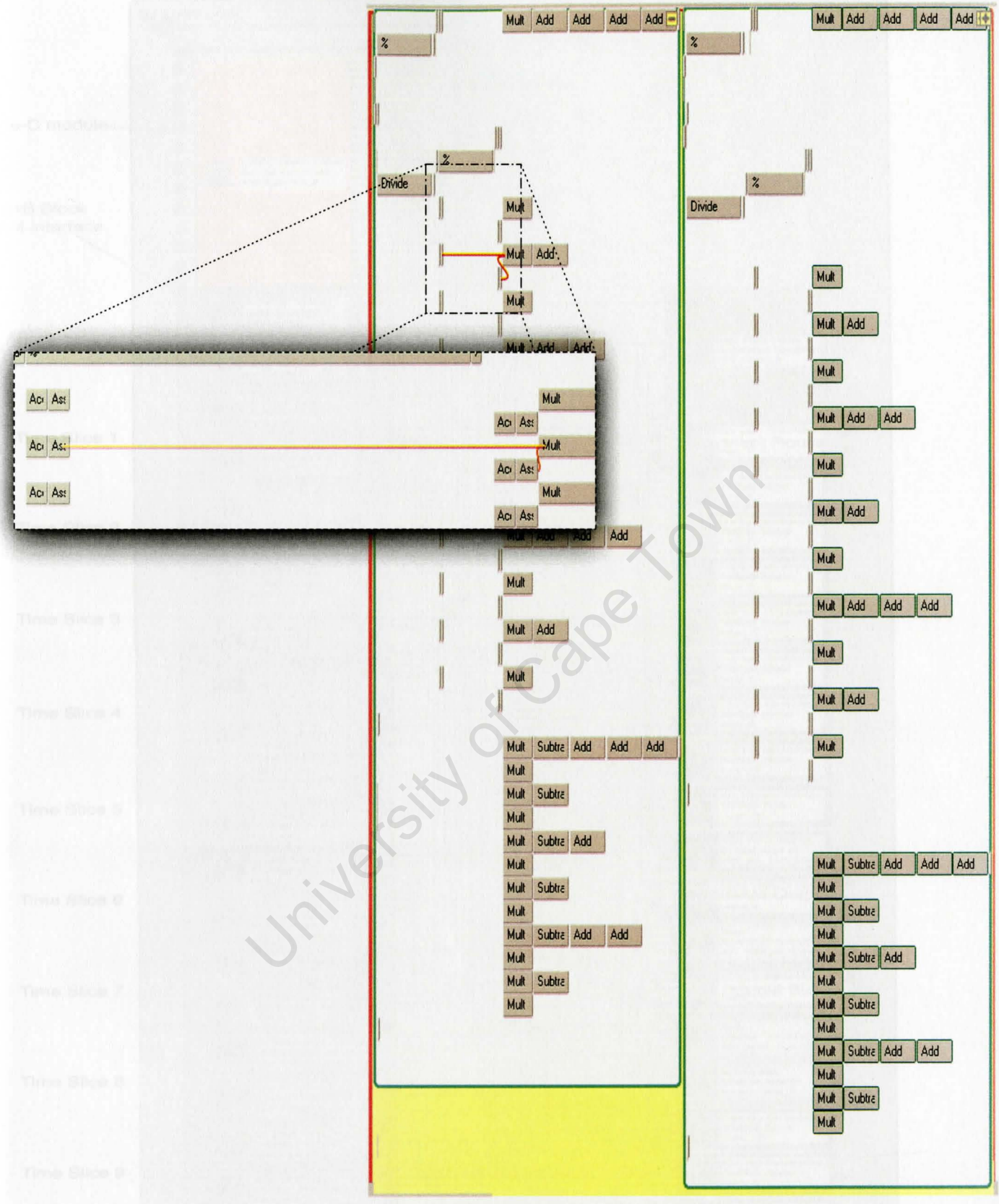



**Figure E.5 – Single Loop Implementation with Double Buffering** Visualisation produced by Dime-C. The close-up showing both inputs being read from BRAM (orange solid line), therefore double buffering of input is needed. See Chapter 4 for details.

