Applying Artificial Intelligence Planning to Optimise Heterogeneous Signal Processing for Surface and Dimensional Measurement Systems

ASHLEY CUSACK

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the Degree of Doctor of Philosophy

The University of Huddersfield

June 2021

Copyright Statement

Abstract

The need for in-process measurement has surpassed the processing capability of traditional computer hardware. As Industry 4.0 changes the way modern manufacturing occurs, researchers and industry are turning to hardware acceleration to increase the performance of their signal processing to allow real-time process and quality control.

This thesis reviewed Industry 4.0 and the challenges that have arisen from transitioning towards a connected smart factory. It has investigated the different hardware acceleration techniques available and the bespoke nature of software that industry and researchers are being forced towards in the pursuit of greater performance. In addition, the application of hardware acceleration within surface and dimensional instrument signal processing was researched and to what extent it is benefitting researchers. The collection of algorithms that the field are using were examined finding significant commonality across multiple instrument types, with work being repeated many times over by different people.

The first use of PDDL to optimise heterogenous signal processing within surface and dimensional measurements is proposed. Optical Signal Processing Workspace (OSPW) is presented as a self-optimising software package using GPGPU acceleration using Compute Unified Device Architecture (CUDA) for Nvidia GPUs. OSPW was designed from scratch to be easy to use with very little-to-no programming experience needed, unlike other popular systems such LabVIEW and MATLAB. It provides an intuitive and easy to navigate User Interface (UI) that allows a user to select the signal processing algorithms required, display system outputs, control actuation devices, and modify capture device properties.

OSPW automatically profiles the execution time of the signal processing algorithms selected by the user and creates and executes a fully optimised version using an AI planning language, Planning Description Domain Language (PDDL), by selecting the optimum architecture for each signal processing function.

OSPW was then evaluated against two case studies, Dispersed Reference Interferometry (DRI) and Line-Scanning Dispersed Interferometry (LSDI). These case studies demonstrated that OSPW can achieve at least 21x greater performance than an identical MATLAB implementation with a further 13% improvement found using the PDDL's heterogenous solution.

This novel approach to providing a configurable signal processing library that is self-optimising using AI planning will provide considerable performance gains to researchers and industrial engineers. With some additional development work it will save both academia and industry time and money which can be reinvested to further advance surface and dimensional instrumentation research.

Acknowledgements

The last six years have been most challenging but most rewarding of my life, my life has changed significantly since undertaking this PhD and I am incredibly grateful for the people in my life who have made me who I am today.

I would like to thank my supervisors, Haydn Martin and Jane Jiang, for their support, guidance and belief in me throughout this project, especially at times when I had little belief in myself. Also, special thanks must be given to James Williamson and all my colleagues for their help and friendship while working in the CPT.

Additional thanks must be given to Andrew Crampton, Peter Mather and Nigel Schofield for continued support and guidance throughout the latter years of this research. Also, a special thank you to Richard Baron and David Spencer for their support during the completion of this thesis.

Finally, I must thank my parents Julie and Steven and my partner Sarah for their love, unbelievable support and continued encouragement throughout my studies and everyday life, without you I could have never completed this work.

This work is dedicated to my grandma, Lilian Walker, who sadly passed during this period.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADL	Action Description Language
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ARIMA	Autoregressive intergraded moving average
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CAPP	Computer Aided Process Planning
CCD	Charge-Coupled Device
CMOS	Complementary Metal-Oxide-Semiconductor
CPG	Consumer-Packaged Goods
CPS	Cyber Physical System
CPU	Central Processing Unit
CSV	Comma Separated Values
CUDA	Compute Unified Device Architecture
CUFFT	CUDA Fast Fourier Transfer
DDL	Dynamic Link Library
DLP	Digital Light Projector
DRI	Dispersed Reference Interferometry
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FIFO	First-In, First-Out
FLOPS	Floating-point Operations Per Second
FPS	Frames Per Second
GB	Gigabytes
GCS	General Command Set
GFLOPS	Giga-Floating-point Operations Per Second (10^9)
GPGPU	General Purpose Graphics Processing Unit
GPL	General Public License
GPOS	General Purpose Operating System
GPU	Graphical Processing Unit
HDD	Hard Disk Drive
HDL	Hardware Description Language
HPC	High Performance Computing
HSFFT	High Speed Fast Fourier Transform
HTN	Hierarchical Task Network
IFFT	Inverse Fast Fourier Transform
IMS	Intelligent Manufacturing System
IIoT	Industrial Internet of Things
IoT	Internet of Things
IPC	Inter-process communication
JCUDA	Java Compute Unified Device Architecture

KPI	Key Performance Indicators
LiDAR	Light Detection and Ranging
LPG-td	Local Search for Planning Graphs Timed-initial-literals and Derived-predicates
LSDI	Line-Scanning Dispersed Interferometry
LUT	Look-Up Table
MDI	Multiple Document Interface
MIP	Mixed-Integer Programming
MLP	MultiLayer perceptron
MQTT	Message Queuing Telemetry Transport
MSVC	Microsoft Visual C++
NVCC	NVIDIA CUDA Compiler
OCT	Optical Coherence Tomography
OS	Operating System
OSS	Open-source Software
OSPW	Optical Signal Processing Workspace
PDDL	Planning Domain Definition Language
PI	Physik Instrumente
PLC	Programmable Logic Controller
PPS	Process Planning and Scheduling
PSI	Phase Shift Interferometry
R2R	Roll-to-roll
RAM	Random-Access Memory
RFID	Radio-Frequency IDentification
RTOS	Real Time Operating System
SCE	Software Cost Estimation
SDF	Standard Data Format
SDK	Software Development Kit
SLD	Super-Luminescent Diode
SM	Stream Multiprocessor
SPE	Signal Processing Engine
SSD	Solid State Drive
STRIPS	Stanford Research Institute Problem Solver
SVN	Support Vector Machine
SysML	System Modelling Language
TFLOPS	Tera-Floating-point Operations Per Second (10^12)
TI	Texas Instruments
TRL	Technology Readiness Level
UI	User Interface
UML	Unified Modelling Language
VHDL	Very (High Speed Integrated Circuits) Hardware Description Language
VRAM	Video Random Access Memory
WLI	White Light Interferometry
WSI	Wavelength Scanning Interferometry
XML	Extensible Markup Language

1 Thesis Overview

1.1 Introduction

1.1.1 Surface and Dimensional Measurement Systems for Future Manufacturing

Surface and dimensional measurement is defined as the measurement of surface texture and surface form of an object and also the physical size and distance from said object (Gao et al., 2019). A large variety of measurement instruments are available commercially, from contact methods using mechanical stylus', to non-contact instruments such as optical methods and Atomic Force Microscopy (AFM) (Whitehouse, 2011). This research concentrates on the need to accelerate the signal processing of optical instruments to realise the ever-increasing manufacturing requirements, as there is the requirement to capture and process significantly more data as the transition to smart manufacturing takes place.

In the shift from traditional manufacturing to smart manufacturing (referred to as Industry 4.0 or the fourth industrial revolution) the requirement of in-process and on-machine measurement has increased across the production line. Smart manufacturing is the data technology driven approach to a production lines that allow real-time on-machine measurement and process control to meet the changing demands of modern manufacturing. In-process metrology is the process of measuring the work piece without stopping the production line, allowing the manufacturing to continue, thus maximising efficiency. This increase in measurements is seen in newer manufacturing such as additive and nanoscale manufacturing, but also in traditional manufacturing methods such as cutting, grinding and polishing (Gao et al., 2019). Furthermore, enabling closed looped feedback from these sensors, greater process control can be achieved by feeding results back into the numerical control systems (CNC), reducing scrap rates and increasing yields. However, to achieve this, the signal processing needs to be faster than ever before.

Optical sensors are ubiquitous in surface and dimensional measurement due to their non-contact measurement and high speed. Industry 4.0 has led manufacturing to become data driven, with ever increasing data sizes, in combination with the increase in the number of sensors, data can no longer simply be stored, it must be processed in real-time with only the most crucial of data recorded once processed. The move to on-machine measurement also means that the apparatus used for processing the data needs to be compact with the maximum possible performance per square meter of production-line real-estate possible. These requirements have led researchers to use hardware acceleration techniques to increase the processing speeds of their algorithms.

For many years, signal processing on standard computer hardware with software such as MATLAB and LabVIEW has been the standard approach, but the data processing requirements for smart manufacturing are increasing due to optical sensors getting larger and faster, and traditional sequential processing is no longer providing enough performance. Researchers and industry are searching for alternatives to increase the performance of their signal processing and, for industry, increase the performance their production lines. There are three main choices instrumentation manufacturers consider when requiring an increase signal processing performance, they are: Digital Signal Processors (DSPs), Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs).

Each of the three have their advantages and disadvantages, such as performance, cost, or complexity to implement (Figure 1-1). Given the right application, and enough time to optimise for the instrumentation, each should provide an increase in performance over all-purpose software packages such as MATLAB and LabVIEW. FPGAs should provide the largest performance increase, but are expensive are time-consuming to program, but are an excellent choice for a single application that does not change over time. DSPs are a cheaper option but offer less computation than FPGAs, they are small and portable just like FPGAs but are easier to program. Just like FPGAs, DSPs are great for a single application but may allow more flexibility than FPGAs (HajiRassouliha, Taberner, Nash, & Nielsen, 2018).

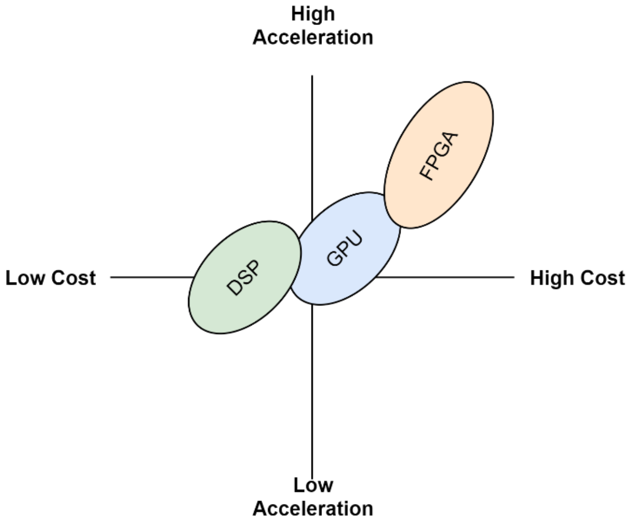


Figure 1-1: *Approximation of cost and acceleration possibilities of different architectures*

GPUs require other hardware such as a PC or server to operate, these often run on general-purpose operating systems (GPOS) such as Windows or Linux, whereas FPGAs and DSPs use their own real-time operating systems (RTOS). Therefore, the cost considered here is for the GPU itself, which can range from a couple of hundreds to a couple of thousand GBP, and within that range offer a wide range of acceleration. Although upgrading to a high specification of GPU later down the line should be much simpler of an upgrade than it would be for either the DSP or FPGA (HajiRassouliha et al., 2018). Due to their ease to program, relative low-cost and capability of acceleration, a significant amount of literature document their use of GPUs for acceleration, across the entire field of surface and dimensional sensors and instrumentation.

However, whether it is a researcher in a laboratory, or an engineer in industry, all the software to design signal processing requires significant time and financial investment to learn a programming language. Furthermore, with the requirement to use hardware acceleration becoming necessary to increase the performance of the signal processing, even more time is needed to ensure that the algorithms that are developed are efficient and optimised. This is being asked of engineers and scientists, who find themselves in this position as a biproduct of their research and do not necessarily have any background in computer science.

1.1.2 The Requirement for Optimisation in Signal Processing

Whether in a manufacturing plant or an academic research lab, time is expensive. If a software package takes twelve months to design, that is twelve months a measurement instrument is not ready to use or sell, not to mention the cost of employing a software engineer for twelve months. If the processing of a set of measurement data takes five minutes, it may be the bottleneck in a production line causing delays, or perhaps the processing is not undertaken at all causing products to potentially not meet the design specifications, all of which costs companies money. These reasons are why industries and academics frequently use commercial software packages such as MATLAB and LabVIEW even though they have their shortcomings.

MATLAB and LabVIEW have their performance issues, designing a “one size fits all” product must make trade-offs for features such as ease of use, a high level of abstraction from programming, and to provide a large library of prebuilt functions to support multiple fields. These trade-offs are at the cost of performance, and while they are becoming more efficient each new release, they still fall behind in performance compared to lower-level programming such as C. In 2010 MATLAB aimed to improve performance by supporting GPUs in their parallel toolbox, followed by LabVIEW’s GPU analysis toolkit in 2012.

However, both these options present the user with a binary choice, CPU or GPU, and while there is the option to measure the execution time and compare which is fastest for a given subset of functions, for large solutions this may be unmanageable. Also, in the case of MATLAB’s parallel toolbox, memory transfers to and from the GPU need to be configured by the user. Depending on the application, it may be advantageous to use a combination of both the CPU and GPU for a given function or set of functions. This is known as a heterogeneous computing.

The execution time (latency) of signal processing can be monitored and recorded and used as a heuristic to determine its performance. For example, two different solutions are shown in Figure 1-2, firstly, a purely CPU based execution (Figure 1-2a), where each function is called in turn and executed sequentially on the CPU and secondly a solution using the GPU for each function (Figure 1-2b). The GPU implementation may be faster or slower than the CPU solution depending on hardware specification and processing requirements. In this example, all memory transfers are considered of equal computation and a time of 5 ms is used for these.

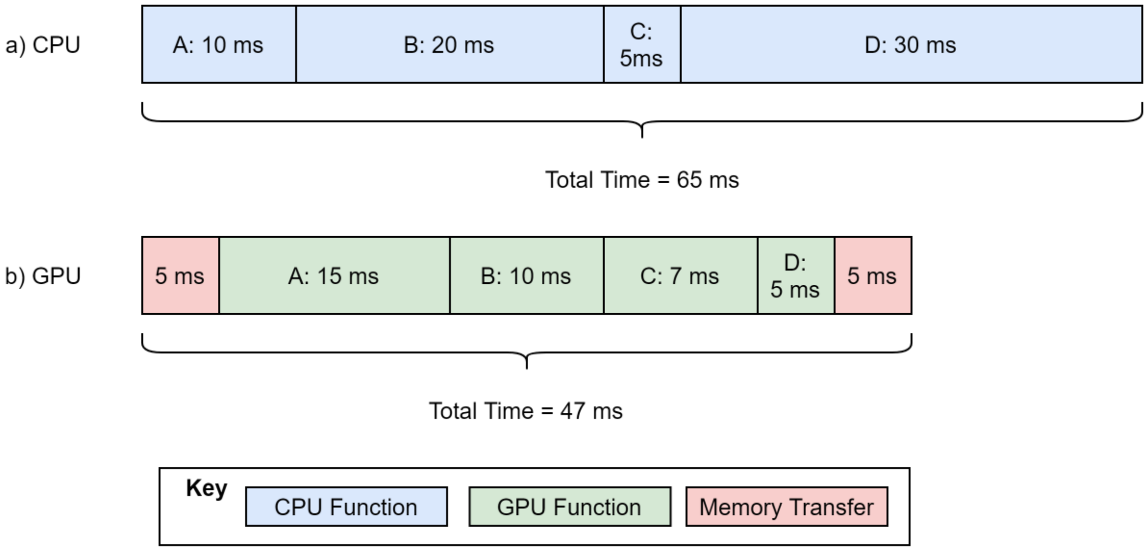


Figure 1-2: *Visual Example of Execution Times in a Heterogeneous Application*

From the example data in the figure above it an improved solution would appear to be to execute functions *A* and *C* on the CPU (15 ms vs 2 ms), but function *B* and *D* on the GPU (17 ms vs 50 ms). However, it is not as straightforward as comparing two functions and selecting the fastest. Figure 1-3a uses the fastest platform for each individual function, however, when the memory transfers are introduced, the solution is actually slower than either the individual CPU or GPU configurations from Figure 1-2.

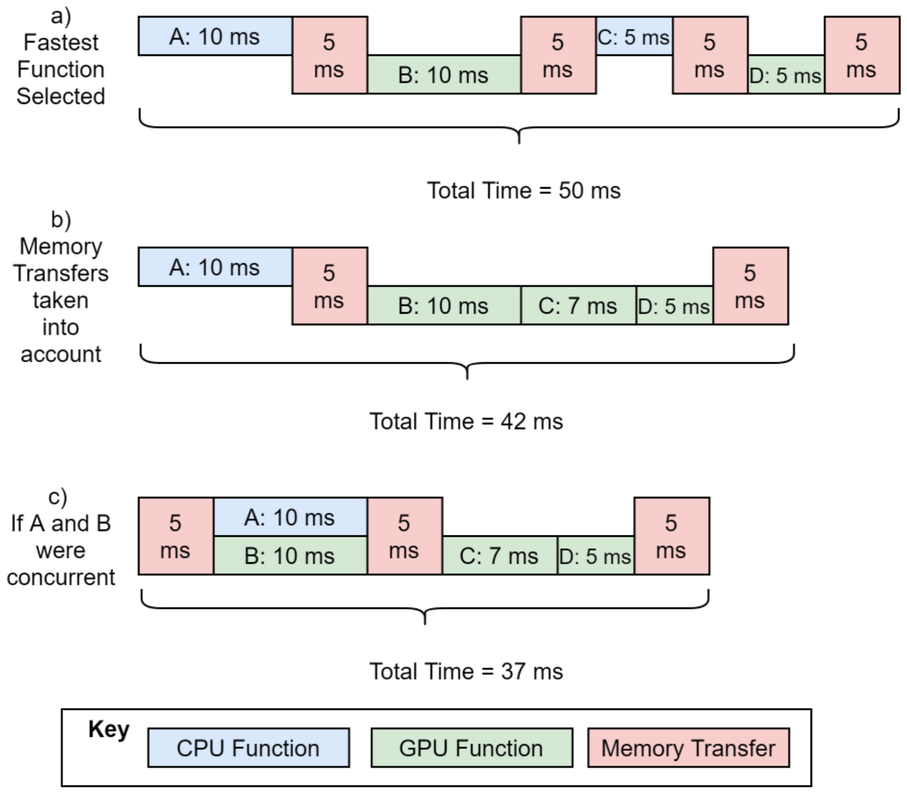


Figure 1-3: *Theoretical Optimisation of Execution Times in a Heterogeneous Application*

This is because function *C* is only marginally faster on the CPU (2 ms faster) but the two memory transfers are required to switch to the GPU and back again adds an extra 10 ms therefore providing a weaker solution. When accounting for memory transfers, the fastest possible solution ($CPU(A)$, $GPU(B)$, $GPU(C)$ and $GPU(D)$) is 35% faster than the original CPU solution and 5 ms (11% faster) than the earlier GPU solution (Figure 1-3b). Finally, Figure 1-3c shows a potential concurrency scenario where multiple functions could execute simultaneously (if they are mutually exclusive) providing an even further benefit to the overall execution time. In the example above, executing function *A* on the CPU simultaneously with function *B* on the GPU would provide a further 5 ms (12%) performance saving compared to the next best solution (Figure 1-3b)

However, as the number of functions in a signal processing sequence increase, the number of permutations across both architectures, in addition to any memory transfers required, increases exponentially creating a problem that cannot be solved by simply checking each path in turn. This problem is similar to the "travelling salesperson problem" and can be solved using the execution time as the heuristic (the parameter we want to optimise for) and artificial intelligence planning to reduce the number of permutations to consider when searching for a solution.

This reduction of permutations, referred to as branch pruning, can be achieved by placing constraints on actions (signal processing functions) excluding them from the search for a solution if their predicates are not met. Once an action has been undertaken, the effects of that action update the state of various parameters to allow the search to continue, enabling new search paths and closing off paths that do not need to be considered anymore.

1.2 Research Questions

This thesis aims to answer the following questions:

1. Why is software such as MATLAB and LabVIEW so ubiquitous in academia and industry, when its performance is suboptimal?
2. What are the reasons behind the increase in requirement for hardware acceleration in surface and dimensional measurement?
3. Is there commonality in the signal processing requirements of optical sensor measurement such that a pre-existing configurable signal processing library would save valuable time and money?
4. Can AI Planning find an optimum solution for executing signal processing on heterogenous CPU-GPU system, give an informed heuristic to minimise latency?

1.3 Aim

To provide the most optimum signal processing sequence on a heterogeneous platform for a given set of algorithms by using artificial intelligence to select the optimum execution. This will support the increase of sensors used to measure dimensions and profiles which are generating larger datasets to facilitate the increasing requirements within manufacturing from the movement to Industry 4.0.

1.4 Objectives

- Investigate the requirements of Industry 4.0 and determine the most important features that researchers and industry require for future manufacturing software.
- Examine how hardware acceleration is currently being used in surface and dimensional sensors and instrumentation to process the increasing data sizes and improving signal processing performance.
- Development of an easy-to-use software package for use by researchers and industry to advance the future of manufacturing by prototyping signal processing quickly and efficiently.
- Demonstrate the use of AI Planning to find the optimal solution for a given signal processing sequence and hardware combination across a heterogenous, CPU-GPU architecture.

1.5 Thesis Structure

This thesis is laid out as follows:

- Chapter 2 reviews how Industry 4.0 is changing the landscape of manufacturing and increase in requirement for in-process measurement. It investigates how hardware acceleration is being utilised in surface and dimensional sensors and instrumentation to processing signal processing from optical sensors and the benefits provided by DSPs, GPUs, and FPGAs. It also investigates the state-of-the-art software used to process signal processing and why there is a transition from commercial software packages to mid-level languages such as C++, CUDA and OpenCL for GPU acceleration.
- Chapter 3 introduces artificial intelligence to solve the optimisation problem. For a given set of signal processing functions, there are exponentially many possible permutations. By investigating search and planning a solution to reduce the total possible permutations to only those that provide a valid result will be determined.
- Chapter 4 provides a case study, Dispersed Reference Interferometry (DRI), to observe the performance benefits of hardware acceleration using a GPU for the single point technique. It also illustrates that developing configurable accelerated algorithms is a steep learning curve and outlines what steps need be taken to remove this programming time from an user.
- Chapter 5 brings together all the literature and development work and a software package has been created that enables users to configure a signal processing sequence for their needs. It details the creation of the software and how it provides flexibility without impacting on performance.
- Chapter 6 outlines the process of profiling, measuring the time taken to run each algorithm on a heterogeneous platform. This profiling information can then be used as a cost function do evaluate which architecture is optimum for each signal processing function.
- Chapter 7 evaluates the heterogeneous solutions found through LPG-td, a PDDL planner, and the performance benefits it provides over a single architecture. The planner should return an optimal solution for the current hardware given a sequence of signal processing and hardware availability.
- Chapter 8 presents an appraisal of the proposed software solution to optimise performance. It evaluates both the improvements that this research provides, and the limitations to the approach taken.
- Chapter 9 considers future investigations and improvements that may provide further performance benefits including changes to the PDDL heuristics, concurrent actions, and asynchronous memory transfers between architectures.

1.6 Contribution to Knowledge

The novel work reported within this thesis is summarised as follows:

- New approach to providing a library of hardware accelerated signal processing containing algorithms used across surface and dimensional sensors and instrumentation, for the future of manufacturing, that outperforms commercial alternatives.
- In contrast to other open-source alternatives, this research provides the capability to configure the signal processing library to the requirements of a user without the need to learn a programming language.
- First demonstration of the use of an AI planning language, PDDL, to optimise the signal processing and data requirements using a heterogenous approach to the available hardware.

1.7 Publications

Conference Presentation:

A. Cusack, H. Martin and X. Jiang, *Optical Signal Processing Workspace (OSPW); a self-optimised on-line system*. The Enlighten Conference: Optical Metrology for Smart Decision Making, 2019, Coventry

2 Future Manufacturing & Industry 4.0

2.1 Introduction

This chapter investigates how Industry 4.0 is impacting high-value manufacturing, specifically the changes to signal processing techniques. It explores how hardware acceleration is being applied in surface and dimensional sensors and instrumentation and the benefits that researchers and industry can achieve for their signal processing requirements. This chapter also investigates and evaluates the different hardware used for processing big data and providing signal processing acceleration through the use of FPGAs, DSPs, and GPUs. Their strengths, weaknesses, and their usage within surface and dimensional sensors and instrumentation is analysed to determine the current state of the art of hardware acceleration and how this technology is being implemented. Finally, this chapter examines Autonomic Computing, a term coined by IBM to describe the shift to systems managing themselves without human interaction.

Jiang, Scott, Whitehouse, and Blunt (2007) define surface metrology as "the science of measuring small-scale geometric features on surfaces: the topography of the surface." Whereas dimensional metrology "ensures the size of the workpiece conforms to the designer's wish" including "the measurement of length, position and radius" (Whitehouse, 2011, p. 1). To measure the surface or dimensions requires two stages, the first is the capturing of information by an input with the second stage being to analyse the data captured. Non-contact optical sensors are used across a wide range of fields from surface measurement of automotive parts to analysis of orthopaedic implants to extend their longevity in the human body. The role of surface and dimensional measurement can be thought of as "what needs to be measured in order to enable a workpiece to work according to the designer's aim – one as to measure in order to be able to control" (Whitehouse, 2011, p. 1)

Industry 4.0 relies on continuous feedback from multiple sensors, monitoring multiple parameters of associate with the workpiece, at multiple points in the manufacturing process. Vacharanukul and Mekid (2005) outline the three types of measurement used in the manufacturing which have since been updated by Gao et al. (2019). There are defined as post-processing, on-line and in-process measurement.

Post-process

Post-process or offline measurement describes the use of instruments located away from the manufacturing process. This significantly increases the time taken from an object being created and being measured increasing overall production time and cost, this method may also require a person to move the object to the measurement area and start the process. It does however allow control of parameters related to the object or machine as it isolated from the machinery, such as temperature or humidity.

On-line

On-line or on-machine measurement is defined as when the part is measured within the production line without the need of intervention. Although this means the production line does have to stop to do the measurements it still has a latency benefits over offline measurement. Measurements can be taken at several stages of the production line without constantly removing or resitting the part in manufacture.

In Process

In-process or on-machine metrology can be described as measurements taken during the manufacturing process without the need to stop production. This saves even more time again than on-line or offline processing but does come with its own challenges. This requires instrumentation to be computationally fast to take the measurement and process the result without slowing down the production line, it also needs to be robust to survive the harsh conditions it could be placed with potential debris and noise such as vibration potentially being present.

Maximum efficiency within the production line can only be achieved when the measurement takes place at the closest point to the manufacturing line, either on-line or in-process (McKeown, Wills-Moren, & Read, 1987). In recent years, in-process, on-machine processing is having an ever-increasing important role in process control, feeding information back into the machines from the measurements taken. This closed-loop feedback allows manufacturing lines to increase production yields and be notified of machining errors such as tool-ware (Takaya, 2014). As the move towards Industry 4.0 continues, the more important on-machine measurement is becoming. As technologies such as additive manufacturing increases in popularity so does the number of measurements required to ensure a product conforms to its design.

The two categories of measurement instruments used in surface and dimensional metrology are either contact or non-contact. Contact based stylus method was the original method of profiling a surface and is obtained by measuring the geometry of the stylus as it moves across the surface. It is therefore important to select the correct shape and size of stylus to ensure an accurate measurement, with these parameters being a limiting factor of the measurement. Another factor limiting stylus based measurement is its slow nature, if the surface is soft the stylus could damage the object and is also limited by the area of which it can measure (Whitehouse, 2011).

Therefore, non-contact techniques such as optical techniques and Atomic Force Microscope (AFM) aim to improve upon the existing contact-based methods. Optical methods remove the possibility of damage to the surface, measure faster and are more robust than stylus based methods (Whitehouse, 2011).

2.2 Review of the Technologies Driving Industry 4.0

The term Industry 4.0, the fourth industrial revolution, initiated in Germany in 2011 and has since attracted significant attention in recent literature (Lu, 2017). This fourth industrial revolution aims to achieve higher levels of operational efficiency, productivity, and automation to adapt to the changing nature of manufacturing (Thames & Schaefer, 2016), including other advantages such as reduced costs and provide more flexibility. Industry 4.0's aim also encompasses other technologies and paradigms including Cyber Physical Systems (CPS), Internet of Things (IOT), Cloud-based Manufacturing and Big Data (Lu, 2017).These technologies are all vital to achieve Industry 4.0's vision of a smart factory, as the number of sensors increases, from measurement instruments to environmental monitoring, "smart-technology" will be required to autonomously process the vast amounts of data that will be generated (Lasi, Fettke, Kemper, Feld, & Hoffmann, 2014).

One of these technologies, Cyber-physical systems, coined by the US National Science Foundation in the mid-2000s, builds upon earlier concepts in fields such as embedded systems and mechatronics. Since its coinage, its use has grown within the literature to over 1000 articles, and while the Internet of Things (IoT) and CPS have several similarities and overlapping principles, IoT has seen an order of magnitude more articles discussing its use compared to CPS. Although IoT did have a head-start being first discussed around 1999 at the Massachusetts Institute of Technology (MIT) (Greer, Burns, Wollman, & Griffor, 2019).

Both CPS and IoT are crucial for achieving the goal of a smart factory, the relationships between CPS, IoT, and Cloud Computing is shown in Figure 2-1. While CPS's are used at shop floor level to integrate manufacturing processes, enable machine-to-machine communication, and physical-to-digital comparisons, IoT uses sensory and smart object interconnection to connect objects, machines and people together to create the internet of things and internet of services (Pisching, Junqueira, Filho, & Miyagi, 2015).

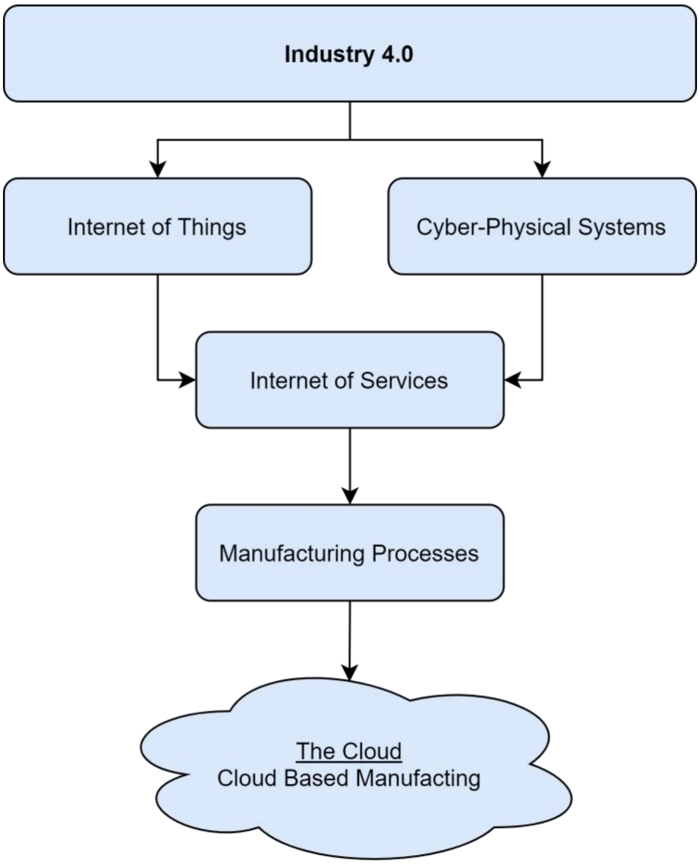


__Figure 2-1:__ The relation between CPS, IoT, and cloud computing within Industry 4.0 (Pisching et al., 2015)

2.2.1 Cyber-physical Systems (CPS)

Due to the increase in availability and reduction in cost, the number of sensors in today's manufacturing facility is greater than ever before. The competitive nature of industry has driven companies to invest in high-technology methodologies to increase efficiencies, reduce costs and become more autonomous. This increase in sensors however has resulted in the continuous generation of large data sets, requiring CPS's to use hardware acceleration and cloud computing to be able to process the data in real-time (Lee, Bagheri, & Kao, 2015; Lee, Lapira, Bagheri, & Kao, 2013).

Lee et al. (2015) outlines a five-level CPS structure for development of CPS systems, named the 5C architecture. This 5C architecture outlines the hierarchy of a CPS and the attributes within each level, reminiscent of the levels of automation for car driving. The 5C architecture starts at level one, smart connection, where multiple sensors/systems are connected and share data, at this level data is collated but no action is taken. The 5C model then increases in complexity through data-to-information, where the data captured is processed to provide smart analytics and diagnosis information about the connected machines, sometimes referred to as Intelligent Manufacturing System (IMS). Level three, Cyber, uses specific analytical tools to extract more information about the machine(s) to enable self-comparison against other machines (horizonal integration) or a digital twin.

The concept of Digital Twins, Digital Factories etc. require the mapping the technical, and sometimes the business processes into the digital world (Kádár et al., 2010). These digital counterparts then allow real-time feedback by comparing the live system to a model, these comparisons can be executed in three different modes:

1. Offline: The analysis of the uncertainties prior to the execution of the machine.
2. Proactive: Anticipating change by recognising any deviation from the model.
3. Reactive: Analysing what disturbance has occurred and how what actions can be taken to minimise losses (Monostori et al., 2016).

All these by themselves allow the system to inform a human that a deviation is about to or has occurred. However, without Cognition or Configuration, no decisions can be autonomously made on the corrective action, these are levels four and five on the 5C architecture by Lee et al. (2015). Providing feedback from the cyber space, digital twin, digital factory etc. to the physical space is the final level on the 5C model and provides the closed loop system required for the smart factory. This allows the system to self-configure, self-adapt and/or self-optimise autonomously based on decisions made by the data captured, whether mechanical or environmental.

As businesses move higher up the 5C the more sensors that are required to map the physical world and compare it against the digital mode. The increase in sensors ensures the physical world is being mapped correctly and therefore decisions can be made accurately without the need for human intervention. The Internet of Things (IoT) is advancing the mapping of physical spaces by providing connectivity between machines.

2.2.2 The Internet of Things

The term, Internet of Things (IoT), was first coined by Kevin Ashton within the context of supply chain management, using RFID in Proctor & Gamble's supply chain (Ashton, 2009). Since then, it has expanded to include applications from healthcare, to transport, and manufacturing. Although the definition of "things" has expanded in the two decades since its first use, the aim has stayed the same, for a computer to sense information autonomously without the need for human intervention (Gubbi, Buyya, Marusic, & Palaniswami, 2013). A study by I. Gartner (2015) forecasted that by 2020 there would be nearly 21 billion connected "things" worldwide, over three times the world population.

Georgakopoulos, Jayaraman, Fazia, Villari, and Ranjan (2016) states three opportunities that IoT can have on manufacturing plants. Firstly, real-time monitoring of key performance indicators (KPIs), providing personalised KPI visualisations to personnel and analysing the production process to identify improvement opportunities. Secondly, closer to the original IoT definition, smart inventory management, electronically tagging products to enable identification and tracking across the plant and beyond. Finally, automation of complex activities for quality control using specialised technologies. They also state these solutions have the added benefit of also potentially increasing plant safety by monitoring employees and equipment.

Zhang et al. (2015) used the term Industrial Internet, which later became Industrial Internet of Things (IIoT), to describe the control and decision making during manufacturing operations (Electric, 2013). IIoT is enabling unparalleled levels of productivity and performance by integrating machine-to-machine communication with industrial data analytics. As a result, industrial companies in power generation, manufacturing, utilities, construction, aviation, and a variety of other sectors are seeing significant operational and financial improvements.

Kanawaday and Sane (2018) provides an example of using IIoT on the production line of a slitting machine used to produce packaging with their aim being to predict failures before any major losses are incurred. To achieve this, they monitor the various parameters using a Programmable Logic Controller (PLC) sampled at 1 Hz pushing the data to the cloud using Message Queuing Telemetry Transport (MQTT). Although the authors do not state why this data is sent to the cloud for processing, their results using an Autoregressive intergraded moving average (ARIMA) model allowed them to predict future failure points with a 94-99% accuracy.

Strauß, Schmitz, Wöstmann, and Deuse (2019) also investigated the use of IIoT architecture and machine learning for predictive maintenance, however their specific aim concerned the “Brownfield” of old systems. They noticed that manufacturing plants are often equipped with older machines that do not contain sensors or any form of connectivity and present results from a study claiming 75% of industrial companies are not willing to heavily invest in these technologies. The authors suggest a low cost, central embedded device, BananaPi, that contains temperature and vibration sensors and communicates over a 5 GHz Wi-fi connection.

For their case study, Strauß et al. (2019) connected BananaPi with an additional sensor kit containing two acceleration sensors, three inductive current clamps, and three temperature sensors each of which monitored one of 60 identical cranes on a BMW production line. Their data was sampling at 10 Hz coming from a total of 480 sensors across all 60 cranes, averaging around 1.2 GB per hour. By using various machine learning algorithms to compare the results against one another, they found supervised algorithms such as Decision Trees and MultiLayer Perceptron (MLP). A neural networks MLP worked best to classify single faults when given historical fault data. Finally, they determined using a Support Vector Machine (SVM) gave the best live detection of faults, detecting several defective installations within the control cabinet.

A research article by Tanuska, Spendla, Kebisek, Duris, and Stremy (2021) investigated smart anomaly detection using AI. Their focus was on the predictive maintenance of automotive assembly line conveyors that transport car bodies during the entire assembly. They state the most common problem of carrier cessations is the jamming or seizing of a wheel baring and is up to the maintenance crew to visually identify the damaged carrier. To resolve this problem and identify the fault before it causes downtime, they implement a host of sensors including temperature sensors for the carrier wheel, an RFID tag for each station for identification and a microphone for sound measurement.

Their initial implementation was to implement the sensors on each of the 59 carriers across the 1.25 km production line. However due to plant constraints they implemented a single stationary measuring station that took measurements each time a carrier past by it, reducing the frequency at which they could measure from 100 frames a day on each carrier, to once every two hours per carrier. All this data was transmitted to an Industrial PC (IPC) and was analysed by a MLP neural network to detect the failure of the bearings.

Tanuska et al. (2021) managed to detect a bearing failure approximately two to four hours before critical damage just from temperature sensors alone, without their neural network. When processing with their neural network this increased to four to six hours, finally including sound data from the microphone increased this to eight to ten hours before critical damage occurred. This increased warning window gave the maintenance crew a powerful tool and allows the assembly process to avoid significant downtime which often led to financial loses.

IIoT is increasing the number of sensors on a production line, but also in the entire manufacturing plant. These sensors provide vital data to enable predictive maintenance, lower energy consumption and monitor KPIs in real time. But these sensors are creating a new problem, data processing requirements and data storage, this problem is referred to as Big Data (Al-Fuqaha, Guizani, Mohammadi, Aledhari, & Ayyash, 2015).

2.2.3 Big Data

With the push towards an interconnected manufacturing factory, through CPS and IoT technologies, data is becoming more accessible. This continuous capturing of real-time information from many sources has the negative impact of significant storage issues and is often referred to as “Big Data”. Cemernek, Gursch, and Kern (2017) described big data as “a ‘term’ describing large volumes of high velocity, complex and variable data requiring advanced techniques to enable the capture, storage, distribution, management and analysis of the information”.

This definition is based upon research set out by Laney (2001) who was the first to outline the three dimensions of big data, known as the three V's: Volume, Variety, and Velocity, however, other authors have since increased this number adding various other properties, such as IBM including Veracity and Oracle also including Value (Cemernek et al., 2017). While the number of characterising factors is up for debate, what is certain is advance data analytical techniques are required to compute the vast sizes of data captured by the ever-growing number of sensors on a production line. The processing of these data sets is often referred to as data mining.

While the size (volume) of data being captured is important, all the different properties of the data need to be considered. The Variety of data describes the different data formats that may be recorded, CSV data, databases, XML files, etc, with velocity describing how often the data is being captured. Veracity, defined by IBM as "the unreliability inherent in some sources of data", is often viewed as the property that defines the data quality (Cemernek et al., 2017). Oracle's value property is slightly more abstract, whereby the data itself has a value, a value that changes dependant on the analytics used to process the data. The more advance the data analytics are, the more economically worthy the insights and the benefits of the data captured are saving development time and eliminating possible defects prior to production (Alcácer & Cruz-Machado, 2019; Obitko, Jirkovský, & Bezdíček, 2013).

Capturing and storing the data is one challenge, the much harder challenge is processing the information acquired using useful analytical tools. Obitko and Jirkovský (2015) state "the single retrieval of [a] time series data point…is usually not an issue, on the other hand, a task such as 'what was the temperature trend during afternoons of this week'…is a more complex analytics task". In this example, the data processing needs to include pattern recognition over many different periods for things like predictive machine maintenance, or product defect predictions.

Finally all this analysis needs to be processed in real-time, a whitepaper by GE (Platforms, 2012) stated a Consumer-Packaged Goods (CPG) company captured 5000 data points every 33 milliseconds. Even at 1 byte per sample, which is highly unlikely, that CPG company would generate 4TB (terabytes) of data each year. However, this whitepaper is nearly a decade old, and the data size captured per second is likely to have significantly increased along with the number of sensors capturing data. There is no argument that this data is useful, but the analytical tools used need to gather and store the useful data and dispose of everything else otherwise huge storage devices are going to be required at great expense (J. Wang, Zhang, Shi, Duan, & Liu, 2018).

2.2.4 Summary

It is clear big data is only getting larger, and the benefits of data mining are numerous, from saving energy, increasing product yields, and shortening development time. However, as the data velocity and volume increase, traditional, CPU-bound processing is no longer fast enough, and industry is looking towards Hardware acceleration to solve its ever-growing data problem.

2.3 Investigation into Hardware Acceleration Technologies

Hardware acceleration is the use of computer hardware that has been specifically made to perform tasks more efficiently than general-purpose Central Processing Units (CPUs). Hardware acceleration is used to either decrease the latency or increase the throughput of signal processing. Other advantages can include reduced power consumption, increased parallelism, and reduction in physical size. The hardware used for acceleration ranges from basic, low-cost, single-core DSPs to highly specific, expensive to develop FPGA solutions, and GPGPU processing offering excellent price to performance. However, selection of the most suitable hardware for a given application is challenging, there are significant differences in the cost of each type, their performance capabilities and the amount of time required to implement such hardware. Therefore, a review of the different acceleration options is undertaken, the usage of these technologies over the last two decades is illustrated in Figure 2-2.

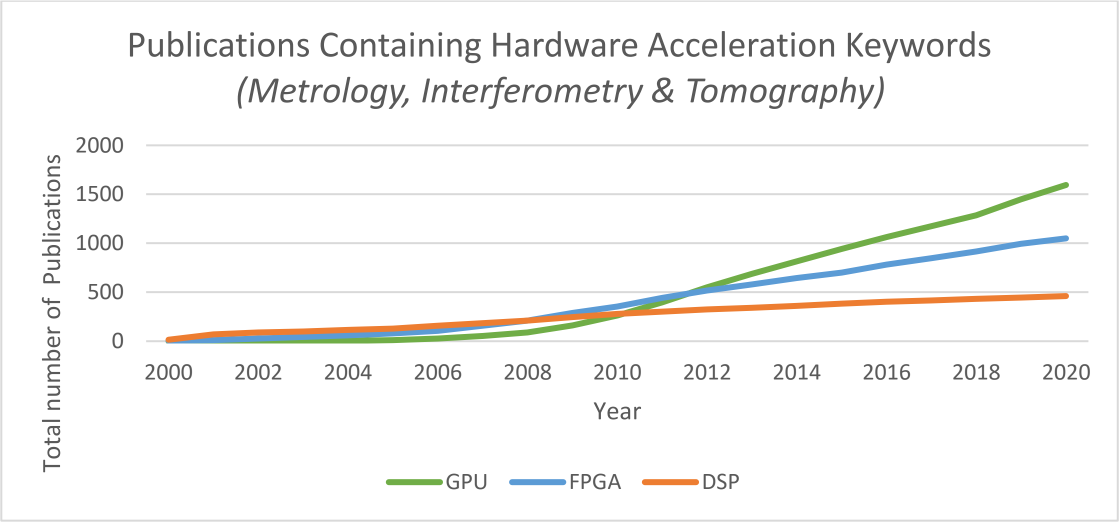


Figure 2-2: *Publications containing Hardware Acceleration Keywords*

2.3.1 Digital Signal Processor (DSPs)

Digital Signal Processors (DSPs) were developed due to the rapid development of digital computer technology in the 1960s in well-funded industries such as oil, space, and medical applications. Due to hardware limitations in the 1960s, DSPs did not run in real-time, taking longer to process the data than the data capture itself, this would change in the 1970s, and became popular in many consumer products such as aircrafts, calculators and barcode scanners (Khan, Hasnain, & Jamil, 2016). Over the next forty years, DSPs have become smaller, cheaper, and faster and are used in a significant amount of consumer devices sold today including mobile phones, televisions, and computers.

Modern DSPs, sold by companies such as Texas Instruments (TI) and Microchip, can be split into four categories:

1. Ultra-low power
2. Power-optimised
3. Digital Media Processors
4. Multi-core

Ultra-low power DSPs are cheap, but limited in their computational power, which limits these processors to simple signal processing algorithms with either small data, or few signal processing steps. Power optimised DSPs are used in devices where low power is one of the most important features, such as mobile and portable devices, but can process more complicated algorithms or more data than their ultra-low power alternatives. Digital media processors are used very specifically in video and image processing, encoding and decoding hardware codecs such as H.264 and JPEG. Finally, Multi-core DSPs are rare in consumer electronics and are normally reserved for High Performance Computing (HPC). They consist of multiple cores allowing them to compute tasks in parallel, although this does allow them to be used for much larger applications, the trade-off is their increase is cost due to the increase in core count (HajiRassouliha et al., 2018).

DSPs would not be as ubiquitous if they did not have their advantages. Their small physical size and low development cost make them attractive for simple computer vision and image processing tasks. Texas Instruments (TI), one of the main manufacturers of DSPs, offer free licenses for their software library tools used to program the chips, increasing their usage. Being low power makes them great for battery operated devices, specifically mobile devices, for their handling of communication protocols such as USB and Bluetooth. However, their processing capability is still limited, and although multicore DSPs exist, they are more suitable for sequential programming. For a more powerful, but still low-cost, option for signal processing is needed then General-Purpose computing on Graphics Processing Unit (GPGPU) might be a better solution.

2.3.2 General Purpose Graphics Processing Unit (GPGPU)

The first graphics accelerators were developed for accelerating high-quality renders of complex scenes using Open Graphics Library (OpenGL) (Montrym, Baum, Dignam, & Migdal, 1997). Although their implementation has changed significantly over the last two decades, the methodology of using graphics accelerators has stayed the same, performing fast matrix calculations in parallel. The increase in performance over the last two decades has been driven by the gaming industry, who are using continually improving hardware to render their gaming scenes in higher and higher resolutions and at greater frame rates. This drive by the gaming industry has also allowed prices to stay competitive, as this hardware is sold directly to consumers and is produced in much larger quantities therefore reducing the price (HajiRassouliha et al., 2018).

There are three major companies in the market offering GPU hardware: Nvidia, ATI Technologies, and Intel. While ATI and Nvidia have focused the mid-to-high range market, Intel focused on integrated graphics processed on their CPU packages. According to an article in 2020 by Forbes (2020), Nvidia have just under 70% market share, making them the leader in the GPU space. However, these figures include graphic cards sold for gaming for consumers, therefore, it is unclear what percentage of graphics cards used for scientific data processing are Nvidia or ATI Technologies. One finding that is clear, is the market is getting larger each year, driven by fields such as machine learning and artificial intelligence with Nvidia and their proprietary CUDA application programming interface (API) leading the way (Nvidia, 2019). Nevertheless, ATI graphics cards can still be used for GPGPU, with OpenCL, an open standard defined by the Khronos Group (Khronos, 2019), allowing CPU and ATI GPGPU heterogenous platform computing.

Both OpenCL and CUDA are free to use and both use C++11 for programming the APIs. Both provide function libraries for common image processing tasks, such as Fast Fourier Transforms (FFT) and linear algebra. Both ATI and Nvidia also offer graphics cards at many different prices and performance levels, with many having orders of magnitude more compute power and memory than the previously discussed DSPs. As of 2021, Nvidia’s current most powerful GPU can process 38.7 TFLOPS (Tera-Floating-Point Operations Per Second) with up 48 GB of Video Random Access Memory (VRAM) (Nvidia, 2021).

In two different articles examining the performance difference between OpenCL and CUDA, Karimi, Dickson, and Hamze (2010) and Fang, Varbanescu, and Sips (2011) both found CUDA to be as fast or faster than OpenCL (Table 2-1). Karimi et al. (2010) suggests that the choice can be made on factors such as “prior familiarity” or the number of “available development tools for the target hardware”. However, the results by Karimi et al. (2010) and Fang et al. (2011) appear to be application specific as Cheik Ahamed and Magoulès (2017) found through a variety of different tests that OpenCL and CUDA switch places for fastest when using small quantities of single precision (floating-point) but CUDA opened up a lead when larger data sizes were used.

Table 2-1: CUDA Performance compared to OpenCL - Green highlights the fastest architecture (Karimi et al., 2010)

Qbits	GPU Operations		Kernel Running Time		Data Transfer Time	
	CUDA	OpenCL	CUDA	OpenCL	CUDA	OpenCL
8	1.97	2.24	1.96	2.23	0.009	0.011
16	3.87	4.75	3.85	4.73	0.015	0.023
32	7.71	9.05	7.65	9.01	0.025	0.039
48	13.75	19.89	13.68	19.8	0.061	0.0086
72	26.04	42.32	25.94	42.17	0.106	0.146
96	61.32	72.29	61.1	71.99	0.215	0.294
128	101.07	113.95	100.76	113.54	0.306	0.417

However, all that performance comes at a cost. GPUs not only cost more than most DSPs, but they require more power and are significantly larger. It is not just the physical cards which are larger, but they require to be a part of a heterogenous system, taking up more space, a potential issue on a small production line. The heterogenous system is also a GPUs main bottleneck, having to transfer data between the host CPU and the GPU is often the slowest part of the program, poor optimisation leads to slow performance. GPUs are designed for massive data parallelism and their performance decreases significantly when met by either a non-optimised algorithm, or an algorithm that requires serial iteration through an array. Finally, although there are significant resources online, provided both by Nvidia and the public, programming with CUDA or OpenCL requires significant development time if the algorithms and data transfers are to be optimised.

While GPUs do offer the best price to performance, when compared against DSPs and FPGAs, they are frequently outperformed by hardware that can be specifically customised to the required task.

2.3.3 Field Programmable Gate Array (FPGA)

FPGAs are the most expensive and most specialised of the three hardware acceleration methods being reviewed. In contrast to DSPs, GPUs and CPUs, FPGAs do not have a pre-structured architecture, instead the architecture is configured by programmatically interconnecting logic gates to perform the task required. FPGA programming languages are also different from DSPs and GPUs, due to their architecture, they are programmed using Hardware Description Languages (HDLs) such as Very (High Speed Integrated Circuits) Hardware Description Language, VHDL, and Verilog. FPGAs, like DSPs, are low powered, however, they can process significantly more data, important for real-time applications where latency is critical (Bailey, 2019).

The two main suppliers of FPGAs are Altera (Intel, 2020) and Xilinx (Xilinx, 2020) who have both been in the market for almost twenty years. Unfortunately, pricing of development tools is high, much more than DSPs and GPU, furthermore, high-level functions often require expensive licenses, resulting in algorithms being developed using unoptimised basic functions. These licensing restrictions, in addition to the fact that HDL languages differ significantly from C++, not only make development difficult for beginners, but require a considerable amount of training time. However, this upfront training and development cost may not be a concern if the intention is to mass produce the chips later. FPGA code can be adapted into Application Specific Integrated Circuits (ASICs), reducing the per unit cost substantially as the majority of the cost is not the chip itself, but the time spent creating it. However, for that time and cost, very efficient, high-throughput processing can be achieved with the lowest computation to power ratio of the three methods, important for portable applications.

2.3.4 Summary

If portability or power efficiency is a design goal, without any financial or time restrictions, then FPGAs provide a very high level of performance when compared to DSPs and GPUs. If portability is still desired, but there is a requirement for low cost, then DSPs provide a respectful level of acceleration at a fraction of the cost, with a faster development time. Finally, the best value to performance is a GPU, if the processing does not need to be portable, and a high level of acceleration is required. A competitively priced GPU offers a quick development path providing the ability to periodically transfer the signal processing to new hardware much easier than the other options discussed.

Having analysed the various methods of hardware acceleration available, a sub-set of literature in the use of hardware acceleration to process surface and dimensional sensor data will be investigated to see how researchers and industry are using hardware acceleration, why they find a need to use it, and finally, which of the hardware acceleration methods are most prominent and why.

2.4 Review of Hardware Acceleration in Surface and Dimensional Measurement Processing

As Industry 4.0 advances high value manufacturing, the optical sensors on production lines are not only increasing in number, but the data sizes they are capturing are also getting larger. This is creating problems for both researchers and industry as traditional, sequential processing methods are no longer providing fast enough signal processing. Therefore, there is an increase in the use of hardware acceleration methods in improve the performance of signal processing to achieve real-time, or better performance.

Early in this research, a review into the signal processing used within optical measurements was conducted to evaluate the commonality of the algorithms used. However, that review included techniques such as Digital Holography, Adaptive Optics and Simulation which no longer relate to this research. Therefore, the literature reviewed discussing the use of hardware acceleration has been limited to only include optical sensor measurement techniques within surface and dimensional measurement or that have significant commonality of their signal processing algorithms. The techniques reviewed below are Interferometry, Structured light, and OCT to determine the extent of hardware acceleration in the processing of optical sensor data to improve process and quality control.

2.4.1 Interferometry

Interferometry is the measurement method using the interference of waves, usually light. Most interference methods start with a basic Michelson interferometer but form the basis of one of the most important tools in metrology. However, until recently, this would have been dimensional metrology, rather than surface metrology. Interferometry covers instruments from high-speed monochromatic single point techniques such as Dispersed Reference Interferometry (DRI) (Williamson, 2016) up to multi-image, three-dimensional data, for example Wavelength Scanning Interferometry (WSI) (Muhamedsalih, 2013). The different ends of this scale have very different requirements, small data at a high speed is more suited for DSPs or FPGAs, optimising for latency. Whereas, at the opposite end, high throughput of large data sets is crucial and while an FPGA may still be an option, Muhamedsalih (2013) selected a GPU for its low cost and its parallelism capability, beneficial when processing across multiple images.

Purde, Meixner, Schweizer, Zeh, and Koch (2004) were one of the first to accelerate surface profile measurement with a GPU, calculating FFTs for speckle interferometry. They chose a GPU as “neither a board design nor an extra high speed data link is necessary”, although it is not clear how much of a performance advantage they achieved, they state they gained an increase in performance over their serial program. Finally, they predicted that “modern graphics are growing with a factor of 3.0 to 3.7 every 18 months” (p. 1119) and while that figure turned out to be a little bit high when comparing operations per second (around 2.1x year-on-year from 2004 to 2018), they did correctly predict the future increases in GPU acceleration in interferometry.

Phase unwrapping is an area that has seen a great amount of attention in GPU hardware acceleration due its high parallelism possibility. Karasev, Campbell, and Richards (2007) used an Nvidia 8800 GTX programmed in C for Graphics (Cg), as this was pre-CUDA. They compared a MATLAB, CPU, and GPU implementation of a phase unwrapping algorithm for different grid sizes. They found that a GPU had an increase of 5-35x the performance of the CPU and around 7500x faster than MATLAB. Almost a decade later, Zhong, Tang, and Zhang (2015) also used a GPU (Tesla C2050) for a 2D phase unwrapping algorithm. Using more modern hardware then Karasev et al. (2007), and with the introduction of CUDA in 2007, Zhong et al achieved similar increases in performance, between 12-105x that of their Intel X5650, a powerful six-core CPU.

Schneider, Fey, Kapusi, and Machleidt (2011) also used an Intel X5650 and Tesla C2050 to calculate height data of interferograms captured from a white light interferometer. Unlike previous discussed works, Schneider et al results are mixed, with the Intel CPU outperforming the GPU by almost double in a contrast algorithm, with the Tesla GPU being the clear fastest in the sliding average and bucket method algorithms. They author states this is due to the GPU's memory bandwidth being limited, and the contrast algorithm only using "one arithmetic instruction". This bandwidth limitation was also reported by Tomczewski, Pakula, Van Erps, Thienpont, and Salbut (2013) using low-coherence interferometry and calculating interpolation, phase shifting and curve fitting. Although they did still see a speedup of 40x that of a single core, and 28x of multi-core, this bandwidth limitation is likely to be common amongst fast algorithms, with the data transfer taking as long, or substantially longer than the algorithm itself.

Finally, Muhamedsalih, Jiang, and Gao (2011) used an Nvidia GTX 280 and CUDA to process multiple 2D images from a wavelength scanning interferometer computing algorithms such as FFTs. As previously mentioned, applying a GPU's parallelism capability to a problem where repeated processes are applied to multiple images is advantageous, and this was shown by the author who achieved from a 21.5x to a 66x speedup against MATLAB, dependent on the number of frames processed at one time.

This collection of literature shows GPUs are being used across different interferometric techniques successfully and providing performance increases in orders of magnitude greater than single-core CPUs and in the case of Karasev et al. (2007), thousands of times faster than commonly used software package MATLAB. However, FPGAs are also being used within interferometry, but not as extensively.

Three different implementations by Peng, Xue, and Gao (2015), Karpiński, Khoma, Khoma, and Więcław (2017) and Scholz, Rosenberger, and Notni (2019) have all used Xilinx FPGAs to accelerate white light interferometry and achieved real-time processing speeds. However, none of them provide more than a couple of results and with no comparison to other architectures it is difficult to draw conclusions as to the benefit they have achieved by using an FPGA over a CPU, GPU, or DSP.