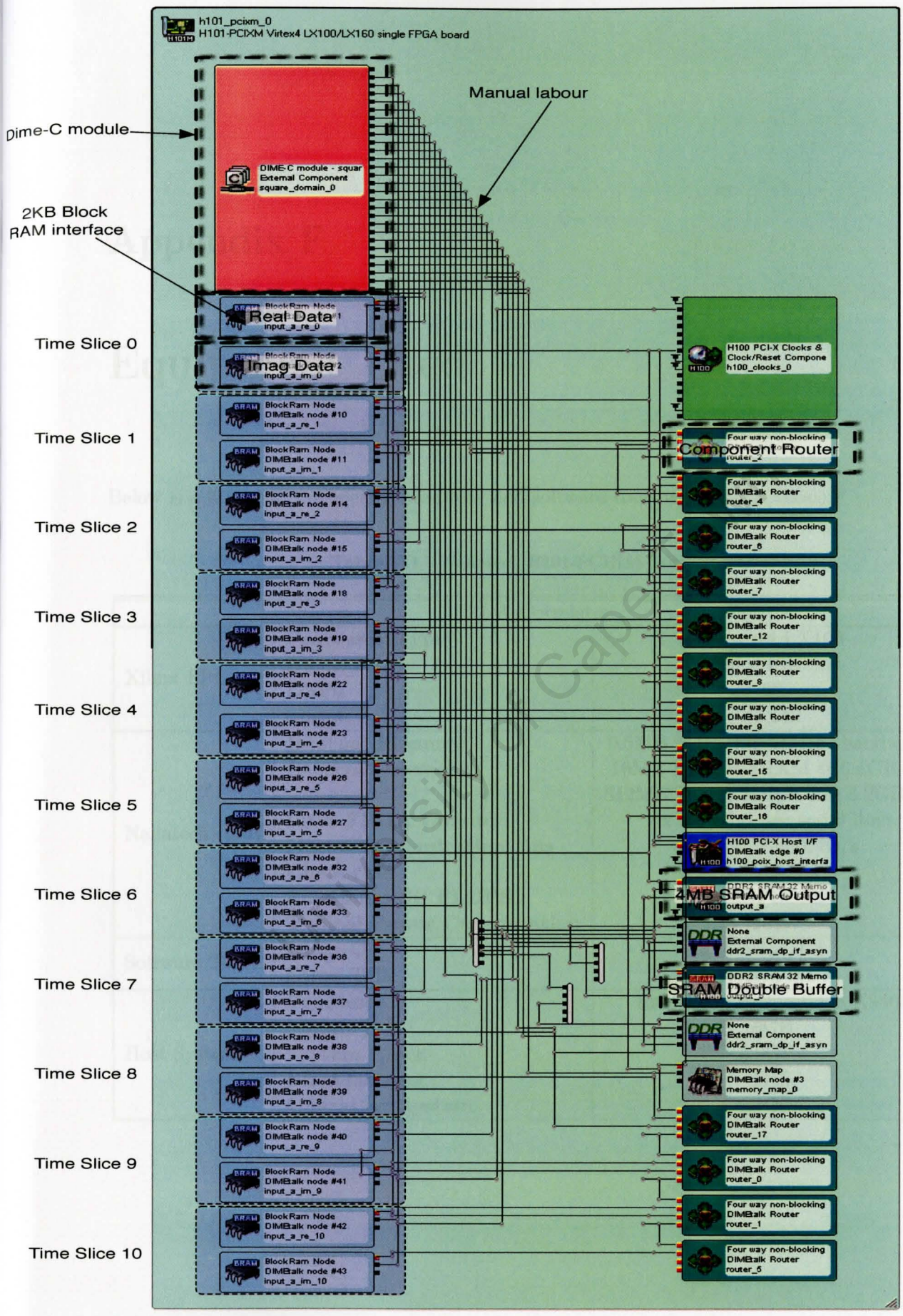



**Figure E.6** – Dime-Talk network used to construct the desired firmware interfaces to the H101 board and connect them to the Dime-C block. This must be done manually by the user.

# Appendix F

# Equipment Used

Below is a list of all the specific hardware and software tools used in this thesis:

**Table F.1** – Nallatech H101-PCIXM Correlator

| FPGA Correlator | | |
|---|---|---|
| Xilinx FPGA | **Processor Type** | Virtex-4 LX100 |
| | **Block Ram** | 240 x 18Kbits |
| | **DSPs** | 96 |
| | **Slices** | 49,152 |
| Nallatech H101 | **Internal Memory** | 0.5MB @ 0.5 TBytes/sec bandwidth |
| | **External Memory** | 16MB DDR-II SRAM @ 6.4GB/sec |
| | | 512MB DDR2 SDRAM @ 3.2GB/sec |
| | **Inter FPGA Comm.** | 4x 2.5 Gbit/sec serial links |
| | **Host Communication Bus** | PCI-X @ 400MB/s |
| | **Clock rate** | 100-200MHz |
| | **Maximum SP FLOPS** | 20GFLOPS |
| | **Typical Power Consumption** | 25W |
| Software Tools | **Dime-C** | Version 1.3 |
| | **Dime-Talk** | Version 3.1.7 |
| Host System | **Processor** | Intel Xeon Harpertown X5450 |
| | **Memory** | 8GB |
| | **System Clock** | 3.0GHz |
| | **Manufacturer** | Dell |
| | **Operating System** | CentOS 5.2 |

**Table F.2** – Nvidia 9800GT Correlator.

| GPU Correlator | | |
|---|---|---|
| Nvidia GPU | **Processor** | 9800 GT GPU (G92)<br>112 SPs (14 MPs) @ 1.5GHz |
| | **Internal Memory** | 8192 32bit Registers/MP<br>16KB Shared Memory/MP |
| | **Memory interface** | 256bit |
| | **Host Communication Bus** | 16 lane PCI-E 2.0 @ 8GB/s |
| | **Maximum SP FLOPS** | 504 GFLOPS |
| Zotac Board | **Onboard Memory** | 512MB GDDR3@ 57.6GB/sec |
| | **Maximum Power Consumption** | 105W |
| Software Tools | **CUDA** | Version 2.0 |
| Host System | **Processor** | Intel Core 2 Duo E6750 |
| | **Memory** | 3GB |
| | **System Clock** | 2.67GHz |
| | **Manufacturer** | Dell |
| | **Operating System** | Ubuntu 8.10 |



**Table F.3** – Intel Harpertown Correlator.

| CPU Correlator | | |
|---|---|---|
| Intel CPU | **Processor** | 3.0Ghz Xeon Harpertown X5450<br>Quad Core |
| | **Internal Memory** | 12MB L2 Cache |
| | **Onboard Memory** | 8GB DDR2 |
| | **Memory interface** | Dual Channel 2x64bit |
| | **Maximum Power Consumption** | 120W |
| | **Manufacturer** | Dell |
| | **Operating System** | Ubuntu 8.04 x64 |
| Software Tools | **Intel Performance Primitives** | Version 5.3.1 |

# Appendix G

# Derivations

## G.1 Computing Complex Input

$$\begin{aligned} & S_i[v]S_j^*[v] \\ &= (a+jb)(c+jd)^* \\ &= (a+jb)(c-jd) \\ &= (ac+bd)+j(bc-ad) \end{aligned} \tag{G.1}$$

## G.2 Commutative Conjugate Multiplication Derivation

$$\begin{aligned} & (a+jb)(c+jd)^* \\ &= (a+jb)(c-jd) \\ &= (ac+bd)+j(bc-ad) \end{aligned}$$

$$\begin{aligned} & \Big((a+jb)^*(c+jd)\Big)^* \\ &= \Big((a-jb)(c+jd)\Big)^* \\ &= \Big((ac+bd)-j(bc-ad)\Big)^* \\ &= (ac+bd)+j(bc-ad) \end{aligned}$$

$$\therefore (a+jb)(c+jd)^* = \Big((a+jb)^*(c+jd)\Big)^* \tag{G.2}$$

## G.3 Correlator Output Derivation

$$\begin{aligned}\mathbf{C}_{(a_{n+1},i,j)} &= \mathbf{C}_{(a_n,i,j)} + S_{(a_n,i)}[v_n]S^*_{(a_n,j)}[v_n] \\ &= \mathbb{R}\{\mathbf{C}_{(a_{n+1},i,j)}\} + \mathrm{j}\,\mathfrak{I}_{\mathfrak{m}}\{\mathbf{C}_{(a_{n+1},i,j)}\}\end{aligned}$$

$$\begin{aligned}\mathbb{R}\{\mathbf{C}_{(a_{n+1},i,j)}\} &= \mathbb{R}\{\mathbf{C}_{(a_n,i,j)} \quad + \quad S_{(a_n,i)}[v_n]S^*_{(a_n,j)}[v_n]\} \\ &= \underbrace{\mathbb{R}\{\mathbf{C}_{(a_n,i,j)}\}}_{\mathbf{P_{a_n}}} \quad + \quad P_{ij}\end{aligned}$$