Two more papers that discuss FPGAs capturing interferometric data are by T. Hussain et al. (2017) and X. Li, Xiao, Zhou, Ni, and Wang (2019). As with the research listed above, results are scarce in these papers, and are written to demonstrate a proof of concept of processing with an FPGA, rather than comparing results to a different, previously used, architecture. However, both sets of authors aimed to achieve capture and process data captured at 6 MHz and 10MHz respectively and their results seem to suggest they achieved this.

Pacholik, Muller, Fengler, Machleidt, and Franke (2011) compared processing of WLI on a CPU, GPU, and FPGA, using an Intel E8600 (CPU), Nvidia 280 GTX (GPU) and Virtex 5 (FPGA). They are computing three different tasks on the different hardware, Pre-processing, Demodulation and Gaussian fitting. While some are more suited to the GPU than others, compared to the CPU they achieved speed-ups of between 5-30x over the CPU and when comparing throughput, the GPU also outperformed the FPGA, however the FPGA typically outperformed the CPU. They state due to the GPU having a processor clock 10x the speed of the FPGA, throughput was going to be higher on the GPU, however they express that FPGA's power consumption was substantially lower and would still allow real-time processing capability.

Fowers, Brown, Cooke, and Stitt (2012) also compared FPGA and GPU execution using an Intel Xeon W3520 (CPU), Nvidia 295 GTX (GPU) and Altera Stratix 3 (FPGA). Like Pacholik et al. (2011), Fowers et al. (2012) compares execution times across three different algorithms, sum of absolute differences (SAD), 2D convolution and correntropy. The FPGA outperformed both the CPU and GPU in the SAD and correntropy, with the GPU coming second, whereas the positions were reversed in the 2D convolution. Overall, the authors state that the GPU was faster for the smaller amounts of data, the FPGA gained significantly when it came to much larger data due to the FPGA having "kernel-size independent performance". It is likely that once the kernel was larger than 16x16 at higher resolutions that the GPU became saturated and had to wait for some data to finish processing before it could continue.

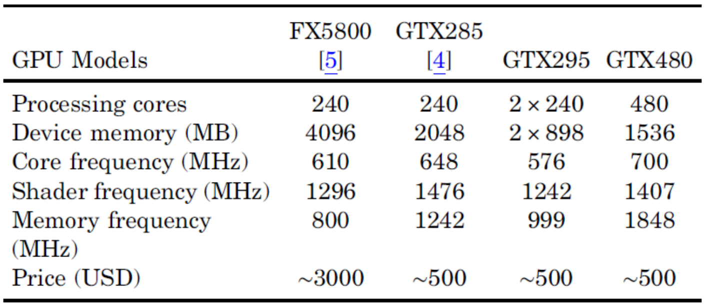
It is clear FPGAs and GPUs are being used within interferometry and achieving performance gains compared to a CPU. FPGAs do seem to have a greater performance advantage when it comes to latency, but GPUs lead the way for throughput due their significantly higher clock speed and core count.

2.4.2 Optical Coherence Tomography (OCT)

Although OCT is traditionally used for medical imaging, it has many similarities to White Light Interferometry (WLI) with regards to the signal processing. The field of OCT is therefore reviewed to not only show the similarities in signal processing with other optical sensor measurements, but also the wide-ranging impact that a generic software package for optical sensor data processing would have outside of manufacturing.

OCT has become a staple of medical imaging capturing two- and three-dimensional images at micrometre-resolution. OCT offers high resolution, cross sectional imaging in fields such as optometry and ophthalmology Yasuno et al. (2005). More recently OCT has been applied to more engineering applications such as defect detection in Additive Manufacturing (AM) and polymer composites and also in micro-electronics to evaluate PCB coating thickness. These recent uses of OCT in non-biological areas are due to OCT's superiority in axial and lateral resolutions compared to other high-frequency ultrasound imaging (Martin, Kumar, Henning, & Jiang, 2020). Due to OCT's extensive usage in medical research, there is a significant amount of literature surrounding accelerating signal processing in OCT, which overlaps with the algorithms used in engineering within interferometry.

Jian Li, Bloch, Xu, Sarunic, and Shannon (2011) used a selection of Nvidia graphics cards (Figure 2-3) to compute DC removal, FFT, and a logarithm amongst a few other steps. They achieved over 2x speedup on the GTX 480 vs the GTX 295, but it is not stated how much faster this is compared to other non-GPU solutions. The major drawback they found was the memory copy time, taking 60% of the overall execution time.



GPU Models	FX5800 [5]	GTX285 [4]	GTX295	GTX480
Processing cores	240	240	2 × 240	480
Device memory (MB)	4096	2048	2 × 898	1536
Core frequency (MHz)	610	648	576	700
Shader frequency (MHz)	1296	1476	1242	1407
Memory frequency (MHz)	800	1242	999	1848
Price (USD)	~3000	~500	~500	~500

Figure 2-3: *Selection of graphics cards used. Note. Reprinted from "Performance and scalability of Fourier domain optical coherence tomography acceleration using graphics processing units" by Li, J, et al., Applied Optics, 50, p. 1832.*

Rasakanthan, Sugden, and Tomlins (2011) also found their data transfers took a significant portion of time, 38% in their implementation of OCT processing. They did, however, minimise the data transfer overhead by batching data together and transferring multiple frames at once. Using a standard technique of processing one frame at a time they achieved a frame rate of 524205 lines/s, but when they batched more than four frames together this increased to 724314 lines/s, a 38% improvement. They also do not state what their previous implementation architecture was before hardware acceleration, or compare it to their new results, but it can be assumed their new GPU accelerated method is faster.

L. Wang, Hofer, Guggenheim, and Považay (2012) have taken an interesting and unique method of programming their Nvidia GTX 580. Rather than re-programming their signal processing in OpenCL or CUDA, including background subtraction, up-sampling, and linearisation, they used a now-discontinued commercial software package called Jacket to convert their MATLAB code to CUDA C++ code. This code is then available via a Dynamic Link Library (DLL) file and can be used either by code packages such as Microsoft Visual Studio, or LabVIEW. They authors used the DLL with LabVIEW to achieve a 90x speedup over their MATLAB code, although due to the number of steps taken to achieve this, inefficiencies may have been introduced into the process, a true CUDA implementation would likely have yielded much better results.

Another paper that compares CUDA execution to MATLAB results is by Xu, Huang, and Kang (2014b). They aimed to achieve real time image construction of 2048 x 1000 images using a combination of a Nvidia GTX 670 and two Tesla C2075 GPUs. As with the other papers discussed, they are using similar signal processing algorithms to other researchers, DC removal, FFT, logarithms, and array padding. Xu et al. manage to achieve a 112x speedup compared to their C++ implementation and a 459x speedup compared against MATLAB achieving their real-time goal. This was published over six years ago, at time of writing, and such processing power is now likely to be available on a single GPU, although their processing requirements or image size may have increased in the elapsed time since this was published.

Others notable papers using GPUs in OCT are by Y. Wang et al. (2012) who managed to achieve real-time processing with an Nvidia Tesla C1060 using algorithms such as interpolation, FFT, and mean. Watanabe, Maeno, Aoshima, Hasegawa, and Koseki (2010) also achieved real-time processing of FFT, interpolation and padding using a CUDA and an Nvidia GTX 295. Xu, Huang, and Kang (2014a) managed to achieve a 5x speedup over their CPU using an Nvidia GTX 580, processing convolutions. Finally, Trojanowski, Kraszewski, Strakowski, and Pluciński (2014) achieved a 12x speedup using a Nvidia GTX 590 to accelerate their processing of FFTs.

FPGAs are also used in OCT to accelerate the signal processing, Ralston, Mayen, Marks, and Boppart (2004) were one of the first to implement this hardware for OCT. They used a Virtex II FPGA to accelerate several low pass filter algorithms achieving real-time processing compared to the 10s it was taking on their unspecified host computer. Desjardins et al. (2009) also used a Virtex II to accelerate their signal processing, consisting of background removal, FFT, and interpolation. They state "The reconstruction speeds achieved with the FPGAs utilized in this study were comparable to those that could be achieved with optimized software running on current desktop CPUs" (p. 1471) and go on to express the potential of FGPA acceleration in OCT for real-time applications.

J. Li, Sarunic, and Shannon (2011) discuss their use of FPGAs and why they chose them over a GPU for their hardware acceleration. This paper is intriguing as the authors discuss their implementation of acceleration using a GPU, which was reviewed earlier. Since their GPU acceleration was 6.9x that of a CPU, they have implemented the same algorithms; DC removal, FFT, and resampling, on three Virtex 5 FPGAs chained together achieved processing speed of 465 MB/s. Although this is lower than their GPU processing rate of 527 MB/s, that did not include data transfer, which when included reduced the overall speed to 207MB/s. However, even though their increase in performance on the FPGA is now over 15x that of their CPU, it is not without its drawbacks, for example, the FPGA only supported 16-bit floating point values, significantly decreasing the quality of the data representation. Importantly, they stated that It took around three months to develop the original GPU code, not an insignificant amount of time, however they estimate the FPGA development time was around four times longer.

2.4.3 Structured Light

Structured Light uses a projection of a known pattern, this can be in the form of horizontal bars or grids, onto an object surface. One or multiple cameras then capture the pattern projected onto the surface and measure how the pattern has been deformed. From that the surface and depth information of the object can be calculated, providing three-dimensional shape data (Geng, 2011).

Karpinsky, Hoke, Chen, and Zhang (2014) developed a 3-D shape measurement system that can capture and reconstruct images at 30 FPS. Karpinsky et al. wanted to develop their processing for laptop computing and therefore chose to use a Nvidia NVS 5400M. Although they do not mention why they chose laptop computing, it did achieve their target of 30 FPS for an 800x600 imaging acquisition. They outline a two-frequency phase-shifting algorithm, two filtering operations and a phase unwrapping algorithm all of which are processed on the GPU.

Hao, Takeshi, and Idaku (2012) built a real-time, structured light 3D scanner capturing multiple coded light patterns that are projected at 1000 frames per second from a Digital Light Projector (DLP). The images are captured through a high-speed camera and passed to a Nvidia Tesla C1060 for processing. They implemented several algorithms including binarization, filtering and triangulation, all which are specified mathematically for future implementation. They conclude that using the GPU, they can output 3D videos of 512x512 pixels at 500 frames per second.

While there are only a few articles for GPU usage in structured light there are large benefits to be had. The majority of CPUs will not be able to process this level of data (>10,000,000 points/s) for real-time processing, this signifies most of structured light data must be being processed offline. By using GPUs this would not online bring these methods online but allow bigger CCD arrays and therefore improve the resolution of the data captured. Karpinsky et al. proved that with a middle-class laptop GPU 30 FPS was possible, for that reason using a up to date GPU like the Nvidia RTX A6000, with over 150x the compute performance, much larger frames could be processed at increased speeds (Nvidia, 2021).

2.4.4 Summary

Interferometry and structured light have seen a significant amount of literature published by researchers using GPU, DSP, and FPGA acceleration to improve the amount of data that can be processed and reduce the time it takes. With other techniques such as OCT also producing significant literature surround the performance benefits through GPU acceleration, this benefit reaches far beyond optical sensors and surface and dimensional measurement. Many of the articles are using the same subset of algorithms, DC removal, background removal, FFT, logarithms, etc. however, each are spending considerable time developing their own algorithms separately.

Given the commonality in signal processing algorithms between interferometry, OCT and structured light, it is likely that other optical measurement techniques in other areas not considered in this research has overlapping requirements. This possible commonality across multiple fields extends the number of academics and engineers having to spend significant time learning a new programming language or languages to develop their signal processing. As J. Li et al. (2011) stated this process could take around a year. This is not just a problem faced in research labs, whereby projects have time restraints, in industry time is money, and companies cannot wait 12-18 months for a software package, they need something they can use straightaway with little to no modification.

The proposed solution to this, is to have a software package, that contains all the signal processing algorithms that are being used for processing optical measurement data, and that provides hardware acceleration in the form of GPGPU processing. This acceleration comes at a relatively low cost, and can provide significant performance benefits over traditional sequential processing. The next area this thesis will review is evaluating the software requirements industry and academics have, the costs associated with software development, and the currently available packages that they are using.

2.5 Analysis of Software Availability and Usage

From the literature reviewed previously, it is evident that hardware acceleration is being used to increase the performance of signal processing within surface and dimensional sensors and instrumentation. Due to their widespread use by the gaming industry, GPUs provide an excellent price to performance level when compared to DSPs and FPGAs and deliver excellent performance of signal processing that is suited to parallelism. However, in most instances, the research discussing hardware acceleration has moved away from established software packages such as MATLAB and LabVIEW and instead, developers have spent significant time programming their own implementations of algorithms in C++, CUDA, or OpenCL. This section investigates the different requirements researchers and industries have for software and what solutions, free or commercial, exist and what benefits or drawbacks there are between developing bespoke software, using commercial products, or harnessing open-source development.

2.5.1 Software Requirements

Industry and academics are often working at different Technology Readiness Levels (TRL), however their software requirements that they have are very familiar. They are often both developing instrumentation and software on an incremental basis and require software that can easily be adapted. The software would still benefit from being as efficient as possible, but another important feature is likely to be configurability. Therefore, any potential solution that provides a signal processing library with hardware acceleration is likely to benefit academics and industrial engineers alike.

Having reviewed the different hardware acceleration methods previously, the use of GPGPU processing offers excellent price to performance levels. That is not to say other methods such as FPGAs are not used, if very high-speed measurement is required, FPGAs may be the only option available, however for most applications GPUs will provide plenty of performance, and certainly more than commercial offering can provide.

However, Small and Median-sized Enterprises (SMEs) made up 99% of private sector businesses in the UK in 2020, with 1.3 million businesses having between 1 and 49 employees (Department for Business, 2020). Therefore, it is vital for these small companies to assess their software requirements and possible solutions carefully, they can do this using a software cost estimation (SCE) model.

2.5.2 Software Cost Estimation

A great deal of literature has been published in the field of estimating software development costs and development time, along with comprehensive reviews of the literature itself. This field of Software Cost Estimation (SCE) is well established and was first discussed in the 1970s by Wolverton (1974) with several hundred of papers published each year attaining to some aspect of estimating the cost of software development. During this time, several models have been proposed that allow companies or researchers wanting to develop software to estimate the resources required such as the Putman model (Putnam, 1978), the PRICE S model (Freiman & Park, 1979), and COCOMO (Boehm, 1984). Each of these models have their benefits and drawbacks, but each provide an estimation to how many "man-months" software development should take given a set of factors required for the software.

Goodcore, a software development company from the UK, compared estimating software costs to asking: "I want to build a house, how much will it cost?". The answer is it depends on many factors, hence why there are so many competing models and significant literature published in the field (Goodcore, 2020). These factors include, but not limited to (Boehm, 1984):

- The number of developers working on the project.
- If there is a time constraint on the development.
- How many progress deliverables are required.
- The access to the hardware to be integrated with the software.
- The complexity of software required.

Each model places different weightings on each of these factors, which ultimately estimates the total man-months required for development. Having previously discussed the almost necessary requirement of hardware acceleration in present day data processing software, this would have a significant impact on development cost and time, requiring higher-skilled developers and an increase in development and testing.

However, without this planning, costs could spiral and without proper project management, the development could deviate from its development schedule, go over budget, or not encompass the requirements of the original software proposal. Sun, Zhou, and Wolf (2001) estimated that 88% of a software development project is personnel costs, therefore it is crucial to plan how much time and money the development of software will cost. According to PayScale, a compensation and data company, the average software developer salary in the UK is around £38,000 a year (PayScale, 2021). Furthermore, costs do not just stem from the software development itself, there are costs associated with creating documentation for users to be able to interact with the software, also there will be potential rework costs and ongoing maintenance. Software is unlikely to be perfect first time and will require iterations to fix any irregularities within the software and may require extra features later down the line (Boehm & Papaccio, 1988).

These potential costs and long lead times often leaves researchers and industry looking towards off-the-shelf commercial software packages that have high levels of abstraction from programming languages to allow them to quickly pull together some signal processing without needing to know lower-level languages like C and Java. These packages will now be evaluated to discuss their benefits and limitations.

2.5.3 Review of Commercial Software

There are several mathematical commercial software packages in the market designed to make creating signal processing easy, without having to learn a low-level programming language. MATLAB and LabVIEW are two of the most popular, but they have taken very different approaches. While

LabVIEW offers graphical programming, with no computer code in sight, whereas MATLAB does require users to use programming script, but is at a very high level. A rough guide to programming language levels can be found below (Figure 2-4). Therefore, if MATLAB and LabVIEW are so easy to use, why are people transitioning to programming language such as C++, CUDA and OpenCL.

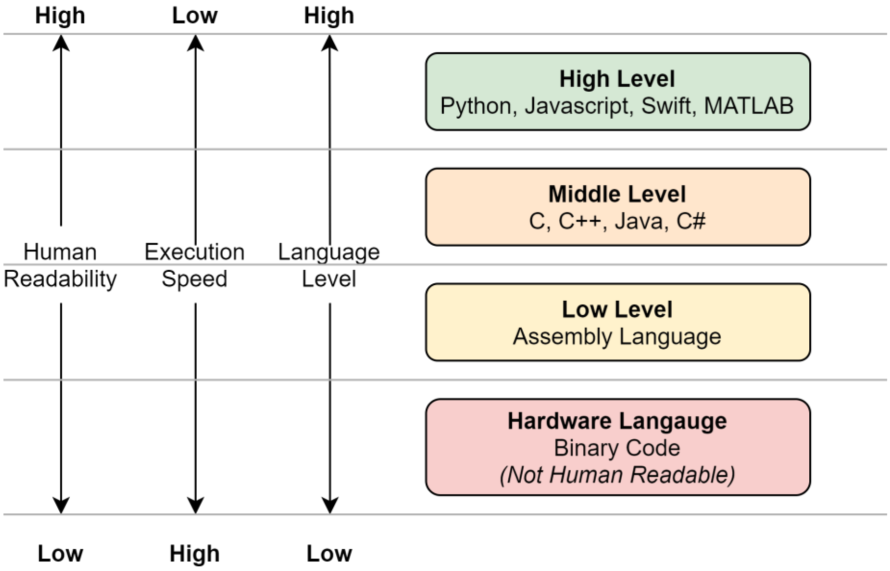


Figure 2-4: *Programming Languages levels and their properties*

2.5.3.1 MATLAB

MATLAB has branded itself as “the easiest and most productive computing environment for engineers and scientists” MATLAB (2020). Whether that statement is true is open to interpretation and a user’s personal preference, however the reasons why they state this are substantiated. They provide an enormous number of functions that are easy to access, rigorously tested, and have documentation written “for engineers … and not computer scientists” MATLAB (2020). These are all beneficial to the researcher or developer needing develop their signal processing working relatively quickly, without spending six to twelve months to a year learning C. Compared to a software developers salary, and the time it would take to develop a bespoke software package, MATLAB’s £2000 license fees is excellent value for money, especially considering the enormous amount of documentation it has, not to mention the community support is has from such wide usage.

Even though MATLAB has an extensive library, helpful documentation, is relatively low cost, and provides parallel computing support, researchers are still shifting to languages like C and CUDA as in the pursuit of faster processing as MATLAB’s flexibility comes at a large performance disadvantage. Many of the literature reviewed in the previous chapter discussed their performance increase moving from MATLAB to C, and even greater from MATLAB to CUDA (Figure 2-5). It should be noted, the graph is a logarithmic plot and that each gridline represents a 10x time increase. Karasev et al. (2007)

illustrates this performance issue showing the differences in execution time between the three methods, they state “reasonable care was taken to optimize all three versions” (p. 577). However, they also state “extensive optimizations that substantially alter the nature of the source code … were not undertaken” (p. 577). This indicates that a more optimised implementation of their CPU and GPU code would have yielded even greater results than they already achieved.

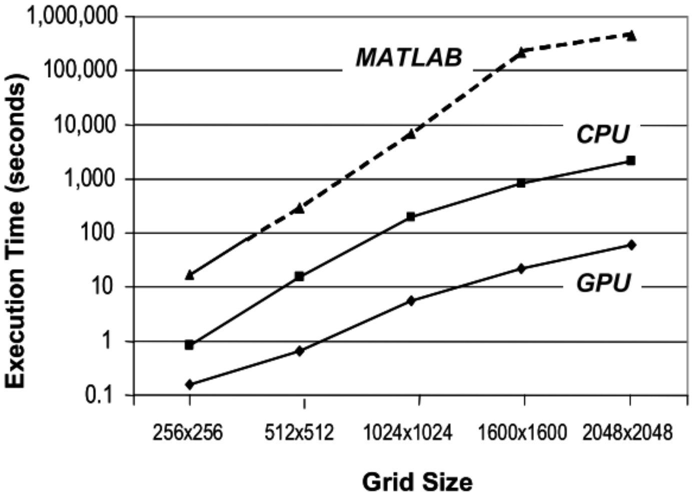


Figure 2-5: Note. Reprinted from "Obtaining a 35x Speedup in 2D Phase Unwrapping Using Commodity Graphics Processors" by Karasev, P.A. et al., 2007, Radar Conference, p. 577

It is this reason that several researchers and companies have developed MATLAB-to-C++ translator tools, including MATLAB themselves. This allows a developer to create their signal processing in MATLAB with all its benefits, and when they finalised their algorithms, they can convert everything to C++. One of these translation tools that has been previously mentioned is Jacket. Jacket (Pryor et al., 2011) by AccelerEyes allowed users to take their MATLAB code and port it to CUDA using an intermediary custom compiler. Jacket contained a large library of GPU-tuned function syntax for MATLAB ensuring that the code that was ported across was optimised as much as possible. While their results showed a significant performance increase above MATLAB, they did not compare Jacket to a pure CUDA implementation, they also used double precision variables, a type not particularly optimised for on a GPU. Unfortunately, it seems MATLAB filed a patent infringement against Jacket and the software was removed from sale.

Due to the lawsuit, and lack of availability of Jacket it is unclear how the translation state between MATLAB and C++ technically worked, however another CPU software API that is currently available is Armadillo (Sanderson, 2010). Armadillo is a C++ library that contains 44,000 lines of code incorporating many of the linear algebra functions that MATLAB also provides. Each of these functions is in a pair, allowing simple conversion between MATLAB and C++, however variable translation is not as straightforward. As C++ is a static language and must be compiled, compared to MATLAB's interpreted language, variables must be explicitly declared at creation, with the type and size being fixed. Nevertheless, Sanderson (2010) reports performance results up to 14.7x greater than MATLAB when using Armadillo rather than MATLAB.

Based upon Armadillo, Paulsen, Feinberg, Cai, Nordmoen, and Dahle (2016) created Matlab2cpp, an intermediary translation tool to parse the input before it is translated to C++ by Armadillo. The authors hope that this parser will increase the efficiency of the translated functions that Armadillo provides. However, in their results they only compare vanilla MATLAB to their C++ results, and not the differences between Matlab2cpp and Armadillo.

Armadillo's limit however is it is CPU bound, and while MATLAB does support GPU execution natively, it is with a smaller subset of algorithms and is still speed restricted due to its high-level nature and while there are other third-party software out there that make programming GPU code from scratch easier, it seems a tool to efficiently translate MATLAB code to CUDA is not available.

2.5.3.2 LabVIEW

LabVIEW was developed as a visual programming language in the 1980s by National Instruments. Using its propriety graphical 'language' called G it has been used by engineers and scientists for decades for testing, measurement, and control applications (LabVIEW, 2019). From the literature reviewed in the previous chapter, only a few papers discussed their use of LabVIEW. National Instruments describe G as "an extremely high-level programming language who purpose to increase the productivity of its users while executing at nearly the same speeds as lower-level languages like FORTRAN, C, and C++".

Although LabVIEW provides a modern looking interface, and able to interact with a large number of actuation tools and capture devices, the graphical programming can start to become complicated. Figure 2-6 shows a LabVIEW example for some basic signal processing (application not specified), and already the number of blocks and 'wires' are starting to look unmanageable. National Instruments, the creator of LabVIEW states that it "executes at nearly the same speeds as lower-level languages", therefore there should be plenty of research using LabVIEW. Instead is a significant body of researchers detailing why they moved away from it.

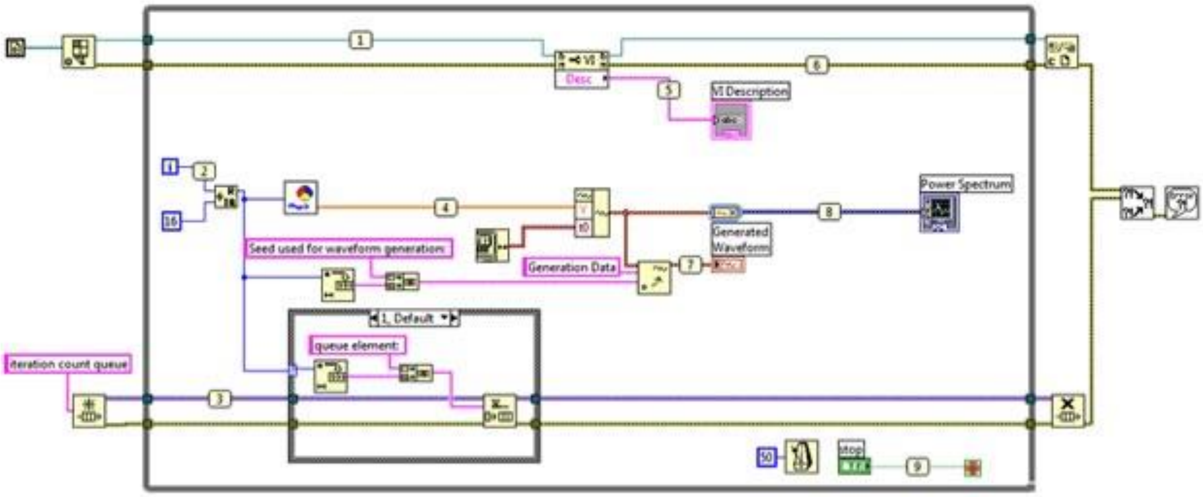


Figure 2-6: *Note. Reprinted from "Benefits of Programming Graphically in NI LabVIEW", by LabVIEW, (2019,03,05). Retrieved from https://www.ni.com/en-gb/innovations/white-papers/13/benefits-of-programming-graphically-in-ni-labview.html*

Tašner, Lovrec, Tašner, and Edler (2012) are one of the few who have moved compared LabVIEW's performance against another piece software, they compared LabVIEW against MATLAB. They compared the two software packages on the same PC (Intel i7-2600) across four different tests, matrix calculation, FFT, bode plot, and a DC motor simulation. Their results are shown in Table 2-2. Half of their tests, FFT and bode plot were faster in LabVIEW than MATLAB, by about 50% each time, with the other tests taking around twice as long. However, even LabVIEW did outperform MATLAB, as previously discussed, the bar was not set very high, and does not yield the results that National Instruments claim. It is unfortunate that these researchers did not compare these results to a third implementation, like C++, but it would be expected from how MATLAB performed, that C++ would also outperform LabVIEW.

Table 2-2: *Note. Adapted from "Comparison of LabVIEW and MATLAB for Scientific Research" by Tašner, T,et al.,2012, Annals of the Faculty of Engineering Hunedoara-International Journal of Engineering, 10, p. 389*

Calculation	MATLAB (s)	LabVIEW (s)
Matrix Calculation	6.23	15.618
FFT	0.289	0.162
Bode Plot	0.188	0.09
DC Motor Simulation	0.215	0.309

Cansalar, Mavis, and Kasnakoglu (2015) also investigated the performances of both MATLAB and LabVIEW. Using an Intel Xeon W3550 running Windows 7, they simulated a simple DC motor controller in MATLAB's Simulink and LabVIEW using a P-controller, PI-controller, and PID-controller. It is clear from the results shown in Table 2-3 that in five out of six simulations, MATLAB outperformed LabVIEW, with LabVIEW only being faster in the smallest simulation. Additionally, as the controller complexity increased, the execution time also increased, and interestingly so did the performance increase that MATLAB provided over LabVIEW (Table 2-4). These results once again demonstrate MATLAB performs faster on average than LabVIEW. For LabVIEW to be outperformed by MATLAB is a tough blow when, as previously discussed, MATLAB itself can be orders of magnitude slower than C implementations.

Table 2-3: *MATLAB vs LabVIEW Results from (Cansalar et al., 2015). Green indicates the fastest software.*

Sim Time (s)	Step Size (s)	Number of Loops	P-Controller		PI-Controller		PID-Controller	
			MATLAB (µs)	LabVIEW (µs)	MATLAB (µs)	LabVIEW (µs)	MATLAB (µs)	LabVIEW (µs)
2	0.001	2000	3940	11600	5011	33303	5692	38885
2	0.01	200	932	1206	1106	3322	1167	4106
2	0.1	20	419	160	694	379	713	449
60	0.001	60000	91262	344453	119900	984330	140620	1176089
60	0.01	6000	9781	34777	12849	99261	14503	116675
60	0.1	600	1561	3616	1859	9895	2058	11986

Table 2-4: *MATLAB vs LabVIEW - MATLAB Speedup – Yellow indicates a performance deficit.*

Sim Time (s)	Step Size (s)	Number of Loops	MATLAB Speedup compared to LabVIEW		
			P-Controller	PI-Controller	PID Controller
2	0.001	2000	2.94	6.65	6.83
2	0.01	200	1.29	3.00	3.52
2	0.1	20	0.38	0.55	0.63
60	0.001	60000	3.77	8.21	8.36
60	0.01	6000	3.56	7.73	8.04
60	0.1	600	2.32	5.32	5.82

It is clear from the results discussed above that although MATLAB and LabVIEW provide very easy to use interfaces allowing programming at a very high level, that it is also their greatest weakness. Their performance figures are trounced by lower-level implementations on the CPU and GPU. If industries or academic institutions cannot invest the money or do not have the time to create their own software, even though commercial software might be easy to use and has a relatively low cost, it often does not provide enough performance. Therefore, the last option available is to evaluate Open-source Software (OSS) options to determine if packages exist that support their requirements.

2.5.4 Survey into the Usage Open-Source Software

While originally in the 1990s there were many critics of open-source out there, such as (Lewis, 1999) who stated “I doubt that open-source will long influence how software is built” in the 20 years since that article was written, open-source is now larger than ever. For example, according Gartner (2021) the an open-source Linux kernel, Android, is used on 86% of all mobile phones. Open-source is now synonymous in everyday life through mobile phones and is used across industry either through the reuse of components in commercial software or the entire software system (Ajila & Wu, 2007).

In 2006, only seven years since Lewis aired his views on OSS, Fitzgerald (2006) coined the term OSS 2.0 having seen radical increase in its use, stating a “Moore’s Law effect seems to be taking place”. While the number of publications may not have continued to double, there has been a significant amount of literature published in the last 20 years regarding open-source software (Figure 2-7). This change from its first iteration to OSS 2.0 was brought on by the introduction of the *Open-source Definition*, removing some of the restrictions developers faced when bundling software together (Lerner & Triole, 2000).

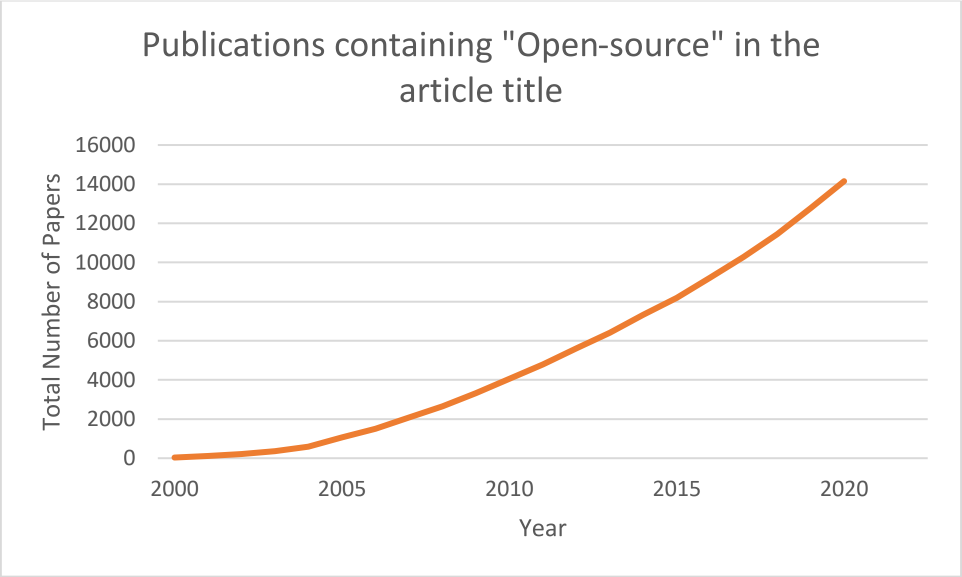
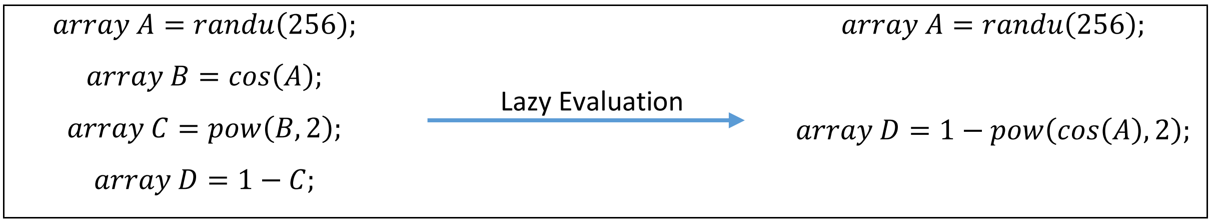


Figure 2-7: *Scopus data showing the total number of publications to contain the phrase "Open-source" in the title since from 2000 to 2020.*

Ajila and Wu (2007) hypothesised and proved there was a strong correlation between the degree of OSS adoption and software product quality, and between the degree of OSS adoption and cost of software development. Therefore, companies and researchers wanting to develop software may be able to turn to open-source alternatives, rather than developing their own, therefore some of the currently available OSS options will be analysed to evaluate what currently exists, and what any new software solution needs to consider having a large impact in optical sensor development.

2.5.4.1 ArrayFire

ArrayFire (Malcolm et al., 2012), created by AccelerEyes, the company behind the legally blocked Jacket software, is a GPU library for C++ and Python. ArrayFire provides high-level abstraction from the GPU through APIs such as CUDA and OpenCL, allowing engineers and scientists to take full advantage of GPU hardware. One of ArrayFire's most useful features is its single matrix object (array). Rather than requiring a defined type for data, ArrayFire analyses the data in the object at compile time and defines the type automatically. Another clever feature that ArrayFire presents is "Lazy Evaluation". Lazy Evaluation monitors the code the user has developed and analyses it in real-time to increase the performance of the formulas. Figure 2-8 shows an example of a user taking four steps to create array *D*, using ArrayFire's patented Lazy Evaluation this is reduced to two steps, reducing the time performing multiple kernel executions and the entire computation is batched together. This technique also reduces memory requirements, before lazy evaluation, the code required four arrays, 256 values wide, whereas the modified kernels only require two arrays, a 50% memory saving.



$array\ A = randu(256);$

$array\ B = cos(A);$

$array\ C = pow(B, 2);$

$array\ D = 1 - C;$

Lazy Evaluation

$array\ A = randu(256);$

$array\ D = 1 - pow(cos(A), 2);$

Figure 2-8: *Lazy Evaluation example*

ArrayFire also contains libraries for displaying the data computed, using OpenGL. By utilising OpenGL, there is no need to copy data from the GPU to the CPU before displaying, OpenGL can render the images or graphs straight from the GPU memory using the GPU performance, further increasing performance. ArrayFire presents impressive performance figures (Malcolm et al., 2012) compared to Intel's Math Kernel Library (MKL) and previously discussed Armadillo (Figure 2-9). However, the drawback of ArrayFire is it does still require learning a programming language, albeit a high-level language that has significant performance advantages over other offerings and has many features to make the process as easy as possible.

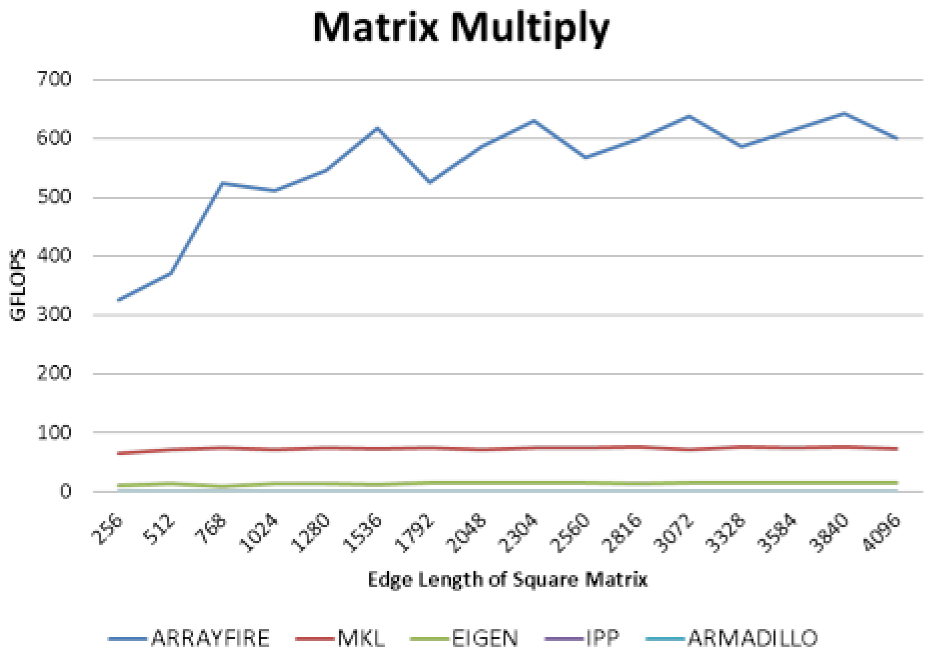


Figure 2-9: *ArrayFire performance figures. Note. Reprinted from "Why ArrayFire?", by ArrayFire. Retrieved from https://arrayfire.com/why-arrayfire/*

2.5.4.2 itom

itom (Gronle, Lyda, Wilke, Kohler, & Osten, 2014) allows researchers to quickly protype software when using various different hardware devices (motors, camera, ADCs etc.). Sensibly, they have used different existing packages where possible, to provide the user with the best experience they could. Using C++ for the software, Qt for the user interface and Python for the programming logic. However, this still does leave the user knowing and understanding Python programming language to be able to use itom, something not too dissimilar than MATLAB's scripting language.

To evaluate itom's performance, it was tested against MATLAB and LabVIEW in three tests, an FFT, a linear filter and a nonlinear filter. It should be stated that itom was created to make prototyping easy, not to provide the fastest implementation, however, in almost every test, at every different kernel size, it was outperformed by both MATLAB and LABVIEW. Gronle et al. (2014) do state that "loop operations should be avoided in script languages, but they are much faster in MATLAB due to the internal just-in-time compiler" (p. 2978), MATLAB was created for matrix calculations after all. However, this does not explain the significant performance deficit that itom had. It could be speculated that all the different layers, Python, C++, and Qt, are having a negative impact on the performance available from the PC.

2.5.4.3 Summary

Open-source software is not without its drawbacks, Morgan and Finnegan (2007) conducted a field study amongst 13 companies such as Nokia, Siemens, and Sony. They found that open-source software often has compatibility issues, where it has been designed with one system, architecture or configuration in mind, and is not compatible with a user's setup. Secondly, they found a lack of expertise exist when implementing the open-source software. Often, people are turning to open-source software as they do not have the expertise to develop their own, therefore they often do not have a team of technicians to get the software working as required.

Secondly, the study also found there were business drawbacks of open-source, some of which pose greater challenges than their technical counterparts. For example, the lack of support of the software, perhaps when first using the software, or after the member of staff that implemented has left the company. The lack of ownership of open-source software also raises similar issues. Finally, it found a significant proportion of open-source software is discovered by word of mouth. As open-source software has no marketing budget, it is hard to get word out of the software that exists.

Even though both ArrayFire and itom both offer advantages, they both fall foul of some of the disadvantages associated with open-source software mentioned above. These have been summarised in Table 2-5. Both at some point fail in terms of both technical and business requirements and may suggest why there isn't a significant body of research discussing the use of them within applications.

Table 2-5: *Evaluation of Drawbacks of Open-source Software*

Technical Drawbacks	ArrayFire	itom
Good Compatibility	Yes	Not Evaluated
Easy to Understand Documentation	Yes	No
Complete Functionality	No	No
Availability of Roadmaps	No	No
Business Drawbacks		
Support Provided	Yes	Limited
Clear ownership	Yes	Yes
Access to source code	No	Yes
Sufficient marketing	No	No

2.5.5 Summary

Industrial engineers and academic researchers who are looking to pair their instrumentation with a software package have a difficult decision to make:

1. Generic commercial software, high level of abstraction, easy to use, inexpensive, but limited performance.
2. Build a bespoke software package, requires significant time and cost, would require mid-level developer(s) to create, would require UI, signal processing, and documentation to be created.
3. Open-source software, free to use, high probability of better performance than generic software, however unless it for its specific application, would need mid-level programming to modify.

It should be noted, that there may be licensing restrictions if using re-packaging commercial software or using open-source for a business's own commercial purposes.

2.6 Conclusion

The literature reviewed illustrates the increase in sensors and the requirement to process the data captured from them arising from the transition to Industry 4.0. With the increase in the use of technologies such as CPS to model a theoretic system and compare it to a physical world, and IIoT increasing not only the total number of sensors, but the variety of parameters these sensors are capturing, the processing requirements of a system are at an all-time high.

As the data size increases, and the type of sensors diversifies, researchers and industry are frequently finding that their traditional, sequential-based implementations, through software such as MATLAB and LabVIEW can no longer meet their requirements. As these requirements exceed the capability of generic commercial software, the field is looking towards bespoke implementations of software using mid-level programming languages to harness the benefit of programming “closer to the hardware”.

As programming moves “away from the hardware” also referred to as a strong level of abstraction, it may use more natural language elements and is much more user readable. This easier to use interface is MATLAB and LabVIEW’s greatest benefit, but the strong abstraction provides an inherent performance deficit and therefore it is also its greatest weakness. Moving into lower-level languages, “closer to the hardware”, removes these levels of abstraction and gives the developer tighter control over processing cycles and memory. However, this is of course at the cost of more complicated programming, longer development times, and an increased skill set being required.

In addition to the benefit of programming in lower programming languages, APIs such as OpenCL and CUDA allow the use of GPGPU processing, a form of hardware acceleration. Although not the only form, with the use of DSPs and FPGAs also increasing, GPGPU processing can offer significant performance increases, as demonstrated by the literature reviewed, at relatively small financial cost.

A heterogenous combination of sequential CPU and parallel GPU execution of signal processing algorithms would allow a “best of both worlds” approach to the processing of surface and dimensional data, other optical sensor techniques, and beyond. However, being able to provide algorithms on multiple architectures introduced a new problem, how to determine which architecture is best for a given function. To find a solution to this problem, the field of artificial intelligence search and planning will be investigated to discover what techniques exist and how they can be implemented into a software package to provide an optimum processing experience.

3 Artificial Intelligence Techniques to Increase Performance

3.1 Introduction

This chapter investigates autonomic computing and artificial intelligence (AI). Autonomic Computing is a computer's ability to manage itself without human intervention. Autonomic computing uses AI to coordinate plans for self-configuration, self-optimisation and self-healing and is being used as part of Industry 4.0 during the manufacturing process (Sanchez, Exposito, & Aguilar, 2020). Industry 4.0 requires high agility and rapid change of the manufacturing process and fast reconfiguration of production lines, but without limiting the performance of the production lines. Therefore, any software package requires both performance and easy configurability.

To maximise processing performance, CPU-GPU heterogenous computing is proposed, however this creates a problem surrounding the choice of architecture for a given function. It would not be efficient to exhaustively test every combination of function and architecture, therefore the field of artificial intelligence, more specifically search and planning, is reviewed to discover what techniques exist to solve this problem is a timely fashion.

A concept defined as "Autonomic Cycle of Data Analytics" outlines the closed loop tasks that work together to enable automation of the processes that they supervise (Aguilar, Cordero, & Buendía, 2018). Firstly, tasks must be monitored by capturing data and information about the systems behaviour, this links back to the previously discussed issue surrounding big data. Secondly, once captured, the data needs to be analysed to interpret, understand, and compare the data against what it knows the process should be doing. Finally, and most importantly, it needs to make a decision, based on its understanding of the process and what is happening, the system must decide on a course of action to take to reduce failures amongst many other factors. Once this "cycle" has been completed, it is restarted starting a new iteration of the cycle (Sanchez et al., 2020).

This research concerns itself with the self-optimisation of signal processing, and to an extent self-configuration based upon the optimisation solution. Scheduling these processing components can be understood as a search problem with various heuristics used to determine what is a successful, optimal path (Klöpper, 2010). Search can be either uninformed, where no further knowledge is assumed of the data, or informed, where other data is known. Automated planning concentrates on how computer interpretable syntax can be created to instruct the planning algorithm what is required of it (Russell & Norvig, 2010).

Potential solutions to this optimisation problem will now be evaluated, starting with search algorithms, both uninformed and informed.

3.2 Evaluation of Search Algorithms

Rather than iterating each possible solution and then selecting the path that returns the smallest number (Figure 3-1), search algorithms use different search techniques, minimum-priority queues, weights and heuristics to reduce the number of permutations that need to be checked. Search starts with an initial node (the root) with each possible other node being linked by branches. Different search algorithms have different techniques for investigating these branches, this section discusses these methods and their advantages and disadvantages. This section uses the concept as asymptotic complexity, also known as O-notation. O-notation describes the behaviour of a function according to how their run time or memory requirements grow as the input size grows (Russell & Norvig, 2010).

Search algorithms reduce the worst-case performance from $O(2^n)$ where n is the number of nodes, to something much smaller. There are many different techniques to searching through a node map such as Depth-first, Breadth-first, Dijkstra, Best-first, and A*.

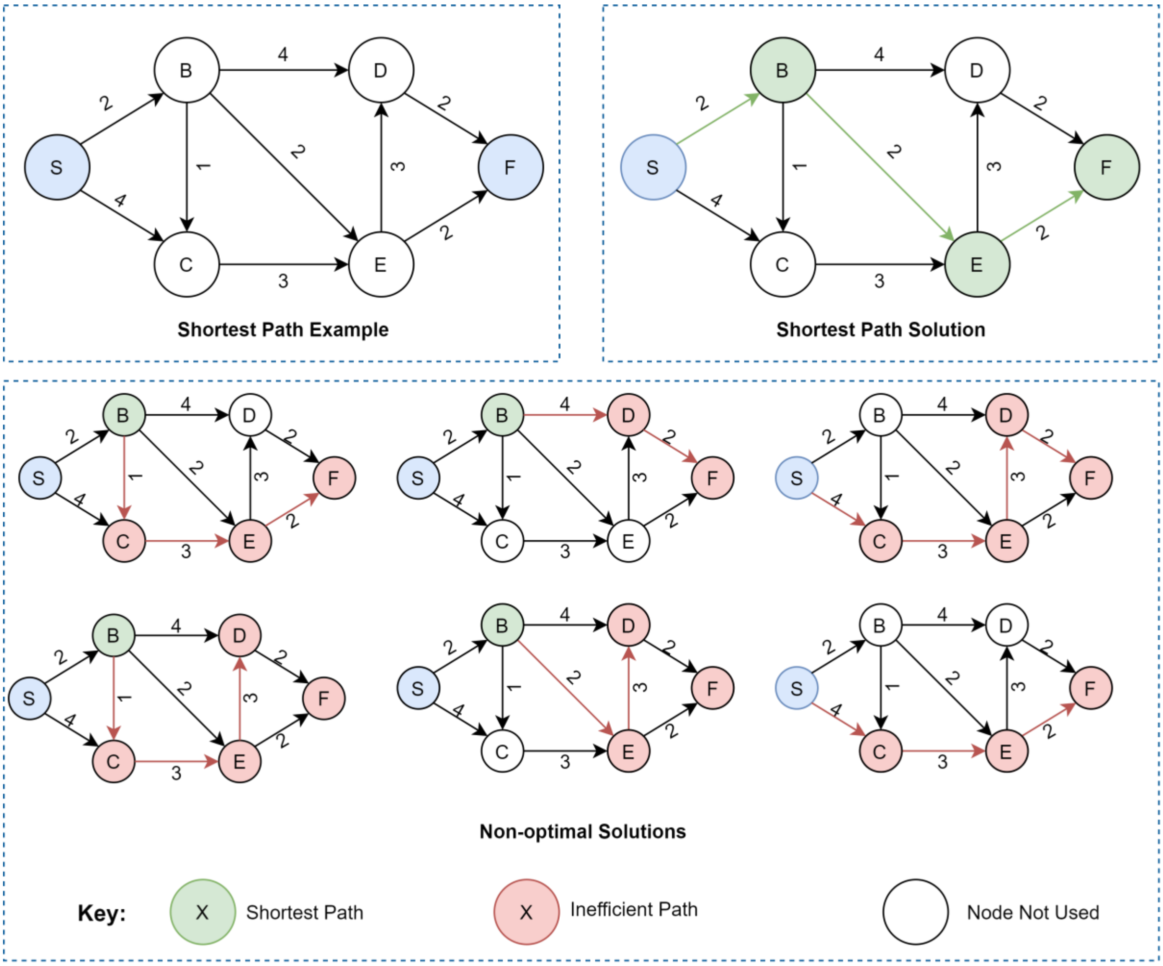


Figure 3-1: *Example of Node diagram, using blind search to evaluate all possible paths.*

3.2.1 Uninformed Search Strategies

Uninformed search strategies, also referred to as blind search, have no additional information about the distance between the start and the goal state. These search strategies are defined by the order which they expand nodes, all they can do is generate successors in order to find the goal state (Russell & Norvig, 2010).

3.2.1.1 Breadth-first

Breadth-first search initially expands the root node and then expands all of its successors. All nodes are given a depth in the search tree and once all the successors at the same depth have been expanded, their own successors are then expanded (Figure 3-2). Any search which stores every expanded node has a space complexity as a factor of the time complexity. For breadth-first, the time and space complexity are both $O(b^{d+1})$, *b* is the branching factor (average) and *d* is the depth of the node graph. This not only means the algorithm takes exponentially longer depending on the depth, but requires an exponential amount of memory to store all the explored nodes (Russell & Norvig, 2010). In the figure below, the goal node would be the 7th node to be expanded.

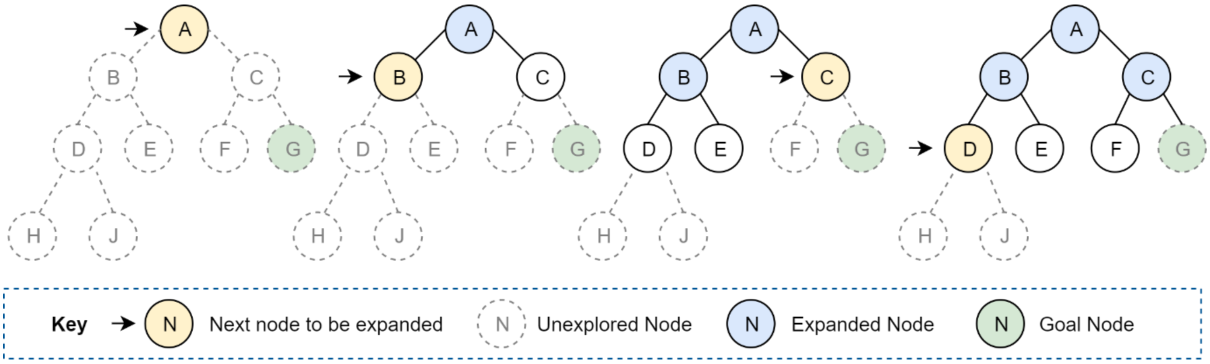


Figure 3-2 *Breadth-First Search on a simple binary tree (Russell & Norvig, 2010)*

3.2.1.2 Depth-first

Depth first search expands each node as deep as possible until the node has no successor, if it reaches the bottom and a goal has not been satisfied it is removed from memory and returns to the highest non-expanded node and restart this process (Figure 3-3). This version of search is non-optimal as it may search the hole of the left-hand branch (to node H), but the goal is on the opposite side of the node graph. If there are multiple goals, it could return a path that is not necessarily the shortest, but simply the one it found first; a path which could have a high depth value. The time complexity of the depth-first algorithm is $O(b^m)$ where *b* is the branch factor and *m* is the maximum depth. This is potentially slower than breadth-first, depending on where the goal lies within the graph, but the space complexity for depth-first is only $O(bm)$, significantly reducing the amount of memory needed (Russell & Norvig, 2010). In the below figure, the goal node would be the 10th node to be expanded.

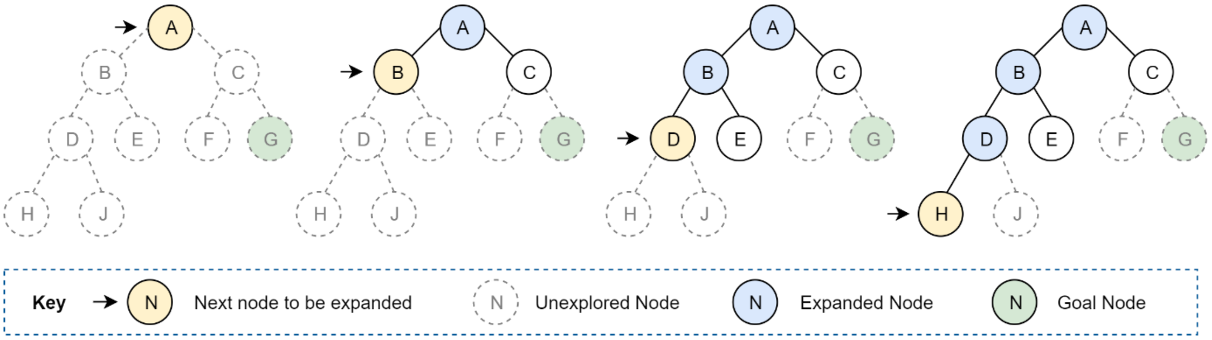


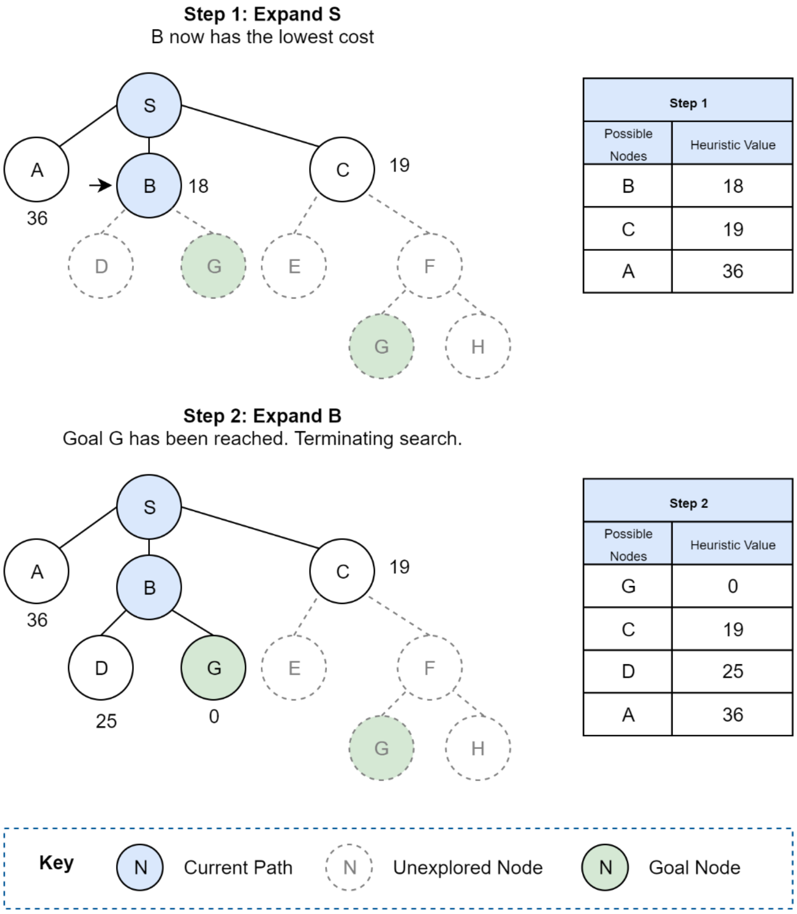
Figure 3-3: *Depth-First Search on a simple binary tree (Russell & Norvig, 2010)*

3.2.2 Informed Search Strategies

Informed search uses problem-specific knowledge beyond the problem itself and uses this information to find a solution more efficiently. These strategies not only consider a cost to travel from the current node to its successors $g(n)$, but also a heuristic value $h(n)$ that helps guide the algorithm towards the most optimum path, given an admissible heuristic (Russell & Norvig, 2010).

3.2.2.1 Best-first

Best-first uses heuristics to prioritises nodes that are heading in the correct 'direction', and only checks others if those routes are unsuccessful (Russell & Norvig, 2010). Figure 3-4 shows a node diagram with a start node *S* and an end node of *G*, also on the graphs are the heuristic values, these have been calculated based on the physical distance between *S* and *G*. Out of the nodes at depth one, *B* has the lowest heuristic value so is expanded first, once *B* is expanded its successors are added to the priority queue in heuristic order. This continues until a goal state is found, or the node has no successors, at which point it continues at the next possible node from the queue. After B is expanded, a goal state is found and therefore the search is stopped; it would not go on to investigate the path $[S \rightarrow C \rightarrow F \rightarrow G]$, which could overall have a shorter path. As best-first only uses heuristics, and not path length, the path length from $[B -> G]$ could be longer than $[C -> F -> G]$, but as the heuristic is smaller it is not considered.