$$\begin{aligned}\mathfrak{I}_{\mathfrak{m}}\{\mathbf{C}_{(a_{n+1},i,j)}\} &= \mathfrak{I}_{\mathfrak{m}}\{\mathbf{C}_{(a_n,i,j)} \quad + \quad S_{(a_n,i)}[v_n]S^*_{(a_n,j)}[v_n]\} \\ &= \underbrace{\mathfrak{I}_{\mathfrak{m}}\{\mathbf{C}_{(a_n,i,j)}\}}_{\mathbf{Q_{a_n}}} \quad + \quad Q_{ij}\end{aligned}$$

$$\therefore\ \mathbf{C}_{(a_{n+1},i,j)} = \mathbf{P_{a_n}} + P_{ij} + \mathrm{j}\left(\mathbf{Q_{a_n}} + Q_{ij}\right) \qquad \text{(G.3)}$$

# Appendix H

# DiFX

The Distributed FX[1] (DiFX) correlator is a popular software correlator implementation. The DiFX correlator was developed at Swinburne University by Adam Deller, and is a parallel, open-source, software implementation of a fully functional radio astronomy correlator [5]. Designed to work with the less processor intensive, very long baseline interferometry (VLBI)[2], the DiFX is an attractive correlator solution for smaller correlator arrays. The DiFX correlator has had a positive response in both astronomy and HPC communities, allowing research to be carried out on standard Linux compute clusters, without sharing or endangering production correlators. The National Radio Astronomy Observatory (NRAO) and Max Plank Institute fur Radioastonomie (MPIfR) have adopted the DiFX correlator for the correlation of their Very Long Baseline Array (VLBA) data [30] [31] and have released their own NRAO-DiFX modification [32].

The original plan for this thesis was to accelerate the DiFX correlator directly using FPGA and GPU co-processors. This would have the potential to create an accelerated correlator to an already existent user base.

By profiling the DiFX we identified hot-spots suitable for acceleration. The profiling uncovered that the DiFX correlator makes many short calls to its software correlation engine. This is not problematic in software, where there is negligible function call over head, however if implemented directly on a co-processor would cause large co-processor call overheads, nullifying any achievable speedup. This could potentially be addressed by buffering the small frequent correlation function calls and transform them into larger, but less frequent co-processor function calls. However, the DiFX correlator is a large software project, and it was easier to first extract the DiFX's core correlation engine and work on it independently, which would avoid the interfacing issues and simplify validation. Although this removes the existing DiFX user base, it provided the simplified platform to investigate the suitability of FPGA and GPU correlation acceleration.

Integrating an accelerated DiFX correlation core is left for future work, however the profile summaries of the DiFX are presented below in Figures H.1, H.2, H.3 and H.4.

[1] FX here refers to how the correlation is performed. FX correlators do a multiplication in the Fourier domain, while XF correlators perform a convolution in the time domain.

[2] VLBI typically uses smaller arrays (<10) with baselines that can span 1000s of kilometers. Since there is relatively small number of data sources, produced at distributed sites it is practical to perform off-line correlation.