Step 1	
Possible Nodes	Heuristic Value
B	18
C	19
A	36

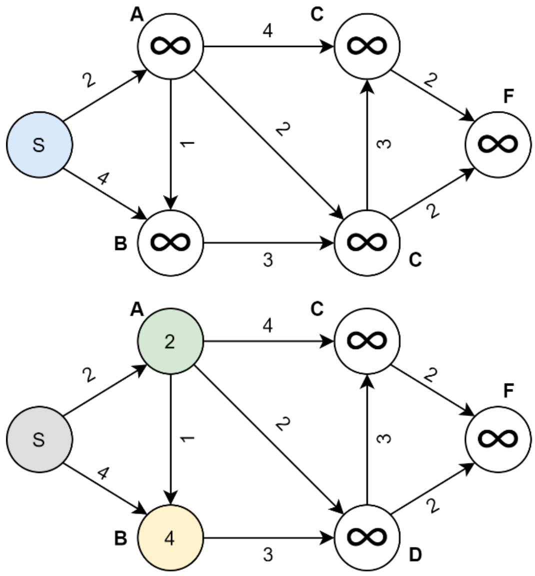
Step 2	
Possible Nodes	Heuristic Value
G	0
C	19
D	25
A	36

Figure 3-4: *Best-first Search on a binary tree*

Although best-first has a potential $O(b^m)$ time complexity, just like depth first, the search cost is likely to be significantly smaller, which is better, as the space complexity is also $O(b^m)$. However, as it should only expand the nodes that are tending towards the goal state, this should reduce the amount of memory needed.

3.2.2.2 Dijkstra

Dijkstra's algorithm (or Dijkstra Shortest Path algorithm) uses path length for finding the shortest path between nodes, giving a worst-case performance of $O(n^2)$ where n is the number of nodes (Dijkstra, 1959). Edsger W. Dijkstra first created this algorithm to find the shortest path between Rotterdam and Groningen in the 1950s, but in the decades since its development, it has been extensively used and many iterations of it exist (Frana & Misa, 2010). Dijkstra uses best-first search to expand the shortest paths first, only expanding other once the current shortest path has a dead-end. An example of how Dijkstra's algorithm works is illustrated in Figure 3-5 to Figure 3-9, each node has a current shortest distance of infinity, as until each node is expanded, the node may not be accessible. In the first figure (Figure 3-5), the first two possible paths (*A* and *B*) are expanded. As the path to *A* is shorter than the path to *B*, this path is further investigated before returning to path *B*.

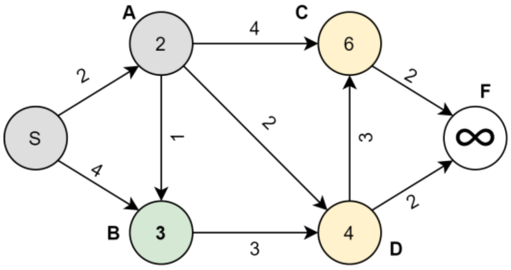


Node	Shortest Distance from S	Previous Vertex
A	∞	
B	∞	
C	∞	
D	∞	
F	∞	

Node	Shortest Distance from S	Previous Vertex
A	2	S
B	4	S
C	∞	
D	∞	
F	∞	

***Figure 3-5:** Initial Node Diagram and First step of Dijkstra's Algorithm to find the shortest path using a min-priority queue*

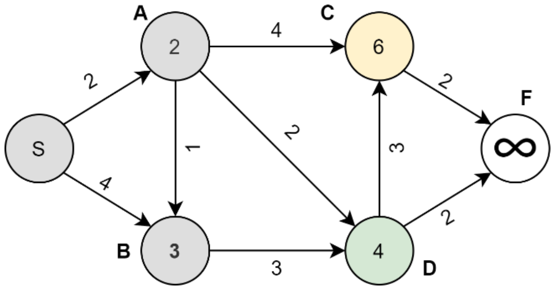
Step two for this example involves expanding node *A*, the current shortest path (Figure 3-6). The three routes to be expanded here are $[A \rightarrow B]$, $[A \rightarrow C]$, and $[A \rightarrow D]$. The route $[S \rightarrow A \rightarrow B]$ now has a path length of 3, therefore it is quicker to perform the route $[S \rightarrow A \rightarrow B]$ than it is $[S \rightarrow B]$.



Node	Shortest Distance from S	Previous Vertex
B	3	A
D	4	A
C	6	A
F	∞	
A	2	S

***Figure 3-6**: Second step of Dijkstra's algorithm for this node diagram - Expanding node A*

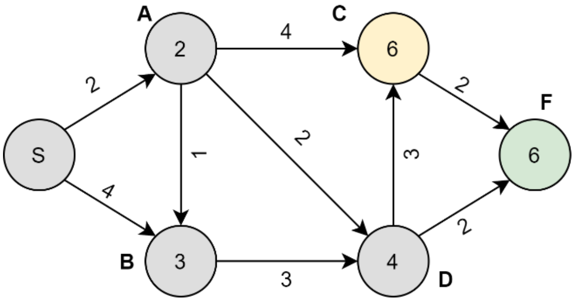
Once node A has been fully explored, $[S \rightarrow A \rightarrow B]$ has the shortest path length, therefore node *B* is expanded next. Node *B* can only be expanded to $[S \rightarrow A \rightarrow B \rightarrow D]$ which would provide a path length of seven. This is already larger than the path $[S \rightarrow A \rightarrow D]$, four, so is ignored, the next shortest path is now $[S \rightarrow A \rightarrow D]$, therefore node *D* is expanded next (Figure 3-7).



Node	Shortest Distance from S	Previous Vertex
D	4	A
C	6	A
F	∞	
A	2	S
B	3	A

***Figure 3-7:** Third step of Dijkstra's algorithm for this node diagram - Expanding node B*

Node *D* can now be expanded in two directions, to node *C* (current path length 6), or to *F*. Expanding to node *C*, giving the path $[S \to A \to D \to C]$, returns a path length of 7, greater than the current path $[S \to A \to C]$ length so is ignored. Expanding to *F* would provide a total path length of six for $[S \to A \to D \to F]$ (Figure 3-8).

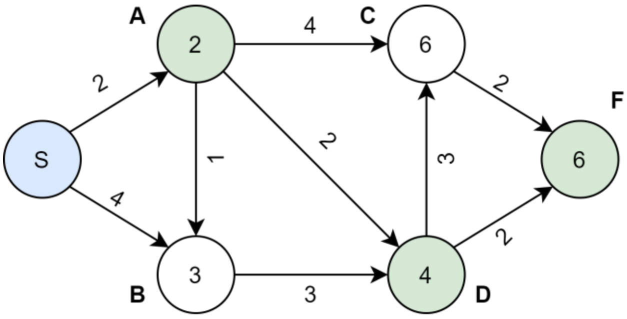


Node	Shortest Distance from S	Previous Vertex
F	6	D
C	6	A
A	2	S
B	3	A
D	4	A

Figure 3-8: *Final step of algorithm for this node diagram - Expanding node D*

The complete path of $[S \to A \to D \to F]$ is the same length of $[S \to A \to C]$, therefore the route $[S \to A \to C \to F]$ would not be investigated as it is not shorter. However, if $[S \to A \to D \to F]$ was greater than $[S \to A \to D \to C]$ or another path was still being considered, all these possible paths would be investigated until the completed path was the smallest path length.

Dijkstra's algorithm works well to find the shortest path in very few steps (Figure 3-9), but when applying it to this research, not all paths are possible. As functions have strict prerequisites, there may only be a couple of valid paths, rather than the multiple paths that Dijkstra would work its way through.



Node	Shortest Distance from S	Previous Vertex
A	2	S
B	3	A
C	6	A
D	4	A
F	6	D

Figure 3-9: *Shortest path to each node and overall shortest path found using Dijkstra's algorithm*

Dijkstra could be used in this manner to have a node for each signal processing function for both CPU and GPU (Figure 3-10). Using a brute-force method of trying each possibility would take $2^n = 2^4 = 16$ steps, whereas using Dijkstra's algorithm it only takes five steps.

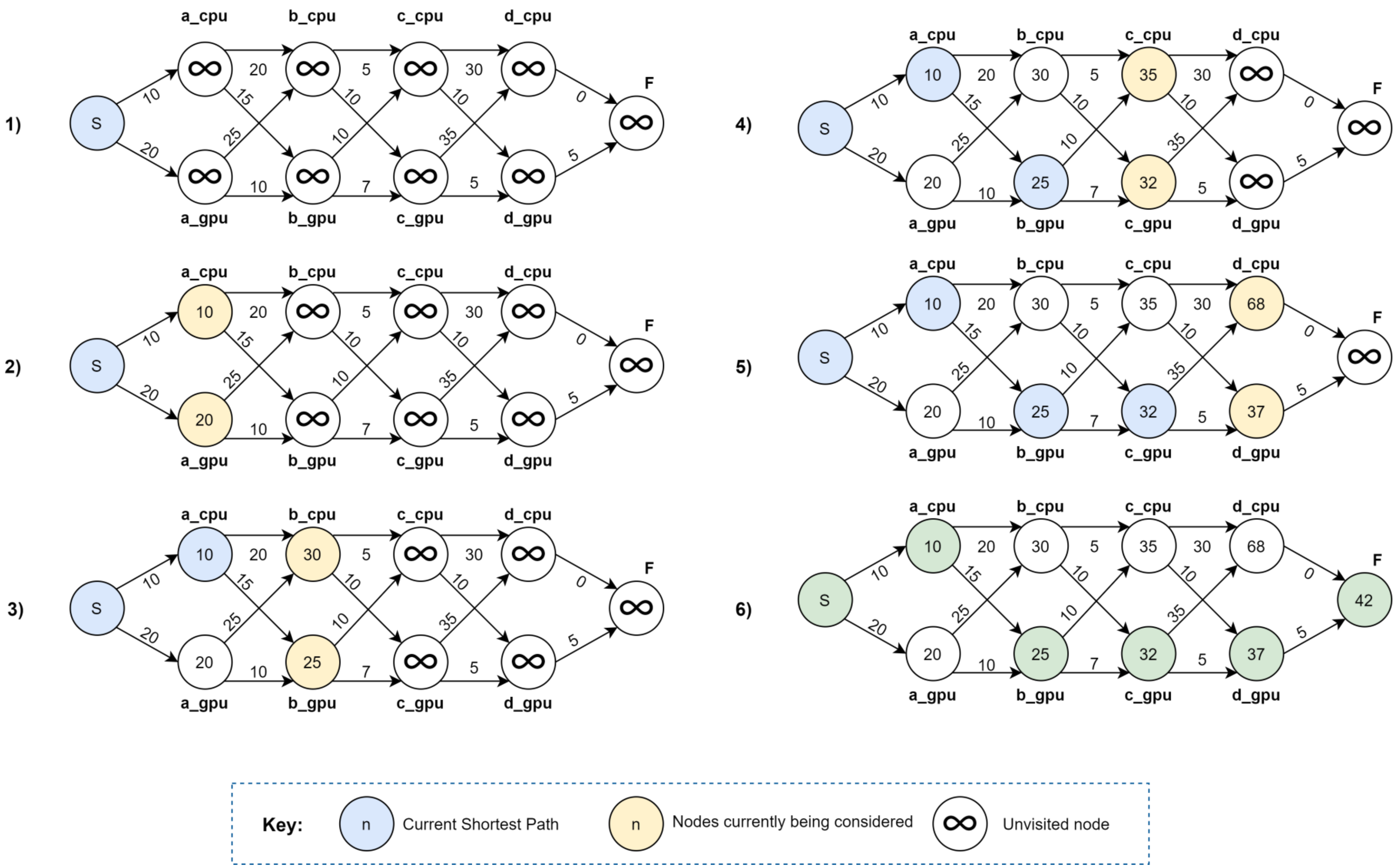


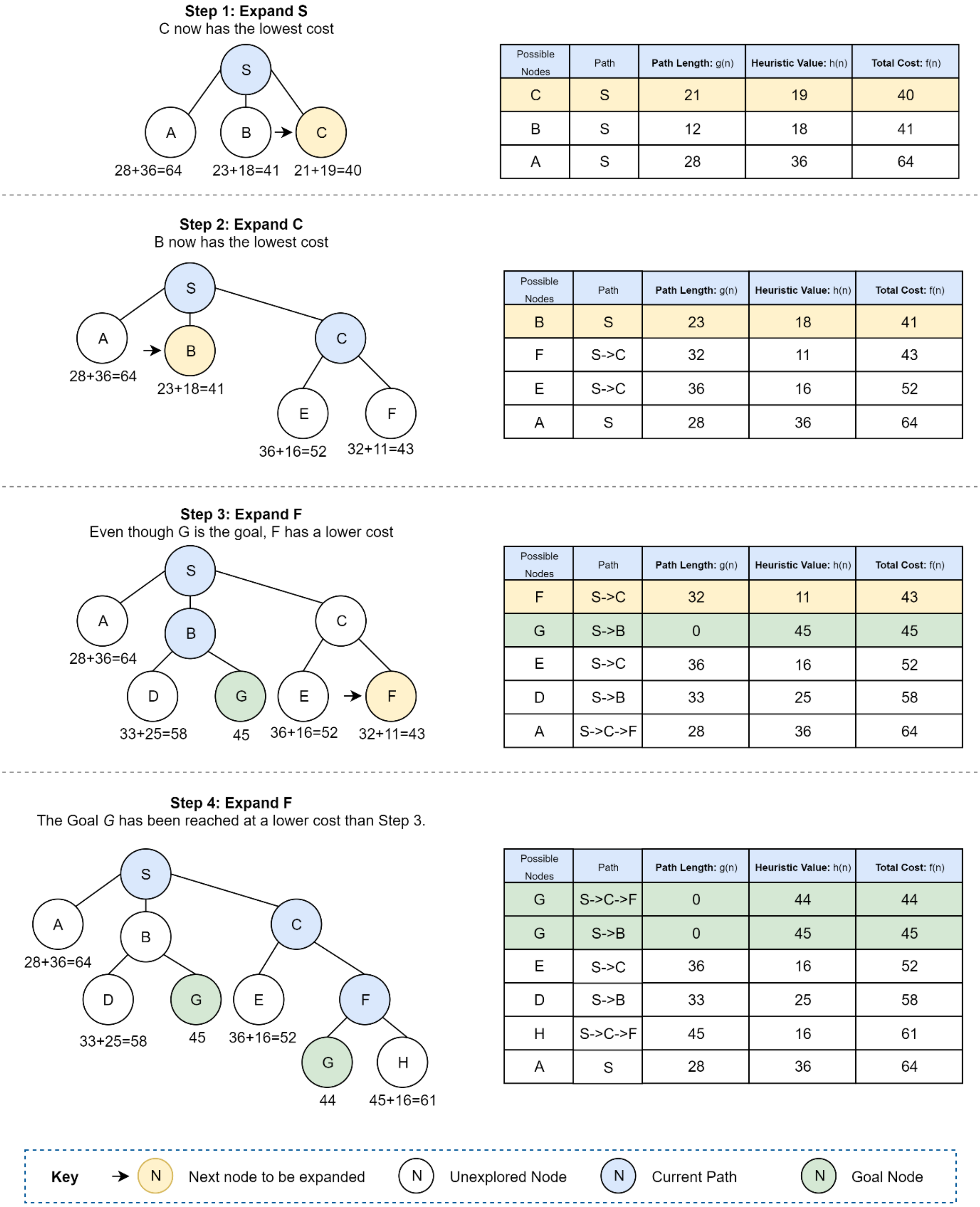
Figure 3-10: Using the example data from , a node graph can be made and solved with Dijkstra's algorithm

Unfortunately, there are a few limitations to Dijkstra's algorithm that means it is not suitable for this application. One of the things Dijkstra requires is rigid structure, requiring node *A* to consistently come before node *B*, and if this is changed, a whole new structure is required for this minor change. Another drawback of Dijkstra is it does not allow parallel actions, if it is advantageous to execute two functions at the same time, one on the CPU and the other on the GPU.

3.2.2.3 A*

The most widely used known format of best-first search is A*. It evaluates which nodes to expand based not only upon the cost to travel to the node (e.g. Dijkstra) but also the heuristic cost to travel from the current node to the goal. A heuristic is a secondary value that can be used to determine whether the current path is reaching the goal state, often this heuristic is a metric like physical distance, for this research the most sensible heuristic to use would be execution time. In A* the travel cost is referred to as $g(n)$, the heuristic cost $h(n)$, with the total cost calculated by $f(n) = g(n) + h(n)$ through node *n*. When finding the optimum solution, it starts with the node with the smallest $f(n)$ value. The main condition of a heuristic used for A* is it must be admissible. An admissible heuristic is one that never overestimates the cost, i.e., it estimates to reach the goal is not higher than the lowest possible cost from the current node (Russell & Norvig, 2010).

Figure 3-11 is an example of an A* node diagram and is compared to Figure 3-4, which uses the same node map, but used the simpler best first instead. As A* uses both path length $g(n)$ and heuristics $h(n)$ to determine which node to investigate, it expands nodes in a different order. Firstly, the start node *S* is expanded giving Nodes *A*, *B* and *C*. Unlike when using best-first, A* next expands node *C* as the even though the heuristic cost is higher, the total cost when factoring in the path length is lower and therefore at the top of the priority queue. Node *C* is then expanded giving *E* and *F*, as *F* now has a total cost higher than *B*, *B* therefore is expanded next. Once *B* is expanded, a goal is found with a total cost of 45, as this goal is not currently the shortest path being considered, $[S\text{—}> C\text{—}> F]$, this path is checked before returning the path $[S\text{—}> B\text{—}> G]$, the path that best-first returned. Now *F* is at the top of the priority queue, it is expanded, also finding a goal state, this goal state has a total path length of 44, one less than the previous goal found. As this goal state has been found, and is at the top of the priority queue, there are no possible shorter paths, and A* returns the shortest path as $[S\text{—}> C\text{—}> F \rightarrow G]$.



Step 1: Expand S
C now has the lowest cost

Possible Nodes	Path	**Path Length:** g(n)	**Heuristic Value:** h(n)	**Total Cost:** f(n)
C	S	21	19	40
B	S	12	18	41
A	S	28	36	64

Step 2: Expand C
B now has the lowest cost

Possible Nodes	Path	**Path Length:** g(n)	**Heuristic Value:** h(n)	**Total Cost:** f(n)
B	S	23	18	41
F	S->C	32	11	43
E	S->C	36	16	52
A	S	28	36	64

Step 3: Expand F
Even though G is the goal, F has a lower cost

Possible Nodes	Path	**Path Length:** g(n)	**Heuristic Value:** h(n)	**Total Cost:** f(n)
F	S->C	32	11	43
G	S->B	0	45	45
E	S->C	36	16	52
D	S->B	33	25	58
A	S->C->F	28	36	64

Step 4: Expand F
The Goal *G* has been reached at a lower cost than Step 3.

Possible Nodes	Path	**Path Length:** g(n)	**Heuristic Value:** h(n)	**Total Cost:** f(n)
G	S->C->F	0	44	44
G	S->B	0	45	45
E	S->C	36	16	52
D	S->B	33	25	58
H	S->C->F	45	16	61
A	S	28	36	64

***Figure 3-11**: A* Node Diagram Example - - Same Node Tree as Figure 3-4. Total Cost: $f(n) = g(n) + h(n)$*

In most scenarios where a sensible heuristic has been chosen, the time and space complexity are still exponential, the same as best-first, giving $O(b^d)$ where *b* is the branching factor and *d* is the depth of the goal. Although, due to the heuristic factor, many of the b^d nodes can be pruned saving time and memory, although memory is still an issue for A* as it keeps all generated nodes (complete or un-investigated) in memory.

3.2.3 Summary

While uninformed search may have space complexity benefits with algorithms such as depth-first, it has its drawbacks in time efficiency and can also return non-optimal solutions, although informed searches can also return non-optimal solutions. Uninformed searches can also fail to return a solution as they can find themselves stuck in infinite loops, depending on the way the nodes are mapped out. Informed search therefore has its benefits, it can be as quick or quicker than uninformed search, it can find solutions which may not be optimal in isolation but may have a lower cost than uniformed search if many goal states exist. The introduction of A*'s heuristic aims to constantly return the optimum, lowest cost goal, but a suitable heuristic is needed for this, A* can still have memory issues due to its need to store all generated nodes (Russell & Norvig, 2010).

On reflection of investigating search, it is not the right solution to the problem outlined in in the introduction. Instead of providing an algorithm with a set of waypoints and allow it to find the best route, there needs to be a mechanism to provide the algorithm a series of constraints. These constrains would contain three things:

1. A loose order in which the signal processing functions need to execute in.
2. Which variables can be used, and where.
3. Where functions could potentially be run in parallel.

These requirements for an algorithm gravitate more towards planning models rather than search.

3.3 Investigation of Automated Planning and Scheduling

Automated planning and scheduling, sometimes denoted as AI planning, arose from research in fields such as robotics and control theory (Ghallab, Nau, & Traverso, 2004). In 1971 Fikes and Nilsson (1971) developed STRIPS, the first major planning system. Since then variations and extensions of STRIPS have been created such as Action Description Language (ADL) (Pednault, 1989) and in 1998, Planning Domain Description Language was introduced as a computer-parsable syntax that allowed users to represent planning problems (McDermott et al., 1998).

Planning languages consist of actions and effects/results, where each action produces a result that then enables one or more further actions; just like node expansion in search, Unlike search, in planning, actions can have preconditions, actions that need to be completed before it can be called itself (Russell & Norvig, 2010). When planning algorithms are searching for a solution, they either use forward-search or backwards-search. Forward-search starts at the first action and looks for other actions for which their preconditions are satisfied. Conversely, backwards-search starts at the goal state, and work backwards, selecting those actions where their result would trigger the current action (Ghallab et al., 2004). These differences are shown in Figure 3-12. By using backwards search, only the nodes that modify the preconditions of the current node are considered, saving time and memory.

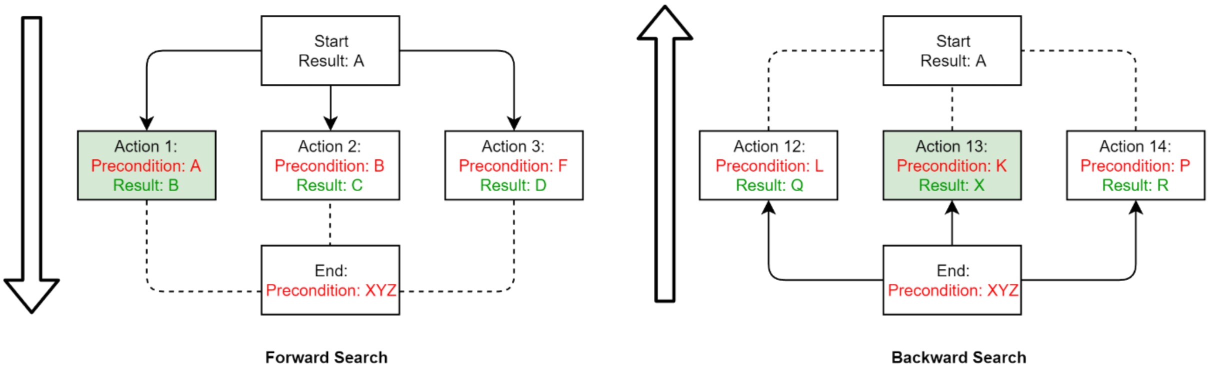


Figure 3-12: *AI Planning search types, where the box highlighted green would be the next action called*

3.3.1 Stanford Research Institute Problem Solver (STRIPS)

STRIPS is a planning automated planner that was created to improve the efficiency of other planning algorithms while reducing the size of the search space. Unlike Dijkstra that uses forward search, starting at the start node and working towards the end node, STRIPS uses backward search, starting at the end node and working its way back to the start. Part of STRIPS's efficiency is that when working backwards, the only nodes need to be considered are those that were a precondition of the current sub-goal. This not only reduces the branching factor, but increases the speed of the algorithm and reduces the memory needed to keep track of all the different branches that are being considered (Ghallab et al., 2004).

When first investigating the problem given, STRIPS (Fikes & Nilsson, 1971) creates a hierarchy of the final goal and its sub-goals (Figure 3-13). These sub-goals are based on the defined preconditions from each parent function, this is how the branching effect is reduced (Ghallab et al., 2004). For the purpose of illustration, two sub-goals (Sub-goal 1.1.1.1 and Sub-goal 2.1.2) have been included, but as no parent function exists with them as prerequisites, these would be ignored by STRIPS. The diagram is shown corrected in Figure 3-14.

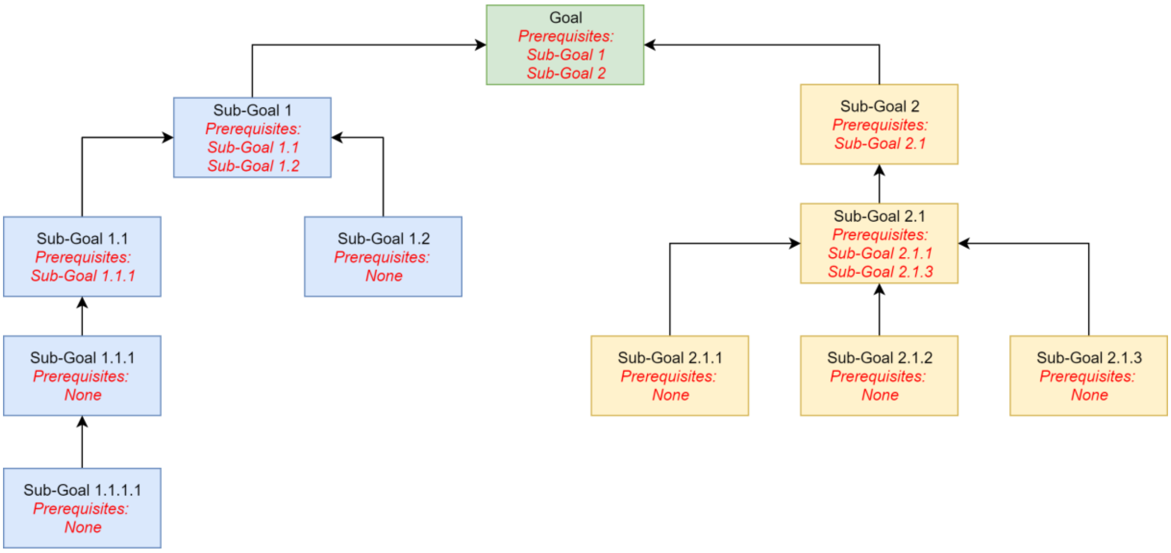


Figure 3-13: *STRIPS Hierarchy Example*

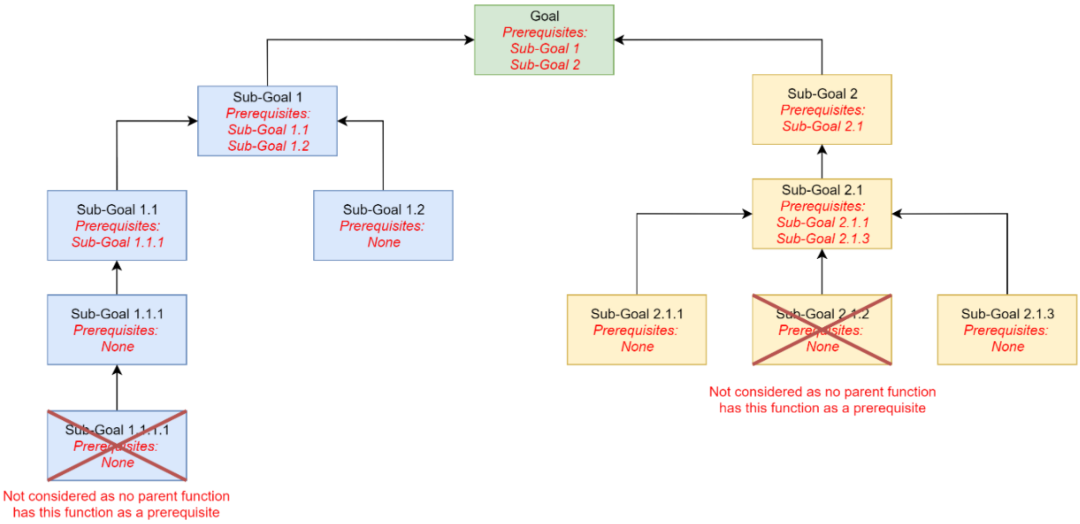


Figure 3-14: *STRIPS Hierarchy Example (Corrected)*

STRIPS then iterates through the various nodes from the main goal, backwards through the sub-goals until all goals in the network have been satisfied. Goals are therefore solved one at a time, this sometimes causes STRIPS to run into problems, such as when two variables are linked and interchanging (Ghallab et al., 2004). Probably the best known example of this is the Sussman anomaly (Sussman, 1973) (Figure 3-15), where two sub-goals are defined from the overall goal of $on(A, B) \wedge on(B, C)$. These sub-goals are defined as:

1. $on(A, B)$
2. $on(B, C)$

Sub-goal o (Figure 3-16a) is to place A on B, simply move C off A and then place A on B, this has satisfied the first goal, although sub-goal 2 cannot be achieved without undoing sub-goal 1. Instead try and to achieve sub-goal 2 (Figure 3-16b) Simply move block *B* on top of blocks *C* and *A*, again, to now achieve sub-goal 1, sub-goal 2 must be broken. This means that the algorithm needs to be recursive to break the goal, execute the action(s) then re-check that it satisfies both sub-goals, giving a worst-case performance of $O(n^3)$, although unlikely in applications where these conflicts do not exist (Ghallab et al., 2004).

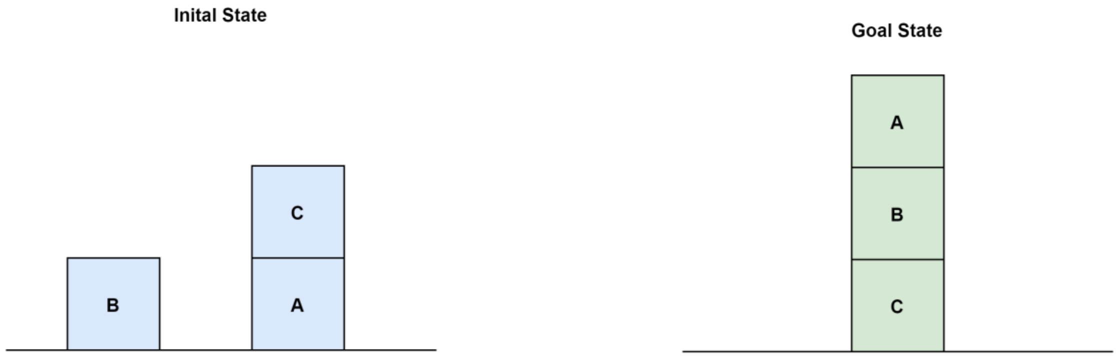


Figure 3-15: *Sussman Anomaly (Russell & Norvig, 2010)*

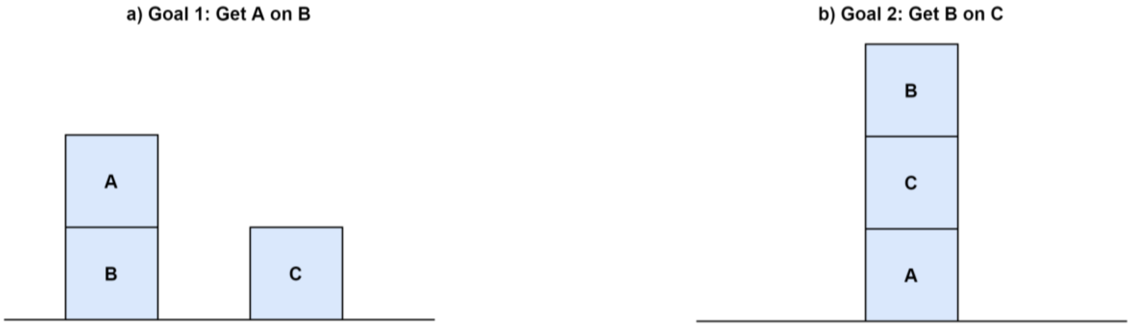


Figure 3-16: *Sussman Anomaly: Sub-goals (Russell & Norvig, 2010)*

The worst-case performance of $O(n^3)$, where *n* is the number of nodes, is unlikely for the application within this research. Therefore, it is likely to be nearer Dijkstra's value of $O(n^2)$ or $O(|V| + |E|)$, where *V* is the number of nodes and *E* is the number of edges (Ghallab et al., 2004; Russell & Norvig, 2010). It is more likely to be closer to A*'s worse case performance value of $O\left(b^d\right)$ where *b* is the branching factor and *d* is the depth (Russell & Norvig, 2010). As these algorithms are going to be dealing with signal processing functions, where each function is represented by a node, the number permutations increases exponentially, so the performance of these algorithms does need to be considered. However, this pathfinding algorithm only runs a single time once the user has settled on the signal processing steps needed for their application. Therefore, the overhead time for pathfinding is going to be relatively small in respect to the time this plan is used for once created. The overall time saved by using this type of method more than outweighs the time needed to find the shortest path in the first place, with the pathfinding algorithm only needing to re-run when there is a software or hardware change.

In conclusion, STRIPS provides backwards search capability reducing the number of active nodes under investigation and therefore reduces the need to check every single branch with its Breadth-first branching reduction technique. Combining STRIPS with a shortest path algorithm based on Dijkstra or A* would provide a method of finding the shortest path between the start and the end meeting all the sub-goals along the way. To achieve this, a formal planning language is needed to be able to formalise the goals, prerequisites and paths needed to successfully complete the signal processing requirements set.

3.3.2 Planning Domain Definition Language (PDDL)

PDDL provides a computer interpretable, standardised syntax for representing planning problems created as part of the International Planning Competition which it has evolved from over the last two decades (McDermott et al., 1998; Russell & Norvig, 2010). Its purpose was to try and standardise AI planning languages inspired by the languages of STRIPS (Fikes & Nilsson, 1971) and ADL (Pednault, 1989), with ADL being an advancement of STRIPS.

A PDDL model is split into two parts, a problem, and a domain. The problem consists of what the goal(s) are, it defines all the objects that are to be used and initialises them (Russell & Norvig, 2010). An object can have predicates (logical facts) applied to them; these states are assumed false unless initiated as true. Furthermore, the problem file is the goal states of different objects, for example: When making a cup of tea (a common example for PDDL), the goal is a hot cup of tea, but the user might want to program that the kettle is returned to its base. Therefore, the goals would be $Hot\ Cup\ of\ Tea\ \wedge Kettle\ on\ Base$. An example goal for the LSDI would be to calculate the height of the measured object, making sure that the measurement is on the CPU at the end. This way, even if the GPU functions outperform the CPU, the data is transferred back to the CPU before finishing. Thus, the goal is not just $Calculate\ Height$ it would be $Calculate\ Height \wedge Height\ is\ on\ CPU$, here the height is the object and the predicate is checking the object is *onCPU*.

The second half of the PDDL model is the domain file. The domain file contains the list of possible actions, not all of which may be needed to satisfy to goal(s) specified in the problem file, this is where STRIPS comes in useful to remove the unnecessary branches (actions) that are not needed for the end-goal (Russell & Norvig, 2010). Each action in the domain contains three parts.

1. Parameters

These are instances of the objects defined that needed for use in this action. Instances are simply a copy of the object, allowing a user to have multiple instances of the same object type, just like in programming with classes and objects. *This implementation defines an object to a single instance and therefore object/instance can be used interchangeably.*

2. Preconditions

These are the same as the prerequisites discussed earlier, these are the conditions that must be met before this action can be used.

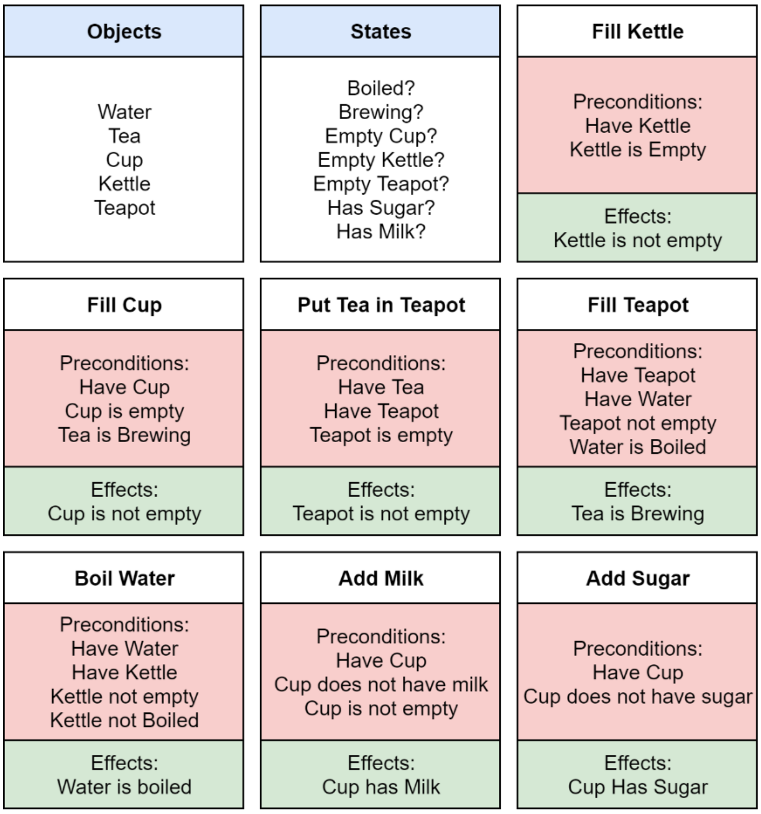
3. Effects

Effects can be viewed as the opposite of preconditions; these are what should happen once the action has been completed.

These three parts are the structure of each action and allows PDDL to start with the final goal and work backwards based on the preconditions needed for the current node to find the start point and find a path.

3.3.2.1 Example: Cup of Tea

Using the example of making a cup of tea using STRIPS and PDDL shows how PDDL can navigate a path based on set goals and constrains. Figure 3-17 shows a graphical representation of PDDL code for the domain. The domain contains seven different actions, all part of the process of making a cup of tea. Each action has preconditions and effects, there also objects, the components needed to make a cup of tea, and states these objects could be in. In PDDL, a state is either true or false, therefore duplicate states for *Cup is Empty* and *Cup is Full* are not needed as either one can be used, with the other one being *Cup is* ***not*** *Empty* or *cup is* ***not*** *Full*.



Objects

Water
Tea
Cup
Kettle
Teapot

States

Boiled?
Brewing?
Empty Cup?
Empty Kettle?
Empty Teapot?
Has Sugar?
Has Milk?

Fill Kettle

Preconditions:
Have Kettle
Kettle is Empty

Effects:
Kettle is not empty

Fill Cup

Preconditions:
Have Cup
Cup is empty
Tea is Brewing

Effects:
Cup is not empty

Put Tea in Teapot

Preconditions:
Have Tea
Have Teapot
Teapot is empty

Effects:
Teapot is not empty

Fill Teapot

Preconditions:
Have Teapot
Have Water
Teapot not empty
Water is Boiled

Effects:
Tea is Brewing

Boil Water

Preconditions:
Have Water
Have Kettle
Kettle not empty
Kettle not Boiled

Effects:
Water is boiled

Add Milk

Preconditions:
Have Cup
Cup does not have milk
Cup is not empty

Effects:
Cup has Milk

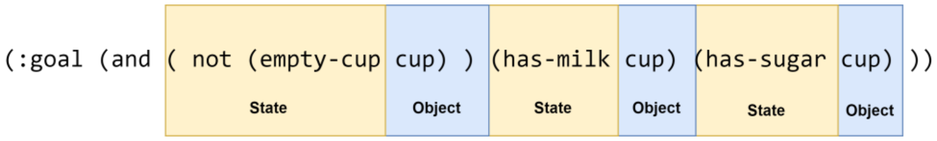
Add Sugar

Preconditions:
Have Cup
Cup does not have sugar

Effects:
Cup Has Sugar

Figure 3-17: *Graphical Representation of PDDL Domain (Making a cup of tea)*

The problem file is then also used to define what is expected of a PDDL plan. In this example, PDDL is required to find a path that makes cup of tea with milk and sugar. Based on the objects and states that can be used, the end goal must include what states need to be true at the end-goal. In this instance, for a cup of tea, the cup needs to not be empty, the cup needs to contain milk and the cup contain sugar. This would be represented as: $Cup\ Not\ Empty\ \wedge Cup\ Has\ Milk\ \wedge Cup\ Has\ Sugar$. This is shown in PDDL syntax in Figure 3-18, the object is the cup, with each of the states being check if true.



```
(:goal (and ( not (empty-cup cup) ) (has-milk cup) (has-sugar cup) ))
```

Figure 3-18*: Goal PDDL Syntax for making a cup of tea.*

PDDL then takes both the domain file and the problem file and creates a map in memory, starting at the final goal, using the sub goals that were set out in the problem file, to create a path all the way back to the start. Taking each sub-goal and looking at which action effects satisfy the preconditions for that sub-goal. This can be seen in Figure 3-19. There are a few things to note in this diagram, firstly *Add Sugar* can be done at any time, there is no arrow leading to sub-goals before this, therefore it could be done at the start, in the middle or near the end. The milk on the other hand, requires the cup to not be empty so cannot be completed until the cup has been filled, this is shown by the arrow pointing left from the *Add Milk* action. Secondly the same is true for *Fill Kettle* and *Put Tea in Teapot*, neither of these actions have preconditions and therefore can be done in any order.

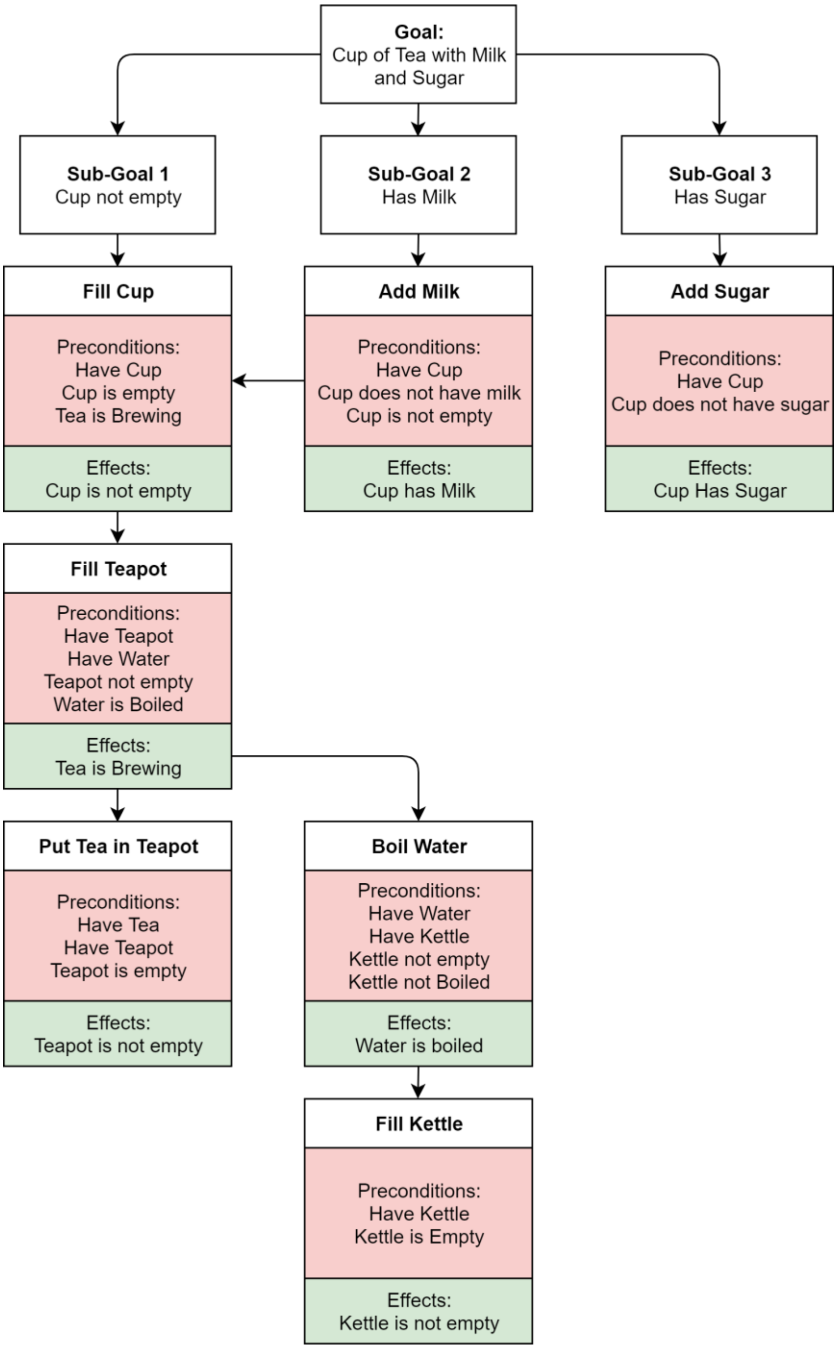


Figure 3-19: *Graphical Representation of PDDL Map (making a cup of tea)*

The solution from PDDL for this example is shown in Figure 3-20. The numbers that preface each action illustrates the order the actions can be completed in; equal numbers state they can be done at the same time in any order, an increase in number means a previous action needs to have completed. As can be seen when comparing Figure 3-19 with Figure 3-20, the generated map was correct and the actions *Put Tea in Teapot*, *Fill Kettle* and *Add Sugar* can be executed in any order. This shows that PDDL can find the solution to a problem given the correct logic and constraints. Furthermore, PDDL 2.1 introduced the ability for actions to have a cost associated with them, hopefully solving the pathfinding problem.

```
0:  (PUT-TEA-IN-TEAPOT T TP) [1]

0:  (FILL-KETTLE K) [1]

0:  (ADD-SUGAR C) [1]

1:  (BOIL-WATER K W) [1]

2:  (FILL-TEAPOT TP W) [1]

3:  (FILL-CUP C TP) [1]
```

Figure 3-20: *PDDL Solution (making a cup of tea)*

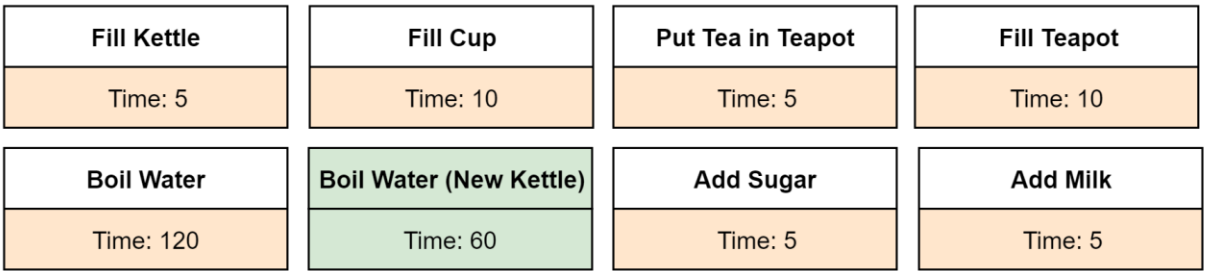
3.3.2.2 PDDL Fluents

PDDL 2.1 (Fox & Long, 2003) introduced fluents, referred to in PDDL as action-costs, which can be used within an action effect to increase/decrease a numerical parameter. This enables PDDL to not only find a possible path but find the best path given a metric to optimise for. Given different applications, different metrics may be necessary to optimise for. A path may need to be a longer, or require a couple more actions, but if it increases accuracy, or decreases cost, that might be the more desirable path. This has a performance impact, as PDDL has to check all possible paths to optimise for the given metric and not simply return the first couple of successful paths that it finds.

Looking back at the cup of tea example, if a (time) cost is assigned to each action (Figure 3-21) and the problem file is instructed to minimise the total cost, it would return the same plan. But if a second type of kettle was added, that boiled water in half the time and the program was run again, it would return the new solution that minimises the time (Figure 3-22). While both the old and new plan are both valid, now that action costs can be defined, the new plan is more desirable.

Fill Kettle	Time: 5
Fill Cup	Time: 10
Put Tea in Teapot	Time: 5
Fill Teapot	Time: 10
Boil Water	Time: 120
Add Sugar	Time: 5
Add Milk	Time: 5

Figure 3-21: *Cup of Tea Example - Action Costs*



Fill Kettle	Fill Cup	Put Tea in Teapot	Fill Teapot
Time: 5	Time: 10	Time: 5	Time: 10
Boil Water	Boil Water (New Kettle)	Add Sugar	Add Milk
Time: 120	Time: 60	Time: 5	Time: 5

Figure 3-22: *Cup of Tea Example - New Action Cost Added - creating more desirable plan*

PDDL has shown using the cup of tea examples that given a goal state, and logical sub-goals, it can find the shortest path given a metric to optimise for, in the examples case, time. It can hold the state of objects in memory allowing for complex logical decisions to be made given an extensive list of actions. Each action only executes if all preconditions are met and returns its own changes in the form of effects to allow PDDL to continue to find the shortest path. There are more extensions of PDDL 2.1 such as durative actions; actions that exist for a specific length of time, that could be used in the future to allow scheduling of actions, but as this research is not taking place on a RTOS, this would not be of much use.

3.3.2.3 PDDL Planner

As PDDL is a language, a parser (called a planner) is needed to compute the files and output a solution. The planner selected for this research was LPG-td (Local Search for Planning Graphs; Timed initial literals and Derived predicates). LPG-td was selected as it allows the use of fluents, introduced in PDDL 2.1 (Fox & Long, 2003), enabling the assignment of a time cost to each action. The problem file can then be instructed to minimise the total-cost of all the actions used, therefore each solution that LPD-td returns has a lower cost than the previous, further optimising the signal processing. LPG-td requires a domain file and a problem file, it then computes the best path to the goal state based upon the constrains given to it in both the domain and problem files. LPG-td first finds a valid path, and then optimise and explore other paths resulting in the final solution being the optimum solution. There are many settings that can be set to also determine when LPG-td stops looking, such as number of path restarts, number of solutions found, or a time limit.

3.3.2.4 PDDL vs Monte Carlo Simulation

Monte Carlo simulations are used to model the likelihood of various outcomes in a mechanism that is difficult to predict due to random variables' interference. It is a method for figuring out how risk and uncertainty effect prediction and forecasting models. A Monte Carlo simulation can be used to solve problems in almost any area, including economics, engineering, supply chain management, and research. A conventional optimisation problem that is regularly discussed alongside Monte Carlo simulation, is the travelling salesperson problem. This problem investigates the optimum travel choice, given the factual distance between each point across a set number of "destinations".

The problem outlined in this chapter has many similarities to this conventional problem. The CPU time that is used as the heuristic would be the distance between the two points, with the goal of visiting each destination translating to visiting each function. However, this is where the similarities end. Figure 3-23 illustrates a sample of two functions with two variables (A and B), with the permutations of a third function also shown. The choice of the first function is one of five, $CPU1$, $GPU1$, $GPU1$ (A), $GPU1$ (B) and $GPU1$ (AB). However, the number of permutations soon rises, as for the second function, the choice is one of 30, and one of 175 for the third function, with only the memories transfers considered that have not be performed previously.

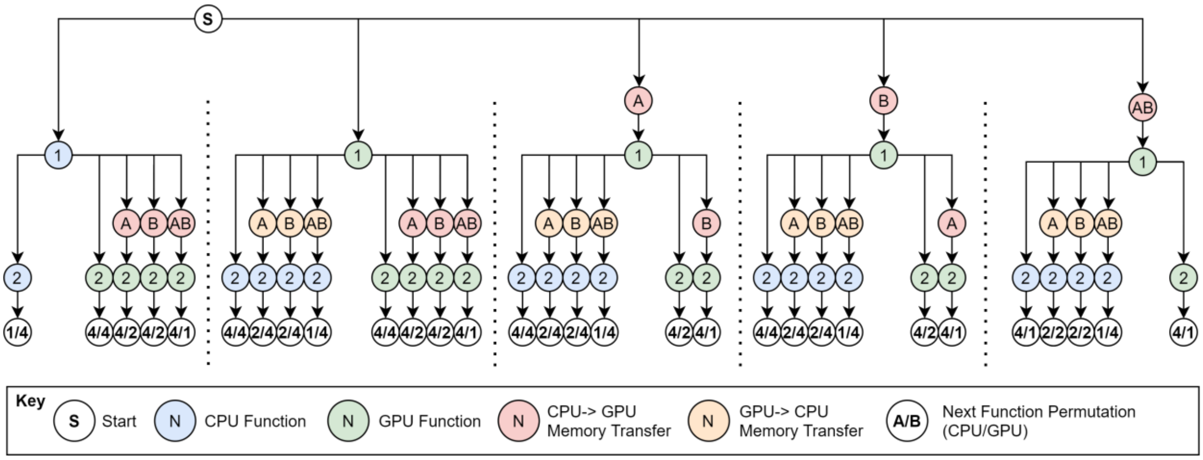


Figure 3-23: All Possible Permutations of three functions with necessary memory transfers

The result of this indicates if all potentially memory transfers are considered at each branch, the branching factor is between $m^2 + 1$ and $m^2 + 2$, where m is the number of variables. Therefore, the time and space complexity increases from $O(2^n)$ to $O\big((m^2 + n)(m^2 + 1)^{n-1}\big)$, where n is the number of functions. To reduce the branching factor and the increase in time and space complexity, a PDDL domain with will be used with predicates and actions tracking which memory transfers are required and which are already in the correct place. This should hopefully reduce the time complexity back towards $O(2^n)$ at the cost of increasing the space complexity to track which memory transfers are required, and which have already occurred. In addition, PDDL will enable future work for scheduling of asynchronous memory transfers and concurrent kernels to be executed, something a Monte Carlo simulation is not designed for.

3.3.2.5 Summary

AI Planning, or more specifically PDDL, has shown that it can find a solution to a problem, given a list of constrains. Not only can a solution be found for a problem, but it can find the optimal solution for the given constrains. By providing the problem file with a cost metric to optimise for, PDDL finds the solution with the smallest cost, in the case of this research, that cost is function execution time. Not that PDDL is without problems, if even a small change to an action or the problem is made, the whole plan needs to be re-computed to check a valid solution still exists. While this process does take time to run, by using backwards-search and breadth first to reduce the branching factor does reduce the amount of time needed. Although this time penalty is minimised, the planning time is very small compared to the amount of time that plan would be valid for, only needing updating when a software or hardware change is needed.

Now the search and planning problem has been realised, and a potential solution in PDDL has been found, the use of AI in software optimisation will be reviewed to examine the current state-of-the-art methods to increase the performance of signal processing.

3.4 Investigation of Software Optimisation using Artificial Intelligence

Optimised software executing across single and heterogenous platforms requires consideration of many parameters at either runtime or compile time. Researchers has proposed different approaches for the heuristics used in optimisation, such as latency, throughput, and energy consumption and indeed the processes used to optimise the software itself (Memeti, Pllana, Binotto, Kołodziej, & Brandic, 2018). The main methods for software optimisation processes can be split into two categories:

1. Offline training data is used to optimise new, unseen signal processing.
2. Offline real data is used to optimise itself.

The first of these was implemented by Z. Wang and O'Boyle (2009), using training data to optimise parallel CPU execution (OpenMP) using a feed-forward Artificial Neural Network (ANN) and Support Vector Machine (SVM) to solve their scheduling problem. They used an offline supervised learning scheme presenting the ANN with program features and desired mapping decisions for the parallel threads. By using this training data and a set of specific heuristics, they trained their ANN to be able to predict the optimal number of threads for a new, unseen problems with up to an 8x performance increase relative to their sequential alternatives.

Dastgeer, Li, and Kessler (2013) also used offline training data to help optimise new problems, however, unlike Wang et al., Dastageer et al. was optimising CPU-GPU heterogenous multicore systems. They proposed a system named *PEPPHER*, an annotated software module using XML documents to specify parameters and performance metrics. *PEPPHER* was given a set of algorithms, matrix-multiplication, sorting, path finding, and back propagation, at different sizes and records the performance, it can then use this training data to predict the execution time of a new unseen problem. However, even though the author states the reduction in time that *PEPPHER* provides against other training algorithms, they do not state how accurate the model is at predicting execution times.

The second of the above-mentioned categories, offline optimisation using real data, has been investigated by Grewe and O'Boyle (2011). They selected 47 benchmarks including matrix multiplication, convolution, and a program to calculate the coulombic potential, and executed them on a CPU (2x Intel Xeon E5530) and a GPU (ATI Radeon HD 5970) using OpenCL. Using the execution time as their heuristic, they developed a machine-learning predictor to best "partition" the corresponding program, by partition they are scheduling some functions for the CPU, some on the GPU or distributed over the two. In their results they state they achieved a 57% performance benefit over other state-of-the-art scheduling/partitioning methods, but more importantly a minimum improvement of 55% by using partitioning instead of a single architecture.

Another investigation into optimising for real-time processing is by Albayrak, Akturk, and Ozturk (2013). They used a greedy mapping algorithm to generate a kernel map to minimise the total execution cost, by splitting functions between the CPU and GPU where it was advantageous to do so. Like Grewe et al., Albayrak et al. used OpenCL to process a set of benchmarks on both the CPU (Phenom II X6) and GPU (GTX 460). They first processed the benchmarks on the CPU and GPU separately then using their greedy algorithm to decide as to which architecture provided the optimum execution on a function-by-function basis. They compare their results to a commercially available Mixed-Integer Programming (MIP) based solver and while their results were slightly lower than the MIP, they did achieve an up to 2x speedup compared to their CPU-only or GPU-only executions.

3.5 Review of Artificial Intelligence to Optimisation Manufacturing

Previously the use of AI to optimise the processing side of the data has been discussed, however there is significant research in the field of optimising the manufacturing process itself. While not directly related to this research, briefly reviewing this area at a high level shows the impact AI is having throughout the whole manufacturing industry.