**Figure H.1** – DiFX Overview [74, 75].

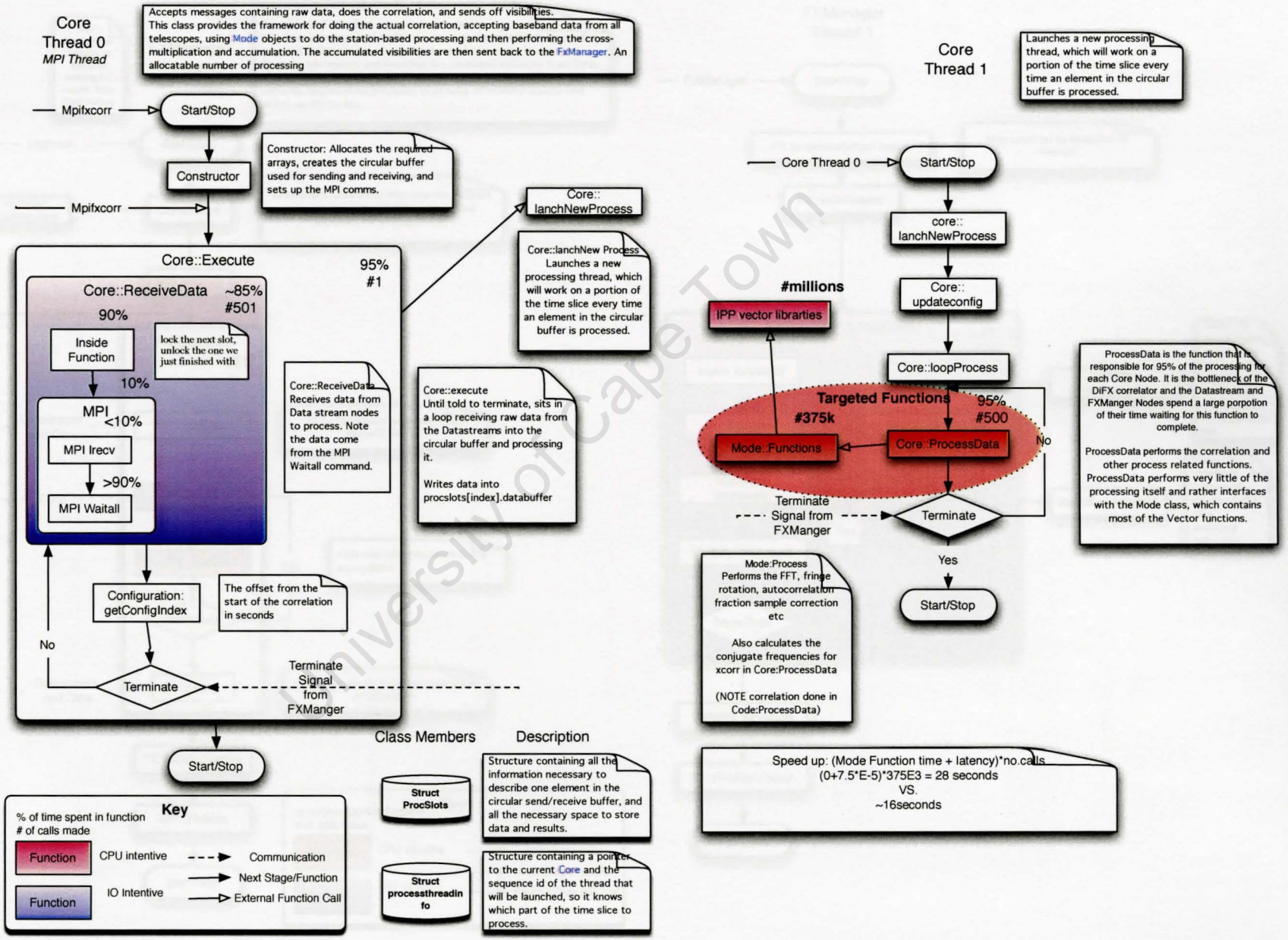



**Figure H.2** – DiFX Core Classes [74, 75].

**Figure H.3** – DiFX FX Manager Class [74, 75].

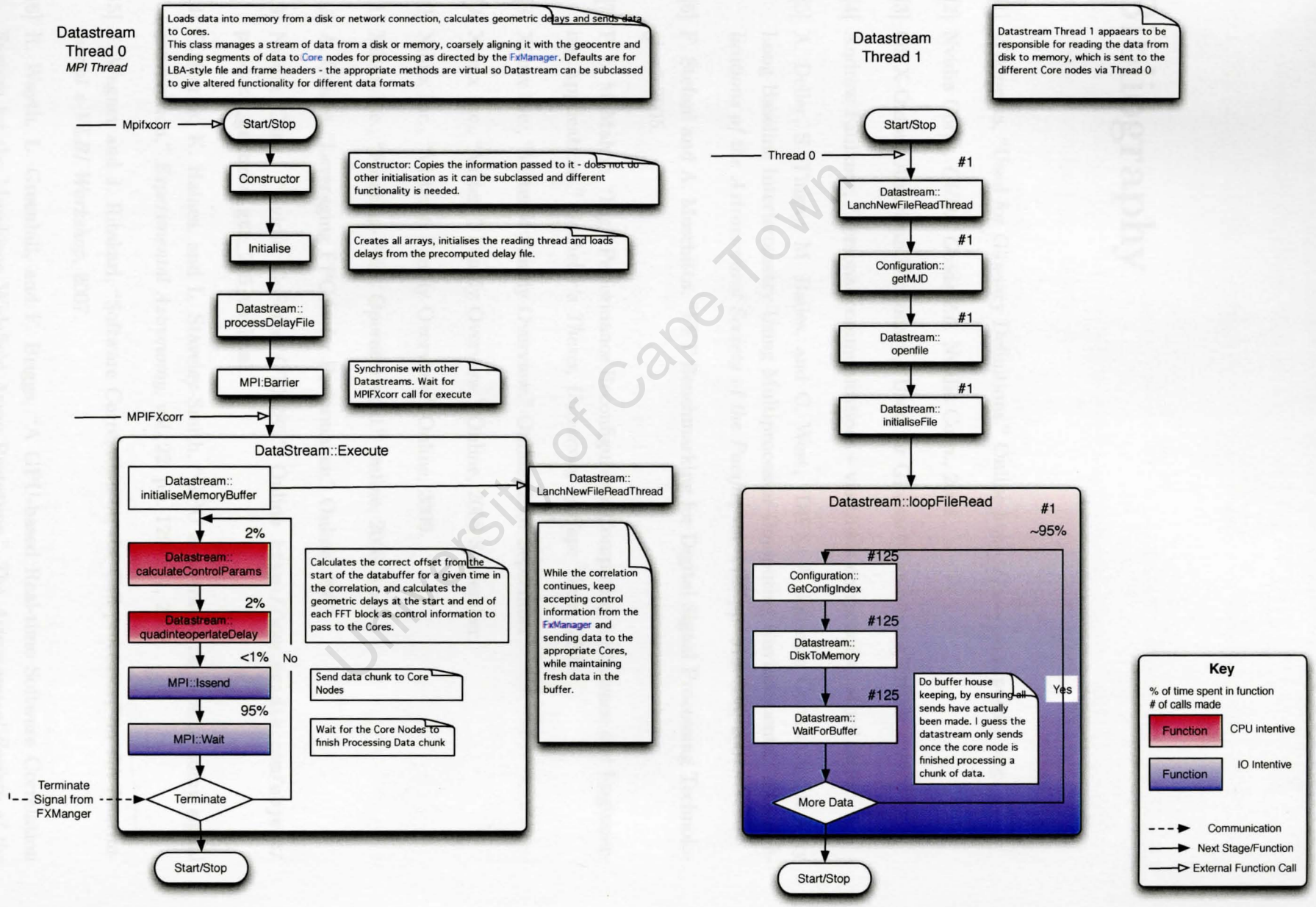



**Figure H.4** – DiFX Data-stream Class [74, 75].

# Bibliography


[1] Wikipedia, "Used for Glossary Definitions." Online `http://www.wikipedia.org`.

[2] Nvidia Corp., *CUDA Guide 2.0.* Nvidia Corp., 2008.

[3] OS X Oxford American Dictionary, "Used for Glossary Definitions," 2009.

[4] Andrew Faulkner, "Personal communications - via thesis corrections," April 2010.

[5] A. Deller, S. Tingay, M. Bailes, and C. West, "DiFX: A Software Correlator for Very Long Baseline Interferometry Using Multiprocessor Computing Environments," *The Publications of the Astronomical Society of the Pacific*, vol. 119, pp. 318–336, 2007.

[6] F. Stefani and A. Moschitta, "FFT Benchmarking for Digital Signal Processing Technologies," 2005.

[7] P. L. McMahon, "High Performance Reconfigurable Computing for Science and Enginnering Applications," *Bachelor's Thesis, University of Cape Town*, 2006.

[8] Xilinx Inc., "Virtex-4 Family Overview." Online, 2007 September.

[9] Xilinx Inc., "Virtex-5 Family Overview." Online, 2008 September.

[10] Xilinx Inc., "Virtex-6 Family Overview." Online, 2009.

[11] Xilinx Inc., "Floating-Point Operator v4.0." Online, 2008 April.

[12] A. Cantle, "Leveraging FPGAs for Performance." Online, 2007.

[13] Nvidia Corp., "Nvidia GeForce GTX285." Online `http://www.nvidia.com/object/product_geforce_gtx_285_us.html`.

[14] C. Harris, K. Haines, and L. Staveley-Smith, "GPU accelerated radio astronomy signal convolution," *Experimental Astronomy*, vol. 22, pp. 129–141, 2008.

[15] J. Wagner and J. Ritakari, "Software Correlation on the Cell processor," in *6th International e-VLBI Workshop*, 2007.

[16] R. Wayth, L. Greenhill, and F. Briggs, "A GPU-based Real-time Software Correlation System for the Murchison Widefield Array Prototype," *The Astronomical Society of the Pacific*, vol. August, no. 121, pp. 857–865, 2009.

[17] A. Thompson, J. Moran, and G. Swenson, *Interferometry and Synthesis in Radio Astronomy.* Wiley-VCH, 2nd ed., 2004.

[18] B. F. Burke and F. Graham-Smith, *An Introduction to Radio Astronomy.* Cambridge University Press, second ed., 2002.

[19] J. L. Jonas, *The 2326 MHz Radio Continuum Emission of the Milky Way.* PhD thesis, Rhodes University, 1998.

[20] R. Perley, "10th Synthesis Imaging Summer School," June 2006. University of New Mexico.

[21] R. van der Merwe and R. Lord, "Correlator Flow Diagram," Members of KAT Computing Team, Pinelands, Cape Town.

[22] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell, A. Mattingly, B. G. Elmegreen, R. van Nieuwpoort, G. van Diepen, and H. Sips, "Radioastronomy Image Synthesis on the Cell/B.E.," in *Euro-Par*, pp. 749–762, 2008.

[23] David Brodrick, "Fringe Dwellers." Online `http://fringes.org/`.

[24] R. Wayth, "Correlation for Radio Astronomy with GPUs: Mostly worth it.," tech. rep., ICRAR.

[25] W. Brisken, "10th Synthesis Imaging Summer School," 2006.

[26] M. P. Rupen, "11th Synthesis Imaging Summer School," 2008.

[27] C. Chang, J. Wawrzynek, and R. Brodersen, "BEE2: A High-End Reconfigurable Computing Systems," *IEEE Design and Test of Computer*, vol. 22, pp. 114–125, 2005.

[28] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Crescini, C. de Jesus, C. Dick, P. Droz, D. MacMahon, K. Meder, J. Mock, V. Nagpal, B. Nikolic, A. Parsa, B. Richards, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright, "PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy," *Asilomar Conference on Signals, Systems, and Computers*, vol. Oct-Nov, pp. 2031–2035, 2006.

[29] L. de Souza, J. D. Bunton, D. Campbell-Wilson, R. J. Cappallo, and B. Kincaid, "A Radio Astronomy Correlator Optimized for the Xilinx Virtex-4 SX FPGA," *Field Programmable Logic and Applications*, vol. Aug, pp. 62–67, 2007.

[30] J. Romney, "A VLBA Upgrade Conforming to VSOP-2 Specifications," in *VSOP-2*, Dec, 2007.

[31] W. Alef, D. Graham, H. Rottmann, and A. Roy, "Software Correlator at MPIfR: Status report," in *European VLBI Group for Geodesy and Astrometry Meeting*, 2007.

[32] W. Brisken, "A Guide to Software Correlation Using NRAO-DiFX Version 1.0." Online, Feb 2008.

[33] Arstechnica. Online `http://arstechnica.com/old/content/2000/04/ps2vspc.ars/5`.

[34] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," tech. rep., Electrical Engineering and Computer Sciences University of California at Berkeley, Dec, 2006.

[35] Xilinx Inc., "Xilinx History." Online.

[36] B. de Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "Fpga accelerator for real-time skin segmentation," in *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pp. 93–97, 2006.

[37] Z. K. Baker and V. K. Prasanna, "Efficient hardware data mining with the Apriori algorithm on FPGAs," in *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 2–12, 2005.

[38] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "A banded Smith-Waterman FPGA accelerator for Mercury BLASTP," in *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, pp. 765–769, 2007.

[39] "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Application Specific Processors*, vol. June, pp. 101–107, 2008.

[40] M. Herbordt, B. Sukhwani, M. Chiu, and M. A. Khan, "Production Floating Point Applications on FPGAs," in *2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*, 2009.

[41] G. Genest, R. Chamberlain, and R. Bruce, "Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C," *Adaptive Hardware and Systems*, vol. Aug, pp. 280–286, 2007.

[42] Nallatech, *Dime-C User Guide 1.3.* Nallatech.

[43] Nallatech, "H100 Series FPGA Application Accelerators: Product Brochure." Online `http://www.nallatech.com`.