Leo Kumar (2017) outlines the two main areas that AI is impacting on the manufacturing process, Computer Aided Processing Planning (CAPP) and Process Planning and Scheduling (PPS). CAPP is the bridge between Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) and is used to achieve more effective use of manufacturing resources. The use of expert systems to help optimise the manufacturing process is nothing new and was first discussed back in the 1980s by Weill, Spur, and Eversheim (1982). Whereas PPS concerns itself with organising, managing, and maximising work and workloads of equipment, supplies, and resources. As this research's focus is process planning and scheduling, it is this area of the literature that this sub-section will focus on.

Zhou, Tang, Zhu, and Wang (2020) presented an AI scheduler used for online and dynamic scheduling of manufacturing jobs in a smart factory. Reinforced learning provides the framework with self-organizing and self-learning capabilities under uncertainty, allowing the AI scheduler to learn and optimise development schedules for multiple objectives. The proposed AI scheduler employs a manufacturing value network (MVN) to estimate state-action values from multi-dimensional sensor data of manufacturing items, and then performs real-time scheduling based on available machines and pending jobs. Their results show that their AI scheduler not only enhances multi-objective performance metrics in scheduling of production jobs, but also effectively deals with unexpected events in manufacturing systems (e.g., urgent workorders, machine failures).

Wally et al. (2019) presented an automatically generated production plan from a production system model that describes production resources, their availability, and their capability, as well as the materials to be handled and produced. To automatically generate this production plan they used PDDL which was able to schedule the use of four, 6-axis robots to manipulate materials. Wally et al. not only wanted to determine the most optimum path, but they also wanted to use durative actions to provide a minute-by-minute schedule for the operation. Their preliminary results, although successful show how problematic durative actions become, with their most optimum schedule taking 4 hours to find, however they did use a computer almost a decade old which will have impacted their possible results, for what was a relatively simple Lego pickup car comprising of three parts.

Another use of PDDL within manufacturing to control robotic arms is by Huckaby, Vassos, and Christensen (2013). In this paper they used an extension of UML, System Model Language (SysML) to model the various operations that can be performed on an assembly line. They then took the SysML model and convert it into PDDL terms for the robotic arms, material parts, assembly stations, and manufacturing tools. Their case study was the preparation of a car-door part for an assembly into a car using a planner called OPTIC capable of PDDL3 domains, however no durative actions were used. Although the author does not state how long these solutions took to find, they successfully found two different solutions depending on different constraints on the reach of the robotic arm, showing that the planner is able to deal with configuration and process change.

Finally, AI, and more specifically PDDL, was used to calibrate machine tools (Parkinson, Longstaff, Fletcher, Crampton, & Gregory, 2012). They proposed two models and test them again both an academic expert and an industrial expert. Their proposed models are for sequential calibration and potential concurrent calibration of a fixe-axis gantry machine with the aim to reduce machine downtime. Previously to their PDDL implementation, Parkinson et al. proposed a Hierarchical Task Network (HTN) method and compared it to their current PDDL results. The academic expert's solution would take 12.5 hours and the industrial expert's plan was 11.5 hours, each algorithm was then given ten minutes on a AMD Phenom II X4 970 running Ubuntu 11.04 to find a solution to the given problem.

They found that the HTN did provide a 45-minute benefit over the academic expert, however the solution was still suboptimal compared to the industrial expert. However, the PDDL implementation was 53 minutes shorter than the previous HTN provided plan, and 38 minutes shorter than the industrial expert. Furthermore, the second, concurrent model, was 72 minutes shorter than the best plan (industrial expert), results of which are illustrated in Figure 3-24 below.

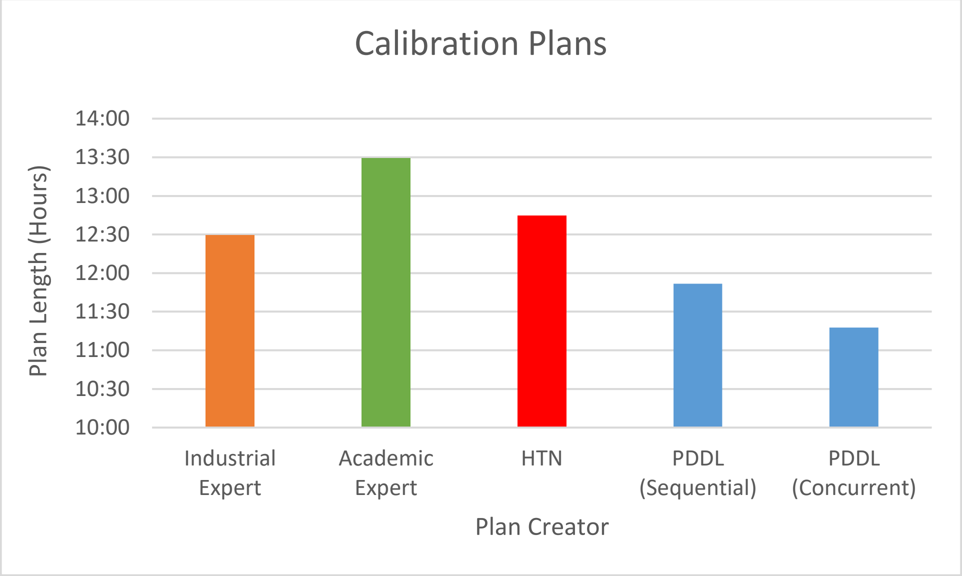


Figure 3-24: *Calibration Plan Time Comparison from Parkinson et al.*

3.6 Conclusion

Automated planning and scheduling can be traced back to the 1960s and 1970s. In its infancy, new search methods were being developed, each tacking an issue from a different angle with benefits for certain applications. It was not until informed search with heuristics came along that the use of AI in this field started to develop. Since this point AI has been used to optimise everything from calibrating machines, scheduling productions lines, and optimising the software that manages everything in between. The use of PDDL and its ability to optimise for a given goal (heuristic) given a set of requirements, whilst having the ability to change and adapt to the various parameters it has access to is beneficial for this research.

The literature reviewed has not only shown that this is possible but has found solutions that are more optimised than those generated by academic and industrial experts. PDDL is able to take timing data mined from the execution of signal processing and generate a solution instructing the software on which architecture is the local optimum for each individual function to achieve an overall optimum execution. It will also be able to adapt when the timing data changes, perhaps due to hardware changes, or a change in the signal processing requirements. It manages this by simply re-processing the domain with updated information and will provide an up-to-date solution that is optimum for the requirements of researchers and industry.

4 Case Study: Dispersed Reference Interferometry (DRI)

As an initial case study, early in this research, Dispersed Reference Interferometry (DRI) is undertaken to allow the author to become familiar with some of the signal processing requirements of optical instruments and learn about the various configurability and optimisation challenges that laid ahead. This chapter evaluates the challenges of providing an optimised library of signal processing functions for use by researchers and industry that also includes hardware accelerated algorithms to increase the overall performance.

4.1 Introduction to Dispersed Reference Interferometry

DRI is a single point interferometric measurement technique that uses a broadband super-luminescent diode (SLD) to measure surfaces at nanometre resolution. DRI was developed with the aim of being used for on-machine metrology as a remote fibre probe could be used to separate the measurement apparatus from the manufacturing line. The DRI architecture is based on a Michelson interferometer which has a large amount of dispersion applied to the reference arm with a separate measurement arm (Figure 4-1). As the measurement arm (M1) moves in the x-axis, the centre of the interferogram shifts left and right, using signal processing it is possible to calculate the distance to the measurand based upon where the centre of the interferogram lies (Figure 4-2). The interferograms are captured by a CMOS sensor spectrometer and then analysed to calculate the measurement (Martin & Jiang, 2013). In order to achieve an absolute high-resolution measurement, two techniques are combined, one achieving the absolute position and the other accomplishing the high resolution (Williamson, Martin, & Jiang, 2016).

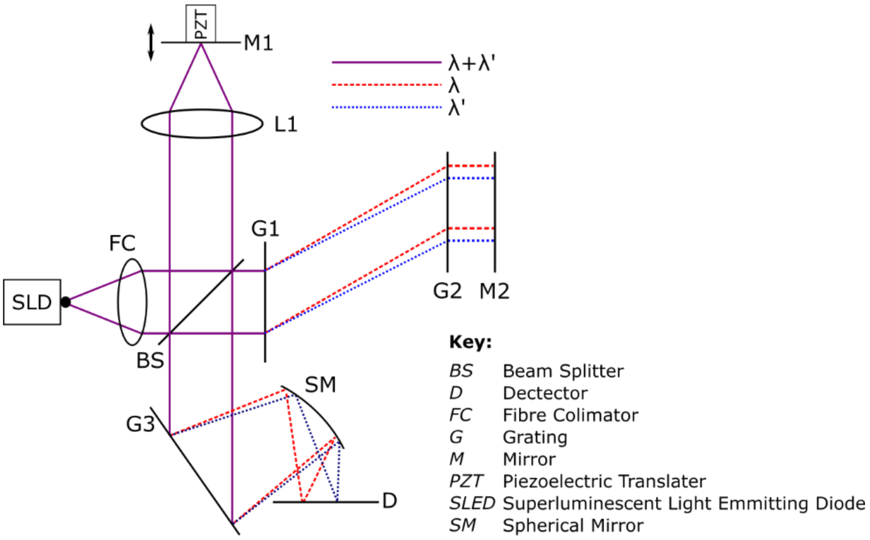


Figure 4-1: *Schematic diagram of DRI bulk optics interferometer based upon a Michelson Interferometer (Williamson, 2016)*

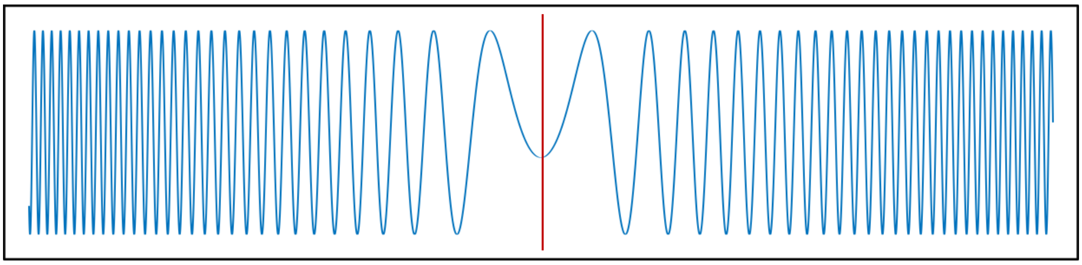


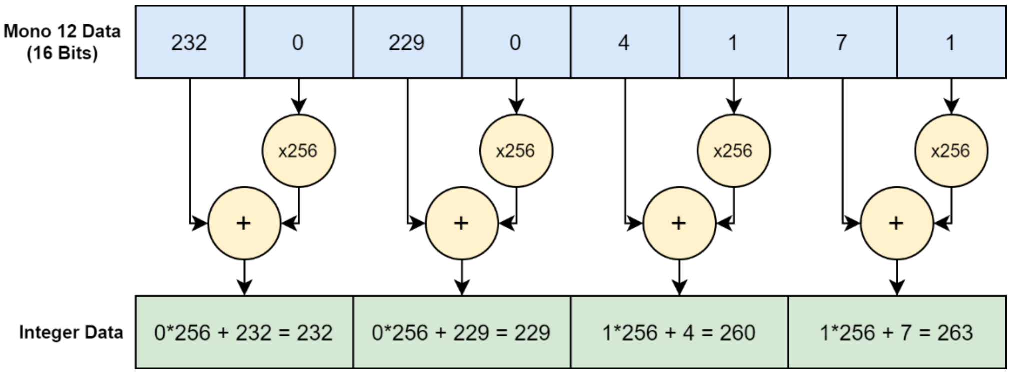
Figure 4-2: *Sample Interferogram generated in MATLAB. The red line notes the point of symmetry.*

The spectral interferograms captured by the detector have a point of symmetry directly linked to an absolute surface position providing an axial resolution of 279 nm over a range of 285 µm. An improvement to this resolution can be achieved through the process of template matching (Williamson et al., 2016). Template matching compares the captured interferogram against a set of pre-calculated templates resulting in a much-increased level of resolution (0.6 nm) across the range.

4.1.1 Low Resolution, Absolute Measurement

The detector used for the DRI is a Basler 8192-pixel 12-bit line sensor (Balser, 2020), giving a value of zero to 4096 for each pixel across two bytes of information. Once this is data is captured it is passed through the following processes:

1. The first processing step in the signal processing chain is to combine the two bytes of data into a single pixel value, this is also converted into a variable type such as an integer (int) or a floating-point type (float) (Figure 4-3). Once the signal is in a desirable format, it is passed through a number of signal processing steps to achieve the low-resolution absolute position.



Mono 12 Data (16 Bits)	232	0	229	0	4	1	7	1

Integer Data	0*256 + 232 = 232	0*256 + 229 = 229	1*256 + 4 = 260	1*256 + 7 = 263

Figure 4-3: *Converting Camera Data into computable values*

2. The derivative of the signal is then computed by calculating the difference between each pixel. As the derivative approaches the centre of the interferogram it tends towards zero. This central point is the interferogram's point of symmetry (Figure 4-4).

3. Once the differential is calculated, it is passed through a low-pass filter using a moving average algorithm that averages each pixel with its nearest neighbours. The width of this window can be increased or decreased as required to remove background noise in the original signal.
4. The filtered signal is then convolved with itself (auto-convolution) to find the symmetrical point of the original signal.
5. Finally, the minimum point is found giving the 279 nm resolution. Additionally, another step is introduced to increase the resolution further by fitting a polynomial to the minimum point and finding the minimum point of the polynomial.

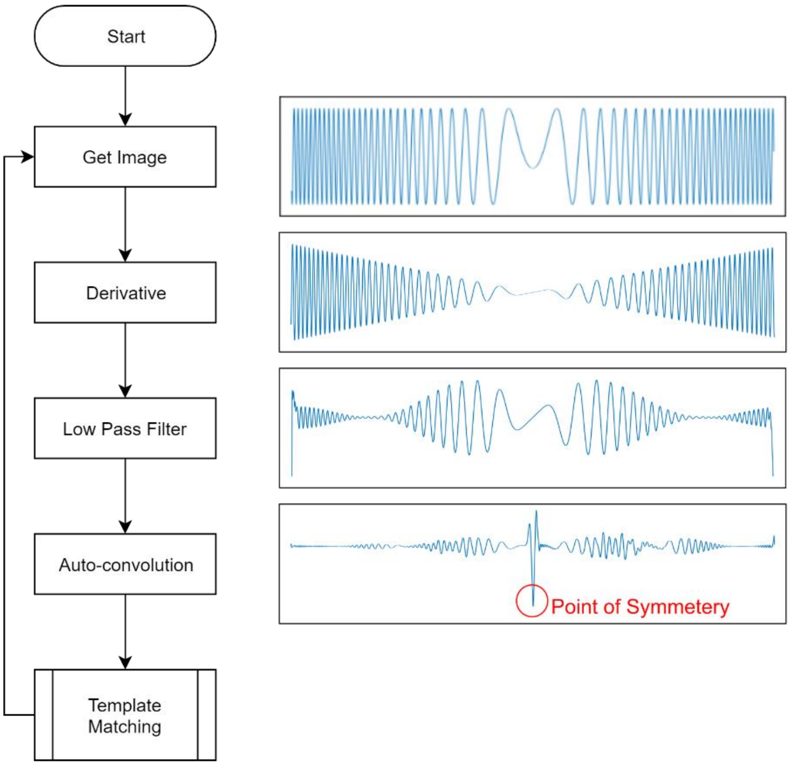


Figure 4-4: *Signal processing to determine the relative position*

The calculated symmetrical point is a vertical low-resolution, absolute measurement, with the pixel resolution being 279 nm, this can be improved by the polynomial fitting. This resolution is not appropriate for many of the on-machine applications that this technique was aimed at, therefore further signal processing to extract phase data to achieve higher resolution is required (Williamson, 2016).

4.1.2 High Resolution, Relative Measurement

The method used to achieve a high-resolution result is template matching, whereby the captured interferogram is compared against a collection of simulated different templates to ascertain which it most closely resembles. This is achieved in several steps:

1. The interferogram is windowed around the point of symmetry, using itself and the 700 pixels either side resulting in a new windowed interferograms 1401 pixels wide.
2. A correlation algorithm is used to compare the new windowed interferograms with a collection of 800 templates.
3. An array of correlation values is then generated, and the first maximum (highest correlation) point is then found.
4. A second-order polynomial is then fitted to the maximum point and the maximum value of the polynomial is found and is used as the wrapped phase value (this is a relative position).

By using 800 simulated templates, each 1401 pixels wide, a resolution of less than 1 nm can be achieved, much higher than the absolute position. Unfortunately, as this high-resolution technique only provides a relative position, it needs to be combined with the absolute measurement in order to have both, high resolution and absolute positioning.

4.2 Signal Processing Requirements of Dispersed Reference Interferometry

Previously to this research, a 2048-pixel CCD sensor was used and was processed on-line in LabVIEW. LabVIEW enables researchers to quickly create signal processing and receive initial results, however this was at a performance cost. LabVIEW was processing between three and five frames per second (to achieve a high-resolution, absolute measurement). Not only was this not fast enough, (the target being 10 kHz), but a wider sensor was needed to increase the optical performance. The increase in data being acquired and processed has a negative impact on processing speed. Therefore, moving away from LabVIEW was the only option. In order to obtain processing speeds closer to the 10kHz target, some form of hardware acceleration would be required.

This became the starting point for this research; to develop a piece of software that provides a similar level of performance of bespoke software while keeping the configurability and ease-of-use of software such as LabVIEW. It also needs to utilise any hardware acceleration if it is available such as GPUs, and parallel processing using multiple CPU threads.

4.3 Investigation into the Software Requirements for Signal Processing

The software requirements were mainly influenced by the signal processing requirements. The computation needed to be fast, be able to provide multi-core and/or GPU processing and support polymorphism. The software should also provide an easy-to-use user interface that is intuitive to use and allows the user to customise the signal processing for their requirements.

4.3.1 Computation Requirements

The main constraint of a programming language is its capability of both multi-core and GPU processing. Having earlier evaluated both CUDA and OpenCL, CUDA, with its propriety optimisations between hardware and software, should provide the best acceleration despite its hardware restrictions. This selection therefore significantly reduces programming languages available for programming the serial code in. CUDA provides a Software Development Kit (SDK) for both C/C++ and Python and has been ported to Java by a third-party called JCUDA (Yan, Grossman, & Sarkar, 2009).

Python is a high-level programming language, this refers to the level of abstraction away from the details of the computer, whereas C/C++ require more intricate knowledge of hardware and are mid-level languages (Figure 2-4). Therefore, Python excels in its ease-of-use but due to its high-level nature it has a negative impact in performance. Whereas with C/C++ or Java, it is the other way around. Java is a mid-level language like C, but was designed to be easier to use, and therefore has had features removed to simplify the experience, such as memory pointers, type definitions and templates. However, Java does have other useful features such as run-time interpreter, rather than C's compiler, event-driven programming and native multithreading.

Contrasting between C++ and Java, both provide different useful features, however, due to the author's previous experience in C++, initially it was the chosen language whilst continually assessing its capability for the processing needed. This allows the use of CUDA in its native supported form, hopefully providing an excellent level of acceleration without having to rely on third-party libraries. As C++ does not have parallel programming natively supported, a third-party interface must be used to have a multithreaded workload.

OpenMP is an Application Programming Interface (API) that supports multithreading in C and C++ on Linux, Windows and macOS (Dagum & Menon, 1998). Multithreading capability is important as nearly all desktop processors sold today have multiple cores on a single chip. Although the speedup for multiple cores is not linear, and varies depending on thread saturation rate, utilising these extra cores can provide a significant speed increase for no additional hardware cost (Figure 4-5). Not only does the main thread fork into multiple threads, OpenMP also automatically splits the memory and can create new memory only accessible by each individual thread allowing easy memory management.

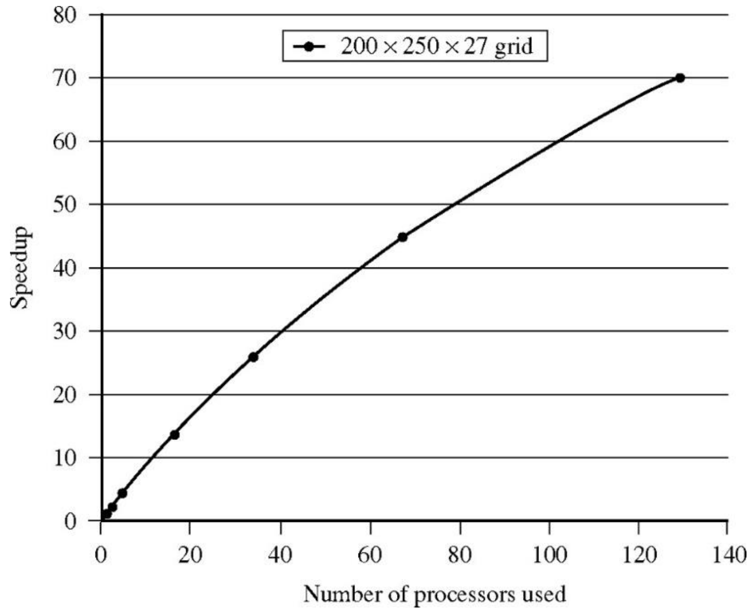


Figure 4-5: *Note. Reprinted from "Parallel Programming in OpenMP" by Chandra, R., 2001, p.3, San Francisco, CA: Morgan Kaufmann Publishers*

4.3.2 User Interface (UI) Requirements

A Signal Processing Engine (SPE) needs an easy-to-use UI to be able to quickly develop the signal processing chain that users require. The user interface needs to provide three main features:

1. The ability to configure the SPE.
2. Graphically display the data output by the SPE.
3. Provide a method for controlling actuators.

Unfortunately, there is typically a trade-off between being easy to use and providing a usable level of functionality. For example, MATLAB and LabVIEW provide flexibility, but MATLAB requires the user to learn an additional coding language, while LabVIEW has a similar learning curve for its associated UI. When creating a UI it for researches and industry, it should not require them to learn a new programming language but must allow them to harness the signal processing available in the SPE.

The first option is to include the UI as part of the SPE program, but Microsoft Foundation Class (MFC) is an outdated framework and requires third-party libraries to achieve some of the above outlined features, although it would be advantageous to use CUDA to render the outputs. There are non-Microsoft design tools available, such as Qt, but these require paid licenses, and these were outside the scope of this research. Therefore, early on in this research, the decision was taken to use C# and .NET framework to develop the UI, enabling an easy to use, modern looking UI.

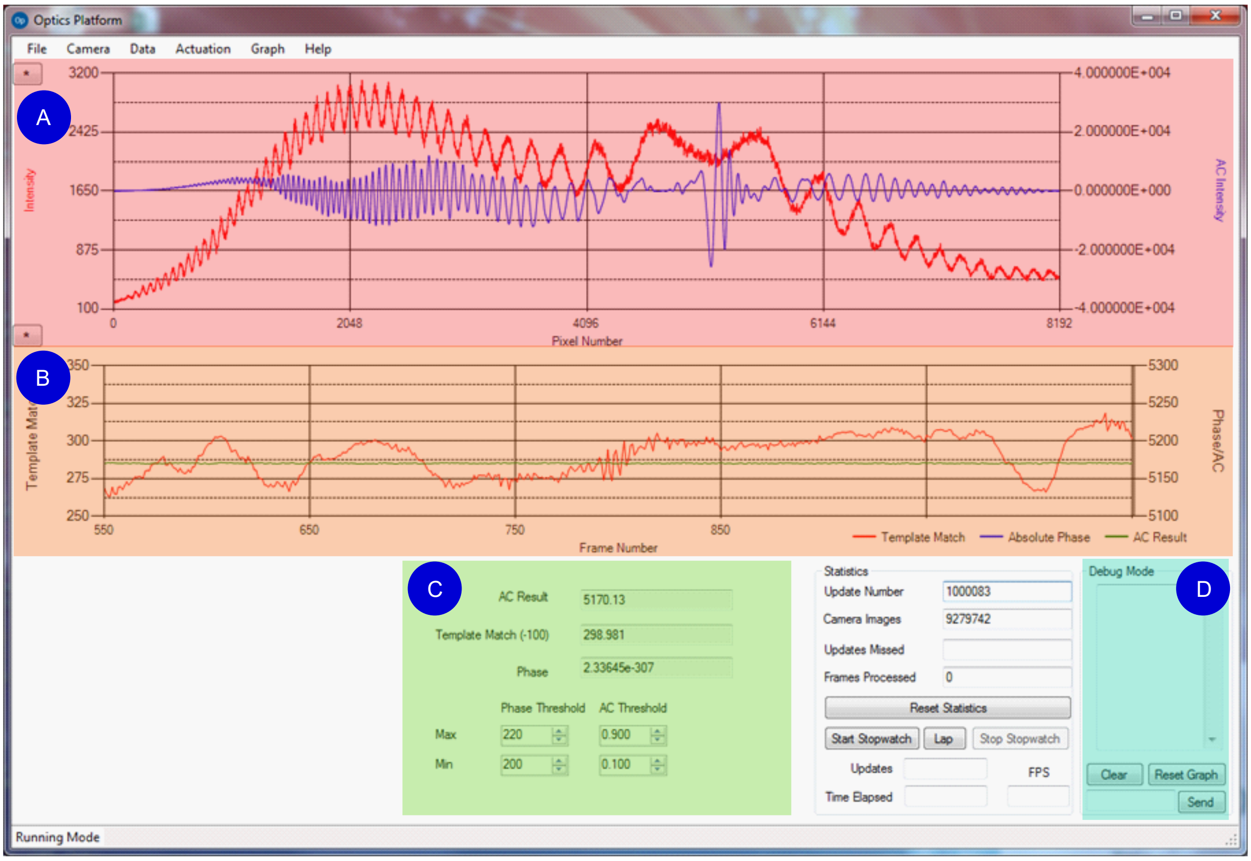
.NET Framework by Microsoft, is made up of tools, programming languages, and a library for many different applications, and can be programmed in C#. C# is an objected oriented programming language also created by Microsoft in 2000, designed to be simple, modern, and general-purpose. Part of the .NET Framework is Windows Form Application, a GUI class library providing an easy interface to develop windows applications for desktops and tablets. C# is significantly more suitable for UI elements rather than C++/MFC and has optimised pre-built libraries for buttons, textboxes, graphs, datasets etc. and does not rely on third-party libraries. However, the need for two separate programs raises data access difficulties between the two processes and any configuration data created needs to be sent from the UI to the SPE and any data to be shown to the user is transferred to the UI. This is covered in more detail in chapter 4.5.

4.3.3 Summary

C++, CUDA and OpenMP were selected as the programming languages for the SPE. They retain performance whilst allowing a useful level of customisability. The UI has been programmed in C# using Windows Forms in a separate program to harness the benefits that the .NET frameworks provides for the graphical side, while maintaining the speed the low-level nature C++ provides. This should allow for the “best of both worlds” approach while hopefully minimising any overhead created by moving data between the two processes.

4.4 Initial Proposal of User Interface Design

The first iteration of the UI was specifically designed for the DRI, this allowed the author to gain experience programming in C# and .NET framework. Throughout the design of the UI, features were designed to be application non-specific, e.g. graphs (Figure 4-6a), others, e.g. the data readout (Figure 4-6c), are specific to the DRI but are available in a more generalised way in future iterations.



AC Result 5170.13

Template Match (-100) 298.981

Phase 2.33645e-307

	Phase Threshold	AC Threshold
Max	220	0.900
Min	200	0.100

Update Number 1000083

Camera Images 9279742

Updates Missed

Frames Processed 0

Updates

FPS

Time Elapsed

Figure 4-6: User Interface v1

The UI has three main purposes, data visualisation, SPE configuration and actuator control. Data visualisation for the DRI consists of three outputs, the raw interferogram, the auto-convoluted signal (absolute position), and the correlation value (relative position). The interferogram and AC signal can be plotted on the same graph (Figure 4-6a), the AC result (the minimum point of the AC signal) and the template match result are both plotted against time so they can also share a graph (Figure 4-6b). The graph data is imported from the SPE by the UI, to save time not every data point is plotted on the axis (Figure 4-6a), instead the UI determines the screen resolution and then decimates the data by the nearest factor smaller than the ratio (Equation 4-1). This data reduction allows the graphs to be drawn quicker and more frequently, while still showing more points than the screen resolution can reproduce, so is unnoticeable by the user. The graphs also auto-scale the first time they plot data in both the X and Y axis', and when plotting against time it shows the last 500 results (this may need to be a factor of the data rate in future).

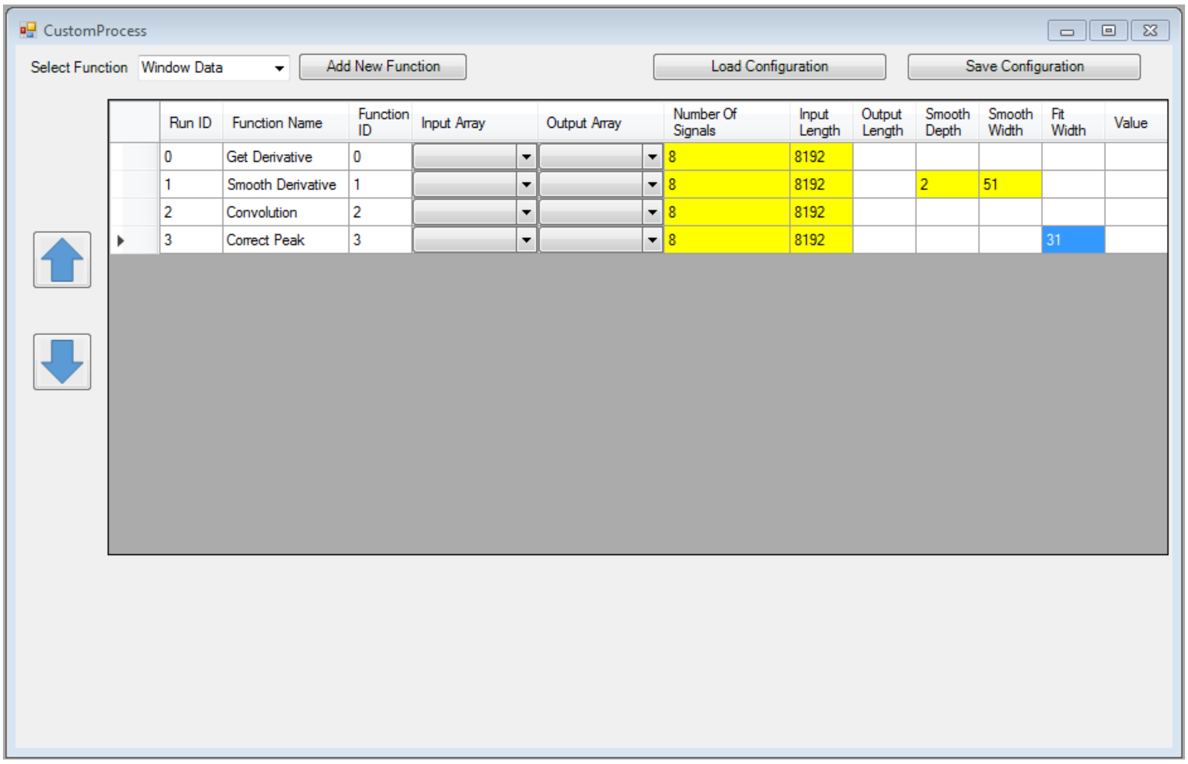
$$DecimationRatio = \left\lfloor \frac{DataLength}{ScreenResolution} \right\rfloor$$

Equation 4-1

If the user requires data to be saved, this can also be done through the UI, by selecting only the outputs required. To reduce wasted time, the different signals and data points that are being displayed can be individually turned on and off to only save the data that is required. Once the outputs are chosen, they can be saved in three different modes:

1. Save every *n* frame for *m* number of frames.
2. Save every frame for *n* seconds.
3. Save every frame until stopped.

The SPE configurator (Figure 4-7), allows users to select the functions they require and set the function parameters. As previously mentioned, this is currently designed for the DRI only, however, with expansion and generalisation in mind, yet currently the only parameters that can be set are specific to the DRI and are hard coded into the UI. All the available functions are loaded into the UI at runtime, once added to the run list, the configurable parameters (highlighted yellow) can be set, along with the input and output arrays. Once all the required functions are added, and placed in the correct order, the configuration can be saved and used by the SPE to execute the chosen signal processing.

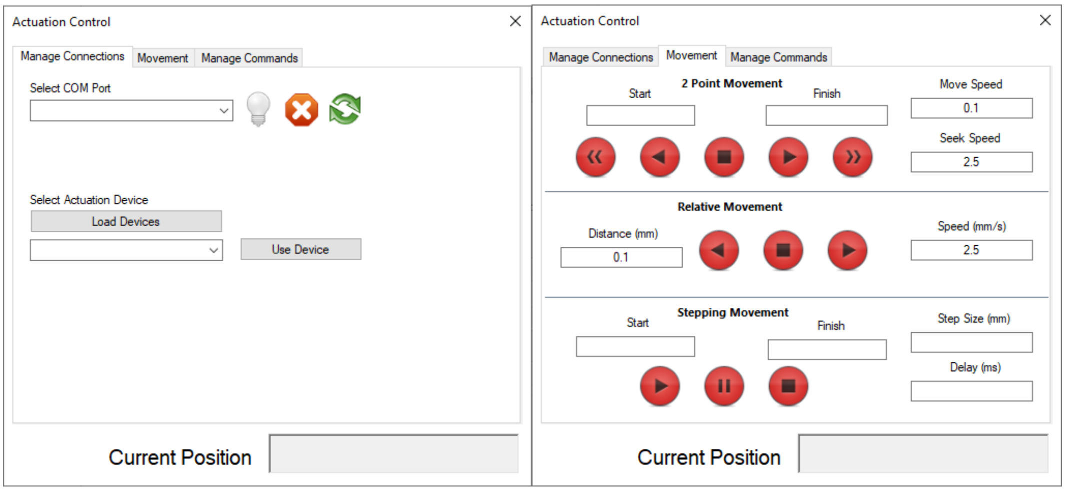


Select Function: Window Data

	Run ID	Function Name	Function ID	Input Array	Output Array	Number Of Signals	Input Length	Output Length	Smooth Depth	Smooth Width	Fit Width	Value
	0	Get Derivative	0			8	8192					
	1	Smooth Derivative	1			8	8192		2	51		
	2	Convolution	2			8	8192					
▸	3	Correct Peak	3			8	8192				31	

Figure 4-7: Function Configuration v1

Finally, the last feature of the UI is the actuator control (Figure 4-8). This allows users to move mechanical stages to scan objects. Currently this is only setup to move in one axis, but multiple of these could be used together to create 2D/3D movement.



Move Speed 0.1
Seek Speed 2.5
Distance (mm) 0.1
Speed (mm/s) 2.5

Figure 4-8: *UI Actuator Control Panel*

Several commands have been added to the UI to communicate with the actuation devices. As the actuation devices use General Command Sets (GCS), a translation Look Up Table (LUT) has been added so that when an action is called, it is translated to the command for the actuation device being used. These are shown in Table 4-1. The actuators can be moved either linearly from two points or stepped by a set distance from a start to an end point. When recording data, the actuation position can also be saved so the data can be matched back to the actuator's position at the time of capturing the data.

Table 4-1: *Look Up Table of General Command Set (GCS) for different actuators.*

UI Command	PI E709	Newport SMC100
Go To	MOV Z *N*[1]	1PA*N*
Move Step	MVR Z *N*	1PR*N*
Get Position	POS?	1TP*N*
Set Speed	VEL Z *N*	1VA*N*
Stop	STP	TP
Terminators	\n	\r\n

4.4.1 Future Improvements

The basics for the SPE configurator have been established successfully, but modifications are needed to adapt to the increase of permutations available when the function library increases. The user will need to create arrays to enable better data flow throughout their configuration. There might be occasions where multiple arrays are used, rather than a single input and single output. The configuration parameter names need to be adaptable for each individual function, therefore knowledge of the function from the SPE needs to be imported into the UI so the right text can be displayed. This needs careful consideration, for the data to be in a readable format by both the SPE

[1] N is the decimal value sent with the command, the units for this are set by the device.

and the UI and to not simply duplicate any data, as this would cause memory read and write issues if the data did not match correctly. Finally, as the number of possible functions, arrays and the function list increase in size, the more complicated the UI may become, therefore a drag-and-drop user interface may be beneficial for the configurator.

4.5 Exploration of Data Management Between Two Processes

The configuration data is managed between the UI and the SPE though files saved in Extensible Markup Language (XML). XML is used for two reasons, firstly it is a user readable format which in this initial stage is useful for debugging possible problems and errors; secondly, C# structures can be serialised directly into XML removing possible programming errors when converting between stored data and XML format. The configuration XML file contains the order of functions, which arrays are used for each function and the data points that go with them in Figure 4-7. Upon completion of the configuration, it is saved, and the user can start the SPE, which has an argument passed to it detailing the location of the configuration file for it to be imported.

The serialisation of the C# data is very useful during the export process, because if the data was incorrectly transferred between the two separate processes, it would be very likely that the functions would not work. The wrong array could be used, the wrong data points, or even worse, e.g. an array of length 100 is asked to record 200 values causing a memory overflow along with a whole host of other problems. Therefore, when the SPE is importing the data, it needs to perfectly replicate what the user requested and what the XML file recorded. Unfortunately, there is no inbuilt method in C++ for importing C++, so custom functions were created using a third-party XML parser (TinyXML, 2015) to read in the data.

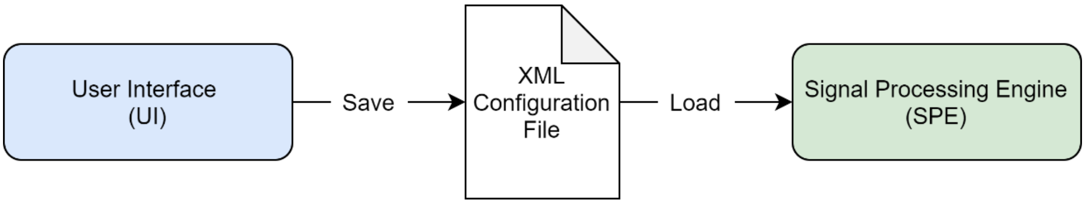


Figure 4-9: *Configuration Files*

The problem with this first version of the import/export system is the class that holds the data, the class has a member for every parameter used across all the functions that exists for the DRI (Figure 4-10). Once the library includes more functions with many different parameters, this class will be very complicated and confusing to maintain and therefore needs improving. However, the import configuration system does work correctly, and SPE imports the correct Signals and DataPoints into the correct function, with the signal processing executing as expected. Finally, to reflect any changes made in the SPE, both the configuration manager and also in the exporting of the configuration file will need to be modified to maintain consistency.

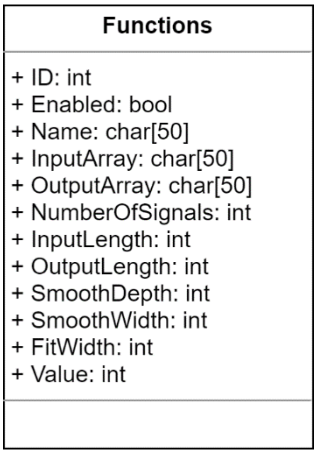


Figure 4-10: *Function Class that data is imported into*

Processed data is transferred from the SPE to the UI, this is done through a process called Inter-Process communication (IPC). There are several ways IPC can be implemented on Windows, these include standard files, memory mapped files, and Named pipes. Standard files such as comma separated value (CSV) can be used to store numerical data with a comma placed between each data point (Figure 4-11) stored on a hard drive.

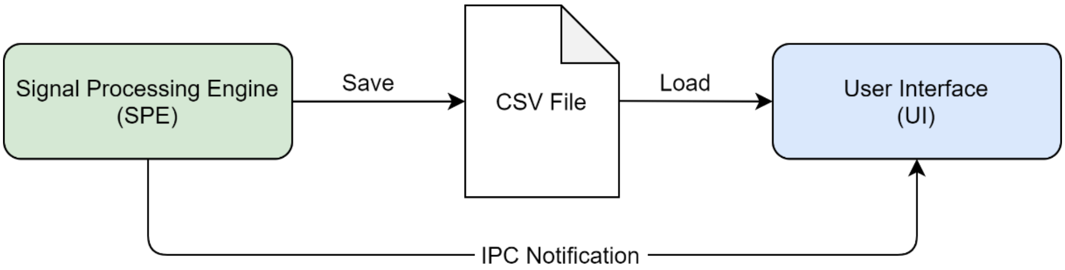


Figure 4-11*: Inter-process communication using CSV files*

CSV allows for easy importing and exporting of data between two processes as both C++ and C# have inbuilt functions for dealing with CSV data. However, this is very inefficient as for each data point a comma is needed (one byte), increasing the file size by 12.5% for 8-byte double precision and 25% for 4-byte integers. Even using this inefficient method, the UI needs to know that there is a file ready to be read and where that file exists. Furthermore, the files were originally being saved to a mechanical hard disk drive (HDD) further limiting performance, this has since been switched to a solid-state drive (SSD) which has seen significant improvements. However, the standard file format is still an issue and needs improving before larger sets of data can be outputted efficiently.

The other type of IPC mentioned, Named Pipes is a First-In-First-Out (FIFO) communication method. Between the two processes, one creates the pipes (the parent), while the other process (the child) searchers for pipes to connect to. Each of these pipes are simplex communication but can be full-duplex by combing two together one in each direction, both created by the parent. This allows data to be saved in a CSV file and then a notification message can be sent down the pipe from the SPE to the UI to say the data is ready to be read and the location of said data. This removes any conflicts where the SPE may be writing to the data file at the same time the UI is attempting to read from it, as the UI can also let the SPE know when it has finished reading and that it can write new data to the file.

4.6 Signal Processing Considerations

There are three processing methods that have been implemented, serial CPU execution, parallel multi-core CPU execution and CUDA GPU execution. Each has its own set of requirements and challenges and requires the ability to be able to switch between all three to evaluate which of the three is best for each given algorithm.

4.6.1 Serial Signal Processing

Firstly, the camera image needs to be acquired, the camera used for the DRI is the Basler racer raL8192-12gm which captures an image 8192 pixel wide and a single pixel high. Basler provides an SDK which allows developers to communicate to the device capturing additional properties such as image number and timestamp. This captured image can be acquired in three different formats, Mono8, containing eight bits of data per pixel, Mono12, containing twelve bits of data across two bytes and also Mono12 packed, containing three bytes of data for each two pixels (Figure 4-12). The format chosen moving forward was Mono12, therefore capturing 16 KB images, the advantage of this was Mono12 packed required longer to unpackage the data the benefit it provided in camera data rate. However, if only image capturing is required, using Mono12 packed would yield a smaller latency and increased frame rate, however, it isthe signal processing is the bottleneck for the DRI. Finally, in the next section, multithreading is introduced rendering the time taken to acquire a camera image irrelevant, therefore the faster signal processing is more advantageous.

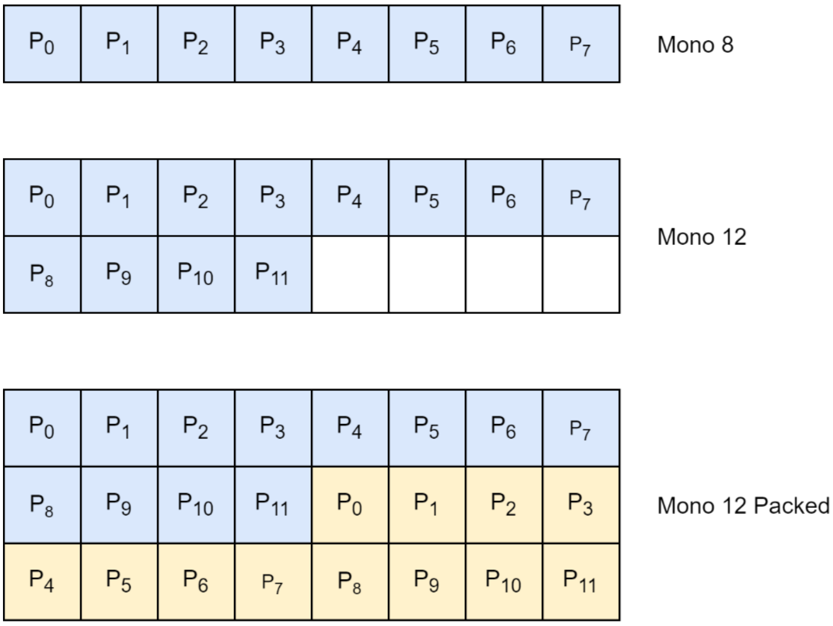
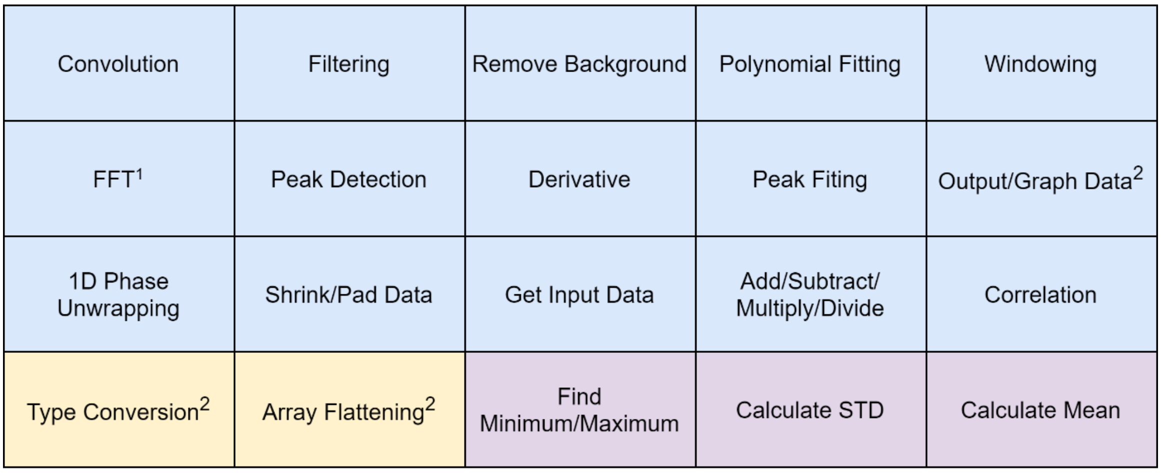

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
Mono 8
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11}
Mono 12
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11} P_0 P_1 P_2 P_3
P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11}
Mono 12 Packed

Figure 4-12: *Camera Data Formats*

Each of the functions that were required for DRI measurement were defined by Williamson (2016), these fulfilled the requirement to be able to measure a high-resolution absolute position by combining data from a convolution and correlation algorithm. All the algorithms were programmed and verified from pre-existing MATLAB code or freely available open-source code, the algorithms created for the DRI are shown in Figure 4-13. The FFT library, High Speed Fast Fourier Transform (HSFFT), is an open-source algorithm for C++ and was implemented with changes made to enable its use with arrays rather than vectors (R. Hussain, 2016). However, this library is replaced by FFTW later for performance benefits.



Convolution	Filtering	Remove Background	Polynomial Fitting	Windowing
FFT[1]	Peak Detection	Derivative	Peak Fiting	Output/Graph Data[2]
1D Phase Unwrapping	Shrink/Pad Data	Get Input Data	Add/Subtract/ Multiply/Divide	Correlation
Type Conversion[2]	Array Flattening[2]	Find Minimum/Maximum	Calculate STD	Calculate Mean

[1] 3rd Party Library *[2] CPU Only*

Figure 4-13: *Initial library of algorithms programmed for the DRI Case Study*

Once all the signal processing has been programmed it is compiled, however, before any data can be transferred to the UI a couple of properties need to be set. Firstly, the signals that the user requires displaying need to be configured, these are set in the UI and imported into the SPE through the XML configuration file loaded at the start. Secondly, all the different outputs need to be packaged together so that only when all the data is ready that the notification is sent to alert the UI to read the files. A function, *OutputData*, takes any signals that the user requires graphing in the UI and caches them until the file is available to be written to, caching these signals and only writing when the files are free prevents data not reaching the UI. However, as multiple frames may be cached, only the latest signal is displayed as data for the DRI is processed faster than human eyesight, so the graph drawing is limited to 25 FPS to reduce superfluous plotting that wastes time.

Once all the serial processing algorithms are programmed and tested, the next step is to investigate if any performance increase can be found by utilising the multiple CPU cores available through parallel programming.

4.6.2 Parallel Signal Processing

The main computer being used for this research contains an Intel i7-4790 consisting of four cores and eight threads. There are two ways that the SPE can utilise the CPU's multithreaded architecture. Firstly, each signal processing function could launch eight threads and split the workload between them equally, this would decrease the execution time of each function individually. However, creating threads (forking) and ending them (joining) has an overhead, and if the function is only short the overhead can cost more than it benefits from (Figure 4-14). Although Amdahl's law (Amdahl, 1967) is strictly associated with the parallel capability of a function itself, this limitation is based on the same principle.

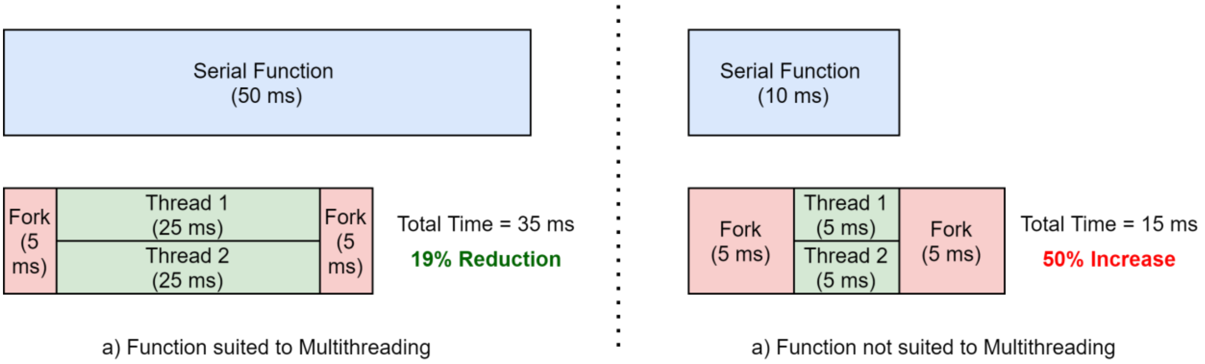


Figure 4-14: *Multithreading example showing two processes being forked from one to two threads*

Depending on the signal processing function, or the size of data processed by the function, parallel processing may not be advantageous in the forking/joining method. There is, however, a second method to use multiple threads, rather than forking and joining threads for each function, specific functions can be run on separate threads for the entire life of the program. Looking at Figure 4-15, running the program serially would take 80ms, that can be improved by forking threads for the main processing saving 15ms, almost 20% of the total execution time. However, an extra 10ms has been introduced due to the time taken for threads to be created and destroyed, reducing or removing this overhead entirely would be advantageous.

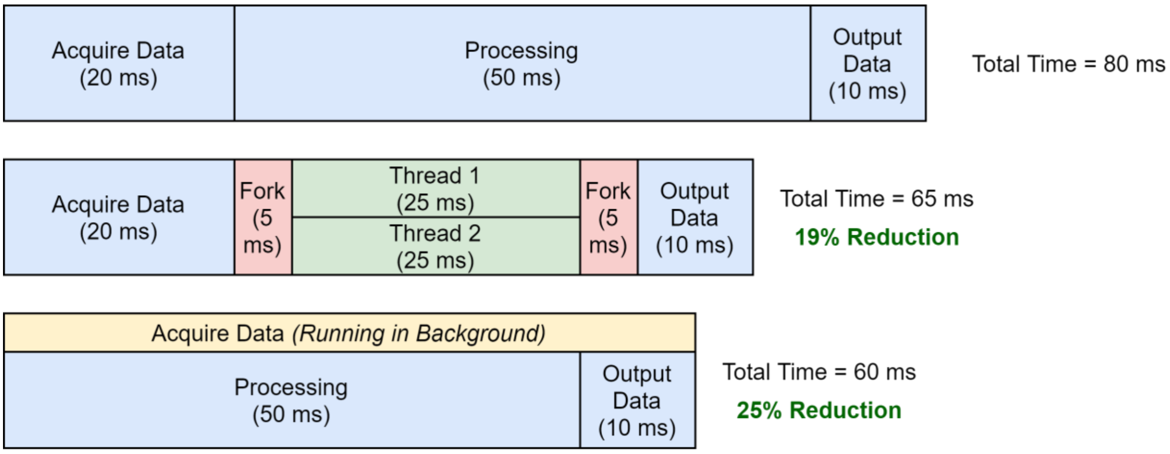


Figure 4-15: *Multithreading can be used without forking/joining threads and can run a single process constantly in a different thread to the main loop*

In Figure 4-15c, rather than splitting each function across multiple threads, the data acquisition is now run on its own thread throughout the process. Although this would not provide a benefit on its first pass, when running in a continuous loop, by the time one set of data has been processed, the next image has already been acquired and is waiting in memory, saving 20 ms, a 25% reduction in overall execution time. This method is frequently not the most efficient and is heavily dependent on the data size and the complexity of the function. As the image size or function complexity increases the more likely that forking and joining is beneficial. However, by using a separate thread for data acquisition the capture latency is reduced to almost zero.

This method of multithreading has therefore been applied to the data capture of the DRI (Figure 4-16). A class, *CameraData*, has been created to hold the images, timestamp, and image number and two instances of this class have been created along with a variable to keep track of which instance is currently being written to. This allows the camera thread (Figure 4-16b) to write to one buffer while the main thread (Figure 4-16c) reads from the other. When the main thread needs to read the buffer, it uses a mutex to stop the camera writing while the *BufferID* is changed, once changed, the mutex is unlocked. Now the camera can resume writing the data it captured to the new buffer, while the main thread reads from the other buffer. This reduces the time spent locked in the mutex to be as small as possible.

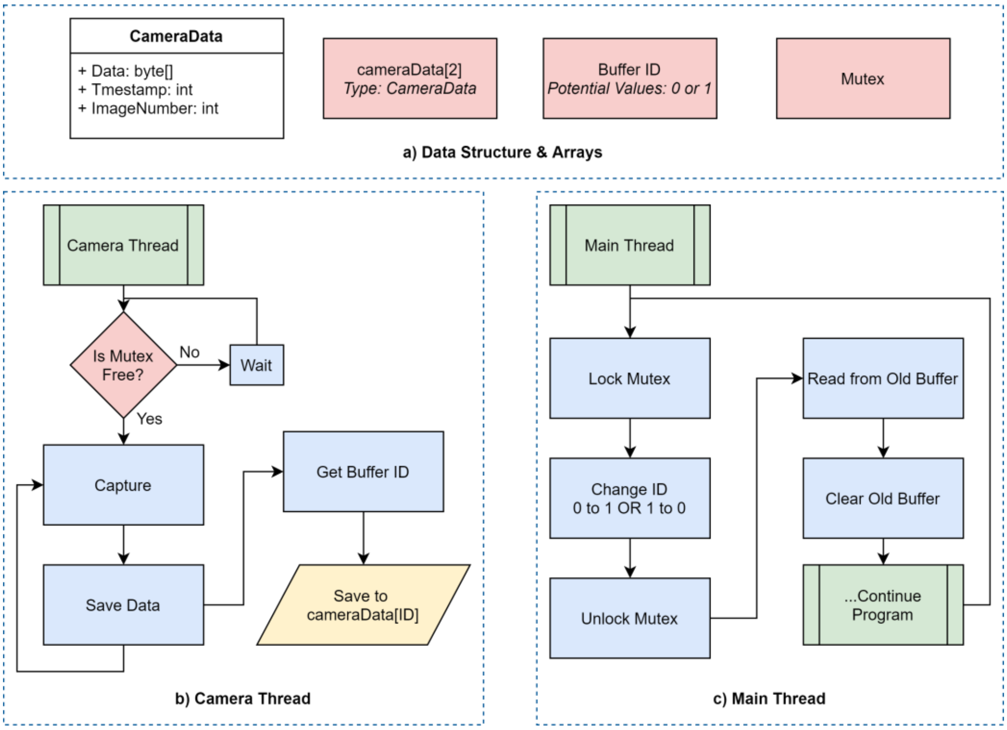


Figure 4-16: *Multiple camera buffer arrays allows the camera thread to write to one buffer while the main thread reads from the other*

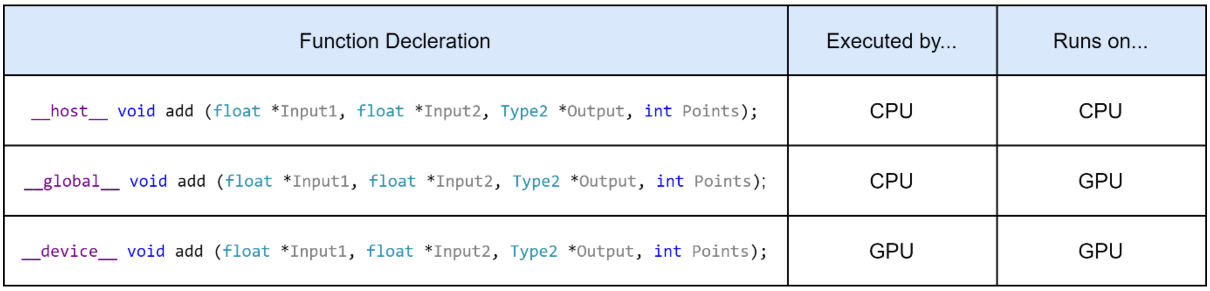
The other function suited to this method of multithreading is the data outputting function. Copying the data into the cached variable needs to be executed by the main thread, but once cached, a similar setup to Figure 4-16 can be used to send the data from the SPE to the UI. The main thread can save to the cached variable, while function *OutputData* is running on a separate thread. Once a notification has been received from the UI to initiate the data transfer, *outputData* locks a mutex preventing the main thread from writing new data to the cached variables while it transfers data to the UI, finally the mutex will be unlocked once this process has been completed. It may also be valuable to have two cached variables that can be switched (like Figure 4-16) to prevent the program from having to stop for long periods while reading or writing.