[44] Xilinx Inc., "The Virtex-4 Power Play," *Xcell Journal Online*, vol. 52, pp. 30–33, September 2005.

[45] W. Wong, "FPGAs Move To 40 nm," *Embedded in Electronic Design*, vol. February, 2009.

[46] D. Thomas, L. Howes, and W. Luk, "A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation," in *FPGA*, 2009.

[47] The Entertainment Software Association, "Essential Fatcs." Online `http://www.theesa.com/facts/index.asp`.

[48] D. Luebke and G. Humphreys, "How GPUs Work," *IEEE Computer*, vol. Feburary, pp. 96–100, 2007.

[49] M. Macedonia, "The GPU Enters Computing's Mainstream," *IEEE Computer*, vol. 36, pp. 106–108, 2003.

[50] J. K. Ìger and R. Westermann, "Linear algebra operators for GPU im- plementation of numerical algorithms," in *ACM Transactions on Graphics*, pp. 908–916, 2003.

[51] N. K. Govindaraju, B. Lloyd, M. L. W. Wang, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 International Conference on Management of Data*, pp. 215–226, 2004.

[52] S.Che, J.Meng, J.W.Sheaffer, and K.Skadron, "A performancestudy of general purpose applications on graphics processors," in *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[53] T. Yamanouchi., "AES encryption and decryption on the GPU," *GPU Gems 3*, 2007.

[54] L.Nyland, M.Harris, and J.Prins, "Fast N-Body simulation with CUDA," *GPU Gems 3*, 2007.

[55] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *SC'08*, 2008.

[56] Nvidia Corp., "Nvidia Geforce 8800GT." Online `http://www.nvidia.com/object/product_geforce_8800_gt_us.html`.

[57] Intel Corp., "Intel performance primitives software library."

[58] Intel Corp., "Intel Xeon Processor X5450." Online `http://ark.intel.com/Product.aspx?id=34446`.

[59] FFTW, "FFT Benchmark Methodology." Online `http://www.fftw.org/speed/method.html`.

[60] Nallatech, "Benchmarking an FFT Complex Multiply IFFT function in DIME C," *Application Note.*

[61] P. Demorest, "National Radio Astronomy Observatory (NRAO) GPU Benchmarking." Online `http://www.cv.nrao.edu/~pdemores/gpu/`.

[62] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU2006 Results." Online `http://www.spec.org/benchmarks.html`.

[63] A. Parsons, D. Backer, A. Siemion, H. Chen, D. Werthimer, P. Droz, T. Filiba, J. Manley, P. McMahon, A. Parsa, D. MacMahon, and M. Wright, "A Scalable Correlator Architecture Based on Modular FPGA Hardware, Reuseable Gateware, and Data Packetization," *The Publications of the Astronomical Society of the Pacific*, vol. 120, pp. 1207–1221, November 2008.

[64] Jason Manley, "Personal communications," Jan 2010.

[65] J. Roy, "The GMRT Software Back-end : GSB," in *HPC in Observational Astronomy*, 2009.

[66] J. Roy, Y. Gupta, U.-L. Pen, J. Peterson, J. Kodilkar, and S. Kudale, "A real-time software backend for the GMRT : towards hybrid backends," in *CASPER Meeting Cape Town*, September 2009.

[67] C. Harris, K. Haines, and L. Staveley-Smith, "GPU FX Spectrometer using CUDA," *AstroGPU*, 2007.

[68] R. Wayth, L. Greenhill, and F. Briggs, eds., *A GPU based Realtime Software Correlation System for the Murchison Widefield Array Prototype*, 2006.

[69] D. Halliday, R. Resnick, and J. Walker., *Fundamentals of Physics.* John Wiley and John Wiley and Sons, 6th ed., 2001.

[70] W. Brisken, "10th Synthesis Imaging Summer School," June 2006. University of New Mexico.

[71] F. G. Stremler, *Introduction to Communication Systems.* Addison-Wesley, third ed., 1992.

[72] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th ed., 2006.

[73] P. L. McMahon, "Accelerating Genomic Sequence Alignment using High Performance Reconfigurable Computers," Master's thesis, Department of Computer Science, University of Cape Town, 2008.

[74] Adam Deller, "The DiFX homepage." Online `http://astronomy.swin.edu.au/~adeller/software/difx/`.

[75] "DiFX Wiki Developer Pages." Online `http://cira.ivec.org/dokuwiki/doku.php/difx/start`.