Another part of the SPE that benefits from running on separate threads are the Namedpipes that are used to communicate between the two programs. Using separate threads allows data to be sent and received without effecting the main thread, allowing notifications from the UI to be received and processed within the main loop at a specified time, action can then be taken depending on content of those notifications. This also allows the SPE to send notifications or messages to the UI, without disturbing the main thread. It may be to immediately lock a mutex and stop the main thread from running, although this is not currently implemented, however, currently it would only stop running at the end of the current frame being processed.

There is the limit to how far multithreading can go at this time, by running the camera acquisition, data outputting and two named pipes all on different threads, including the main thread, a total of five threads are being used without any forking/joining cost. Therefore, the next stage to further increase the SPE performance is to implement General Purpose Graphics Processing Unit (GPGPU) hardware acceleration for the signal processing to examine how much benefit that can provide.

4.6.3 GPU Signal Processing

CUDA can be programmed alongside standard C++ code and compiled with Nvidia CUDA Compiler (NVCC). When using CUDA, code that is executed on the CPU, or variables stored in RAM are known as *host* functions or *host* variables, conversely, GPU functions and variables are known as *device* functions or variables. When using NVCC it separates the two parts of the code allowing the C++ code to be compiled by MSVC and only compiles the CUDA. CUDA functions, known as *kernels* have a definition prefix before the function return type, these are *__host__*, *__device__*, and *__global__* (Figure 4-17). Host functions are called and executed on the CPU, with no intervention from the GPU, conversely, device kernels are called and executed on the GPU independent of the CPU. Finally, global functions, the most frequently used, are called by the host but executed on the device.



Function Decleration	Executed by...	Runs on...
`__host__ void add (float *Input1, float *Input2, Type2 *Output, int Points);`	CPU	CPU
`__global__ void add (float *Input1, float *Input2, Type2 *Output, int Points);`	CPU	GPU
`__device__ void add (float *Input1, float *Input2, Type2 *Output, int Points);`	GPU	GPU

Figure 4-17: *Kernel Definitions (Code)*

Before a global kernel can be launched, the number of threads the GPU should create needs to be calculated and set, once calculated. In CUDA threads are split up into blocks, each block of threads can have up to three dimensions. A single block can hold 1024 threads, and each thread executes a copy of all the code within the kernel. If more than 1024 threads are required, a grid of blocks needs to be configured (Figure 4-18). Both these parameters are then included within the function called, placed between two sets of chevrons (Figure 4-19).

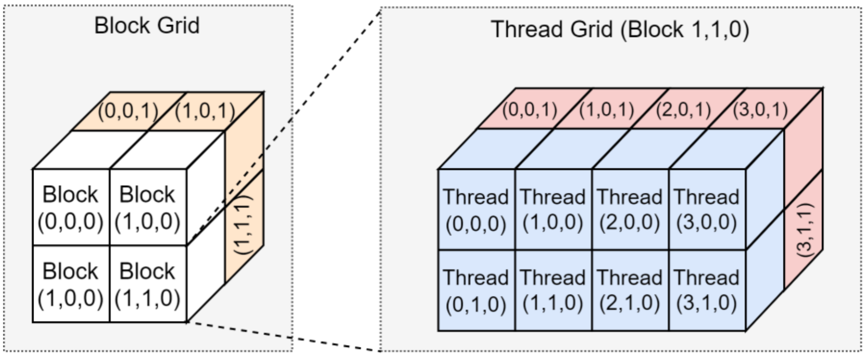


Figure 4-18: *CUDA Thread Organisation*

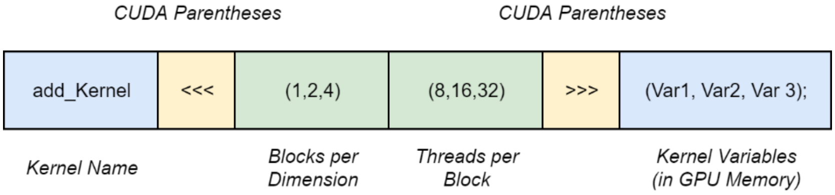


Figure 4-19: *Kernel Call*

Calculating the correct and most efficient number of threads to be launched to achieve the calculation is crucial. Launching too few threads results in missing computation and incorrectly returns zero for parts of the data, launching too many results in wasting time while the GPU manages the unnecessary threads. Although, as GPUs consist of multiple stream multiprocessors (SM) resulting in large numbers of cores per device, each SM has thirty-two CUDA cores so it is advisory to launch blocks of threads that are multiples of 32, even if that means launching a slightly too many threads. For example, an array is 800 by 125 pixels, it may seem sensible to run 800 threads per block and 125 blocks. This would provide each block an efficiency of 78%, as each block is not utilising 224 of its threads. Whereas launching ninety-eight blocks each with 1024 pixels, would mean only the last block is under-utilised, giving an overall performance of 99.6%.

Figure 4-20 shows the process used for calculating the CUDA threads for the DRI signal processing, the number of total threads would first be calculated based on the algorithms requirements and the data size being used. This value is then passed through the allocation function, each time using the maximum number of threads possible in each dimension and subsequently calculating the number of blocks in each dimension of a grid.

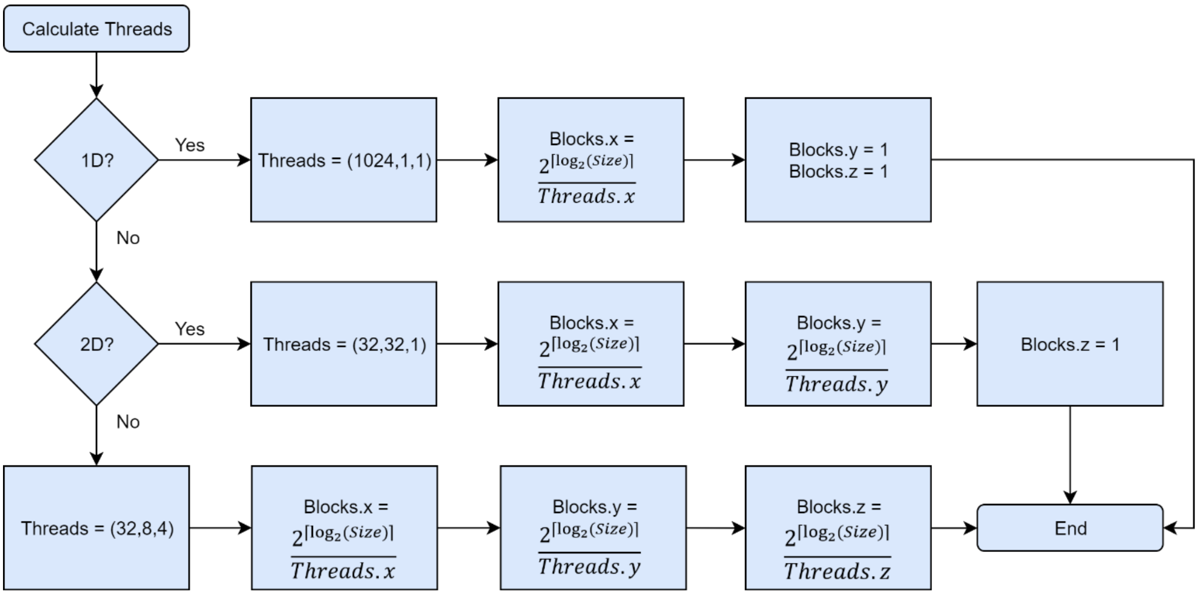

Calculate Threads
1D?
Yes
No
Threads = (1024,1,1)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = 1
Blocks.z = 1
2D?
Yes
No
Threads = (32,32,1)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.y}$
Blocks.z = 1
Threads = (32,8,4)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.y}$
Blocks.z = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.z}$
End

Figure 4-20: *CUDA Thread Allocation Calculation*

When using CUDA, data must be transferred to and from the CPU and GPU using memory pointers. This means it is not advisable to have multi-dimensional arrays storing data, therefore when data has more than one dimension it is stored in a flattened array. For example, if the data is a three-dimensional 2x2 image, each pixel having three colours (red, green and blue) this is flattened into a single 1D array twelve wide and one high (Figure 4-21). This is so the data can be accessed by an array pointer created and allocated memory at the start of the program rather than at compile time. This can then be transferred to the GPU where basic calculations are used to separate out the different dimensions from the original array using the original widths of the data and a stride value, the distance between each dimension.

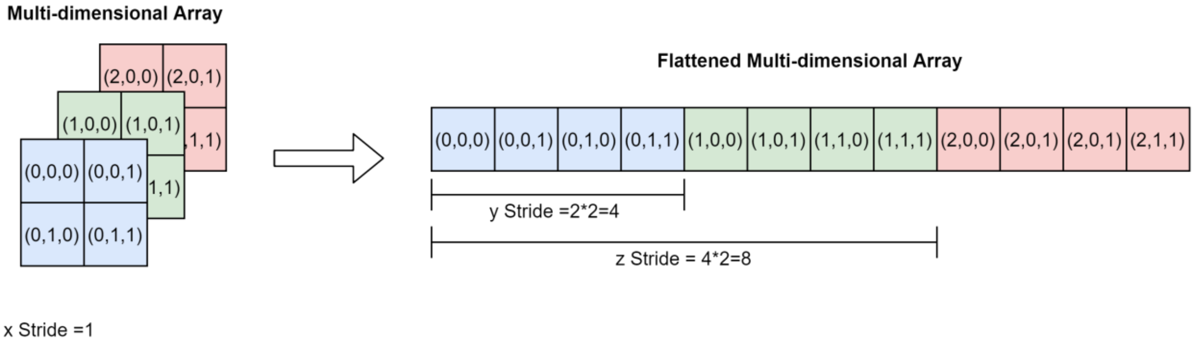


Figure 4-21: *Array Flattening*

The best way to illustrate the differences between a serial and GPU function is to look at an example, Figure 4-22 shows the *getDerivative* function and Equation 4-2 shows the calculation.

$$\mathbf{d} = \left(d_1, \cdots, \mathrm{d}_i, \cdots, \mathrm{d}_m\right)^T \text{ where } \mathrm{d}_i = \left\{ \begin{array}{lll} \left(s_{i+1} - s_i\right) & \text{where} & i = 1 \\ \dfrac{\left(s_{i-1} + s_{i+1}\right)}{2} & \text{where} & 1 < \mathrm{i} < \mathrm{m} \\ \left(s_i - s_{i-1}\right) & \text{where} & i = m \end{array} \right\}$$

$$\text{and index}, i \in \{1, 2, \cdots, m\}$$

Equation 4-2

The serial function first calculates the first and last value in the array as these are calculated differently, then creates a variable, *i*, and loops through each remaining element in the array until all elements have had their derivative calculated. On the GPU, the same code runs on every thread, therefore there are no loops and the check for the bookended elements is also done on each thread. As more threads may be launched than needed, there is an initial check to make sure extra threads do not execute and cause memory overflows. Next, each thread checks if its *threadID* is either zero or *Length-1*, if either are true it executes the code specific to those elements, else the main operation is executed. Once each thread finishes it is destroyed, once all threads are destroyed the kernel exits.

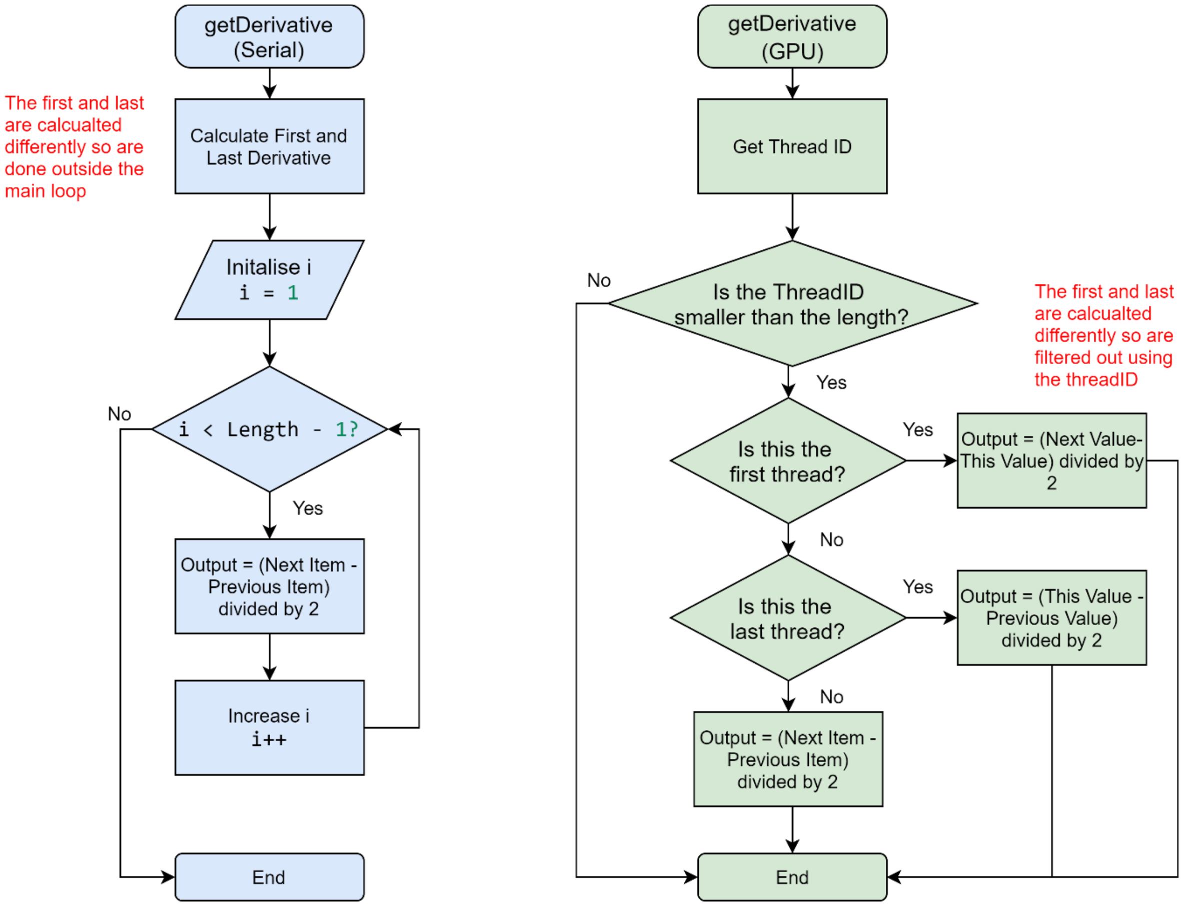

getDerivative (Serial)
The first and last are calcualted differently so are done outside the main loop
Calculate First and Last Derivative
Initalise i
i = 1
i < Length - 1?
No
Yes
Output = (Next Item - Previous Item) divided by 2
Increase i
i++
End
getDerivative (GPU)
Get Thread ID
Is the ThreadID smaller than the length?
No
Yes
The first and last are calcualted differently so are filtered out using the threadID
Is this the first thread?
Yes
Output = (Next Value-This Value) divided by 2
No
Is this the last thread?
Yes
Output = (This Value - Previous Value) divided by 2
No
Output = (Next Item - Previous Item) divided by 2
End

Figure 4-22: *Comparison between how getDeriviative runs in serial execution compared to the GPU execution*

When the CPU launches a *global* GPU kernel, the CPU does not wait for the kernel to finish and continues through its sequential execution, if another kernel is launched before the previous has finished, the CPU adds the kernel to a queue and once again continue. As far as the CPU is concerned this is an asyncronous operation, but the GPU executes the kernels in the order which they were launched. If a kernel is required to finish before the main program flow should continue, this can be achieved by using *cudaDeviceSynchronize* which forces the CPU to wait until all queued kernels have been completed.

As mentioned previously, GPU memory and CPU memory are separate and when executing kernels any variables that are used need to be copied to the GPU before the kernel is launched. Depending on the data flow, these may need to be copied back to the CPU after if they have been modified. If a kernel is launched and the next operation is to copy the memory back to the CPU, the CPU waits until the kernel has completed and is free to be able copy data back, in this scenario, *cudaDeviceSynchronize* is not needed as *memcpy* operations are synchronous.

The function to transfer data to the GPU is *memcpy*, this function can transfer memory from the host (CPU) to the device (GPU) and vice versa. Typically before a kernel, any arrays used are allocated GPU memory and copied to the device. Once all data is on the device the number of threads needed can be calculated and then the kernel can be launched, when complete any modified arrays may need to be copied back to the CPU if used on the host at a later stage. If all functions in a signal processing chain are to be executed on the GPU, it is unlikely any data would be copied between the host and device apart from the data acquired and the data to be output to the UI. A representative example of the steps taken to execute a kernel is shown in Figure 4-23.

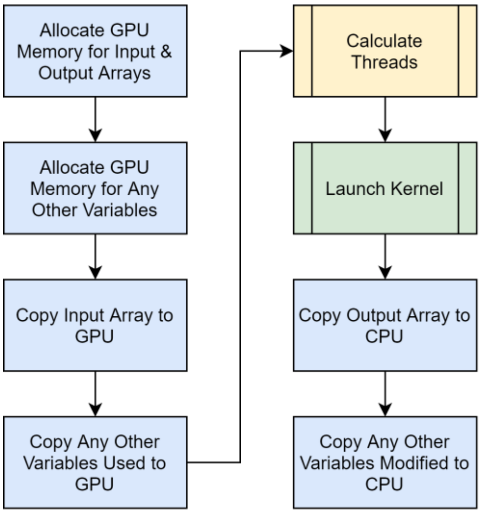

Allocate GPU Memory for Input & Output Arrays
Allocate GPU Memory for Any Other Variables
Copy Input Array to GPU
Copy Any Other Variables Used to GPU
Calculate Threads
Launch Kernel
Copy Output Array to CPU
Copy Any Other Variables Modified to CPU

Figure 4-23: *Order of operations for executing a function on the GPU*

Now all the signal processing has been programmed and tested, the surrounding code that allows all the functionality described above to work together is discussed. This forms the majority of the Signal Processing Engine (SPE).

4.6.4 Signal Processing Engine

A significant amount of code is needed as the framework of SPE in order to be modifiable and to allow configurations that fundamentally changes the program and acquire and output the correct data. The structure of these threads are shown in Figure 4-24 and discussed in more detail throughout this section.

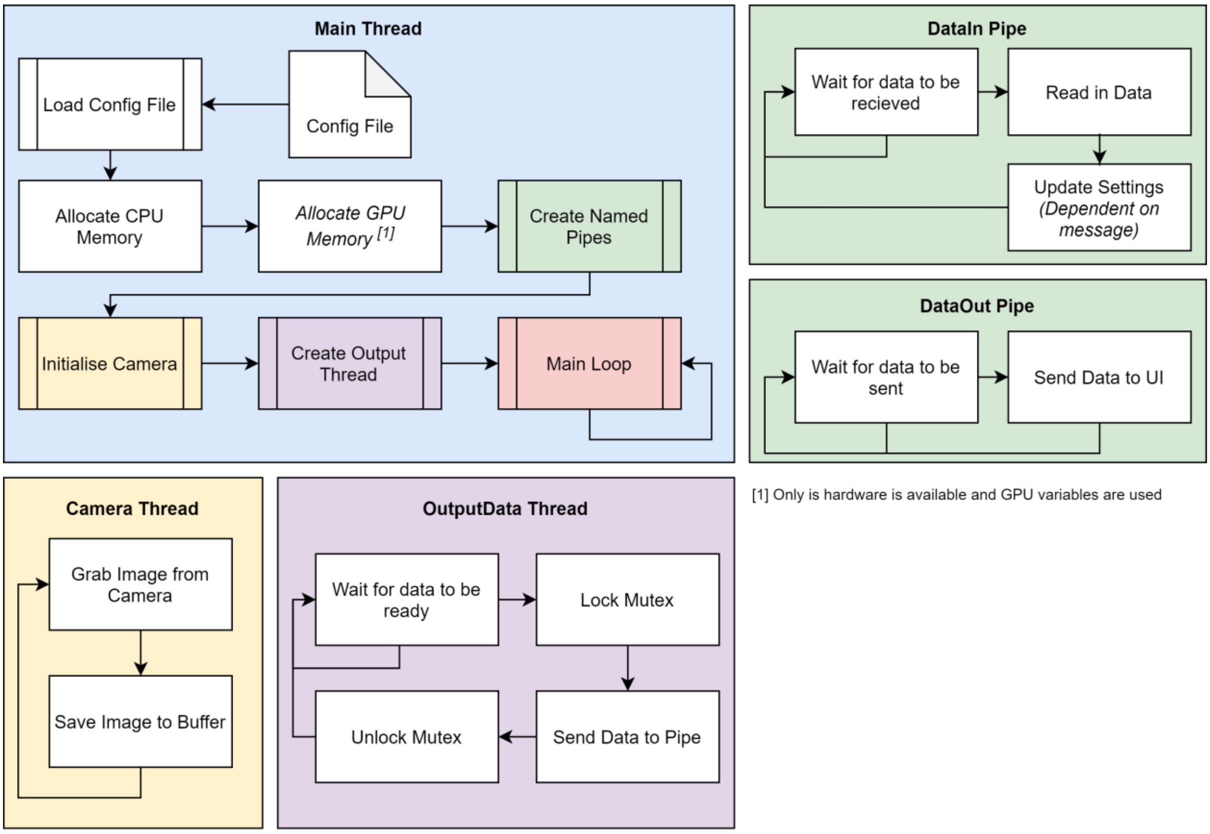


Figure 4-24: *Signal Processing Engine main file flow diagram*

Firstly, the configuration file (Chapter 4.5) is imported, this is an XML file and contains all the information about each function, sample files can be found in Appendix 11.1. The configuration file loading follows a simple process of looping through each element in the XML and importing each line into a list of type *function* (Figure 4-25). This class is not an efficient way of storing the data, as each instance has a member for every different variable needed across the DRI functions, therefore each instance of *function* has many unused members, wasting memory. Moreover, when expanded to include more functions, this is not sustainable for all the possible different variables needed, this is modified in a future iteration discussed in Chapter 5.3.3.

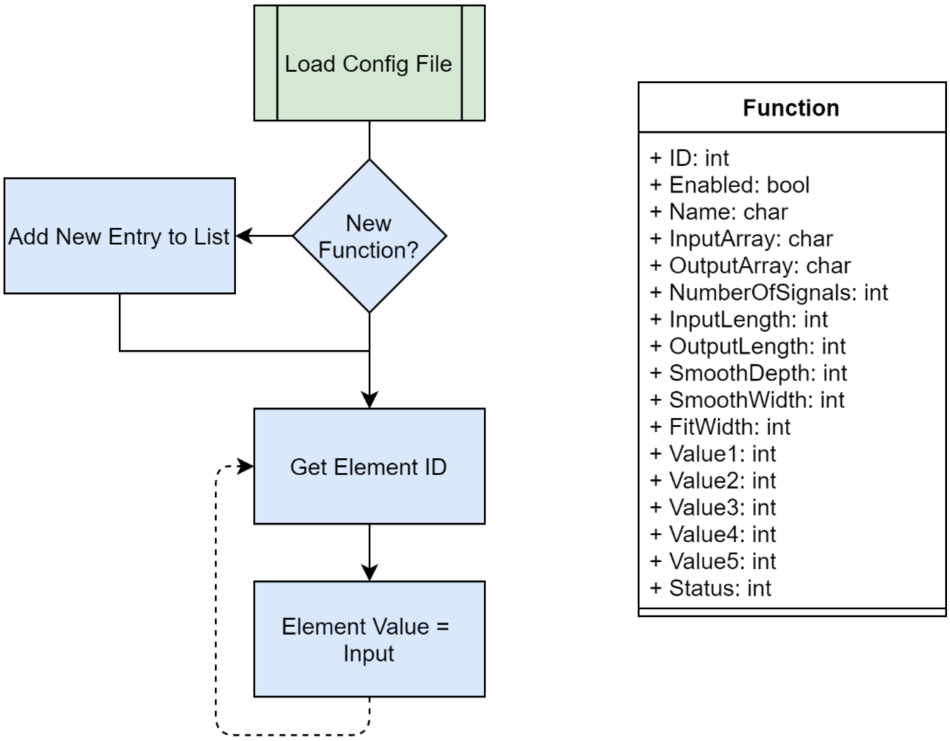


Figure 4-25: *XML Config File Loading and Importing. Function class represented in UML*

The variable names are stored in the function class, but the variables themselves are stored separately. All the arrays used are stored as pointers in a class, then at runtime these are allocated memory based on the width of the captured image, which is currently hardcoded into the SPE. This is inefficient, as arrays that do not require the full length of a camera image still have the same amount of memory allocated, this is not too much of an issue for the current 8192 image size, but this inefficiency would be worse for a larger image. This is also the method for allocating the data on the GPU, where memory can be extremely limited. Furthermore, memory is allocated on the CPU and GPU regardless on which platform is being used, this again is an inefficiency that needs addressing in future work. Finally, the DRI has two stages of measurement resolution, if only relative measurement is needed, all the arrays for absolute measurement are still initialised and allocated, wasting both memory and time.

Once the configuration is imported, memory can be allocated on the CPU, and if available, the GPU, template data is then imported and copied onto the GPU for future potential use. The penultimate step before the main loop can start is to connect to the named pipe to allow data transfer to and from the UI. Finally, a connection to the camera needs to be established, this initialisation of the connection only needs to run once as the data capturing itself is separate. Once all the variables have been allocated and any setup functions have been executed, the main loop can start. The main loop runs in a while loop and cycles through all the objects of *function* in the function list (Figure 4-26). The SPE then uses switch statements to compare the function member *ID* and selects the correct function based upon this attribute.

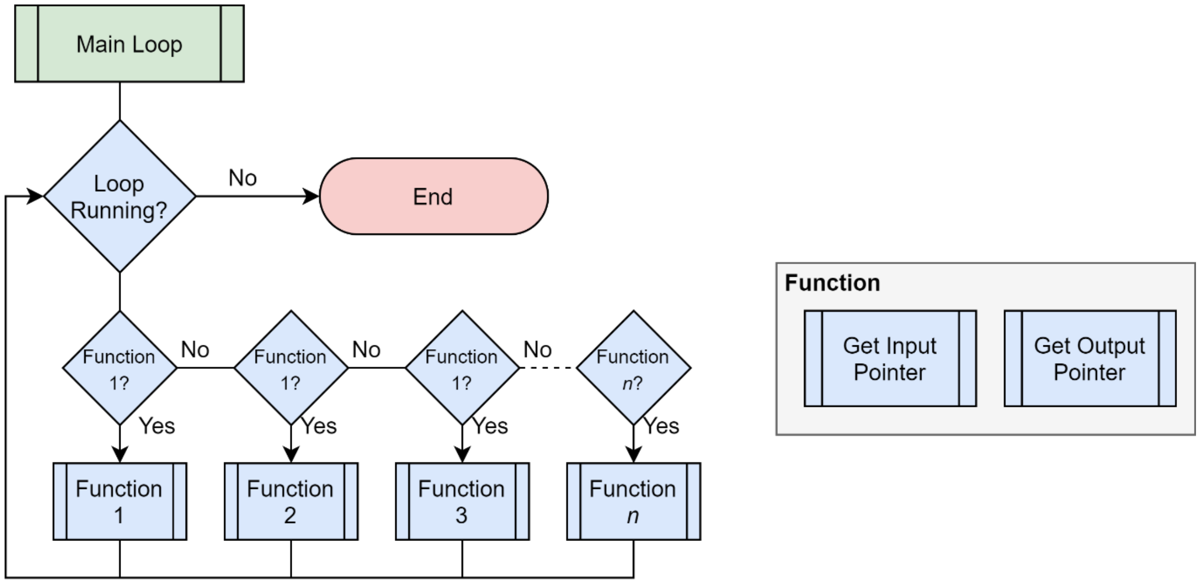


Figure 4-26: *Flow diagram of the main loop of the SPE that runs continually until stopped. It uses switch statements to select the correct function.*

Once the function has been located, the function call does not contain pointers to the memory location where the correct array is stored, instead it contains a secondary function call to a *getPointer* function that takes a string as an input (Figure 4-27). This string is the name of the array needed in plain text form, this was imported from the XML file and is used as the reference to the array. This name is then compared in a series of if statements to find a match against other strings. Once a match is found, the correct pointer is returned and can then be used in the function call, this usually needs to be repeated for the output array as well as the input array.

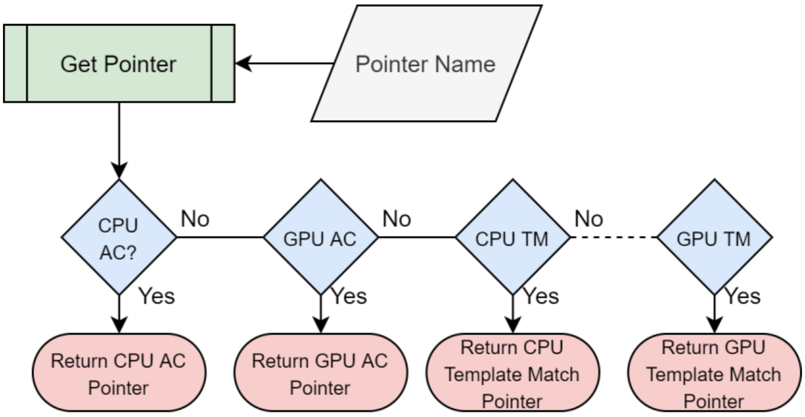
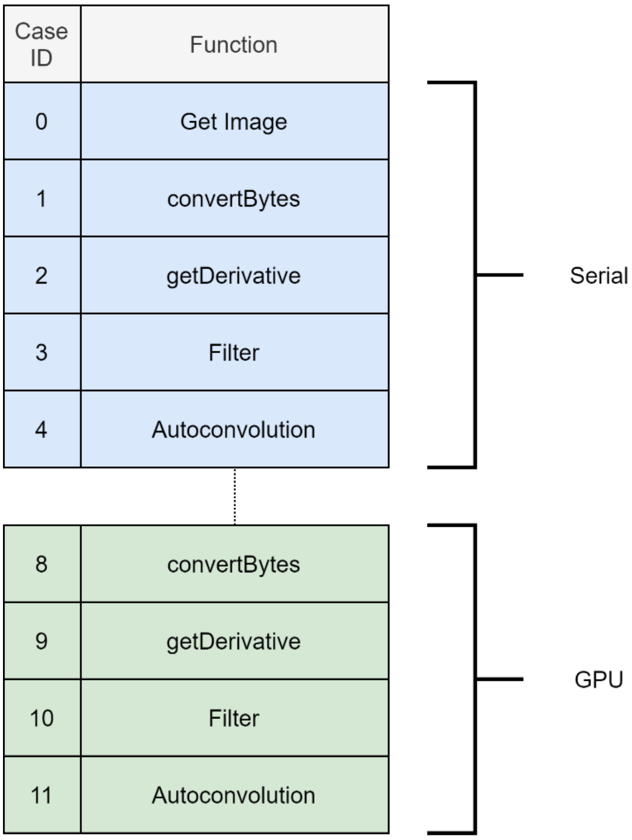


Figure 4-27: *getPointer function. Given the name of a variable returns its pointer*

Just like the function class, this is a very inefficient way of managing this data, not only does the SPE have to compare the name against multiple strings, it must do it every loop. These comparison strings are also hard coded, so any changes made to a variable name would require a full recompile. The last negative is that this function can only return a single type, therefore severely limiting flexibility and also speed. Calculations using doubles requires twice the memory and take at least double the time. This is another section of the SPE code that is improved later and has been reworked as the number of arrays increase and SPEs library expands.

The ability to switch between CPU and GPU signal processing functions is achieved by selecting a different case in the switch statement. Rather than selecting case one for the *convertBytes* CPU function, case eight would be selected for the GPU counterpart (Figure 4-28). In future, the ability to use the same case from the switch statement for both the CPU and GPU function, with some logic inside the switch case to select the correct architecture would be a useful improvement as the function library gets larger. This reduces the number of cases that need to be check each loop by half, increasing the performance as the library gets larger.



Case ID	Function
0	Get Image
1	convertBytes
2	getDerivative
3	Filter
4	Autoconvolution
8	convertBytes
9	getDerivative
10	Filter
11	Autoconvolution

Figure 4-28: *Switch statement for selecting functions in the main loop of the SPE*

4.6.5 Challenges Faced

One of the challenges faced in this early work came about by developing the SPE and the UI in separate programs. By separating these two programs and developing in separate languages has allowed each program to include the benefits of its chosen programming language. Unfortunately, this means having to pass data between the two separate programs, this is not a problem for non-time critical data, such as configuration parameters, but passing signal processing data needs to be done in a quick and time efficient manner to prevent causing a bottleneck. It is not ideal to use a language like C++ to gain efficiencies to then have a slow data transfers just to output the data on a graph in C#. Ideally, as the data is stored in random access memory (RAM), it would be ideal to simply pass the memory location between the two programs, but alas memory pointers are unique to each individual process and should not be accessed from a different program other than the one that created it.

An inefficient and temporary solution to data transferring was to save the data as a CSV file, however, a better option was to save the data as binary (.bin) files. Binary files have no formatting and simply consist of ones and zeros; therefore, the data has to first be converted from integers, single-point precision or double point precision into binary before it can be saved in this format. C++ has these functions built in and therefore are relatively efficient. Using the same hard drive as before, a test was conducted to investigate what the performance benefit would be for saving the data as a binary file rather than a CSV. The results showed that saving in binary format was over twice as fast saving as CSV due to not having to separate the data with commas.

As the binary file does not contain any formatting, header information needs to be placed at the start of the binary stream that can be read by the UI to enable it to convert the data back into its original form (integer, single precision, double precision). Adding this header information does create a small-time penalty but is insignificant when compared to the time lost adding commas when saving as a CSV file. The complication that this adds is the process to add/read the header information must be identical on the SPE and UI, otherwise data reconstruction is not possible. Any changes made in the SPE must be mirrored on the UI, duplicated the work needed to make minor changes. An unintended advantage of saving in a binary file is multiple signals can be saved in the same file at once, provided that this is reflected in the header information. This can provide a significant timesaving when transferring multiple signals or sets of multiple signals compared to saving a single signal. This is because as file sizes increase data can be written to the disk faster as there are less random read/write operations needed when the data is saved in one single block.

Another challenge faced during this early work was dynamically setting which array was to be used at what time and in which function. The ability to import the configuration existed and the name of the array to use was accessible, but the ability to select an array based on its name was not. In an attempt to solve this before each function was executed (every loop) *getPointer* is called for each array in order to select the array to be used. *GetPointer* takes the name of the array, in plain text and compares the string against several if statements, when one of the if statements return true *GetPointer* returns the correct pointer. Although this is a brute force way of achieving this, for the small number of arrays used at this time it worked just fine, however, it was not very efficient and it checks every variable, every loop, but it does allow dynamic array selection. This problem is addressed and solved later in Chapter 5.3.5 as this is an ever increasingly long list as the number of variables grows.

```
double * GetPointer(char Name[]) {
    std::string myString;
    int Size = strlen(Name);
    myString.assign(Name, Size);
    if (myString == "CPU_AC")
        return CPU_Signals[0].AC;
    else if (myString == "CPU_Interferogram")
        return CPU_Signals[0].Interferogram;
    else if (myString == "GPU_Interferogram")
        return GPU_Signals[0].Interferogram;
    else if (myString == "GPU_AC")
        return GPU_Signals[0].AC;
    else if (myString == "GPU_Smooth_Derivative")
        return GPU_Signals[0].Smooth_Derivative;
    else if (myString == "CPU_AC_Result")
        return CPU_Results[0].AC_Result;
    else if (myString == "GPU_AC_Result")
        return GPU_Results[0].AC_Result;
    else
        cout << "Error, Variable Not Found: "
               << myString << "\n";
        system("pause");
    }
}
```

Figure 4-29: *GetPointer Function takes the name of an array and returns the correct pointer*

4.6.6 Summary

The challenges listed above allows for development of the SPE moving forward when the number of functions in the library increases. It has also identified performance benefits that can be made further increasing the SPE optimisation. The serial functions allow signal processing on the CPU without any extra hardware required, and also utilises multithreading if available on the hardware. By dedicating specific functions to their own thread, data acquisition, data outputting and inter-process communication allows utilisation of the multiple cores modern CPUs contain. In addition, by multithreading in this method, any GPU acceleration is an additional benefit, and not instead of, further increasing the efficiency of the SPE.

4.7 Evaluation of Dispersed Reference Interferometry Performance Improvements

Now the SPE is operational, the different processing modes can be benchmarked, and the architectures can be compared against each other.

4.7.1 Image Capture

The first test involves capturing the data, reinterpreting the Mono12 to floating points and then saving to a CSV file. The main thread is responsible for converting the Mono12 data to integers on the CPU, whereas when running on the GPU, any memory transfers needed are also included. The data acquisition from the camera is being executed on a separate thread. As shown in Figure 4-30, the CPU and GPU (Single Frame) have a similar frame rate, with the GPU just edging ahead, even though the signal processing required for converting data is minimal, it was still faster to transfer the data to the GPU and back. As for batching GPU data, this method better utilises the large number of GPU threads available, and therefore provides a significant increase in frame rate. However, this is at the cost of latency, as thirty-two or sixty-four frames are being processed at once, the rate at which the data is output is reduced. The camera can capture in Mono12 mode around 6000 FPS so should not result in a bottleneck with these measurements, but for the batched data, it cannot be ruled out that the main thread had to wait for frames to be ready before it could send them to the GPU.

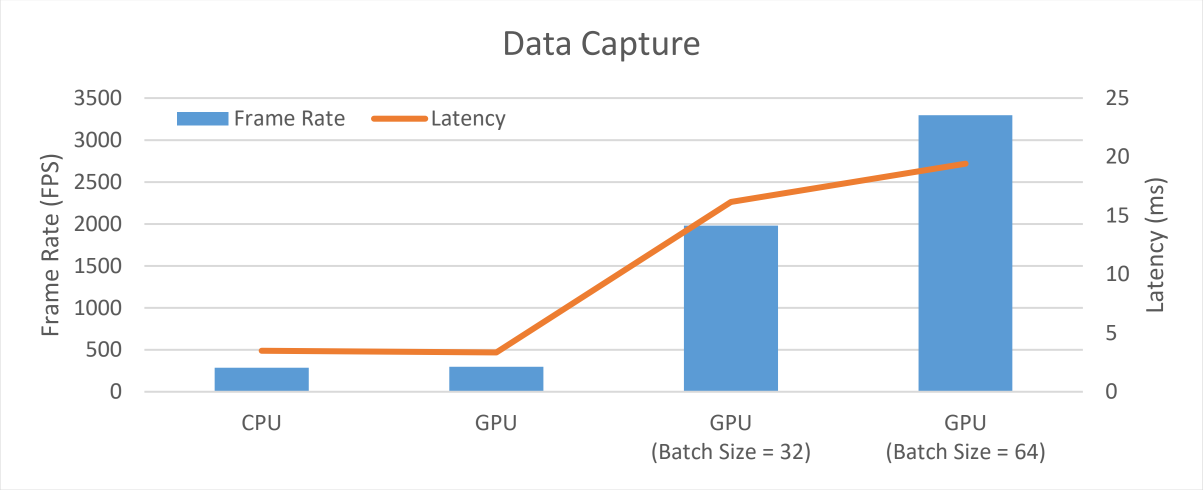


Figure 4-30: *Frame Rate and Latency of Data Capture for DRI*

4.7.2 Absolute Position Measurement

The next is to calculate the absolute position, this involves capturing the data and reinterpreting it to floating points, calculating the derivative, filtering the derivative, and finally convoluting the signal with itself (auto-convolution). This time, the CPU outperforms the GPU (Figure 4-31), which might at first seem peculiar, given operations such as derivative and FFT are prime candidates for GPU acceleration, but there are two reasons for this. Firstly, for the FFT, CUDA has to run a setup each time before the FFT can run, which is inefficient and has been re-evaluated and modified to prevent this in a later update. Secondly, and the main reason, is the filter algorithm is poorly optimised for a single frame on the GPU.

Due to the nature of the filtering algorithm, which is a moving average, there are a number of summing actions that need to be done linearly and do not allow for easy parallelisation within a single frame. However, when data is batched, the algorithm can be parallelised across multiple frames, completing multiple linear summations in different threads for each different frame. This allows batched frames to provide a significant frame rate increase once again, but once again, at the cost of increased latency. These results were also compared to a MATLAB implementation, using pre-acquired data. Although using pre-acquired data might not seem an accurate comparison, the data acquisition is done in a separate thread in the SPE and therefore the cost to acquire data is virtually zero, therefore, MATLAB should have a slight advantage. As expected MATLAB, due to its high-level language, is considerably slower than either the CPU or GPU and therefore a higher latency than either architecture when processing a single frame at a time.

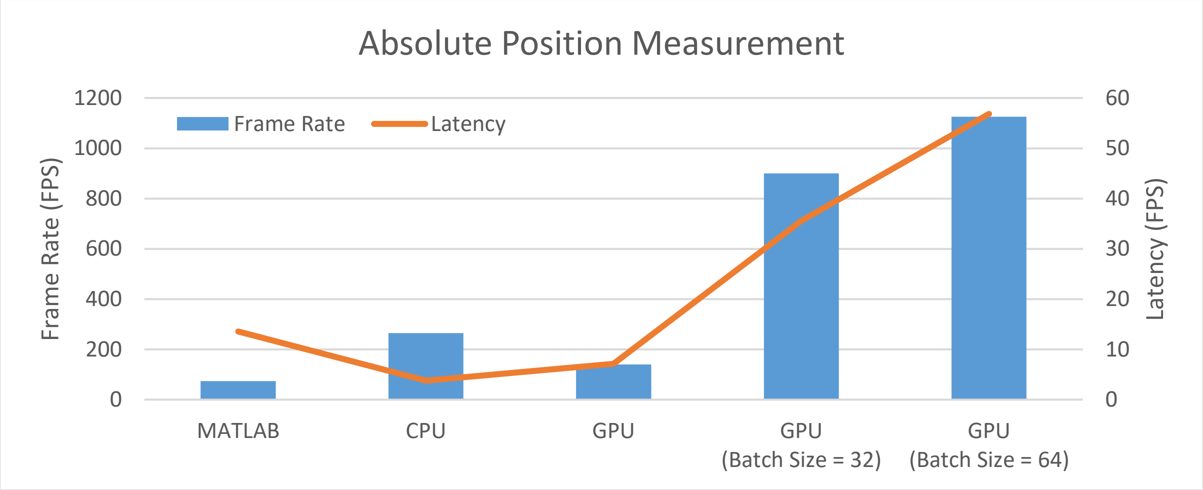


Figure 4-31: *Frame Rate and Latency of Absolute Position Measurement for DRI*

4.7.3 High-Resolution Measurement

Finally, is to calculate the high-resolution measurement, this contains all of the steps from above and also windows the signal and implements template matching (correlation), however, this does not include combining the two measurements together. Although previously (Figure 4-31) the GPU was slower than the CPU, Figure 4-32 shows the GPU out-performing the CPU due to the template matching having some very parallelisable elements. Once again, batching the frames provides significant frame rate increases over single frame, and would indicate that 128, or 256 frames may also have an additional benefit. However, when the GPU becomes saturated, or the memory transfers become the limiting factor, the performance increase batch processing data provides plateaus. Lastly, MATLAB is once again slower than the CPU or GPU with the worst decrease in performance from the absolute position across any architecture.

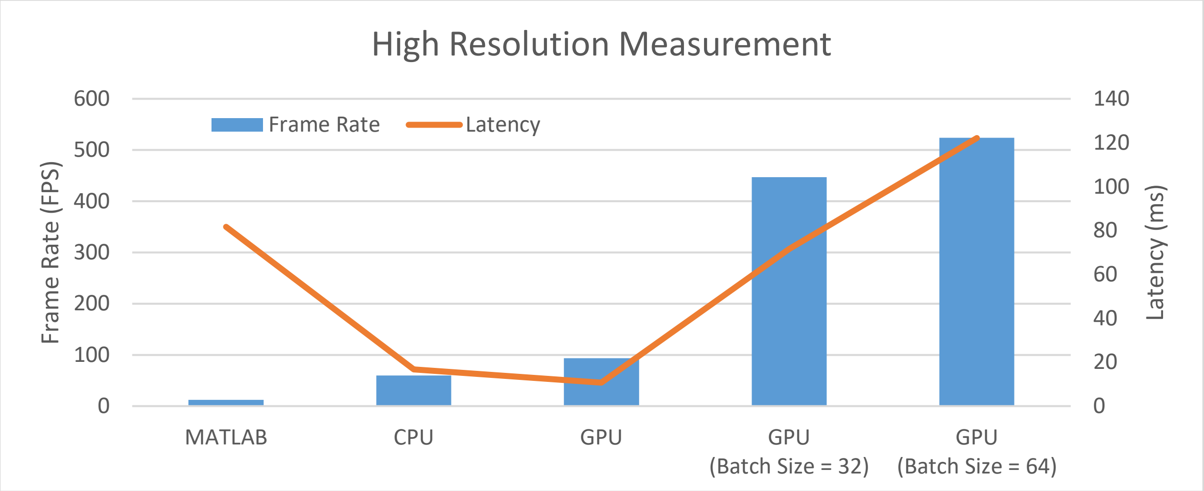


Figure 4-32: *Frame Rate and Latency of High-Resolution Measurement for DRI*

4.8 Conclusion

In all tests, the SPE was significantly faster on the CPU than MATLAB, but when processing a single frame at a time, the GPU was sometimes not the fastest architecture (Figure 4-33). This is due to the overhead associated with executing on the GPU. These overheads are memory transfers to move the data to the GPU for processing and then returning the modified data back to the CPU's memory and also a small overhead to calculate the number of threads needed to launch the kernel. As the algorithms become more complex and take longer to execute, the percentage of time consumed by the overhead decreases, thus using the GPU become more advantageous.

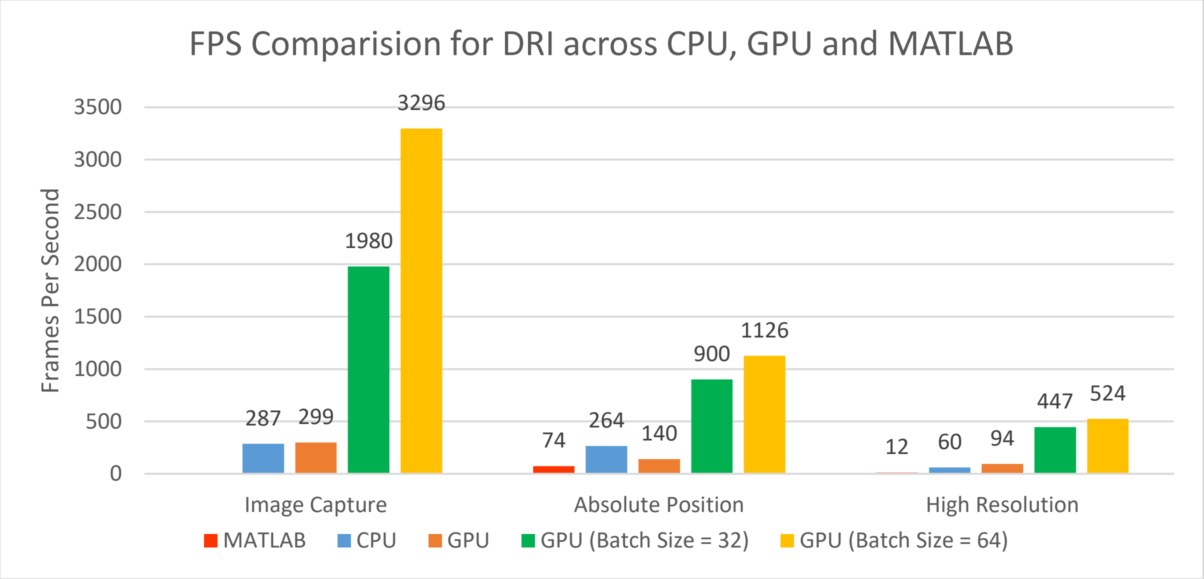


Figure 4-33: *DRI frame rate comparison for different measurements and architectures*

In these tests, processing frames in batches also indicates how 2D data would perform, as, the more data provided to the GPU to compute, the smaller of a percentage the overhead is, and therefore the greater the benefit of using the GPU over the CPU is. In this scenario the GPU really benefits from processing larger data, either 2D or batching 1D data together provides a higher throughput but also higher latency (Figure 4-34).

This initial work in developing a basic UI and SPE for the DRI involved learning C++, C#/.NET Framework, OpenMP and CUDA. With the author having some previous experience in C++ and C#/.NET this was still around a twelve month learning and development process to achieve the above results. While the results demonstrate a significant increase in performance for the GPU over the CPU, the DRI contained relatively few signal processing steps, and the development for a larger signal processing chain would likely be longer. Furthermore, the performance increase potential is likely to be higher than the reported figures above, CUDA optimisation is not a trivial undertaking and requires significant knowledge of thread and memory management to achieve the highest level of performance. Therefore, there is a huge benefit to be able to access a signal processing library with the CUDA efficiencies already established. To be able harness the acceleration of a multicore CPUs or GPU without having to spend twelve to eighteen months programming to develop a piece bespoke software is an enormous benefit to researchers and industry.

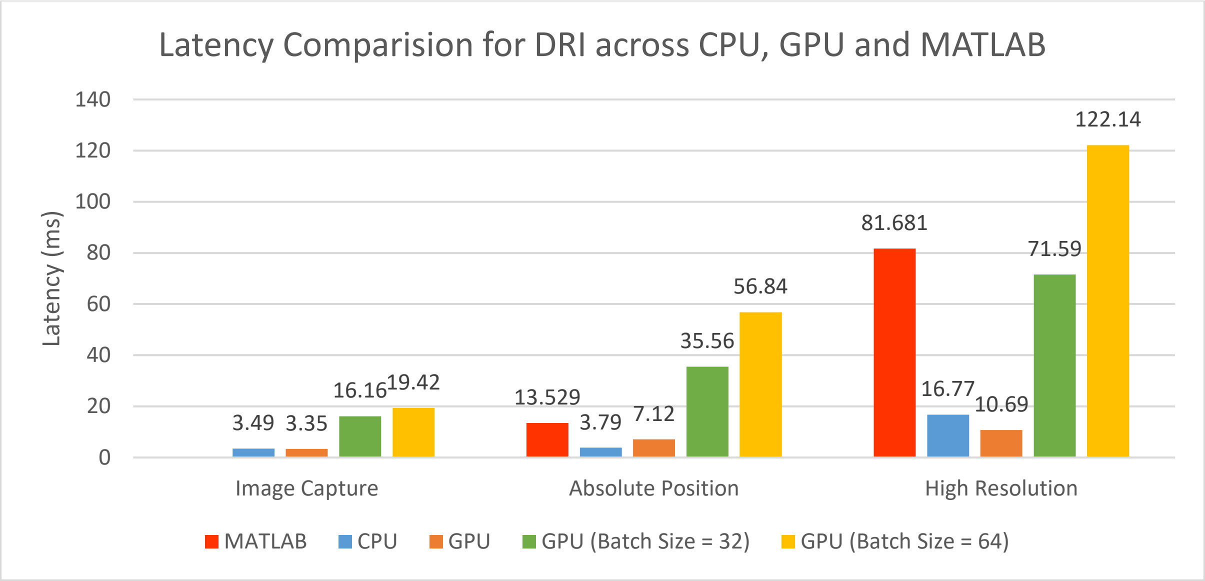


Figure 4-34: *DRI latency comparison for different measurements and architectures*

In the next chapter this case study will be expanded to be more generic and applicable to other optical measurement techniques and many fields beyond. One of the main features required in this software is the ability to profile signal processing functions to be able to compare CPU and GPU execution times against each other for each function. This comparison can then be carried out using AI to narrow down the search criteria and return a solution providing increased performance compared to a single architecture.

5 Consideration of a Heterogenous Signal Processing Library

5.1 Introduction

Continuing from the case study, the consideration of a heterogenous signal processing library is now discussed to meet the requirements laid out in the objectives. Throughout this chapter, a second case study will be undertaken, Line-Scan Dispersive Interferometry (LSDI), introducing two-dimensional data, and three-dimensional data when multiple frames are processed in batches. The introduction of this second case study will help ensure that the signal processing algorithms created can be evaluated for their performance on a real experimental setup.

Creating software containing a prebuilt library of signal processing functions that are configurable by the user at runtime while keeping the performance of bespoke software is a challenging goal. The software should be easy-to-use, while containing the signal processing algorithms that industry and researchers commonly use when processing surface and dimensional sensor data. Some algorithms are available free and open-source from third parties, these algorithms have been extensively tested for performance and accuracy and these have been implemented within the Signal Processing Engine (SPE).The SPE also should also include GPU acceleration using CUDA while offering the same level of configurability as the serial function library. To be able to provide the user with optimum performance algorithms need to be profiled (timed) on both architectures to find which is the best for the signal processing requirements of their measurement.

LSDI uses a broadband light source from a halogen lamp connected to the interferometer though a multi-mode fibre. LSDI provides an environmentally robust technique that is suitable for in-line surface inspection obtaining the surface profile in a single shot (Figure 5-1). LSDI can achieve high dynamic measurements whilst maintaining a high signal-to-noise ratio, a necessity when implementing instruments on manufacturing lines (Tang, 2016).

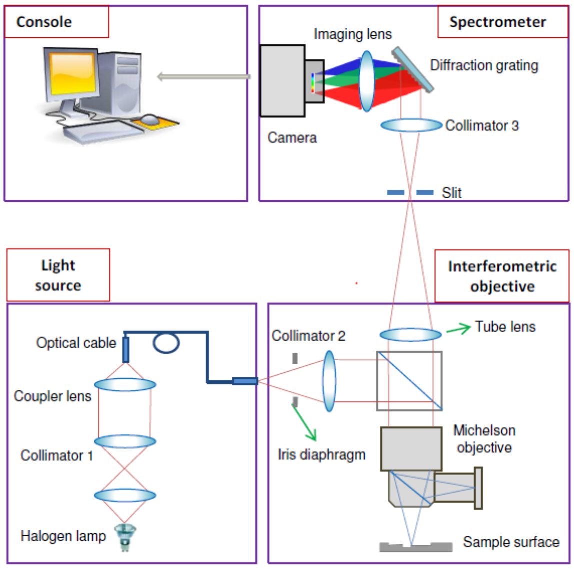


Figure 5-1: *Note. Reprinted from "Investigation of Line-Scan Dispersive Interferometry for In-Line surface metrology" by Tang, D., 2016, p. 59.*

The LSDI captures fringe patterns using a Mono8 640x480 pixel image (Figure 5-2) resulting in a 0.3 MB image, 37x larger than the DRI image. Within this single shot, LSDI can measure a vertical resolution of 100 nm across a range of almost 6 mm. It also contains significantly more signal processing stages, therefore CUDA acceleration should increase the performance over the serial code and provide a significant increase over the MATLAB code previously used by Tang et al.

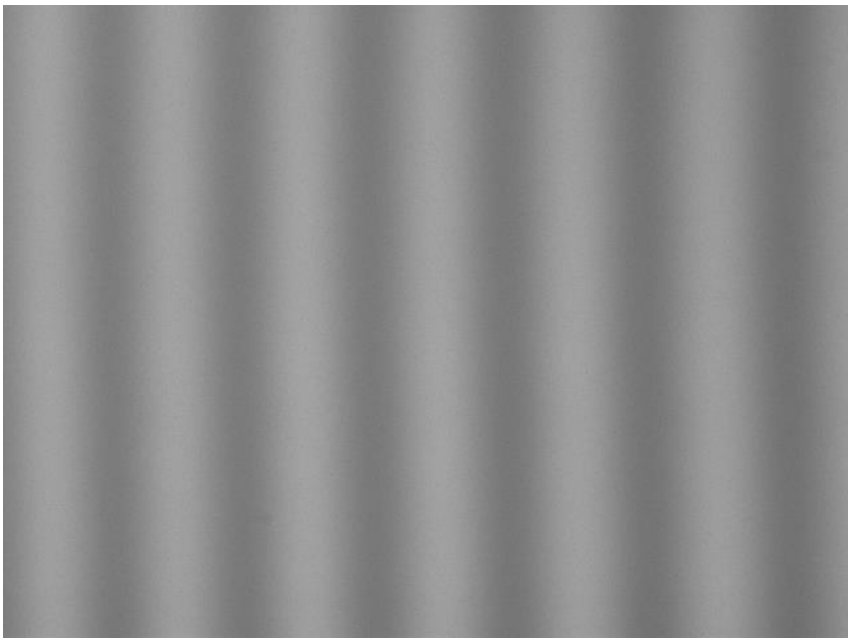


Figure 5-2: *LSDI Fringe Patten*

The proposed software solution has been named Optical Signal Processing Workspace, OSPW. This chapter will demonstrate how this software has been created, the inherent benefits it provides over previously discussed solutions, and the innerworkings of how it achieves its flexibility while not sacrificing performance. The discussion of OSPW is in chronological order of how a user would interact with the software. Therefore, first the user interface is discussed as this would be the first interface and user would see when using OSPW.

5.2 User Interface: Signal Processing Configuration

The first part of OSPW is configurating the Signal Processing Engine, this is done through the Configurator. The Configurator uses information stored in configuration files to enable users to configure the functions available in the library. Once a configuration is created, it is saved in an XML formatted file used by the SPE to execute the configuration created.

5.2.1 Creation and Management of Configuration Files

Each signal processing function has two files associated with it, a function header file (.h or .cuh) used by the SPE and a function configuration file (.xml) used by the configurator in the UI. Both these files have the same file name, the name of the function, with the configuration file describing the specifics of the function, its available architectures, and the parameter information. It is important that the details in the files match exactly, if there is a disparity between them, then the SPE may not execute correctly. Therefore, these files are created and controlled by the UI through the library management. Whenever a new function is added, a function configuration file is automatically created from the function code.

Each of these configuration files (Figure 5-3) contains the name of the function, the switch statement case number and whether there is a CPU and GPU version of the function available. The bulk of the file consists of the parameter information, each parameter of the function is listed with its various properties such as its name, description, whether it is a signal or a data, and the data type. It also includes whether the parameter is read only, to help constrain the data flow, whether memory transfers would be needed to execute the function on the GPU, and finally, if the variable needs to be hidden and not configured by the user, but by OSPW itself.

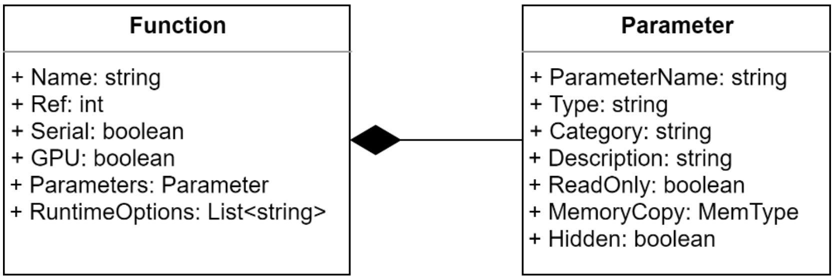
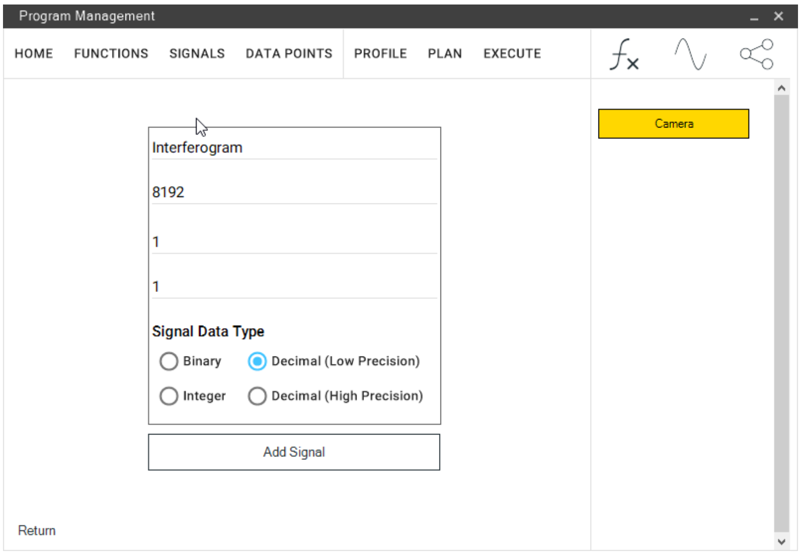


Figure 5-3: *UML Representation of Configuration XML File*

The *function* data type can be any of the primitive data types supported by C++, or a template type allowing multiple possibilities. The list of permutations created by the template type parameters are exhaustively listed in the *RuntimeOptions* list within the function configuration (Several examples of configuration files can be found in Appendix 1.1. By listing each of the possible permutations allows the configurator to check if the permutation chosen by the user is supported by the function. This list of runtime options is also created when the function is added, the user adding the functions, will select the data types they wish the function to accept.

5.2.2 Providing the Ability to Configure Signal Processing

The signal configurator has been made to be as easy to use as possible to allow users to quickly create their chain of signal processing functions. The program management configurator uses a drag-and-drop interface to make the process simple by dragging the signals and datapoints from the side bar onto the function. The creation of *Signal* and *Datapoints* is also straightforward. Selecting *Signals* at the top opens the 'Create new signal' page, where the signal details need to be entered; name, length, height, depth, and data type (Figure 5-4).



Interferogram

8192

1

1

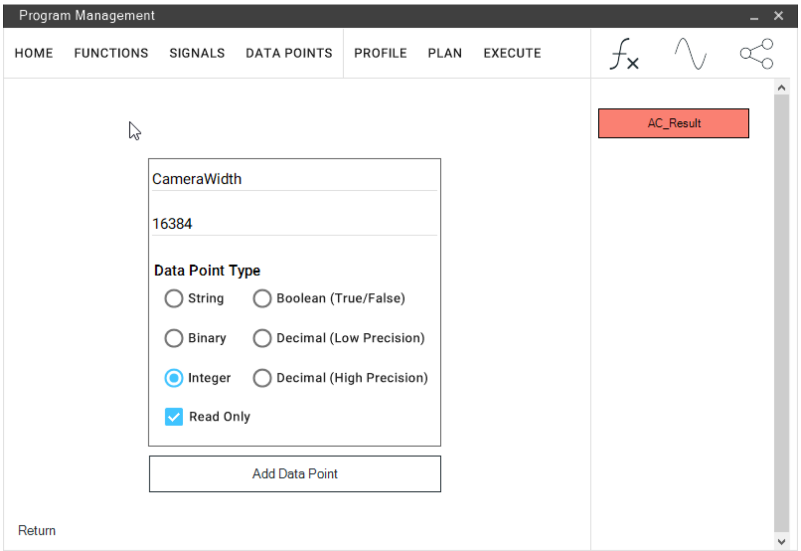
Signal Data Type

○ Binary ● Decimal (Low Precision)

○ Integer ○ Decimal (High Precision)

Figure 5-4: *Configurator: Add New Signal*

Datapoints are created using the same method, however, they have different properties (Figure 5-5). datapoints are a single value and can either be alphanumerical strings or numerical data, their initial value also needs to be set, although if they are modified within a function the value can be initially set to zero. Finally, whether the datapoint is read only or not needs to be specified; by default, the box is ticked. If a value does not change it does not ever need to be copied in or out of GPU memory or have a mutex setup when being modified, saving valuable time and removing constraints when trying to optimise the signal processing.



CameraWidth

16384

Data Point Type

○ String ○ Boolean (True/False)

○ Binary ○ Decimal (Low Precision)

● Integer ○ Decimal (High Precision)

☑ Read Only

Figure 5-5: *Configurator: Add New Datapoint*

The signals and datapoints are the data flow throughout the signal processing chain and need to be added to the functions in the configurator. Functions are split up into categories and can be selected from the dropdown boxes (Figure 5-6). A function can contain *Signals*, *Datapoints* and *Prerequisites* and can be marked as once only. Signals and datapoints can be dragged from the right-hand-side into the multi-coloured boxes.

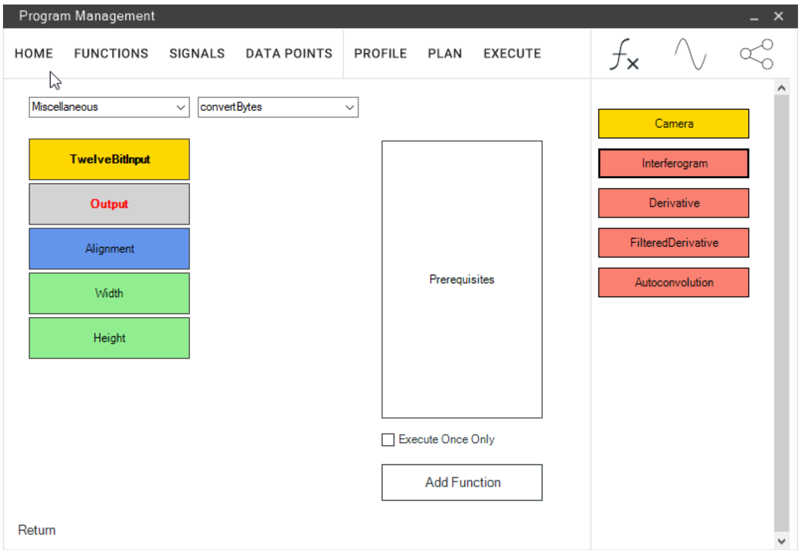


Figure 5-6: Configurator: Add New Function

Generally, the colours of the signal and datapoints must match the boxes for the functions, except for grey boxes. Grey boxes are for variables that use templates and accept multiple data types. When a template variable exists in a function, the compatibility of the data type used is checked against its configuration file before it can be added. Bold labels represent signals whereas non-bold labels represent datapoints, finally red labels signify that this variable is modified by the function. A full breakdown can be found in Figure 5-7.

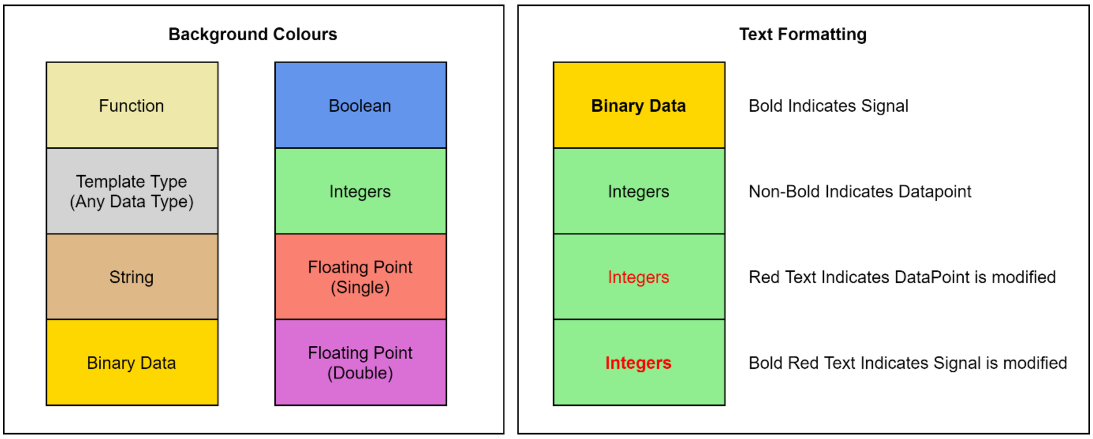


Figure 5-7: Configurator: Colours/Properties of Boxes

Finally, when all the signals and datapoints have been added any prerequisite functions can be added. Only functions that are immediate prerequisites need to be added, with any previous prerequisites being carried forward within the previous function. This adds constraints to check whether or not a function can be executed at a specific time, if this was not implemented it could be assumed that all functions could run in any order. The option "Execute Once Only" marks the function as an initialisation process and only run it once before the rest of the functions. This allows functions like data importing to be run at the start and then use the data throughout, increasing performance by not having to execute it each cycle. A completed function can be seen in Figure 5-8. If the variables used for the template variables match the runtime options specified in the configuration file it is added, otherwise any problematic signals and datapoints are removed.

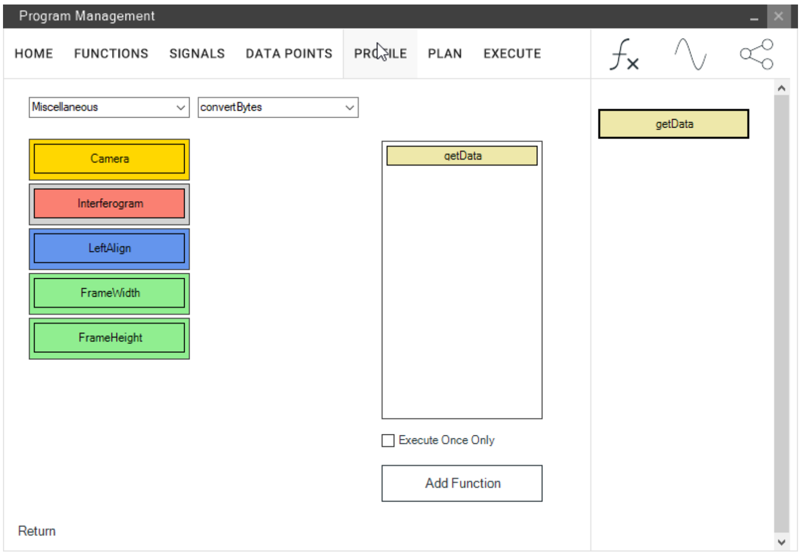


Figure 5-8: Configurator: Add New Function (Completed)

Once all the signals, datapoints and functions have been added, the user can click ‘Profile’ and two configuration files are created; CPU and GPU (Figure 5-9). At the same time, the Configurator places the correct function calls into each function header file (Figure 5-10) for both CPU and GPU profiling, after this the SPE needs recompiling, this is discussed in the future work section. Each of the configuration files (Figure 5-11) is loaded separately into the SPE to begin profiling the signal processing functions selected. These files are loaded by sending the location of the file as a runtime argument to the SPE executable along with the information on what architecture it should profile for and the profiling parameters. These parameters tell the SPE when it should stop profiling; this is discussed in detail in Chapter 6.

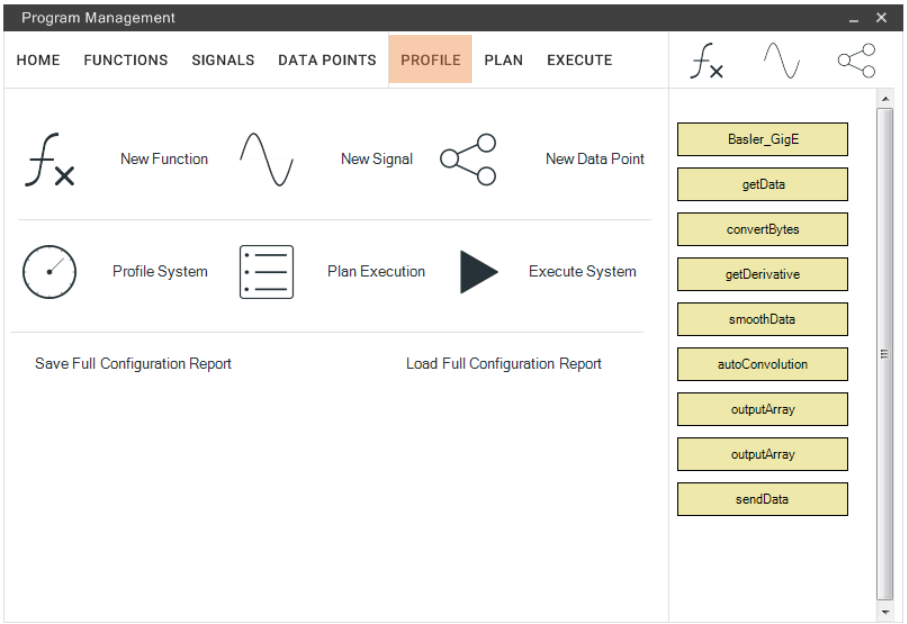


Figure 5-9: *Configurator: Completed Signal Processing Chain Ready For Profiling*

```
Function<float,float>(F[0].CPU_Signals[0]->fData,F[0].CPU_Signals[1]->fData,
F[0].CPU_DataPoints[0]->iData,F[0].CPU_DataPoints[1]-iData,
F[0].CPU_DataPoints[2]->fData);
```

Figure 5-10: *OSPW Created Function Call. This has two Signals and three Datapoints. It also specifies floats to be used as the template types*

```
<Signal>
      <ID>0</ID>
      <Name>Lambda</Name>
      <Type>float</Type>
      <Length>640</Length>
      <Width>1</Width>
      <Depth>1</Depth>
</Signal>
<DataPoint>
      <ID>0</ID>
      <Name>Lambda_A</Name>
      <Type>Double</Type>
      <ReadOnly>true</ReadOnly>
      <Value>-0.0000032722</Value>
</DataPoint>
```

```
<Function>
      <FunctionConfig>
             <Name>windowSignal</Name>
             <Ref>38</Ref>
             <GPU>false</GPU>
             <Hidden>false</Hidden>
      </FunctionConfig>
      <ID>4</ID>
      <Type>1</Type>
      <Platform>0</Platform>
      <Thread>0</Thread>
      <Signals>1,4</Signals>
      <DataPoints>9,10,4,26</DataPoints>
</Function>
```

Figure 5-11: *OSPW Configuration File Extract*

Once these files have been created and updated, they are sent, by refence, to the Signal Processing Engine (SPE), where they are imported and interpreted in C++.

5.3 Investigation of Configurable Signal Processing Methodology

Once the configuration has been finalised it can be imported to the second part of OSPW, the Signal Processing Engine. This is where all the signal processing is executed using C++ for serial code and CUDA for GPU accelerated functions. It takes the XML configuration file created by the configurator and imports it into a selection of custom classes configuring each function to the user's requirements.

5.3.1 Standardised Function Files

Function files follow a standard template (Figure 5-12), the template contains file definitions, file includes and a class that encapsulates the signal processing code and configuration functions. Each function is enclosed within its own class to enable further standardisation when configuring and specific configuration functions to allow the configurability needed in the function, without having to change the processing code. The GPU function call contains the calculation of the blocks and threads needed, and then calls the actual kernel separately, this kernel must be defined outside the class as the class does not exist in the GPU's memory. As each function has a standard format, it allows for the Configurator to edit these files and insert the correct configurations into the CPU and GPU configuration sections. Additionally, if new algorithms are added to the library in the future by the user, OSPW can programmatically create new function files by inserting the relevant CPU/GPU code where necessary.

```
#ifndef [FUNCTIONNAME]_H
#define [FUNCTIONNAME]_H
#include “CPUHeaders.h”
#include “GPUHeaders.h"

//GPU Kernel Header

class [FUNCTIONNAME] {
private:
   //Any Private class objects are defined here
public:
   //Serial Function Call

   //GPU Function Call

   //CPU Config – Add to Queue

   //GPU Config – Add to Queue

}; // - End of Class

//GPU Kernel - This has to be stored outside of
the class
#endif
```

Figure 5-12: *Function Class Template*

5.3.2 Passing Run Parameters for Configuration

To allow OSPW to communicate between the UI and SPE, run-time parameters are added to pass data between them. When the UI is instructed to launch the SPE it passes the file location of the loaded configuration file to the SPE along with any other parameters. These are; whether the SPE should profile or simply execute, the tolerance of the median when calculating the average and how many medians in a row are needed for OSPW to be sure that the median calculated is stable and repeatable (discussed in Chapter 6). These parameters are loaded by the SPE at the very start of the programme and cannot be changed unless the SPE is restarted.

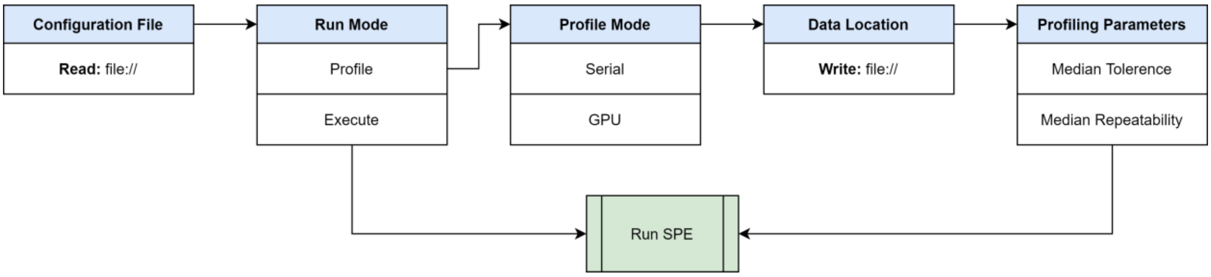


Figure 5-13: *SPE Run Parameters check at launch*

5.3.3 Parsing XML Data to Configure Signal Processing

All the configurable data is stored within the six custom classes listed below (Figure 5-14). For each *Signal*, *Datapoint* and *Function* the user creates in the configurator, it is stored within these classes. When the SPE first loads the configuration files, these classes are populated.

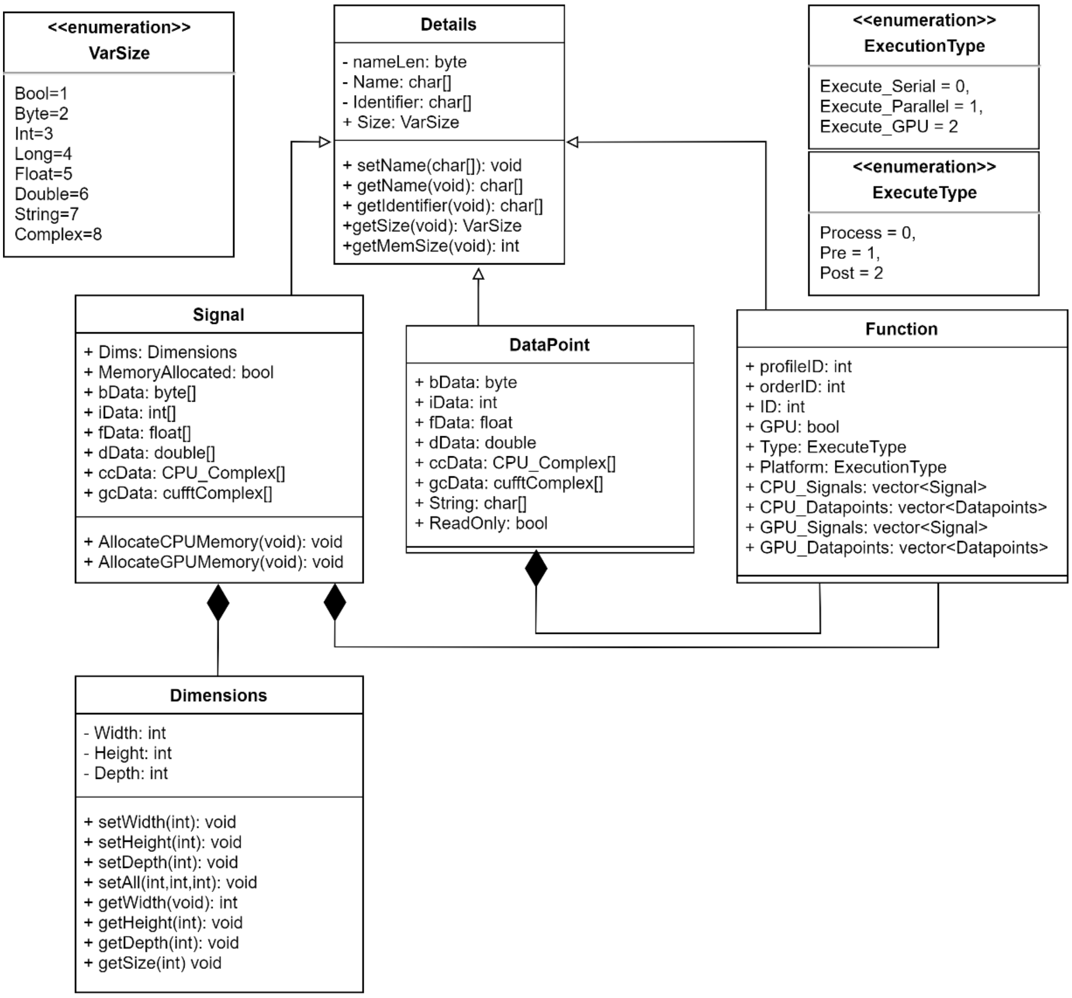


Figure 5-14: *Data Classes*

Dimensions

This sub-class contains the values of the dimensions of the parent class. It is used within the *Signal* class to store up to three dimensions of the signals, these can then be accessed later for calculating GPU thread needs and CPU/GPU memory transfers. It is kept as a sub-class rather than integrated into *Signal* to keep member functions tidy.

VarSize (Enum)

VarSize is an enumeration rather than a new class and is used in the *Details* sub-class to identify the data type being used for that variable. This identifier is used when allocating the memory to ensure the right memory space is given for the variable type used.

Details

This base class is used to store the text data of signals and datapoints and includes the enumeration *varSize* and the three-letter identifier. The identifier is used in the header information when transferring data from the SPE to the UI and is matched to the stream management. The *Details* class is defined as a parent class as it is used in deriving the *Signal* and *Datapoint* classes.

Datapoint

This derived class is for storing single value variables. Variables such as the width of an image being processed or a multiplication factor to be used in a multiply function. These are stored within a union to minimise memory overhead. datapoints can be stored as read-only or read and write depending on if they need to be modified during the SPE execution, this is set in the configurator when creating a datapoint.

Signal

The Signal class is also derived from the details class and is similar to the datapoint class, but has two extra members, *Dimensions* and *memoryAllocated*. *memoryAllocated* is set to true once the memory allocation function has been run and the memory have been allocated to the signal. This is checked before functions with the variable can be used.

Function

The *Function* class is also derived from the details class and contains all the information regarding each function including its name, *orderID,* which instructs the function where it needs to be called within the sequence and the *ID* which stores which case it is in the switch. The execution type informs the SPE whether it is to be run before, after or within the main loop, allowing functions to be called once at the beginning or the end, saving valuable time and the *Platform* selects whether to execute on the CPU or GPU. Finally, there are vectors that store the pointers of the already created signals and datapoints needed for the function. These are stored as pointer references and not duplicated, this allows data to be modified by one function and used in another, without having to pass data between functions themselves, only the pointer.

To create and fill the classes described above, data needs to be imported, from the file created by the configurator. This is then loaded into the SPE upon starting and allows the dynamically created class objects to be configured correctly (Figure 5-15).

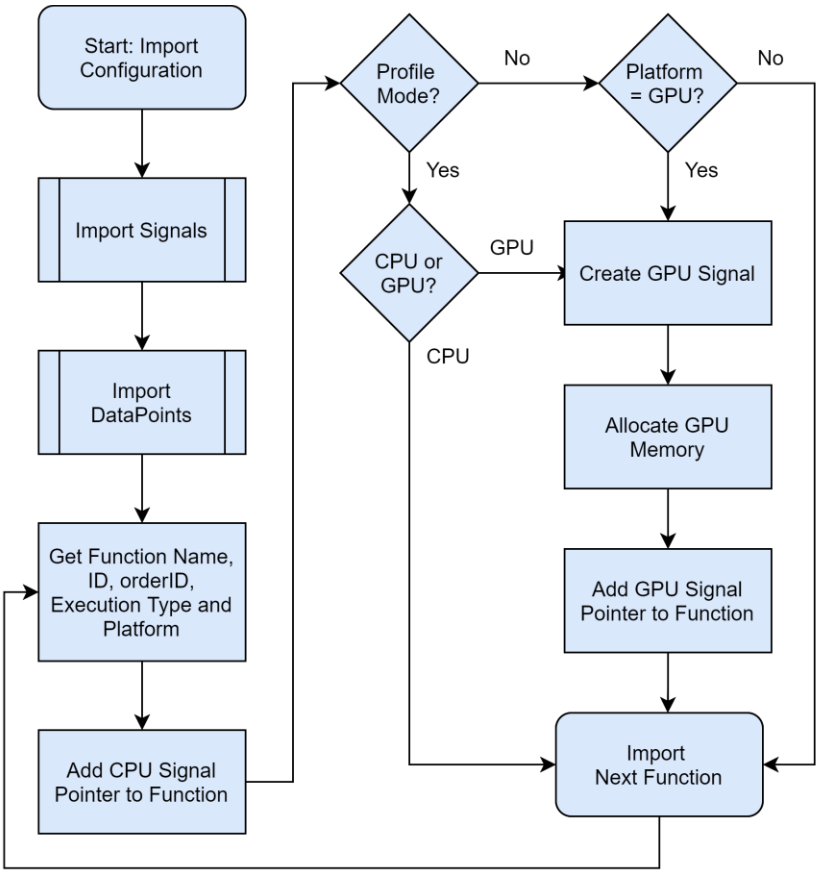

Start: Import Configuration
Import Signals
Import DataPoints
Get Function Name, ID, orderID, Execution Type and Platform
Add CPU Signal Pointer to Function
Profile Mode?
No
Yes
CPU or GPU?
GPU
CPU
Platform = GPU?
No
Yes
Create GPU Signal
Allocate GPU Memory
Add GPU Signal Pointer to Function
Import Next Function

Figure 5-15: *Data Importing Process*

In the SPE there are five vectors created, two for signals (CPU and GPU), two for datapoints (CPU and GPU) and one for functions. First, all the signals and datapoints are imported into their CPU vectors, secondly the functions are imported one-by-one (Figure 5-15). As each function is loaded, a pointer to each signal and datapoint required is placed in the respective CPU vectors (Figure 5-14). Then, if GPU profiling is specified or if the function is configured to be executed on the GPU, a copy of the CPU signal/datapoint is created and added to the corresponding GPU signal or datapoint vector at the same index. This simplifies any CPU to GPU memory transfers as they can be accessed at the same index .

This also makes providing the correct variables to function calls very straightforward, as the compiled code can access the variables in an increasing order, therefore it is critical that in the configuration file, variables are placed in the correct place and required in the correct order.

5.3.4 Enabling Runtime Polymorphism

The signal processing library contains the commonly used functions, but these functions need to be configurable to the user's requirements, otherwise they are just as restrictive as existing methods. To make these functions configurable they must have parameters that can be set at runtime, something that is not as straight forward as it might seem in C++.

The first problem is data types, as the signal processing functions are customisable, one user may require an integer parameter, whereas another may want to use a decimal (floating-point) value. C++ has two methods for achieving this (Horton, 2004), in the first method C++ allows multiple versions of a function with the same name and selects the correct version based upon the data types sent to it; this is called overloading (Figure 5-16). The second method uses templates and allows a single form of the function and different data types to be used (Figure 5-17). This significantly reduces the amount of code repeating and reduces any potential for error, unfortunately both these methods require knowing which variables and type(s) at compile time and not at runtime.

```
void Function(int *Input, int *Output, int Width, int Height, int Parameter);
void Function(float *Input, float *Output, int Width, int Height, int Parameter);
void Function(float *Input, float *Output, int Width, int Height, float Parameter);
void Function(double *Input, double *Output, int Width, int Height, int Parameter);
void Function(double *Input, double *Output, int Width, int Height, float Parameter);
void Function(double *Input, double *Output, int Width, int Height, double Parameter);
```

Figure 5-16: *Overloaded Function*

```
template<class Type1, class Type2>

void Function(Type1 *Input, Type1 *Output, int Width, int Height, Type2 Parameter);
```

Figure 5-17: *Template Function*

To combat the need to know the variables at compile time, multiple classes have been created to house the variables and select the correct one at runtime. Each function has a list of all the signals and datapoints used in the function and is configured to use each signal and datapoint in sequence (Figure 5-18). These signals and datapoints have been populated during the loading and is configured to the users requirements. This gives the compiler a known memory location where the data is eventually going to be stored without knowing data sizes or types but allows the user to configure the function later to their needs.

```
template<class Type1>

void Function(Type1 *F.CPU_Signals[0], Type1 *F.CPU_Signals[1],
int F.CPU_DataPoints[0], int F.CPU_DataPoints[1], int F.CPU_DataPoints[2]);
```

Figure 5-18: *Function with Function Class Variables*

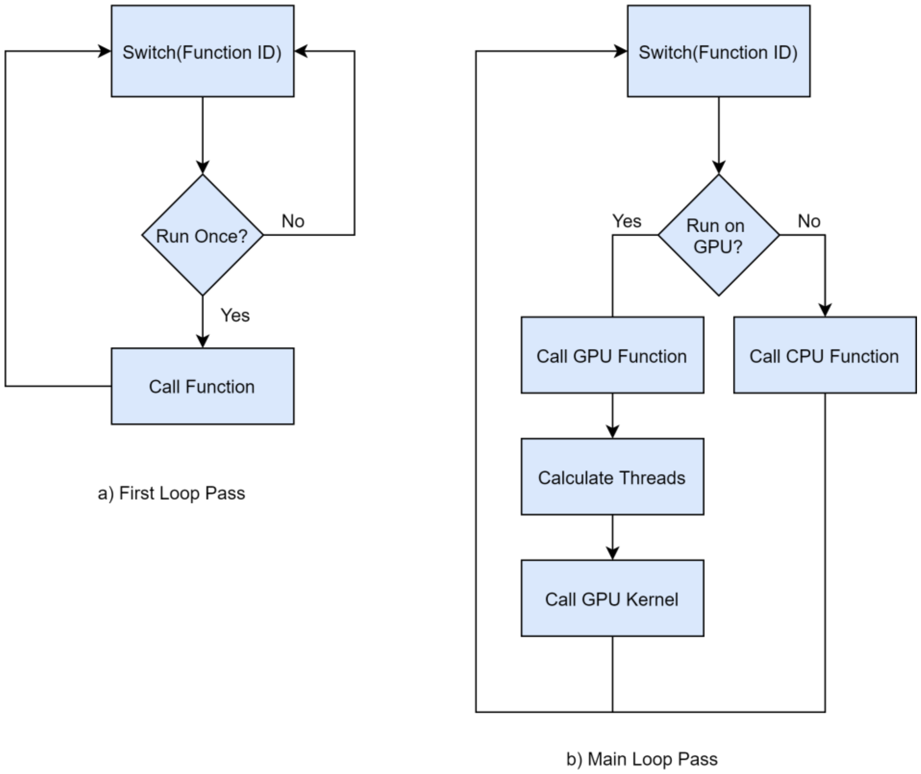
The final part of this configurability is the union structure used for the signals and datapoints, “A union is a user-defined type in which all members share the same memory location” (Microsoft, 2019). To be able to store all the different types of data that the user may want to use, the union allows the storage of all the available datatypes and when loaded allocates the correct size memory to the correct variable. Due to the above limitations, and the need to specify a variable type at compile, the configuration needed by the user is required to be hard coded into the program before compile time. Due to the checks in the UI code, there should be no errors generated as it has all the information it needs to make sure the code it generates is compatible at compile and runtime. This is because each function’s configuration file details the runtime options to ensure compatibility.

```
union {
   byte *bData;
   int *iData;
   float *fData;
   double *dData;
   OSPWComplex *ccData;
   cufftComplex *gcData;
};
```

Figure 5-19: *Variable Union*

5.3.5 Configuring the Execution of the Correct Functions

Once all the functions are imported and the vectors are filled, the functions can be called. Firstly, the loop is run once (Figure 5-20a) to execute any functions that have the *ExecuteType* value of 1, these are functions only to execute once at the beginning. This could be functions like camera setup, importing background profiles etc., these only need to be run once at the start, and this saves time later, rather than unnecessarily running these each time.



a) First Loop Pass

b) Main Loop Pass

Figure 5-20: *Function Calling*

Which of the different functions to be executed is derived from the configuration file that has been imported into the *Function* vector (Figure 5-14). This vector is then iterated though looking at the *ID* parameter and finding the corresponding case from the switch statement used to organise the different function calls (Figure 5-21a). However, this was inefficient, having to search for the matching case statement each time had an impact on performance, therefore a better solution was required.

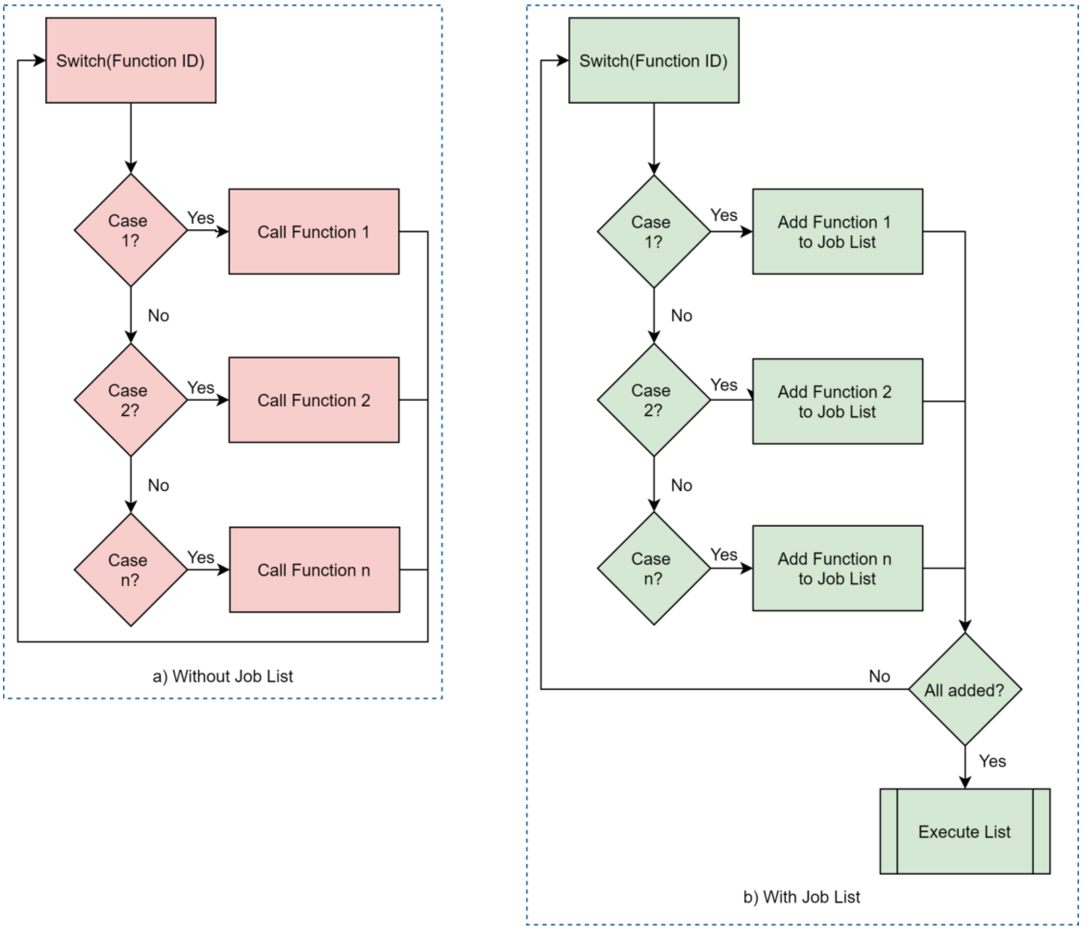


Figure 5-21: *Job Management - Main Switch*

The solution was to create a job management class to manage the profiling and function execution (Figure 5-21b). The function vector still must be looped through once to determine which switch case needs to be called, but instead of calling the function and moving on, it is added to a list in the job management class.

The job management class is possible due to Variadic templates (Microsoft, 2016) and C++ Lambda functions (Lischner, 2013). Function calls themselves cannot simply be added to a list and then executed in sequence, there is no standard datatype that would allow that, as each function is different and has a different number of parameters, that is where Variadic templates are used. Variadic templates allow a variable number of parameters to be sent to a function, but these are still not formatted in a manner to be entered into a list.

To solve this, an intermediary function is employed to convert the standard function call into a void pointer that can then be saved to a list. A void pointer is simply used as a common type, and refers to an object of any type. This intermediary function uses a lambda expression to take the function and its arguments passed via a variadic template, combined into a void pointer and then inserted into the job list along with the function parameters for use later (Figure 5-22). By also including the function parameters, when the system is profiling it can relate the data gathered back to the specific function and its execution within the job list, this removes any potential for error at a later stage, also if a function is called multiple times, it removes any ambiguity as to which iteration it is.

```
class WorkItem {

   function<void()> Func;
   Function *F;
   template<class _Fn, class... _Args>
   void Create(_Fn&& _Fn, _Args&&... _An) {
      Func = [_Fn, _An...]() { (_Fn)(_An...);};
   }

   void Execute() {
      Func();
   }
}
```

Figure 5-22: *Sample Code of Variadic Templates & Lambda Function*

Using the job management class, the *Function* vector is iterated through, and for any function required to run on each loop, its configuration function is called. From this configuration function, it adds the CPU or GPU version of the function to the job list. Once the SPE reaches the end of the *Function* vector, the job list can start executing the CPU/GPU functions. After each job has executed, it is placed in a 'done' list, this continues until the job list is depleted (Figure 5-23). Some housekeeping is undertaken before the job list can be repopulated, such as saving timing data, alerting the UI that data is ready to be sent, once this is completed, the job list is then refilled from the 'done' list and everything repeats.

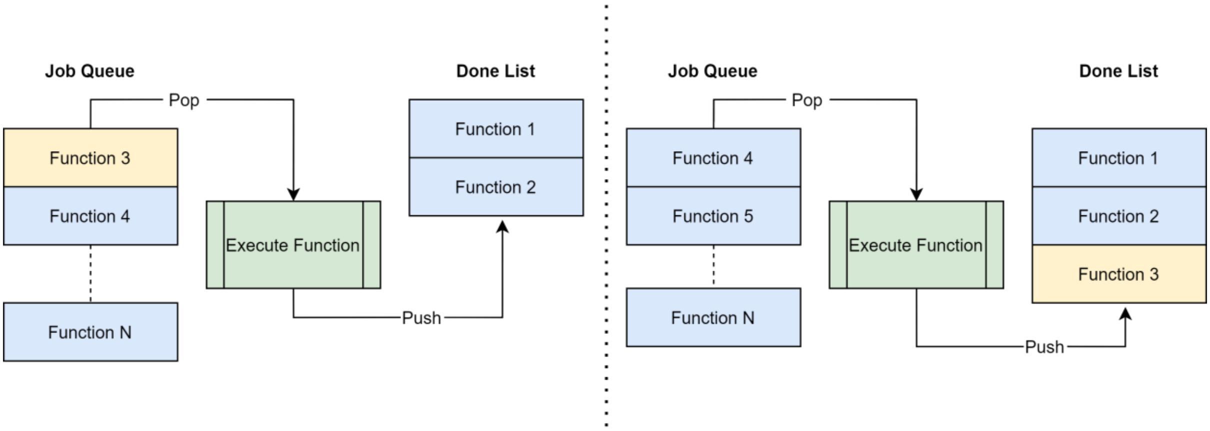


Figure 5-23: *Job Queue Pop & Push*

5.3.6 Introduction of CUDA Accelerated Signal Processing Functions

GPU functions are launched differently to CPU functions and includes two additional steps before the CUDA kernel can be launched. The GPU function has already been configured and added to the job list, but rather than the signal processing code being in the function call, it has an added layer to first calculate the threads needed, before executing the kernel itself. The intermediate layer is there as all the configuration code and serial function are stored in a class, but the kernel cannot be contained within a class due to the class object being created on the host but executed on the device.

Any data required by the GPU function is not transferred as part of the function itself, instead being copied before and after the function where required (Figure 5-24). Separate functions manage the memory transfers between the host and device, which are created by the configurator and inserted into the configuration file. The configurator uses the *MemoryCopy* property from the function configuration files to determine if data needs copying to the device before the function or to the host after the function. This enables the signal processing functions to operate in isolation, and if memory transfers become unnecessary, i.e. if multiple GPU functions are chained together, the memory transfer functions can be removed from the job list, rather than having to re-program the GPU call itself.

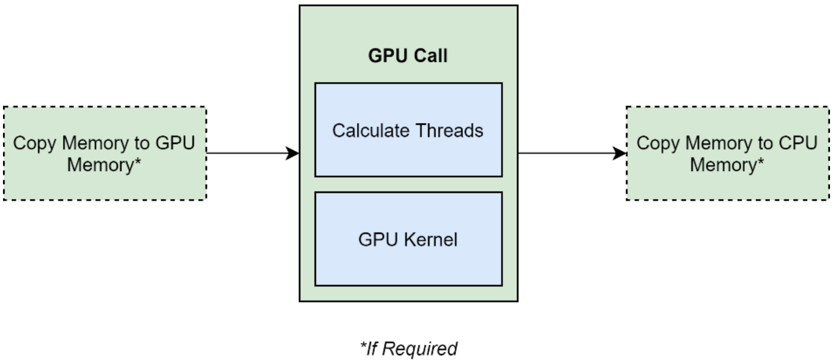


Figure 5-24: *GPU Function call and potential memory transfers*

Threads are calculated as discussed in Chapter 4.6.3, illustrated in in Figure 4-20, and launched in three-dimensional blocks of 1024 threads each. Each of the signal processing functions have parameters defining the width, height and depth of the data passing through it, and the threads are calculated based upon these values. This thread calculation is done every time the function is called and not simply the first time, this is in case the parameters being used to calculate the number of threads is modifiable by a previous function, and therefore the number of threads required changes over time. An improvement that could be made here, is a check on the thread calculating parameters, checking their read and write status, if they are read only, save the thread and block values to be used repeatably, saving time not having to recalculate a fixed value.

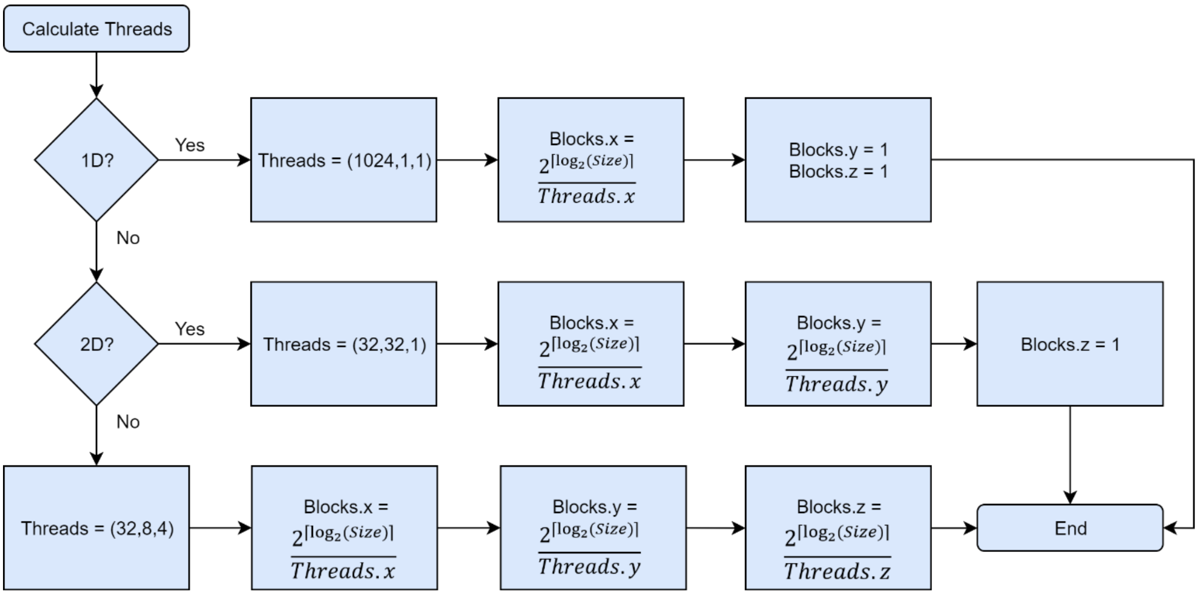
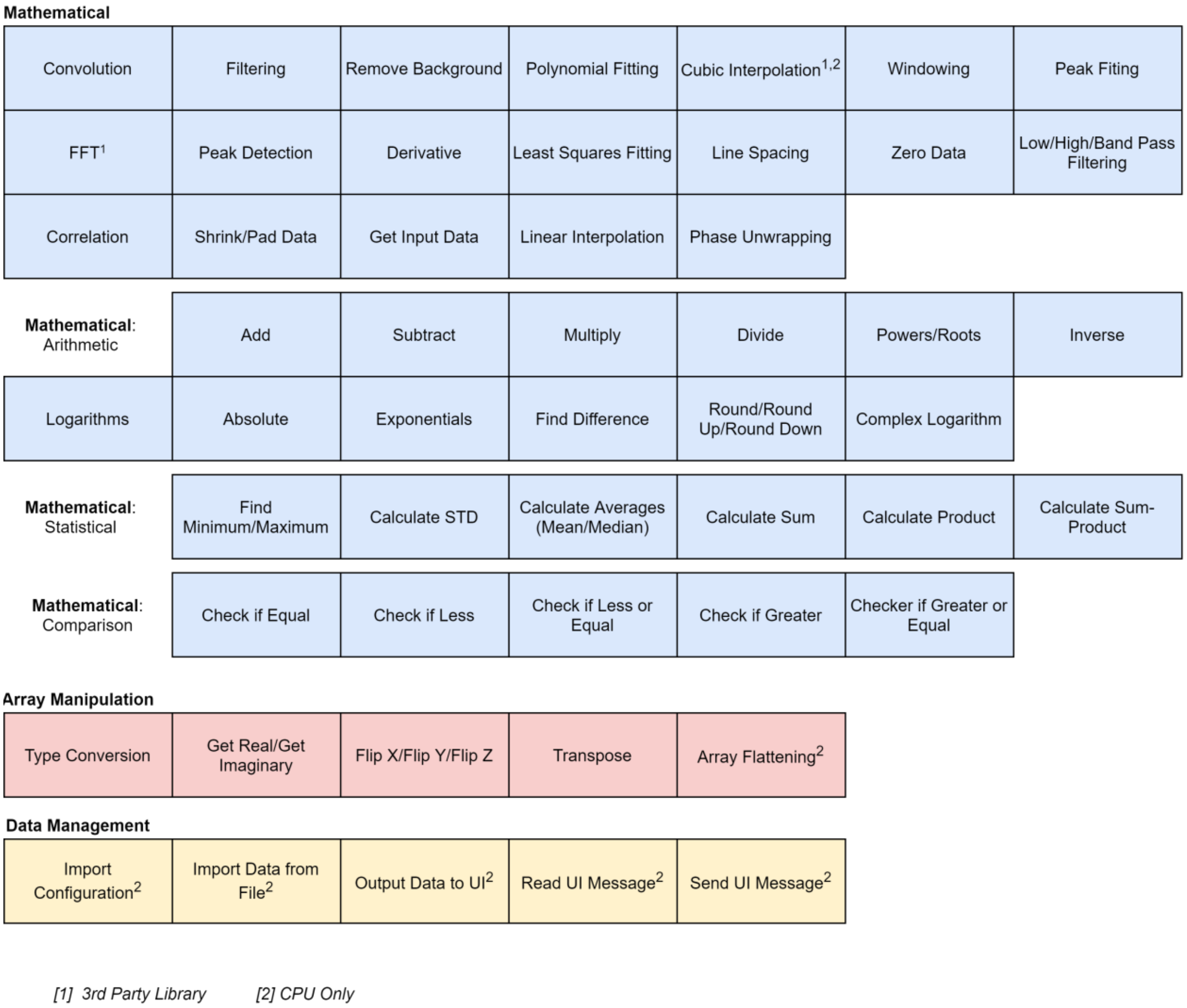

Calculate Threads
1D?
Yes
No
Threads = (1024,1,1)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = 1
Blocks.z = 1
2D?
Yes
No
Threads = (32,32,1)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.y}$
Blocks.z = 1
Threads = (32,8,4)
Blocks.x = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.x}$
Blocks.y = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.y}$
Blocks.z = $\frac{2^{\lceil \log_2(Size) \rceil}}{Threads.z}$
End

Figure 4-20: *CUDA Thread Allocation Calculation*

5.3.7 Prospective List of Signal Processing Functions

From the literature reviewed in section 2.4, several algorithms were highlighted as common across optical surface and dimensional measurements. The majority of these have been programmed in an initial library for use across many measurement techniques (Figure 5-24). All the functions allow up to three-dimensional data inherently for anything from 1D single point to 3D areal measurement or batches of data containing one or two dimensions. The functions created fit into three main categories:

1. Mathematical
2. Array Manipulation
3. Data Management



Mathematical

Convolution	Filtering	Remove Background	Polynomial Fitting	Cubic Interpolation[1,2]	Windowing	Peak Fiting
FFT[1]	Peak Detection	Derivative	Least Squares Fitting	Line Spacing	Zero Data	Low/High/Band Pass Filtering
Correlation	Shrink/Pad Data	Get Input Data	Linear Interpolation	Phase Unwrapping		

Mathematical: Arithmetic	Add	Subtract	Multiply	Divide	Powers/Roots	Inverse
Logarithms	Absolute	Exponentials	Find Difference	Round/Round Up/Round Down	Complex Logarithm	
Mathematical: Statistical	Find Minimum/Maximum	Calculate STD	Calculate Averages (Mean/Median)	Calculate Sum	Calculate Product	Calculate Sum-Product
Mathematical: Comparison	Check if Equal	Check if Less	Check if Less or Equal	Check if Greater	Checker if Greater or Equal	

Array Manipulation

Type Conversion	Get Real/Get Imaginary	Flip X/Flip Y/Flip Z	Transpose	Array Flattening[2]

Data Management

Import Configuration[2]	Import Data from File[2]	Output Data to UI[2]	Read UI Message[2]	Send UI Message[2]

[1] 3rd Party Library *[2] CPU Only*

Figure 5-25: *OSPW Function Library*

5.3.7.1 *Mathematical Functions*

The mathematical functions are made up of five sub-categories; first- and third-party functions consist of the bulk of the signal processing techniques used in data processing. The other three categories concern basic arithmetic operations, statistical operations such as averages and standard deviation and finally comparison operations e.g. if statements.

5.3.7.1.1 First Party Functions

The function library has been extended since the first case study for the DRI adding extra functions some of which are used by the LSDI. The functions initially created for the DRI have been updated to accept three-dimensional data, and all new functions also support this. Almost all of the new functions added were based upon MATLAB code being used for LSDI but have been generalised to allow for any size data and single or multiple dimension images. The majority of algorithms have also been translated into CUDA kernels for GPU acceleration, with the acceptation of the cubic interpolation due to being provided by a third-party library. These have all been tested and their results compared to their serial counterparts to check for mathematical consistencies ensuring each function returns the same values regardless of architecture.

All the functions have been compared to their MATLAB counterparts to check for differences in the results calculated, the largest error calculated was 1.72 nm, below the resolution of the LSDI instrument. Further investigations into why this error occurred found this was down to the differences in the cubic interpolation algorithms, the MATLAB used a third derivative of the signal to calculate interpolation properties, whereas the C++ implementation used the second derivative. This was restricted by the third-party implementation.

5.3.7.1.2 Third Party Functions

In addition to the custom functions that were created some third-party mathematical libraries were utilised for their speed and efficiency. For fourier transforms, CUFFT included in CUDA's toolkit is used on the GPU, and for the CPU, FFTW, created by the Massachusetts Institute of Technology (MIT) (Frigo & Johnson, 2005), is used under the General Public License (GPL). The free edition of ALGLIB by ALGLIB Project was also used, a numerical analysis and data process library, for the cubic interpolation function. Although ALGLIB is free to use, they limit the license to GPL 2+, removing any possibility of commercial distribution without providing source code, something that may need to be reviewed in the future. ALGLIB do offer a commercial version, but the source code of this cannot be distributed, something that may be a problem due to the current method of OSPW configuring the SPE by injecting source code and recompiling.

5.3.7.1.3 Arithmetic Operations

Standard mathematical operations such as add, subtract, multiply and divide were needed to allow for basic maths to be computed within the platform. Although these are all standard C++ operations that can be used, they needed to be placed within OSPW configurable functions, so the user has the ability to configure the functions. Two versions of each of the above functions needed to be created (Figure 5-26). Figure 5-26a allows a signal to be divided by a single number (datapoint) whereas Figure 5-26b divides one signal by another, both these are commonly used. Other mathematical functions that were added include logarithms, exponentials, indices, absolute, inverse, rounding, sine, cosine and tangent.

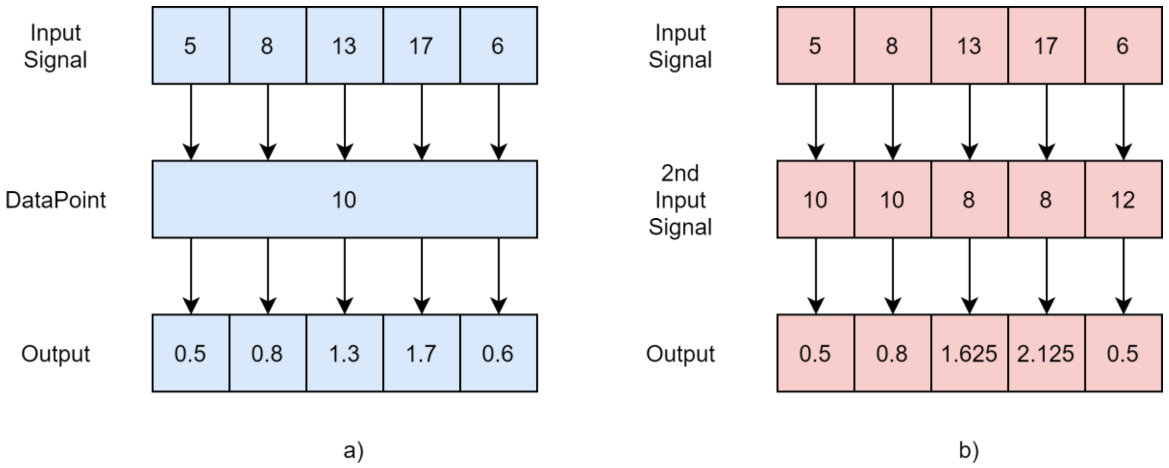


Figure 5-26: *Mathematical Functions Example (1D Representation)*

5.3.7.1.4 Statistical Operations

Statistical functions such as finding the minimum, maximum, mean, median and standard deviation are commonly used in algorithms such as correlation and normalisation. These functions are configurable and can be used to find the statistical method in any dimension or combination thereof (Figure 5-27). Other functions that are grouped within this category also include calculating the sum of an array and the product of an array, both used in other algorithms.

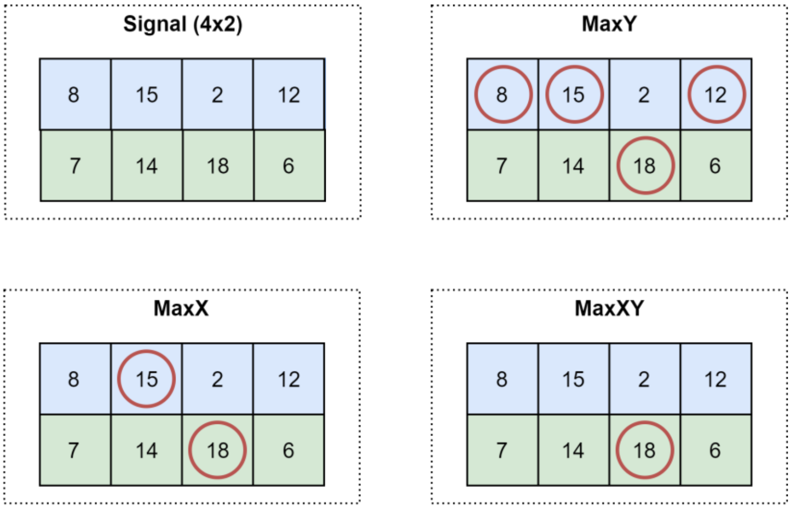


Figure 5-27: *Statistic Functions Example (2D Representation)*

5.3.7.1.5 Comparison Operations

This is the smallest section of functions, the comparison functions allow a u to do one of three things; check each value in a signal against a single datapoint Figure 5-28a, check a signal against another signal Figure 5-28b, or compare two datapoints. Therefore, there are three versions of each of five comparative operations, check if equal, check if less than, check if more than, check if more than or equal, check if less than or equal. Each of them has an option to simply return true or false, or a different value if true or false.

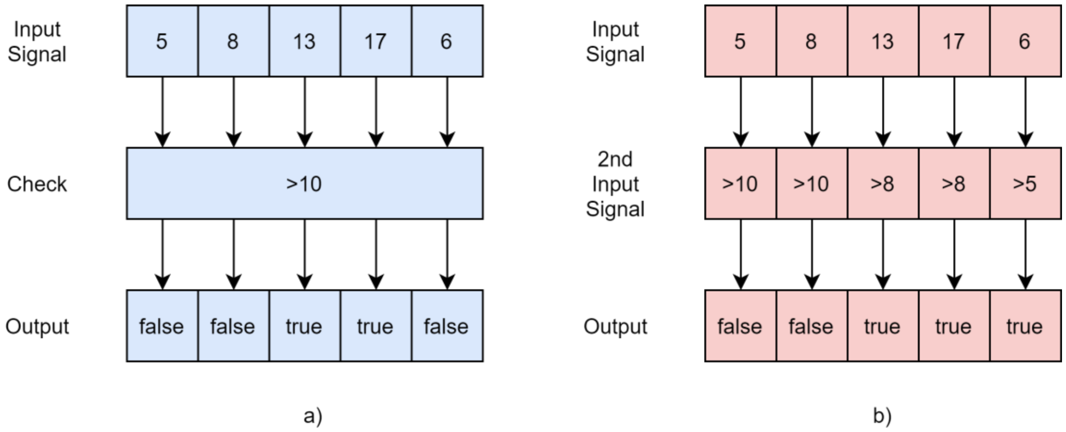


Figure 5-28: *Logic Function Example (1D Representation)*

5.3.7.2 *Array Manipulation*

Array Manipulation algorithms are very important and commonly used in all data processing. Operations such as ‘reverse the signal in any of the three possible dimensions (or combinations thereof) (Figure 5-29) and also transposing a 3D array into any new order. There are a few other functions that fall into this category, such as separating the parts of a complex number (real and imaginary) and also sorting an array highest to lowest or lowest highest (by dimension or overall).

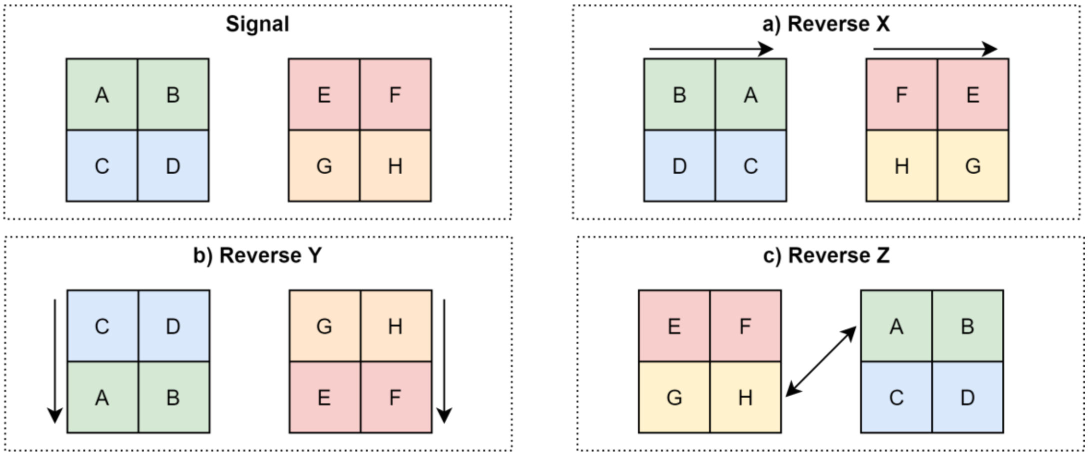


Figure 5-29: *Signal Manipulation Example (2D Representation)*

5.3.7.3 *Data Management*

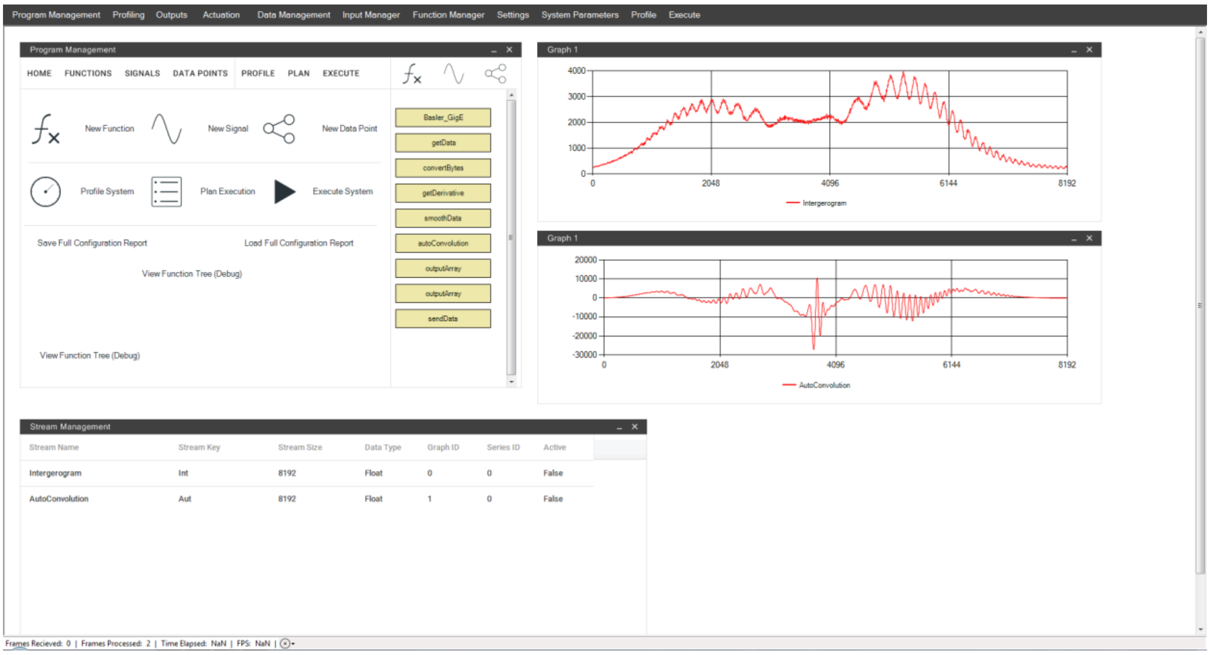
Two functions are responsible for caching the data to be transferred to the UI, *outputArray* and *outputDatapoint*. Both these functions add header information about the data and the data itself to a vector that is transferred to the UI when the job list is empty. The header information contains an identifier to the signal that is created by the configurator and stored on the UI and also the timestamp of when the data was captured (Figure 5-30). Data is then transferred over a named pipe on a separate thread once the job queue is empty, this allows the signal processing to continue and start the next cycle while the data is being transferred. If the data transfer takes longer than the signal processing, multiple frames are cached together and sent at together.

3 Bytes	4 Bytes	Height*Width*Depth*DataSize Bytes
Signal Identifier	Timestamp	Data
INT	1745631523	™µª ¾yd@DÓýÃh@b†ˆh@ø=ôy|

Figure 5-30: *Header and Data sent from SPE to UI*

5.4 User Interface: Visualisation

The final of the three parts that make up OSPW is the visualisation and UI objects. The main UI has changed from a single user form, to multiple windows that can be placed within the main UI window, this is known as a Multiple Document Interface (MDI) (Figure 5-31). Child windows created such as SPE configurator, stream management, function manager, actuation and settings can be placed within this window, but not outside it. To help with window alignment the UI has been split into 25x25 pixel squares that child windows snap to the nearest grid corner. While the SPE Configurator is the focus of the pre-signal processing, the Stream Management is the focus of the post-signal processing and data visualisation. From this window the user can control what is shown on the screen, whether that is graphs or images and also control what data is saved.



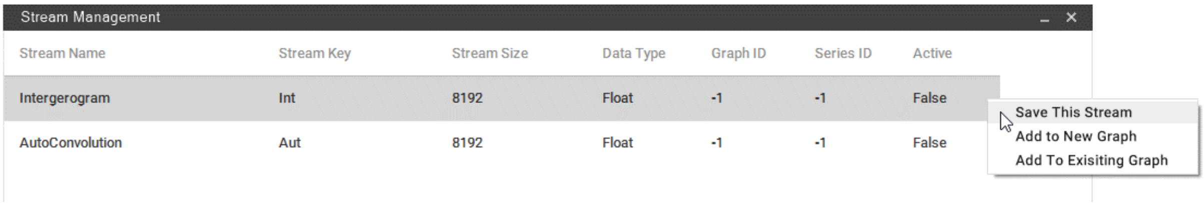
Stream Name	Stream Key	Stream Size	Data Type	Graph ID	Series ID	Active
Intergerogram	Int	8192	Float	0	0	False
AutoConvolution	Aut	8192	Float	1	0	False

Frames Recieved: 0 | Frames Processed: 2 | Time Elapsed: NaN | FPS: NaN |

Figure 5-31: *User Interface Multiple Document Interface (MDI)*

5.4.1 Management of Data Streams Between Applications

When the *outputArray* and *outputDatapoint* functions are configured in the configurator, the signals and datapoints that are referenced are added to the stream management. The signal dimensions are also included enabling the stream management to know what size data to expect when decoding the data received (Figure 5-32). The header information for each signal and datapoint includes the *Stream Key* to identify which variable has been transferred from the SPE but does not need to include and size or data type information. The stream management keeps track of which graph interface the data is being displayed on, with the option to plot multiple signals on the same graph, or different graphs. The stream management is also where users can opt to save the stream data to a file, this data is saved in the raw binary format that is imported to the UI and is saved until disabled.



Stream Management

Stream Name	Stream Key	Stream Size	Data Type	Graph ID	Series ID	Active
Intergerogram	Int	8192	Float	-1	-1	False
AutoConvolution	Aut	8192	Float	-1	-1	False

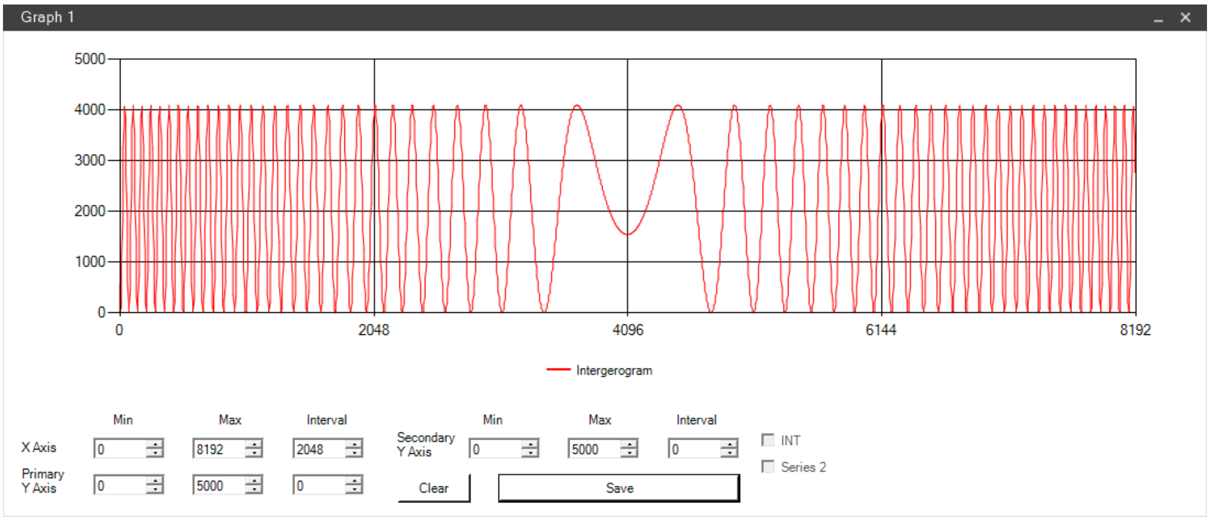
Figure 5-32: *Stream Management Window*

5.4.2 Visualisation Objects of Processed Data

The main aspect of a user interface is the visual graphs and images that represent what is being measured, OSPW provides both these features. Multiple signals can be placed on a single graph and multiple graphs can be displayed at once, also if saving BMP images, these can too be displayed live in the UI.

5.4.2.1 Graphs

Graphs have changed very little since the original case study. They have been split from the single form into separate MDI child modules enabling any number of graphs to be created from the same template (Figure 5-33). This also allows graphs to be placed anywhere on the screen, snapping to the previously described grid . The graphs still auto-scale based on the first data they receive and can be manually changed below. Multiple signals can also be placed on the same graph on either the same or different y-axis', this is all controlled from the *Stream Management* window. As before, the data is decimated by an integer that gives the closest number of pixels to the screen width, to save time drawing each data point (Equation 4-1). Two-and-three dimensional graphs do still need implementing and is discussed in the further work section.



	Min	Max	Interval
X Axis	0	8192	2048
Primary Y Axis	0	5000	0
Secondary Y Axis	0	5000	0

□ INT
□ Series 2

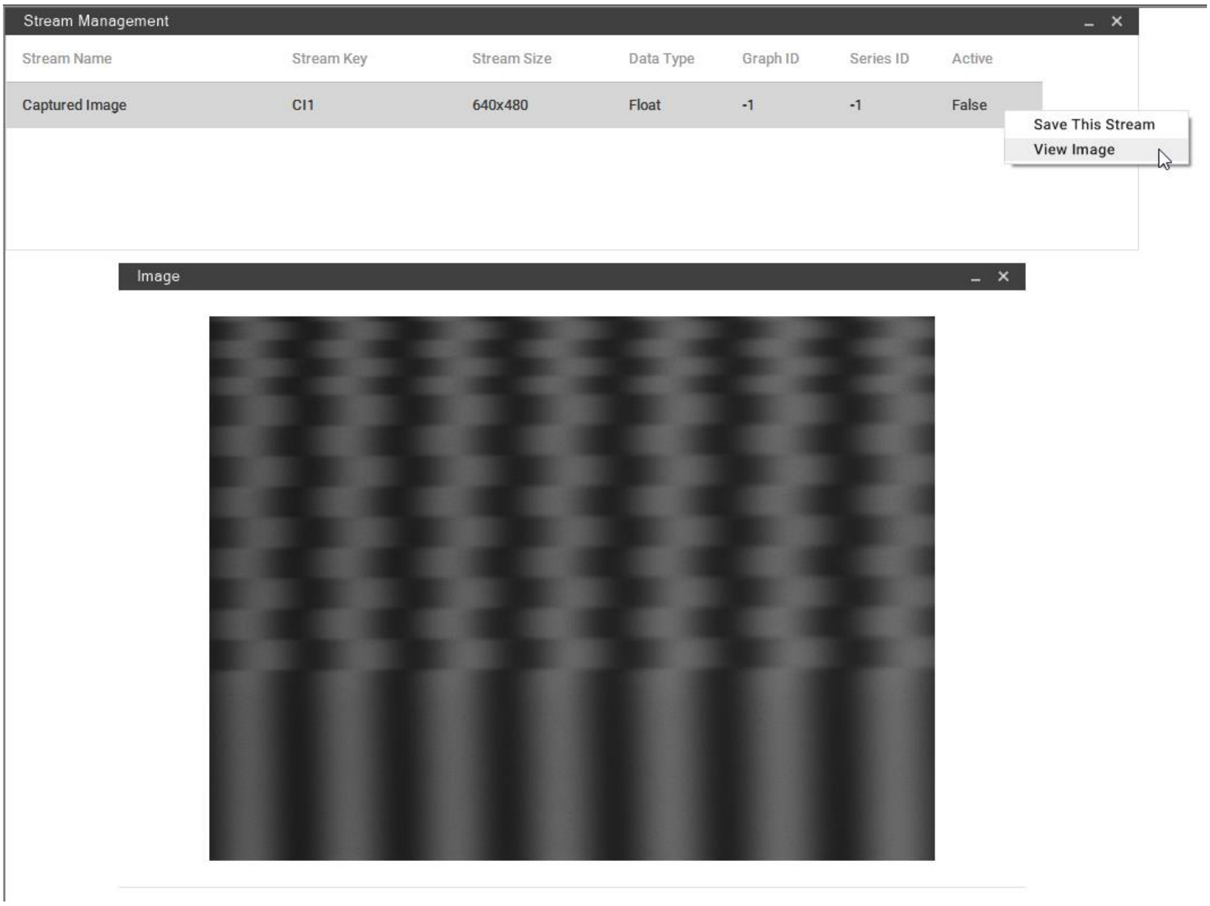
Figure 5-33: Multiple Document Interface Graph Window

$$DecimationRatio = \left\lfloor \frac{DataLength}{ScreenResolution} \right\rfloor$$

Equation 4-1

5.4.2.2 Images

Live 2D images can also be displayed from the stream management (Figure 5-34). If the *saveBMP* function is selected in the SPE configurator, the option to display the image appears in the stream management when the SPE is running. This image is loaded from disk and therefore may have performance limits but should update at the same time as any other data is transferred from the SPE, and when the job list has finished. The ability to visualise 3D Standard Data Format (SDF) files is a possibility for the future and is discussed in the future work section.



Stream Name	Stream Key	Stream Size	Data Type	Graph ID	Series ID	Active
Captured Image	CI1	640x480	Float	-1	-1	False

Figure 5-34: *Multiple Document Interface: 2D Image*

5.5 Conclusion

OSPW provides a simple to use interface to configure the signal processing required by a user without the need for the user to understand or learn a programming language. Using simple drag-and-drop interface, powerful signal processing functions can be configured for one, two- or three-dimensional data and have been validated to ensure their mathematical consistency with MATLAB. OSPW also provides live data visualisation and the ability to save data allowing users to post-process or evaluate data later if necessary.

Currently two versions of each function exists, one for the CPU and another for the GPU, but being able to determine which of these is fastest for a given function, would allow OSPW to consistently execute the optimum signal processing chain given the data and processing requirements given by a user. Having previously discussed the potential use of artificial intelligence in chapter 3, a heuristic is needed for a planner to optimise for. Therefore, the following chapter will evaluate a method of profiling the execution time of a function will be evaluated to discover if execution time can be used as the heuristic for optimisation.

6 Evaluation of Profiling to Determine AI Heuristics

6.1 Introduction

To be able to optimise the signal processing for greater performance, the current performance will be measured to be used as the PDDL heuristic. To measure this execution time, each signal processing function on both the CPU and GPU will be timed (profiled). Profiling of the signal processing may begin once the infrastructure is complete and commonly used signal processing functions have been added to the library, with the majority also having a hardware accelerated version using a graphics card.

OSPW needs to determine which architecture is optimal for an application, this is achieved by measuring the execution time of each individual signal processing function on the CPU and GPU. Immediatly before and after each function is called, a timestamp, the number of CPU ticks since the OS started, is recorded at a resolution of approximately 285 ns. Once each of the functions is profiled, the fastest architecture, or combination of architectures, is identified and used to execute the signal processing in the optimum configuration for the processing requirements and hardware available.

6.2 Consideration of Optimisation Parameters

As previously discussed, whether in an academic research laboratory, or on a production line performance is critical. If a measurement cannot be taken in real-time it might not be desirable to even take the measurement, leading to potential defects and expensive rework. Therefore, this research has primarily focused on execution time of signal processing sequences, or more accurately, the latency of the measurement. The faster the measurement data can be captured and processed, the lower the latency of the result, however if latency is not critical for an application, throughput could provide further performance gains.

GPUs processes data in parallel rather than sequentially like a single-thread CPU would do. Therefore, there can be benefits to processing data in batches to utilise the GPU to its full capacity. Theoretically, computing the signal processing for two frames on a CPU would take twice as long, however on a GPU, if GPU load for a single frame is below 50%, should only take marginally longer than one frame. The signal processing time per frame itself will increase very little, however memory transfer times will scale linearly with the data size once the maximum memory bandwidth is reached.

However, in some cases latency, or throughput, may not be the critical parameter and instead might be memory, storage capacity, network transfer speeds, or power consumption. Due to the requirement for inherent flexibility in the proposed solution, there is a significant memory requirement, due to allocating all possible memory at the beginning, to prevent time lost to allocating and deallocating memory. Nevertheless, unless capturing very large images, or creating a large number of arrays, memory should not be a limiting factor when compared to the 128 GBs of RAM most CPUs support. There is a possibility this may be a limiting factor for GPU memory for very large images; however, GPUs are shipping with more VRAM each generation, with Nvidia's 3000 series shipping with up to 24 GB and their Ampere Quadro cards providing up to 48 GBs (Nvidia, 2021). Memory usage is however easy to calculate and could be used as a secondary heuristic in the PDDL domain.

The consideration of power consumption is growing with the need to be greener becoming part of every company's strategy map, with Wolpert, Kempes, Stadler, and Grochow (2019) estimating that computers make up 5% of all electricity used in the USA and Europe. While transistor sizes have decreased, increasing efficiencies, GPU dies have become larger and power usage has increased over the last decade, with the high-end GPUs consuming around 300 W (Nvidia, 2021). However, the use of AI planning to optimise for power efficiency would is an interesting discussion and will be detailed in the future work section.

Storage speeds and or network transfer speeds are much harder to measure or determine due to various factors. Storage speeds massively differ depending on the size of the data being read or written, for example 1000 files at 1 MB each will take longer than one file at 1000 MB due to the sequential nature of the single file write, compared to the random write of 1000 files. In SSDs, read and write speeds can also change depending on the size of the storage device used, with larger devices offering better performance due to the increase in individual NAND chips. Although recent advances with PCIe generation 4, write speeds can be found upwards of 5 GB/s (Digital, 2021), therefore storage write speeds should not be a bottleneck.

Finally, network speeds are even harder to predict and measure than storage speeds, as depending on the current network traffic can have a direct impact on each individual's IPC capability of accessing or writing to network devices. However, 10 Gbps network connections are not uncommon anymore, with 25 Gbps and 100 Gbps being introduced and is therefore unlikely to be a limiting factor for researchers or industry.

To summarise, the heuristic that is used in the domain file can be any measurable parameter, however due to the requirement for real-time processing, time (latency) is used as the primary cost function in this research. However, both latency and throughput will be evaluated to discover the performance benefits of processing multiple data sets at once for applications that does not require low latency.

6.3 Investigations into Profiling Methodology

Before profiling, OSPW's user interface generates two configuration files for the signal processing required, one for CPU profiling and one for GPU profiling. The CPU configuration file contains the list of signal processing functions needed that run on the CPU (Figure 6-1a), these are run on a single thread, except some input and output functions which operate on separate threads as they can be running while other signal processing is taking place. The second configuration is the GPU configuration and is very different to the CPU configuration. To execute functions on the GPU, any required data needs to be transferred to the GPU memory beforehand, also any changes or new data created must be transferred back again if it is used by the CPU later (Figure 6-1b). Therefore, the GPU configuration contains memory transfer functions, before and after the GPU kernel, to allow profiling of the memory transfers that are required to execute GPU code correctly. This is because if a GPU kernel is faster, but requires a lengthy memory transfer, it can make the overall GPU function slower. When profiling the GPU, any functions that do not have a GPU counterpart will run on the CPU in the same way as the CPU profiling method. This profiling is only run if GPU hardware is detected.

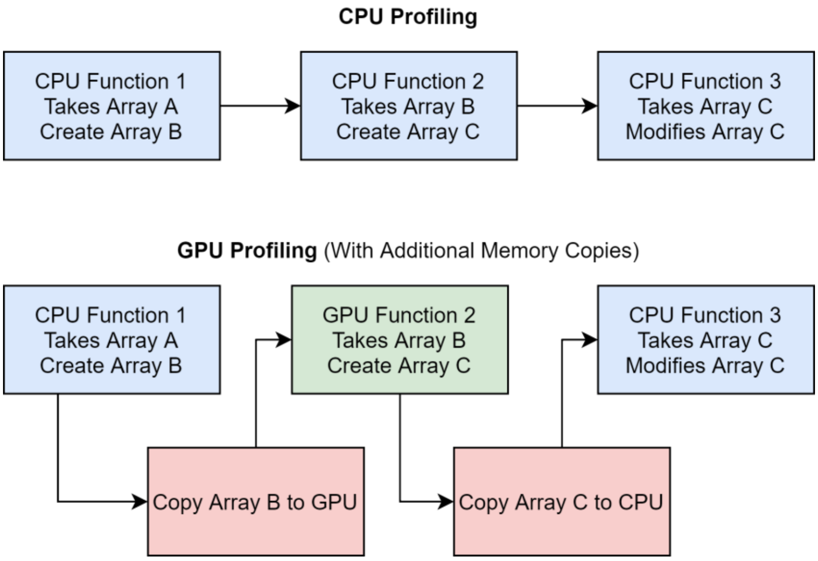


Figure 6-1: *Additional Functions needed for GPU Profiling. Using three arrays as examples, the data needs to be copied backwards and forwards between the CPU and GPU to execute correctly.*

Once the configuration is imported and the function vector is filled, several if statements allow OSPW to operate in a few different modes (Figure 6-2). Initially, it checks whether the system is running in profile mode, it then checks which architecture it should profiling for and then calls the respective CPU or GPU launch function. If profiling has already been completed, and OSPW has created an optimum configuration file, OSPW operates in execution mode and checks which architectural platform it needs to use for each function before execution.

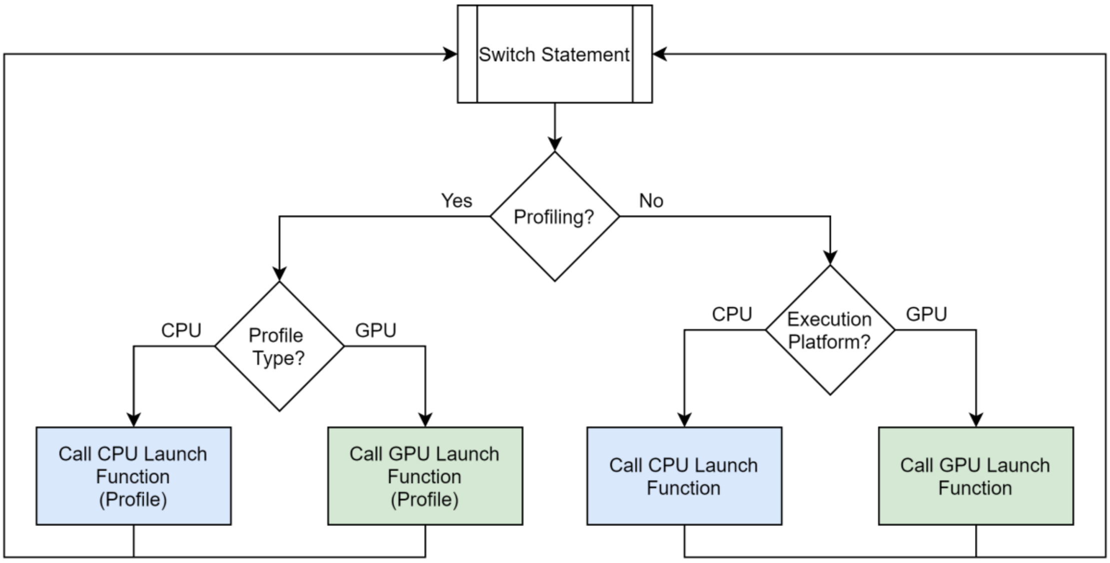


Figure 6-2: *OSPW Profiling Decision – Profiling commences unless an optimum configuration already exists*

Once all the launch functions have been called through the above process, the job queue is populated and OSPW starts iterating through the list. When profiling, OSPW requests the current timestamp from the Windows performance counter. The performance counter enables high resolution (<1 µs) measurement as it counts the number of CPU ticks that have elapsed since system start-up (Figure 6-3). Once each function from the job list is profiled, a second timestamp using the same performance counter is taken, the difference between the two values is then calculated. This value is then used to compare the execution time of a CPU function vs a GPU function.

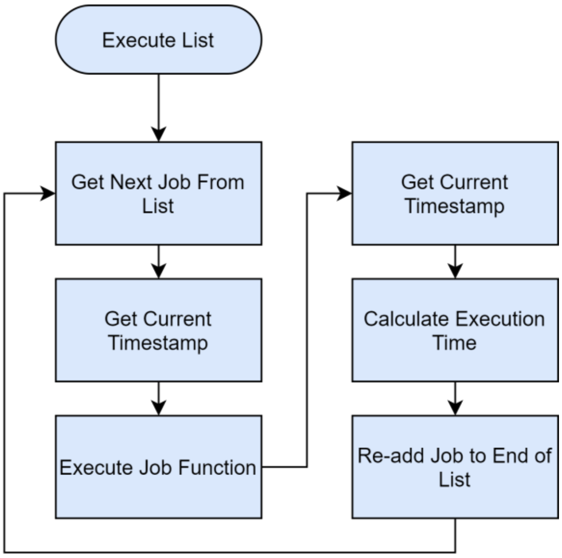


Figure 6-3: *Job Management – Profiling*

As OSPW has been designed on Windows, and not on a Real Time Operating System (RTOS), function execution time varies each time the function runs. Since many events run simultaneously within the General Purpose Operating System (GPOS), with little or no priority given to one single task, software often has to wait for the next free processing cycle, which in turn leads to a fluctuating and unpredictable execution time. To minimise this unpredictability, multiple runs of each function are processed and a statistical measure can be calculated. Although the execution time does fluctuate from the average that is calculated, profiling is designed to determine if, on average, the CPU or GPU is the fastest for that specific function and not accurately predict what the OS will be doing for each microsecond it is running for.

It is predicted that the range of execution times on the GPU is less than that of the CPU as it is likely that the GPU is idle while not being used for this processing. However, secondary functions used for memory transfer functions are required to allow data to move between the CPU and GPU memory which requires CPU processing cycles which is affected by the nature of the GPOS. When developing an algorithm to calculate the average execution time of a signal processing function, several statistic methods were considered; mean, median and 90th and 95th percentile.

To establish which average type returns the most reliable and repeatable value, the signal processing for both the DRI and LSDI has been profiled and the data gathered is evaluated to find the best average type to use to decide on the best architecture for each individual function during profiling. As can be seen from the graphs in Figure 6-4, the data does form a normal distribution but is heavily right-skewed, therefore the mean is situated to the right of the median. This is to be expected when measuring time due to the processing being measured having a finite lowest value, but an infinite upper limit. The data shows that although the mean over a long period of time is possibly more representative, it is skewed due to high-value anomalies. Therefore, the median is a more reliable and repeatable average to use than the mean.

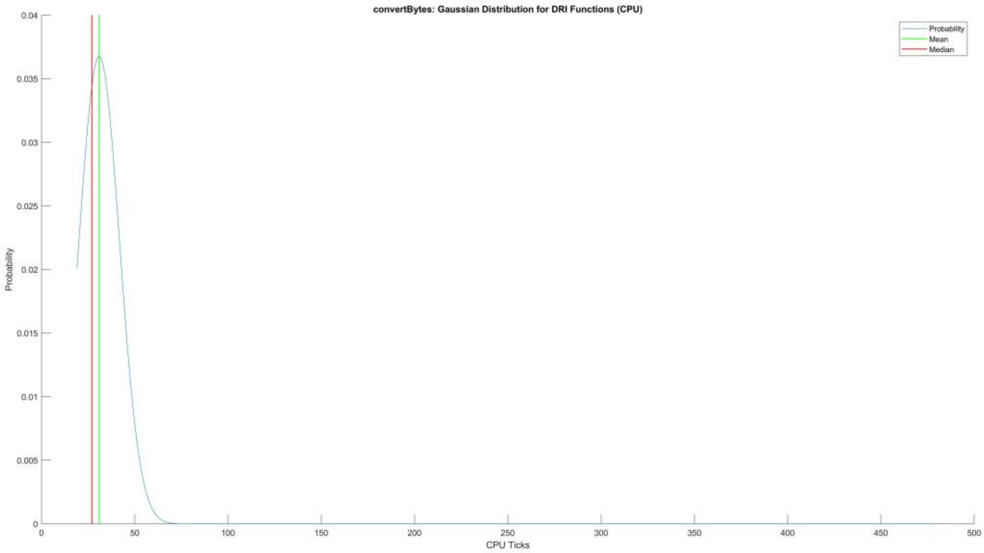


Figure 6-4: *Normal Distribution plot of DRI Function on the CPU*

Next the 90th and 95th percentiles were compared. Analysing the data for two functions; one from the DRI (Figure 6-5) and one from the LSDI (Figure 6-6), after a period, both these percentile measures stabilise. The value at which these parameters stabilise at is not too important, the reliability and repeatability of the data is what is necessary for profiling, to be able to accurately compare the two architectures. Therefore, looking at the graphs, the median stabilises before either of the percentiles therefore profiling the system faster. Calculating the median is much quicker than calculating a percentile, saving time during the profiling of the system too, something that becomes important when profiling many functions on two different architectures.

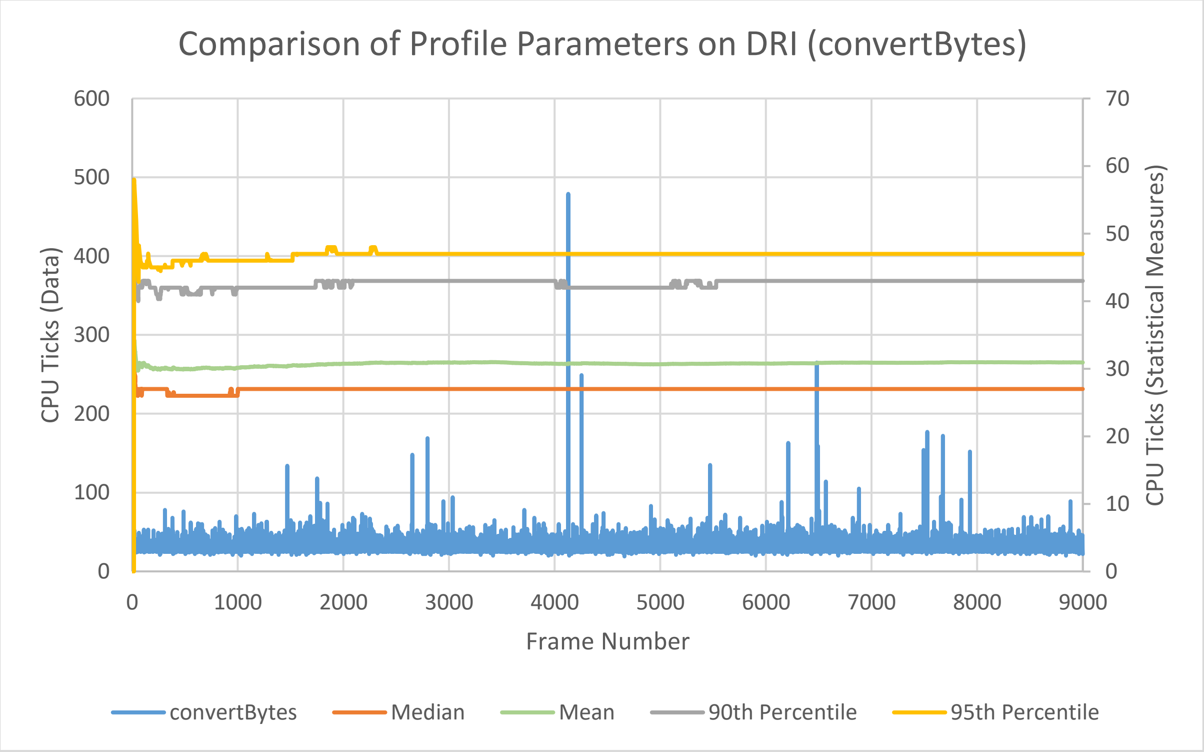


***Figure 6-5:** Statistical Calculations of DRI data on convertBytes function*

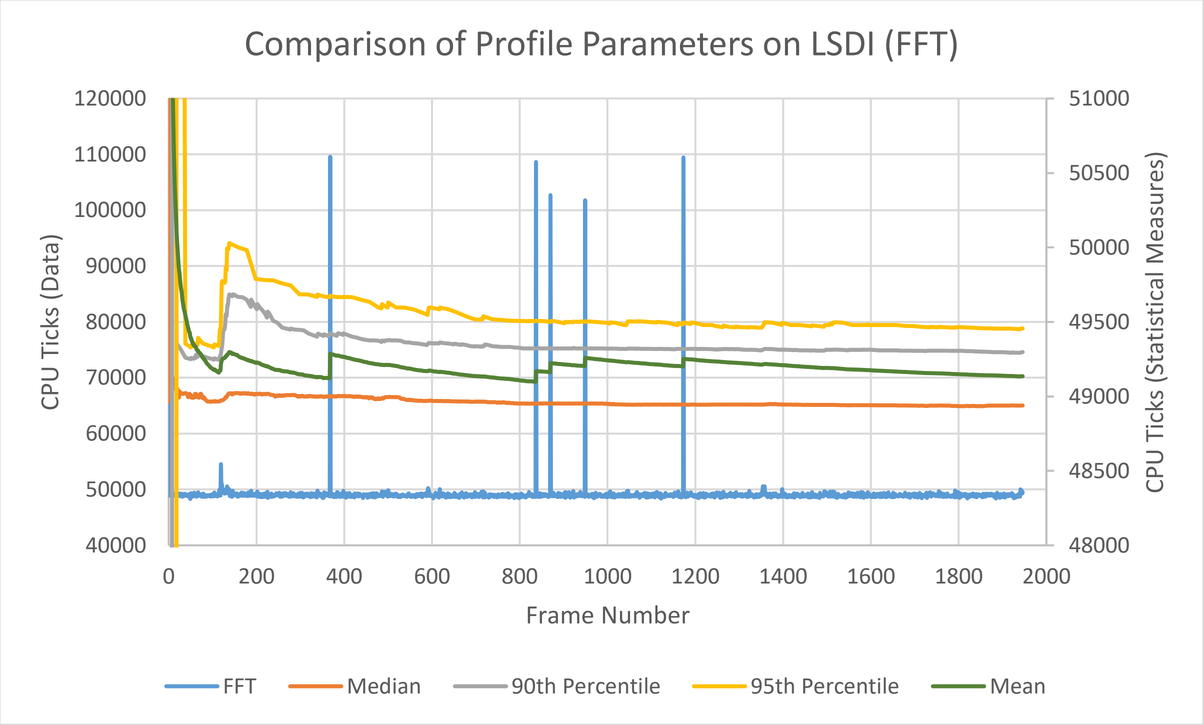


***Figure 6-6:** Statistical Calculations of LSDI data on FFT function*

When profiling, OSPW needs to decide if enough profiling has been recorded, and a stabilised median has been selected to be the indicator, OSPW calculates a median at particular intervals, and when this value has stabilised, OSPW can stop profiling. Reviewing the data recorded in the previous test and evaluating where the median reached a stable value, the DRI ran at around 581 FPS and reached a stable median after around 1000 frames on average, however, some functions took longer to stabilise at around 5000 frames. LSDI ran slower due to the increase in signal processing requirements and data size, running at around 7.4 FPS taking around 900 frames on average to reach a stable median. This means the DRI reached a stable median in a maximum time of ten seconds whereas the LSDI took almost three minutes.

As the function execution takes longer, the range of values increases, and due to there being less data, the median fluctuates more as the probability of median changing is higher at in smaller datasets. Therefore, a median is calculated every 1000 frames or ten seconds, after five consecutive medians that are within 1% of each other, that function is regarded as successfully profiled. Once all functions reach this status, profiling is stopped for the selected platform and the results saved for further analysis.

6.4 Comparison of CPU vs GPUs for DRI and LSDI

Both case studies discussed in this research have been profiled on their respective CPUs and a selection of different graphics cards. DRI was profiled on an Intel i7-4790 CPU, with the LSDI using an Intel i7-7820X. For the graphics cards, a selection of Nvidia graphics cards across different generations were used to represent the hardware researchers may have available, these were; GTX 520 (DRI Only), GTX 650Ti, GTX 780Ti and GTX 1070. Each function was executed, including any memory transfers to/from the GPU that are required for GPU kernels to execute correctly. Additionally, OSPW has been evaluated against standalone software for both the DRI and LSDI to ascertain if any performance deficits exist by using it.

The GPUs being used to test are compared in Table 6-1 and Figure 6-7. Generally, the more CUDA cores, faster memory and higher floating-point performance, the better it should perform. Nvidia's naming scheme for their GPUs is mostly straight forward, the first digit (or two digits for four-digit numbers) denotes the generation, ten is newer than seven which is newer than six. The next digit indicates the tier of graphics card, again larger is better, the last invariably always being zero. Finally, the Ti (Titanium) indicates a slightly more powerful version of the standard card, the GTX 780Ti has more performance than the GTX 780.

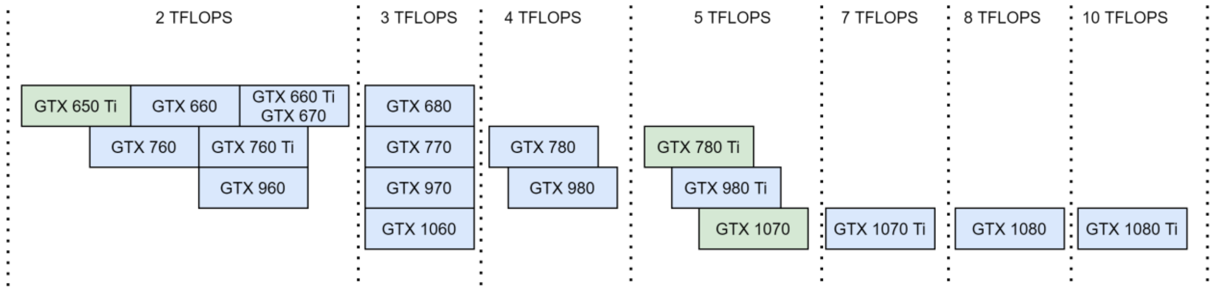


Figure 6-7: *Visual Representation of GPU Card Performance. Cards Highlighted in Green were used for Profiling.*

This is important to note when comparing the GTX 780Ti and GTX 1070, the 1070 is larger number and therefore the assumption is it perform better. Even though the graphics card is two generations newer (Nvidia skipped generation eight), it is two tiers lower, so would be expected to perform about the same, Nvidia tend to improve by one tier each generation. Therefore, when looking at the results, there are likely to be a slight improvement when comparing the two newest GPU, with a larger gap between the rest.

Table 6-1: *Comparison of GPUs used for OSPW Profiling*

	Nvidia GTX 520	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
Clock Speed	810 MHz	928 MHz	876 MHz	1506 MHz
CUDA Cores	48	576	2880	1920
Memory Bandwidth	14.4 GB/s	86.4 GB/s	336 GB/s	256GB/s
Floating Point Performance	156 GFLOPS[2]	1425 GFLOPS	5046 GFLOPS	5783 GFLOPS

6.4.1 Dispersed Reference Interferometry (DRI)

The case study discussed in chapter 3, Dispersed Reference Interferometry , was the first to be profiled to calculate a low resolution, absolute position, and did not contain signal processing to calculate the high resolution as the previous algorithms were no longer being used by the lead researcher. Therefore, profiling is completed for seven signal processing functions, these include one for capturing data from camera and two that output processed data to the UI. The data captured and processed by the DRI is relatively small, 8 KB, and the functions themselves are computationally light, therefore there is no advantage for executing any of the signal processing on a GPU (Table 6-2). There is a time overhead required to launch a kernel on the GPU, this overhead consists of a launch overhead and an execution overhead. The launch overhead is for the CPU to queue up the GPU kernel and prepare it for launching, the execution overhead, the smaller of the two, is associated with initialising the kernel.

[2] *Giga-FLoating Point Operations Per Second*

When profiling the DRI, the overhead is a substantial component of the kernel execution, this is excluding any memory transfers that may be needed if functions were faster on the GPU.

Table 6-2: Comparison of Execution Time (CPU Ticks) for the CPU and various GPUs for DRI – Fastest architecture is highlighted in green

Function	Intel i7-4790 CPU	Nvidia GTX 520	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
getData	103	N/A	N/A	N/A	N/A
convertBytes	14	158	99	107	118
getDerivative	7	160	95	98	117
smoothData	305	1422	627	652	401
autoConvolution	864	7619	1106	5211	1534
outputArray	8	N/A	N/A	N/A	N/A
outputArray	6	N/A	N/A	N/A	N/A

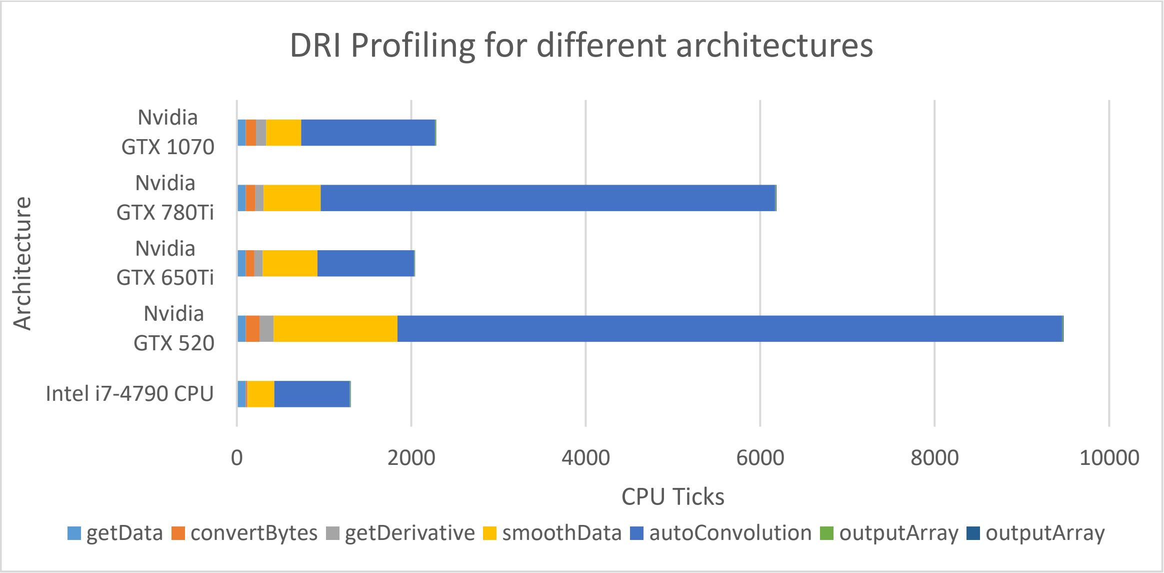


***Figure 6-8:** Comparison of signal processing across different architectures on the DRI – Lower is better*

Closer examination of the GPU execution times, Table 6-3, reveals no single GPU is best across all functions, this is likely due to differences in GPU clock speed and kernel execution overhead time. Most of the functions executed faster as the performance of the GPU increased, except for *autoConvolution* on the GTX 780Ti. Further investigations into these discrepancies showed the auto-convolution took an unexpected amount of time, perhaps due to an outdated driver. However, as long as the metric used is repeatable across the two architectures, the value itself is not as important. A repeatable metric is more important than an accurate metric here. Despite the DRI executing faster on the CPU, the memory transfer times were still measured and recorded and are shown in Table 6-4 and Table 6-5. Again, there is no clear fastest GPU, but as the data is small, there is very little difference across the selection. The largest difference between cards was transferring the auto-convolution signal from the GPU amounting to 1.86 ms between the GTX 520 (the slowest) and GTX 650Ti (the fastest), this likely due to differences in memory bandwidth, memory clock speed or differences in the memory architecture.

Table 6-3: *Comparison of Execution Time (Counter Ticks) for the various GPUs for DRI (Colour scale indicates the range of values, from red to green: worst to best).*

Function	Nvidia GTX 520	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
convertBytes	158	99	107	118
getDerivative	160	95	98	117
smoothData	1422	627	652	401
autoConvolution	7619	1106	5211	1534

Table 6-4: *Comparison of Data Transfers to various GPUs for DRI*

Signal (To GPU)	Nvidia GTX 520	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
Camera	105	97	88	91
Interferogram	111	85	100	93
Derivative	112	85	98	93
Filtered Interferogram	116	85	100	94
Autoconvolution	N/A	N/A	N/A	N/A

Table 6-5: *Comparison of Data Transfers from various GPUs for DRI*

Signal (From GPU)	Nvidia GTX 520	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
Camera	N/A	N/A	N/A	N/A
Interferogram	184	108	128	153
Derivative	169	106	126	145
Filtered Interferogram	160	127	124	132
Autoconvolution	208	118	163	132

6.4.2 Line-Scan Dispersive Interferometry (LSDI)

The LSDI, in contract to the DRI, contains eighteen functions and processes 37.5x more data per function at 0.3 MB per frame. The increase in signal processing steps needed for the LSDI, and the 2D nature of the data, allows the GPU to significantly accelerate some of the algorithms, but not all of them (Table 6-6). A prime example for acceleration is *FFT* (4) and *IFFT* (15) where the NVidia GTX 1070 GPU only takes 1.6% and 2.9% of the time the Intel CPU. By contrast, functions such as *Zero_Start* (7), *Subtract* (11) and *Add* (12) are an order of magnitude faster on the CPU than any of the GPUs, this is expected in these functions contain very few operations and therefore the GPU kernel launch overhead once again outweighs any potential performance benefit.

Table 6-6: *Comparison of Execution Time (CPU Ticks) for the CPU and various GPUs for LSDI - Fastest architecture is highlighted in green.*

Function		Intel i7-7820X CPU	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
1	divide	779	1313	348	275
2	Interpolation	174914	N/A	N/A	N/A
3	FFT	57897	4016	1651	1283
4	Absolute	8926	1169	654	629
5	Zero_Start	18	810	459	467
6	Zero_End	240	808	454	460
7	getMax	5512	15287	13172	10350
8	Logic	18	1195	1251	1102
9	zeroComplex_Start	73	889	514	417
10	zeroComplex_End	937	822	455	392
11	IFFT	49653	5264	2232	1434
12	complexLog	79875	1205	550	463
13	getImag	1979	751	420	399
14	unWrapPhase	5802	16599	15620	6435
15	windowSignal	1263	1017	515	290
16	polyFit	62974	82767	63458	22201
17	calcHeight1	6	506	546	619
18	calcHeight2	196	495	541	619

Evaluating the graph in Figure 6-9 reveals that any of the graphics cards used for profiling provide an overall performance increase above that of the CPU, this includes any memory transfers that are required for the signal processing to work correctly. However, not all functions executing on the GPU are performing optimally, many would benefit from being run on the CPU. Two problems arise from this, firstly for a given function, the CPU may be faster, but the surrounding memory transfers required to move the data between devices may outweigh the CPU performance benefit. Secondly, this problem gets exponentially more complicated, for instance, it may not be optimum to copy data from the CPU to the GPU for a single function, but it might be for the next three consecutive functions, this become a problem that is not easily solved and becomes exponentially harder for the more functions that exist.

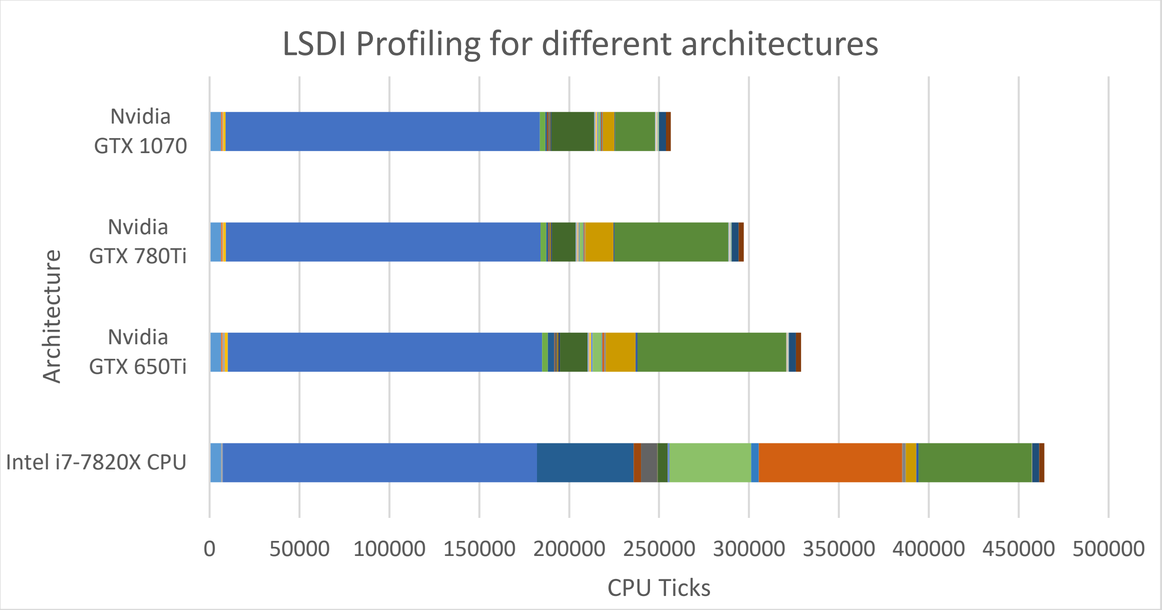


Figure 6-9: *Total Execution time for each architecture (including necessary memory transfers)*

The data in Table 6-7, demonstrates that the Nvidia GTX 1070 is sometimes not the fastest of the various graphics cards tested. Function execution times mostly follow the pattern of getting quicker the higher performance the GPU is, with a few exceptions (functions 17 and 18). These functions only process one data point for each line of the captured image, therefore leading to relatively little signal processing compared to something like *FFT*, therefore even at a maximum of 20% slower, it only equates to 19 µs, and as shown in Table 6-6, for this particular application, the CPU was faster.

Table 6-7: Comparison of Execution Time (CPU Ticks) for the various GPUs for LSDI

ID	Function	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
1	divide	1313	348	275
2	FFT	3586	1176	827
3	Divide2	430	475	456
4	Absolute	1169	654	629
5	Zero_Start	810	459	467
6	Zero_End	808	454	460
7	getMax	15287	13172	10350
8	Logic	1195	1251	1102
9	zeroComplex_Start	889	514	417
10	zeroComplex_End	822	455	392
11	IFFT	5264	2232	1434
12	complexLog	1205	550	463
13	getImag	751	420	399
14	unWrapPhase	16599	15620	6435
15	windowSignal	1017	515	290
16	polyFit	82767	63458	22201
17	calcHeight1	506	546	619
18	calcHeight2	495	541	619

Looking at Table 6-8 illustrates similar anomalies with the Nvidia GTX 1070 periodically not being the optimum graphics card out of the three used for profiling but with larger differences. There are often slight differences due to the OS interrupting the processing, but these differences were measured up to 64%. Although these figures are concerning, this is the exact reason that profiling on the hardware to be used is necessary, these results were run and averaged over five runs to minimise anomalies (Figure 6-10). When an user begins to use OSPW, the profiling measures the likely execution times for the given hardware, therefore, if the hardware is changed at a future point, OSPW would need to re-run the profiling procedure to generate a new optimised configuration.

Table 6-8: Comparison of Memory Transfer Time (CPU Ticks) to and from various GPUs for LSDI

Signal (To GPU)	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
fftResult	13612	13806	13362
fftResult_Real	3825	3905	3837
getHeight	1811	1955	2088
getHeight2	236	274	359
HighCutFreq	347	380	382
ifftResult	11826	12103	11095
interSampleYt1	3086	3178	3063
LowCutFreq	385	408	394
MaxID	353	362	370
Phase	1673	1740	1722
phase_final_c	3701	3749	3299
Polynomial	316	310	375
Signal	859	878	815
unwrapedPhase	3954	3952	3037

Signal (From GPU)	Nvidia GTX 650Ti	Nvidia GTX 780Ti	Nvidia GTX 1070
fftResult	13424	13351	12846
fftResult_Real	3249	3280	3582
getHeight	274	331	440
getHeight2	269	324	440
HighCutFreq	400	409	441
ifftResult	11362	11550	10721
LowCutFreq	405	416	443
MaxID	505	513	577
MaxValue	402	418	448
Phase	2563	2602	2660
phase_final_c	3098	3072	2551
Polynomial	597	620	687
sampleYt1	1504	1433	1287
unwrapedPhase	3542	3394	2931

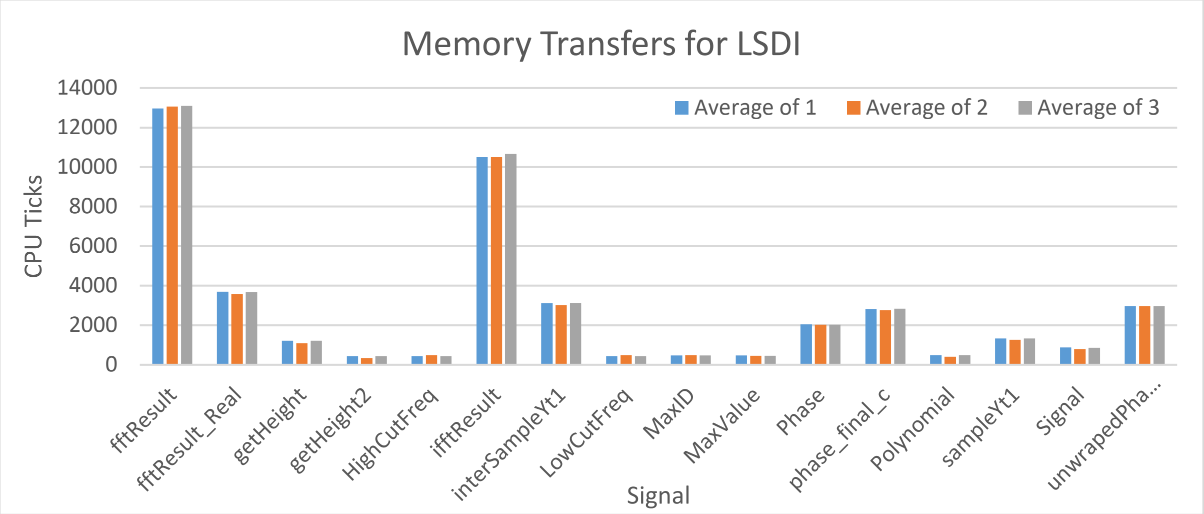


Figure 6-10: *Comparison of GTX 1070 Memory Transfers Over Three Consecutive Runs*

6.5 Performance Comparison of Throughput vs Latency

6.5.1 Dispersed Reference Interferometry (DRI)

As previously discussed, the DRI signal processing is short, and therefore the data size is small, when processing a single frame, the CPU is significantly faster. However, when batching 16 frames together, the GPU starts to have a speed benefit (Figure 6-11). For some applications, batching frames together might not be beneficial, if real-time feedback is needed for example, however in scenarios where latency is not critical, batching DRI frames can have enormous benefits. At 128 and 256 frames the limiting factor becomes the camera capture rate, around 6.5 kHz, but the GPU benefit over the CPU is around 3x the throughput. As for the CPU execution time per frame, this stays between 2100 and 2200 frames independent of the batch size, as is to be expected due to its sequential processing nature, as the data size increases, the processing time will increase linearly.

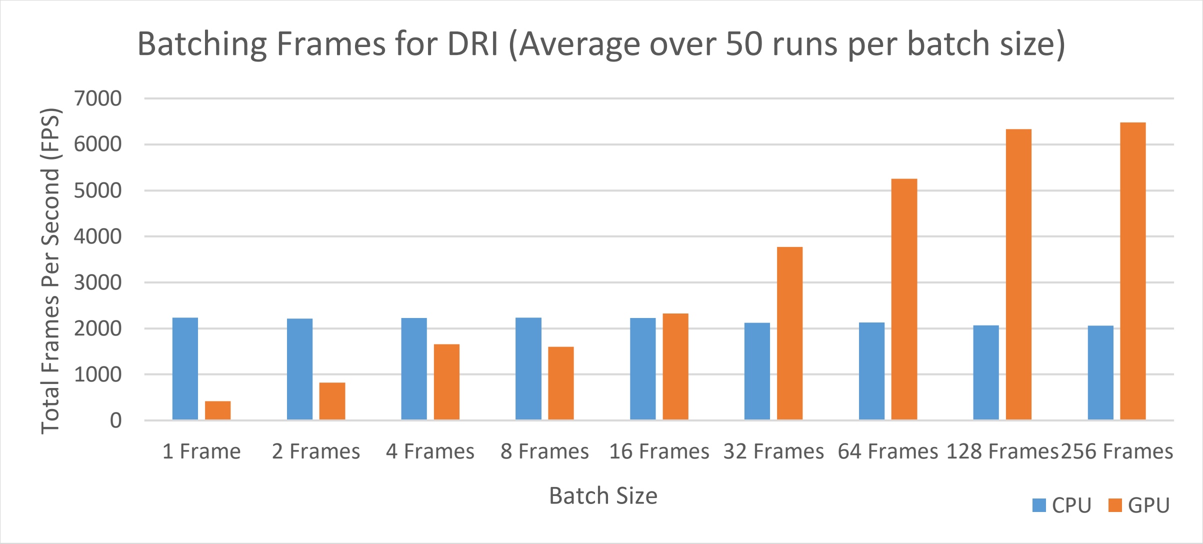


Figure 6-11: DRI Throughput Comparison

6.5.2 Line-Scan Dispersive Interferometry (LSDI)

It was demonstrated in chapter 6.4.2 that using a GPU for processing LSDI data had a significant speed advantage, and this only increases when batching frames together and optimising for throughput rather than latency (Figure 6-12). For this comparison, the CPU limited interpolation algorithm has been ignored to illustrate the GPU accelerated functions only and the benefit they provide the signal processing. Increasing from a single frame up to 16 frames shows a marked increase in performance, around 3.5x that of the single frame GPU and 22x of a CPU.

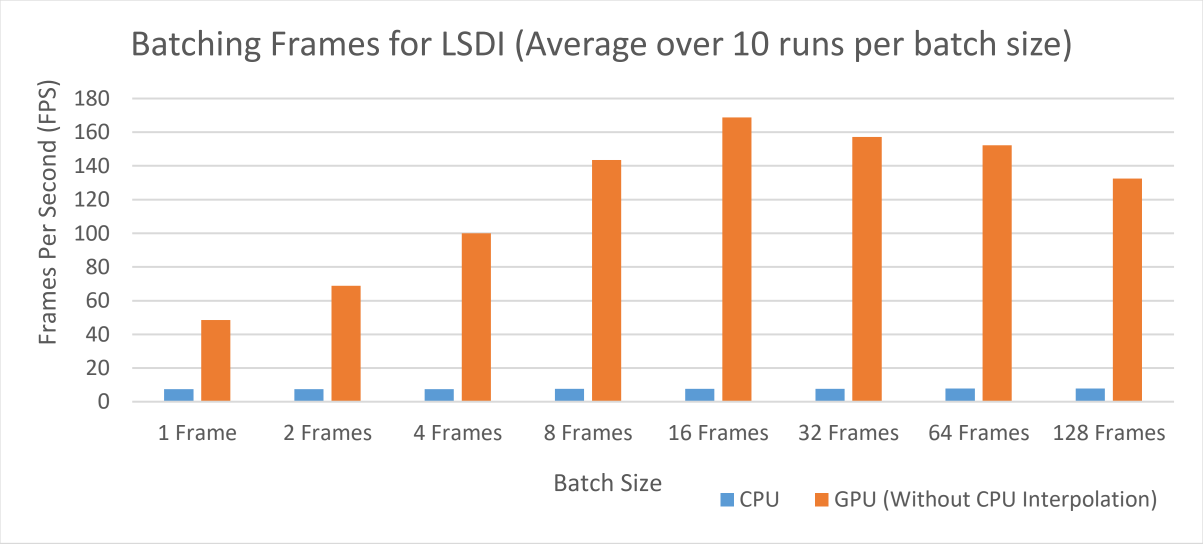


Figure 6-12: LSDI Throughput Comparison

After 16 frames performance unexpectedly decreases, this is likely due to the increase in memory being used when processing the frames. However, upon further inspection this decrease in performance is linked to two specific functions, *polyfit* and *unwrapPhase* (Figure 6-13). These functions are more sequential in nature when compared to their counterparts and this seems to have a major impact on their performance when processing large amounts of data. In future work these algorithms will be reworked and further optimised to reduce any of these performance issues.

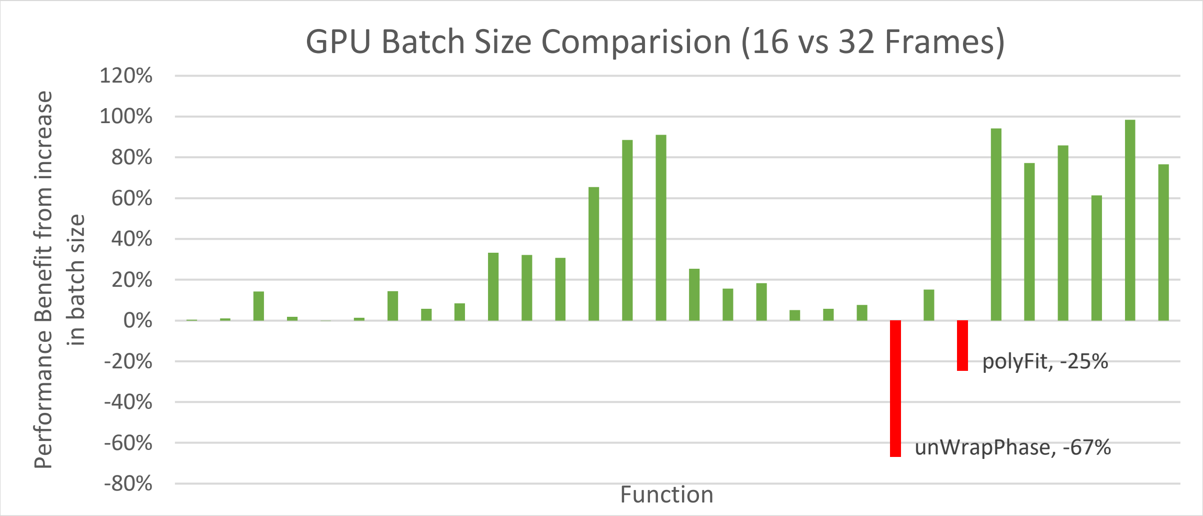


Figure 6-13: *Performance Benefit for GPU frame batching for LSDI*

6.5.3 Summary

The results of these throughput comparisons provide two interesting conclusions. Firstly, if latency is not important, batching frames together for the GPU to process together provides a significant speed benefit. However, both the DRI and LSDI reveal a point of diminishing returns, for the DRI this was around 128 frames, and the LSDI was 16 frames. This plateau, or decrease in the case of the LSDI, is important when deciding on a batch size. The major limiting factor of how large of a batch can be used is likely to be the GPU memory, this is less of a problem with newer graphics cards which are sold with higher and higher memory capacities, however there is still an upper ceiling. In the case of the DRI, the point of diminishing returns was the camera speed, not the GPU memory, around 37 MB/batch for 256 frames), and for the LSDI was the slow algorithms, not the GPU memory, around 850 MB/batch for 32 frames). This point is therefore hard to predict and like other parameters, benefits from profiling the signal processing at different batch sizes (if latency is not critical) to find the optimum configuration. This should be considered when searching for a pathfinding algorithm to compute the different configuration possibilities.

6.6 Evaluation of OSPW Performance

The signal processing algorithms used by the DRI and LSDI were compared by themselves outside of OSPW to simulate the environment they would be used in if they were created by an user from scratch. Therefore, these do not contain dynamic variables, polymorphism, or variadic templates, instead the functions are static as they do not need to change. The standalone software also does not have a job queue and sequentially iterates through a loop; therefore, it does not require launch functions.

6.6.1 Dispersed Reference Interferometry Performance Evaluation

Over 100 runs of the DRI on the CPU, the standalone software averaged a frame rate of 2257 FPS, whereas OSPW returned a frame rate of 2206 FPS giving a deficit of 2.31% by using OSPW while profiling. Although when profiling is complete, and therefore disabled, the deficit is reduced to 1.88%. These results were also compared to a MATLAB script, with the data pre-loaded rather than using than a capture device but keeping the hardware and other processing the same. Under these conditions MATLAB returned a frame rate of 104 FPS, therefore, OSPW provides a 21x increase in performance compared to MATLAB (Figure 6-14).

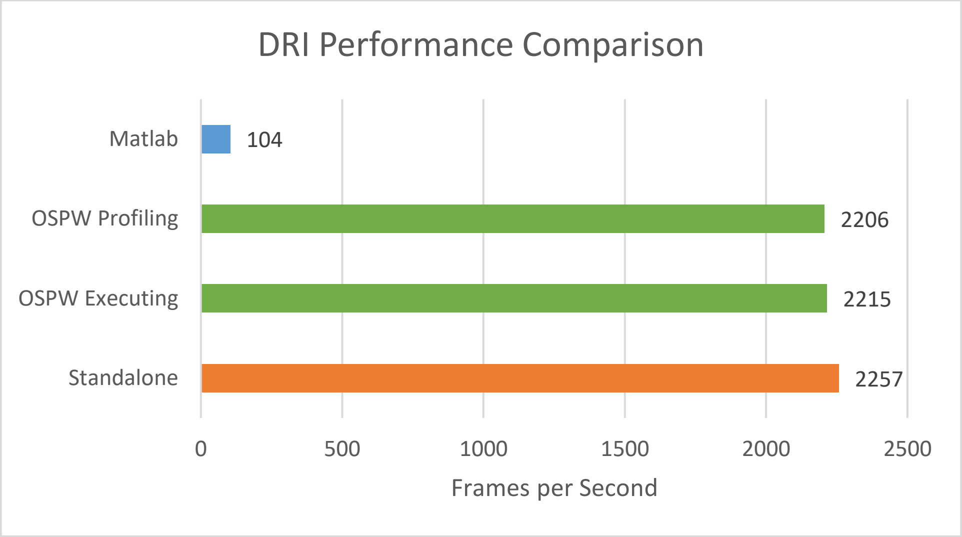


Figure 6-14: *Comparison between OSPW and standalone software for DRI*

6.6.2 Line-Scan Dispersive Interferometry Performance Evaluation

For the CPU data on the LSDI, the standalone software outputted at an average of 7.06 FPS whereas OSPW outputted at an increased 7.59 FPS, outperforming the standalone software. When comparing the pre-configured execution of OSPW against the standalone software, both increase to 7.62 FPS for OSPW but 7.8 FPS for the standalone software. This increase for the standalone software means it outperforms OSPW by 2.18%. MATLAB was once again compared to the above figures, and for a CPU execution, MATLAB ran at an average of 1.43 FPS (Figure 6-15).

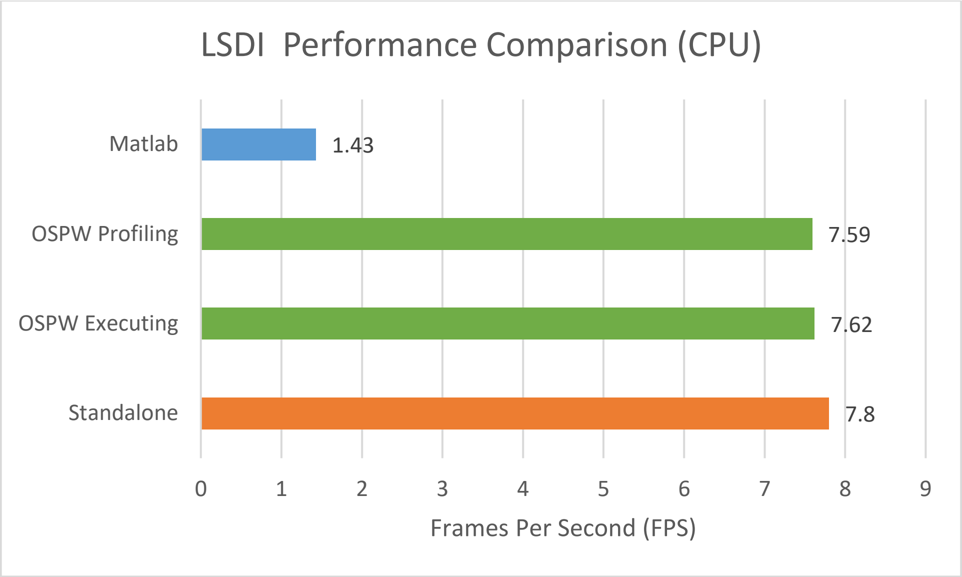


Figure 6-15: Comparison between OSPW and standalone software for LSDI (CPU)

Using a graphics card has a performance benefit for the LSDI, therefore the same tests have been repeated for GPU profiling and execution. When OSPW profiles GPU functions, it also profiles each potential memory transfer to and from the GPU, so this has a significant impact on the profiling results, although the same profiling restraints were placed on the standalone software. When profiling OSPW, it is considerably slower at 5.93 FPS compared to the standalone software at 8.92 FPS, this is an unexpected difference and may have been caused by other unknown factors, such as CPU or memory occupancy, rather than being a true representation of the difference. When executing an optimised configuration (and therefore removing all the unnecessary memory transfers), OSPW outperforms the standalone software at 15.71 FPS compared to 15.45 FPS (Figure 6-16).

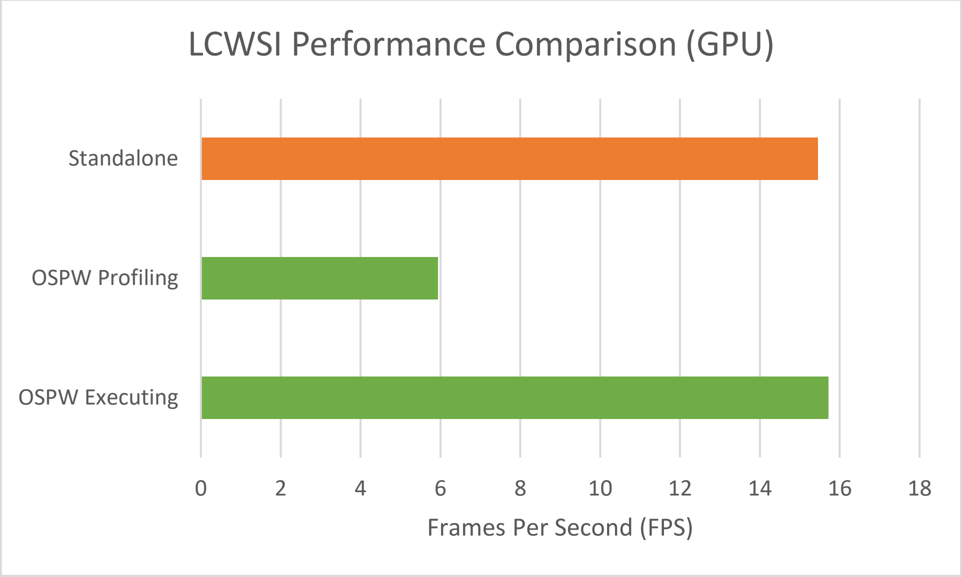


Figure 6-16: Comparison between OSPW and standalone software for LSDI (GPU)

6.7 Conclusion

At around 2% difference in any benchmark between OSPW and the standalone software (Figure 6-17), some of the deficit can probably be attributed to operating system requirements at the time of running these benchmarks. It was initially anticipated that OSPW would have a minor reduction in performance over standalone software due to the extra processing needed to retrieve functions from a job queue and the polymorphism of the functions themselves. Nonetheless, this deficit is outweighed by OSPW's ease of use and configurability. It also performs faster than MATLAB with a 5.3x speed up on the CPU and 11x increase using the GPU for LSDI and a 21x increase for the DRI.

However, if the two different architectures can be used in harmony and retain the fastest functions from each architecture throughout the signal processing, this could reduce any deficit and potentially provide a significant performance benefit over standalone software executing on any single architecture. The next chapter will apply PDDL using the profiling data as the action cost, for the planner to be able to optimise for.

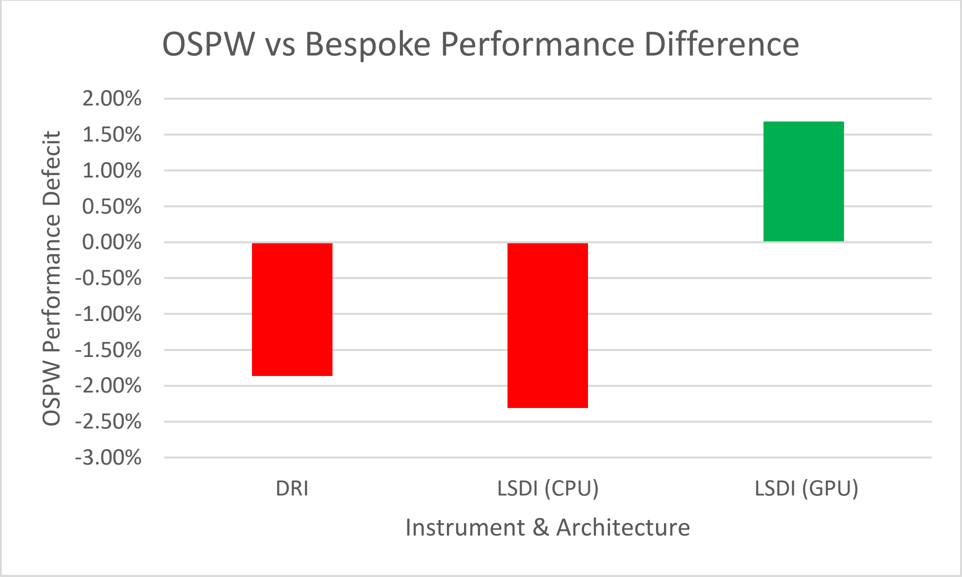


Figure 6-17: *FPS Percentage Benefit of using Bespoke Software*

7 Optimising OSPW Using Artificial Intelligence (AI) Planning

7.1 Introduction

Chapter 6 analysed the profiling results and concluded that making the decision as to which was the fastest architecture was complex. The more functions required for the measurements, the exponentially larger the problem became. This chapter describes the application of PDDL, first discussed in chapter 3.3.2, using LPG-td, to the profiling results to examine what optimisations are possible using multiple architectures together. These results were captured across several different graphics cards to demonstrate a range of hardware that may already be available to users or present the different levels of acceleration different price points can provide.

7.2 Mapping Signal Processing Functions to PDDL Actions

Once OSPW captures the profiling data it automatically creates three PDDL domain files; CPU (standalone), GPU (standalone) and CPU and GPU Heterogeneous and a single problem file. OSPW then evaluates the three solutions that are returned by LPG-td, CPU Only, GPU Only and Heterogeneous, and selects the solution with the smallest total-cost to use for the final signal processing configuration. The solution lengths given are the number of CPU ticks measured by the Windows performance counter (Microsoft, 2018), these were recorded with a counter frequency of 3.51 MHz for the DRI and 3.52 MHz for the LSDI. This gives a tick resolution of 285 ns.

Converting the signal processing requirements into PDDL is about mapping the signal processing parts to be readable by a PDDL Planner (Figure 7-1). Firstly, the variables (signals and datapoints) is the objects, the signal processing functions are the actions, the only other details needed are states. Each signal processing action needs a state ‘flag’ to say that action is completed, it also checks that any precondition actions have also been completed. Other states that is needed are to check whether the data is on the CPU or the GPU, and mutex states to make sure two resources are not attempting to be used at the same time. A mutex is needed to check if the CPU or GPU is in use, as two actions cannot be run concurrently on the same architecture.

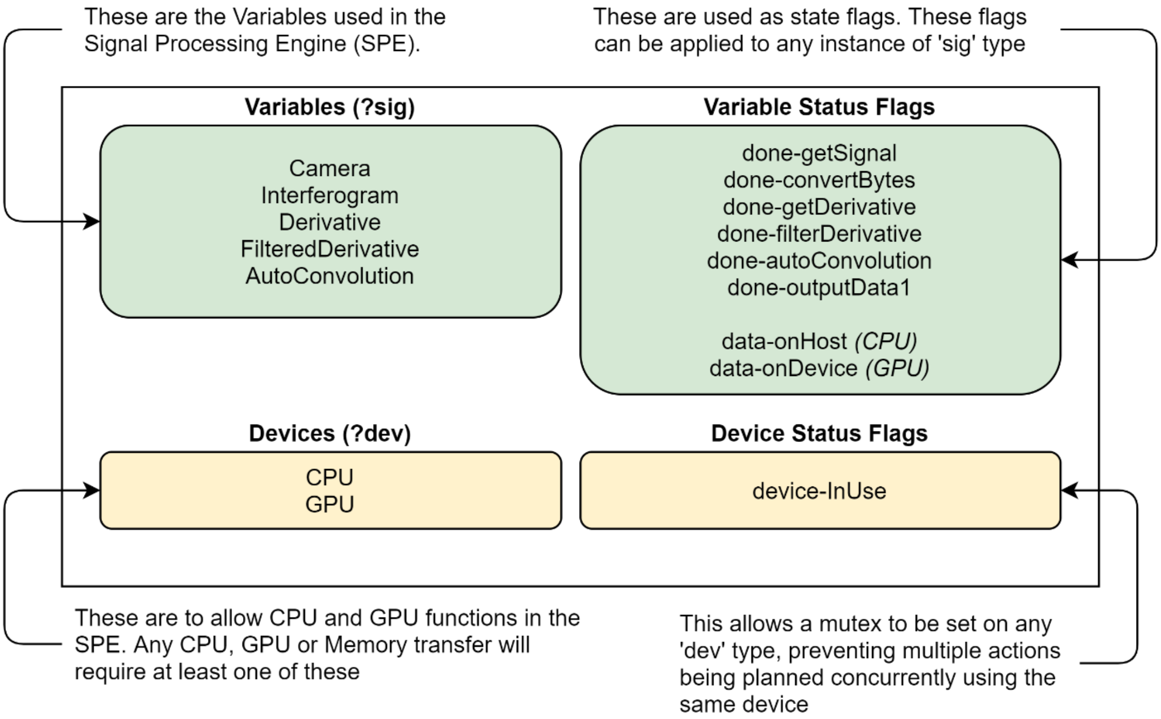

These are the Variables used in the Signal Processing Engine (SPE).
These are used as state flags. These flags can be applied to any instance of 'sig' type
Variables (?sig)
Camera
Interferogram
Derivative
FilteredDerivative
AutoConvolution
Variable Status Flags
done-getSignal
done-convertBytes
done-getDerivative
done-filterDerivative
done-autoConvolution
done-outputData1
data-onHost (CPU)
data-onDevice (GPU)
Devices (?dev)
CPU
GPU
Device Status Flags
device-InUse
These are to allow CPU and GPU functions in the SPE. Any CPU, GPU or Memory transfer will require at least one of these
This allows a mutex to be set on any 'dev' type, preventing multiple actions being planned concurrently using the same device

Figure 7-1: *DRI PDDL Predicates showing the variables, devices and state flags used throughout the domain file*

An action is needed for each signal processing function on the CPU and GPU and also for each memory copy needed between the two architectures for each signal and datapoint object used. Figure 7-2 is a visual representation of DRIs PDDL domain showing a CPU action, it takes three parameters:

1. cpu – this is so it can check the CPU is not currently active.
2. sig – This is an instance of the signal: *Camera.*
3. sig – This is an instance of the signal: *Interferogram.*

In the preconditions, its checks that all of the instances relate to the correct object, if they do not yet exist, they are created. The next precondition group checked is the CPU and the function status. It first checks that the CPU is not busy, then it checks that this action has not already been run for the output signal. The action done status is applied to the output signal, this allows the successor function to check any prerequisites have been completed on its own input parameters. This allows the planner to keep track of which actions have been completed without having to set this state for every object. Finally, it checks where the objects required are in memory. It checks that both the input parameters (Signal and Background) are on the CPU, as this is a CPU function. If all of these preconditions are met the action can be executed.

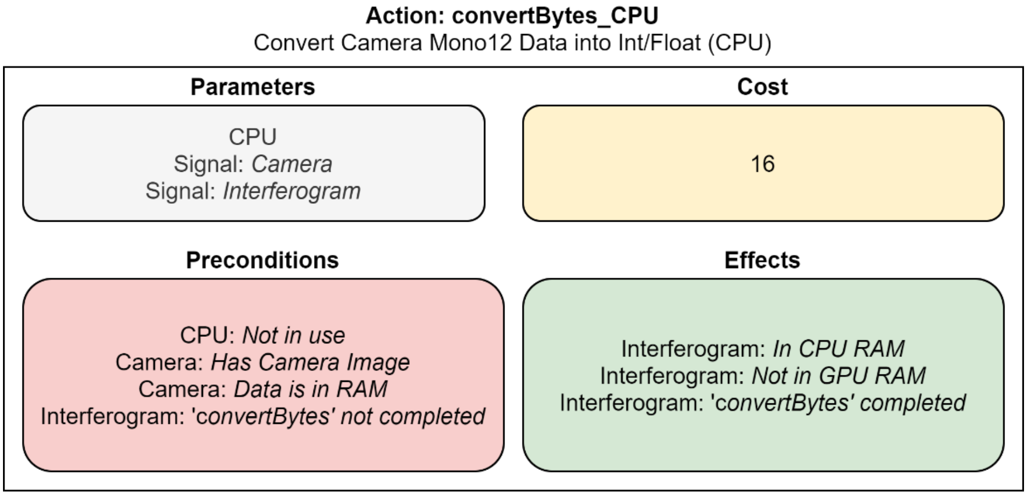

Action: convertBytes_CPU
Convert Camera Mono12 Data into Int/Float (CPU)
Parameters
CPU
Signal: *Camera*
Signal: *Interferogram*
Cost
16
Preconditions
CPU: *Not in use*
Camera: *Has Camera Image*
Camera: *Data is in RAM*
Interferogram: '*convertBytes' not completed*
Effects
Interferogram: *In CPU RAM*
Interferogram: *Not in GPU RAM*
Interferogram: '*convertBytes' completed*

Figure 7-2: *Visual Representation of PDDL action for a DRI function*

The effects this action has are as follows:

1. It sets that the output signal *sampleYt1*'s data is on the host (CPU) and not the device (GPU). Previously this object was neither on the host or on the device.
2. It marks the output signal *sampleYt1* as completed by *divide1* action, therefore when the successor to this function checks its input for pre-completed functions, it returns true.
3. It also increases the metric *total-cost*; this is the median value calculated during profiling. In the problem file it is instructed to minimise this *total-cost* metric, finding the shortest path.

A visual representation of the DRI's parameters, pre-conditions and effects are shown in Figure 7-3.

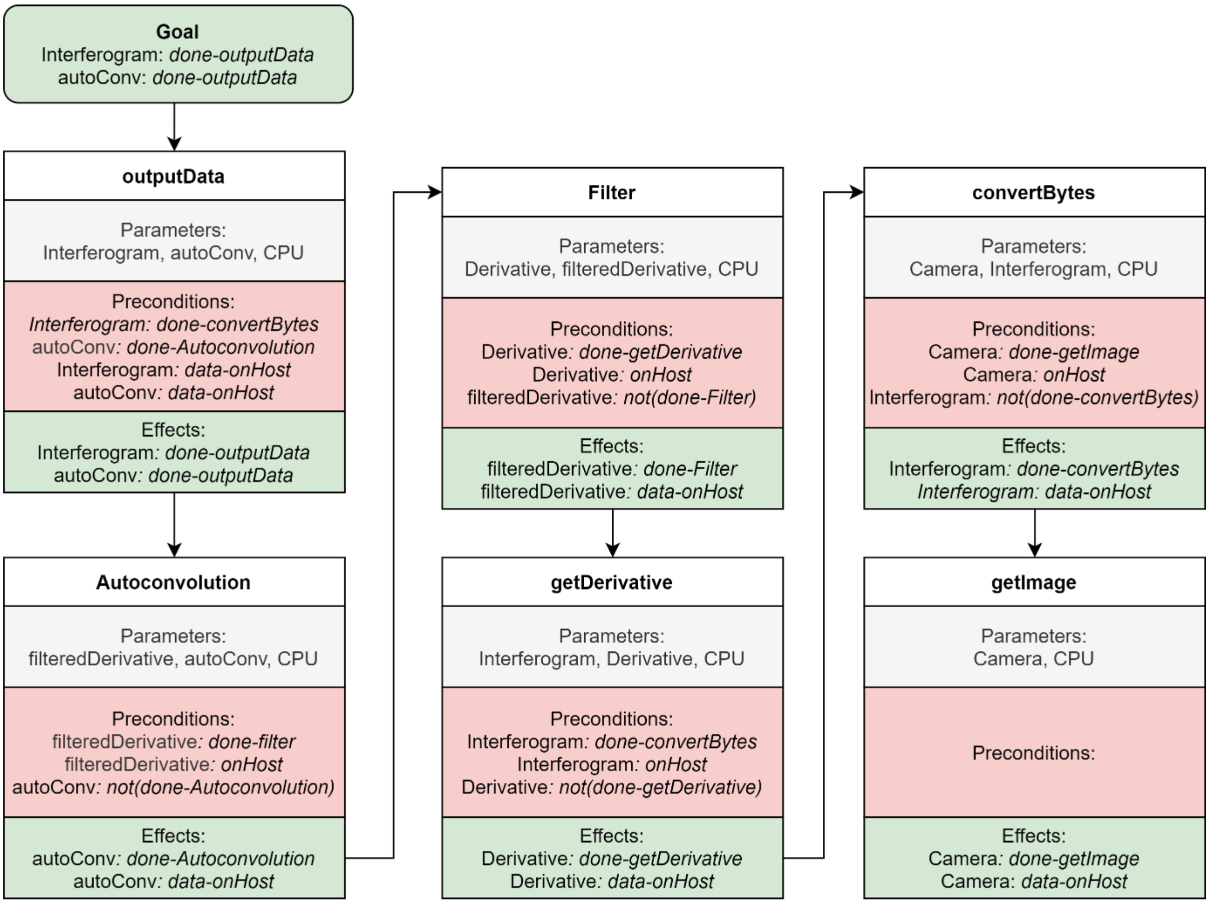

Goal
Interferogram: done-outputData
autoConv: done-outputData
outputData
Parameters:
Interferogram, autoConv, CPU
Preconditions:
Interferogram: done-convertBytes
autoConv: done-Autoconvolution
Interferogram: data-onHost
autoConv: data-onHost
Effects:
Interferogram: done-outputData
autoConv: done-outputData
Autoconvolution
Parameters:
filteredDerivative, autoConv, CPU
Preconditions:
filteredDerivative: done-filter
filteredDerivative: onHost
autoConv: not(done-Autoconvolution)
Effects:
autoConv: done-Autoconvolution
autoConv: data-onHost
Filter
Parameters:
Derivative, filteredDerivative, CPU
Preconditions:
Derivative: done-getDerivative
Derivative: onHost
filteredDerivative: not(done-Filter)
Effects:
filteredDerivative: done-Filter
filteredDerivative: data-onHost
getDerivative
Parameters:
Interferogram, Derivative, CPU
Preconditions:
Interferogram: done-convertBytes
Interferogram: onHost
Derivative: not(done-getDerivative)
Effects:
Derivative: done-getDerivative
Derivative: data-onHost
convertBytes
Parameters:
Camera, Interferogram, CPU
Preconditions:
Camera: done-getImage
Camera: onHost
Interferogram: not(done-convertBytes)
Effects:
Interferogram: done-convertBytes
Interferogram: data-onHost
getImage
Parameters:
Camera, CPU
Preconditions:
Effects:
Camera: done-getImage
Camera: data-onHost

Figure 7-3: *Backwards search through DRI functions to satisfy the goal state*

For GPU functions, the action syntax changes only slightly. Rather than checking if the CPU is free, it checks the GPU is not in use, and the preconditions regarding object locations must be on the device rather than the host, (although these two properties are not mutually exclusive). Finally, when the GPU action has completed, the state of the object is set *onDevice* rather than *onHost*.

The other type of action that exists in the domain file are the memory transfers. Memory transfer actions check that both the CPU and GPU are not in use, as both are needed to complete the memory transfer. It also checks that the object is on the sender's memory and is not on the receiver's memory, if the object is already on the receiver's memory, the action does not need to be executed. Finally, if all the preconditions are met, the object is marked as being on the receiver's memory, but it is not removed from the sender, as currently the data is simply duplicated for now, it is only removed from the sender's memory if the data has been modified by a future action.

In the LSDI's problem file, are all the objects (variables) that are going be used, these then each have an instance created in the domain, this allows PDDL to create multiple instances of the same object, something that is not needed for OSPWs requirements. It also sets the initial state for some objects (variables), for example, a background image is loaded into CPU memory once at the very start, as this is a single call function it is not profiled. Therefore, the default state that PDDL observes for this object is that the data already exist on the CPU, otherwise PDDL cannot return a valid plan. This is because the first function expects this object to either already exist or be created by p action that it needs to call as a precondition. Finally, the goal is set, the goal for the LSDI is to return two different height signals, but the data to be on the CPU so it can be sent to the UI, therefore the sub-goals are:

$$done\_getHeight1 \wedge done\_getHeight2 \wedge onHost(getHeight1) \wedge onHost(getHeight2)$$

This ensures that not only is the correct data calculated using the correct functions, but that data exists in the CPU memory ready to be outputted to the user interface, completing the signal processing chain.

7.3 Evaluation of PDDL Solutions for Dispersed Reference Interferometry

7.3.1 Solution Results

As shown in Chapter 6.4.1 there does not seem any advantage to running the DRI on a GPU due to the small amount of data being captured and processed. Nevertheless, all the profiling data for the Intel i7-4790 CPU and Nvidia GPUs: GTX 520, GTX 650Ti, GTX 780Ti and GTX 1070 was imported into a total of nine separate domain files and processed with LPG-td (Figure 7-4). A selection of these domain files and solutions can be found in appendix 11.3.

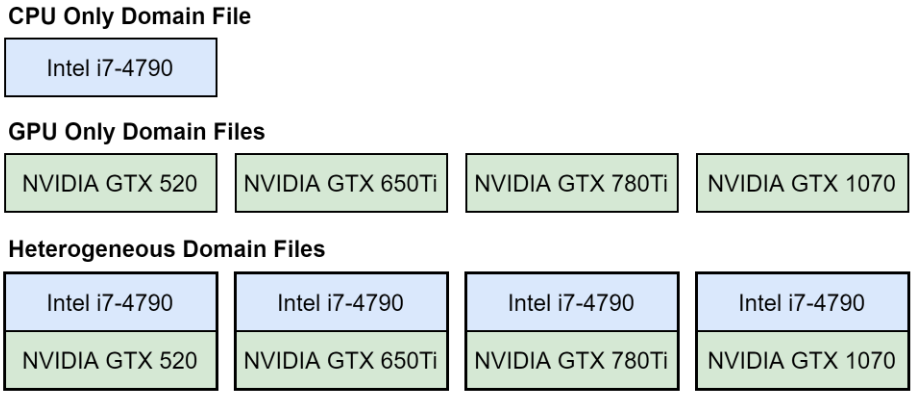


Figure 7-4: *Nine Different Domain Files created for Dispersed Reference Interferometry*

Figure 7-5 shows using the CPU as the only architecture is the best solution, when running solely on the GPU, the solutions are typically slower than the CPU. Finally, when the domain was given both the CPU and GPU functions to choose from it successfully selected all the CPU functions.

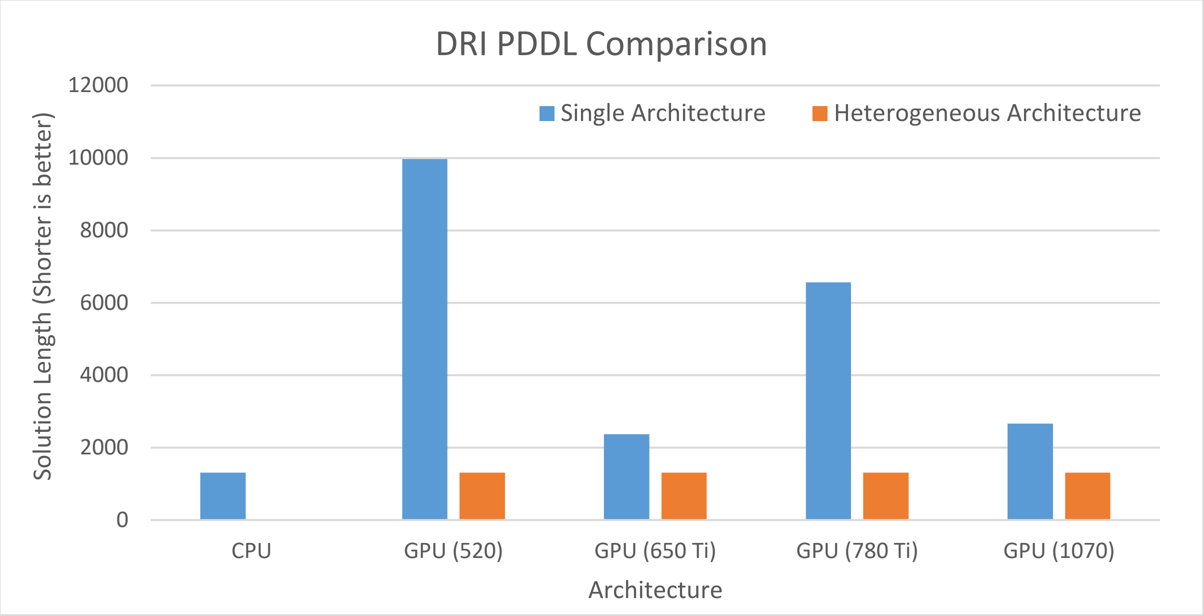


Figure 7-5: *Comparison of PDDL Solutions for the DRI Signal Processing*

One anomaly here is that the GTX 780Ti results are much higher than expected. When investigated further in Chapter 6.4.1, the execution of the Auto-convolution on the GTX 780Ti is the cause, executing at around one fifth of the speed that would be expected (Figure 6-8). It is much more likely to be executing at around the same speed as recorded on the Nvidia GTX 1070 due to their similar specifications. However, these profiling results were gathered multiple times and their values averaged to reduce such issues. Further investigation and repeated tests are needed to solve this, but as this still would not provide an overall benefit it was not explored.

7.3.2 Summary

Although restricted by the data, LPG-td has solved the problem for the DRI, with the best solution to run everything on the CPU. Next the LSDI solutions is investigated to explore what advantage may be yielded when using multiple architectures.

7.4 Evaluation of PDDL Solutions for Line-Scan Dispersive Interferometry

Unlike the DRI, it was clear from the results chapter 6.4.2 that GPU acceleration was beneficial for the LSDI, but it depended on which combination of CPU and GPU as to how much benefit could be gained. For the LSDI, three graphics cards were compared: GTX 650Ti, GTX 780Ti and GTX 1070 with an Intel i7-7820X CPU (Figure 7-6). A selection of these domain files and solutions can be found in appendix 11.4.

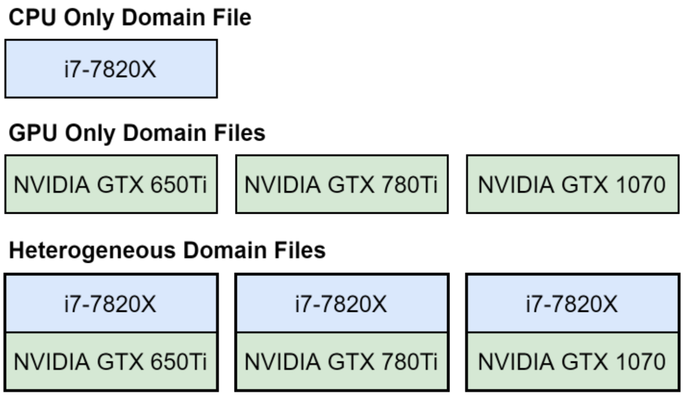


Figure 7-6: *Seven Different Domain Files created for Line-Scan Dispersive Interferometry*

Each set of profiling data for each GPU was imported into a PDDL domain and run against the problem file for LSDI each CPU/GPU combo is analysed separately and then against one another. As there are eighteen functions used for the LSDI, there are 262,144 different permutations that would need to be considered for each architecture combination.

7.4.1 Single Architecture Results

When running PDDL for a single architecture there is only one valid solution, as there are no other permutations to choose from. Therefore, these are very quick for LPG-td to compute, the planner selects all the signal processing functions required for the selected sole architecture. Figure 7-7 shows that the use of any of the three GPUs provides a speed benefit, and the more powerful the GPU (left to right), the more benefit it provides.

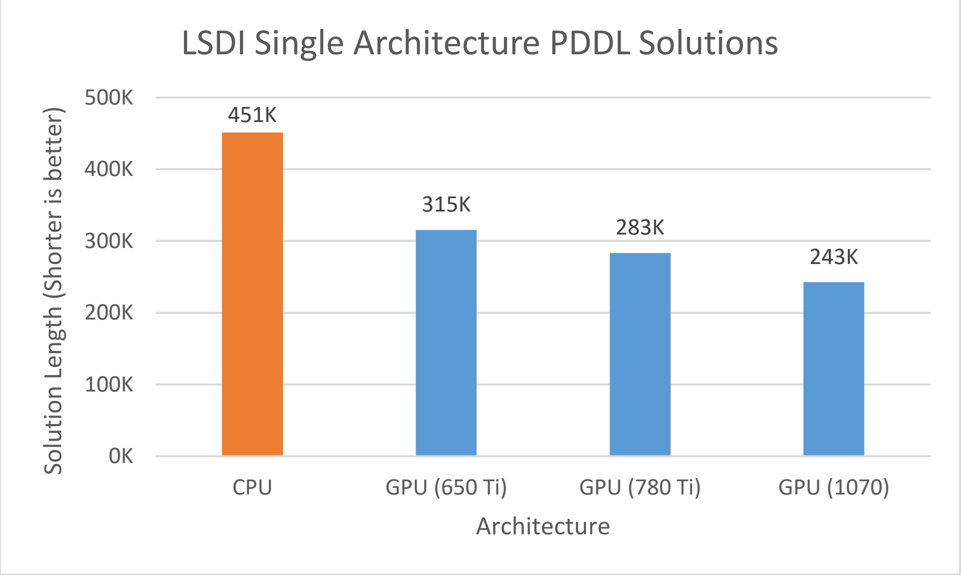


Figure 7-7: *LSDI Single Architecture PDDL Solutions*

7.4.2 Analysis of Heterogenous Execution of LSDI

From the single architecture results, it is clear that the GPU yields a benefit but combining the architectures should provide an even larger performance benefit.

7.4.2.1 CPU and Nvidia GTX 650Ti

As there are many possible paths, LPG-td first finds a valid path before exploring other paths that may return a better solution. As can be seen in Figure 7-8 the first solution was worse than the GTX 650Ti by itself (425K ticks vs 315K ticks), however, the second iteration of the heterogeneous solution is faster and once the final, optimum solution is found, it is 13% faster than the standalone GPU.

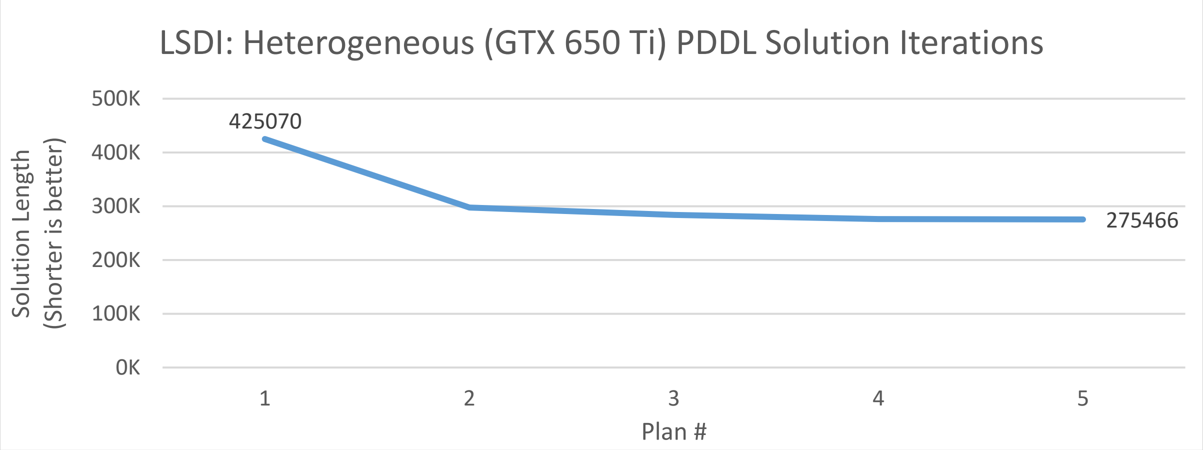


***Figure 7-8**: Iterative LSDI Heterogeneous PDDL Solutions (GTX 650Ti)*

Figure 7-9 shows a visual representation of the way LPD-td found solutions that utilises multiple architecture. By using the GPU for functions three and four (FFT and absolute data) it saves time over the CPU, it also repeats this later for the IFFT amongst other functions. But it also utilises the CPU when it is the optimum architecture. For example, function fourteen (unwrapping phase) is much quicker on the CPU, so it therefore it copies the data back to the CPU memory and continues serially. PDDL and LPG-td satisfied the requirement of finding the optimum solution for this combination of CPU and GPU based upon the profile data provided. A solution that is faster than either architecture individually.

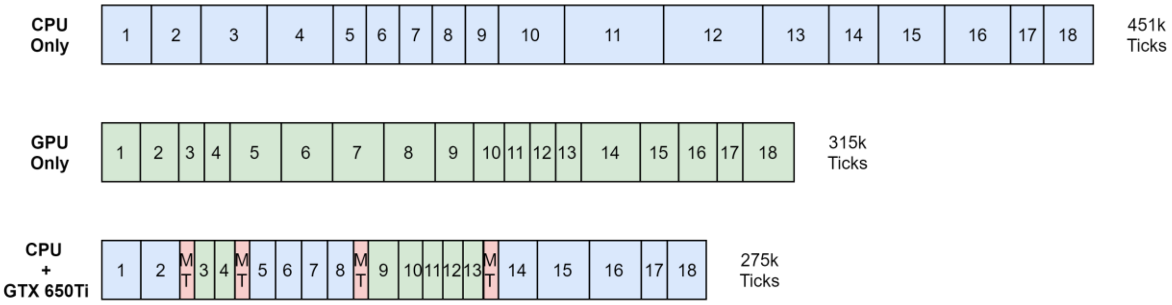


***Figure 7-9:** Visual Representation of LSDI Homogenised Function Execution on GTX 650Ti (not to scale)*

7.4.2.2 CPU and Nvidia GTX 780Ti

The GTX 780Ti is a more modern and a higher specification of graphics card than the GTX 650Ti and this performance advantage is shown in the standalone solutions (283K ticks compared to the GTX 650Ti's 315K). Therefore, there should be a further increase in performance compared to the CPU-only and GTX 780Ti-only solutions. Creating a new domain file with the GTX 780Ti's profiling data creates five solutions, once again starting off worse than either individual architecture, but gradually getting better each iteration (Figure 7-10). By the final and best solution, the heterogeneous solution is almost 6% faster than the GPU by itself, and over 40% faster than the CPU alone.

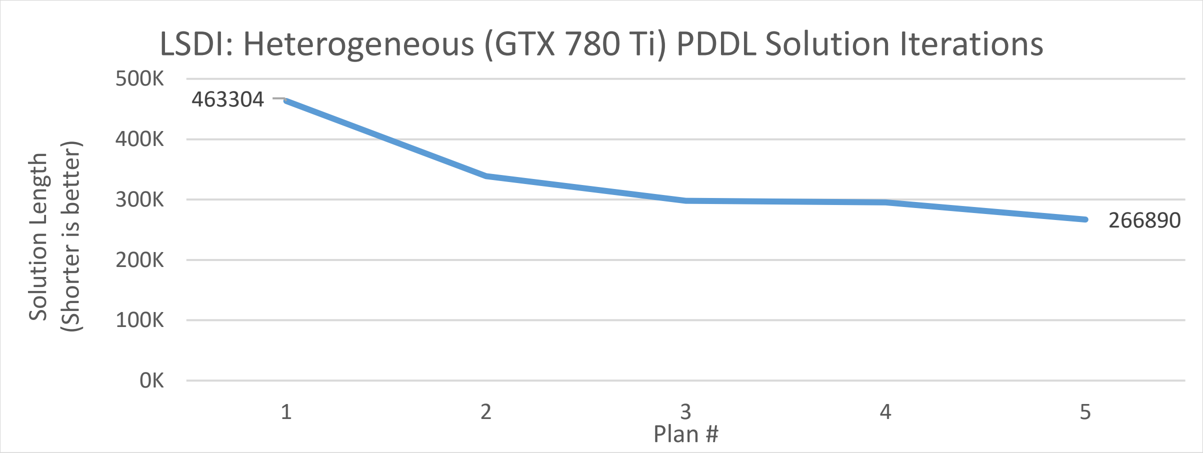


Figure 7-10: *Iterative LSDI Heterogeneous PDDL Solutions (GTX 780Ti)*

When looking at the function breakdown in Figure 7-11, the function split between architectures is exactly the same as the GTX 650Ti, only utilising the few GPU functions that are faster. Until there is such an increase in GPU performance that the GPU is faster (including any memory transfers) for more functions, the solutions look the same. But again, LPG-td was successful in finding the most optimum solution for this particular signal processing and available hardware.

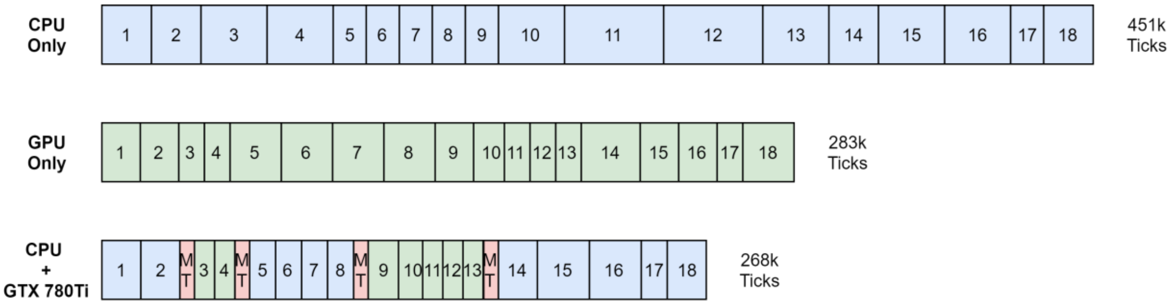


Figure 7-11: *Visual Representation of LSDI Heterogeneous Function Execution on GTX 780Ti (not to scale)*

7.4.2.3 CPU and Nvidia GTX 1070

The GTX 1070 is the most powerful graphics card of the three profiled with about 15% better floating-point performance than the GTX 780Ti (Table 6-1). This should therefore present around a 15%

improvement in PDDL solution from the GTX 780Ti, especially as the GTX 1070 outperformed the GTX 780Ti by 14% when used by itself (Figure 7-7). Unlike previous plans, the profiling data for the CPU and GPU functions was much closer and therefore LPG-td created more plans before settling on the final one (Figure 7-12).

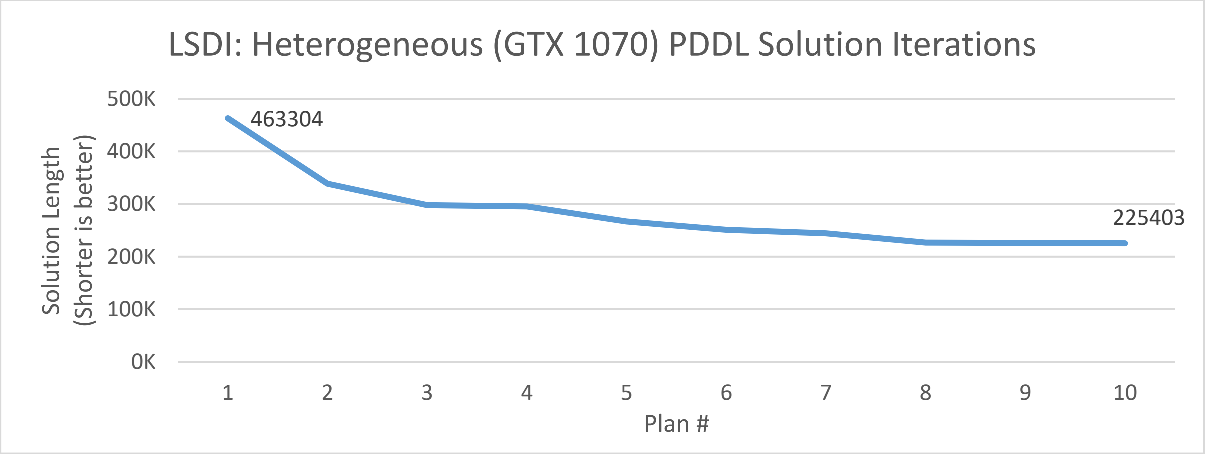


Figure 7-12: *Iterative LSDI Heterogeneous PDDL Solutions (GTX 1070)*

As with each of the previous iterative solutions, the GTX 1070 starts off with a plan worse than either single architecture solution and then gradually improves until reaching an optimum result of 225K ticks. This solution is 8.5% better than the GPU only solution and 50% faster than the CPU only solution. As can be seen in Figure 7-13, the majority of the functions selected are using the GPU, hence why the difference between the GPU only solution is less than 10%. To further investigate how the plan went from 463K to 225K, each of the ten solutions have been analysed to illustrate how the solutions progressed.

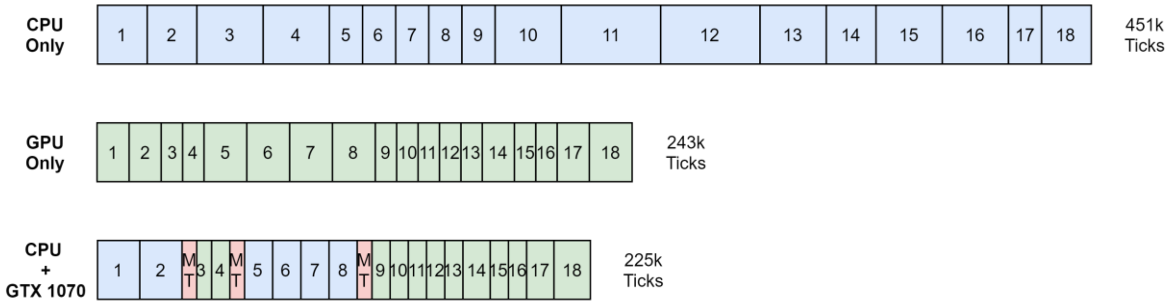
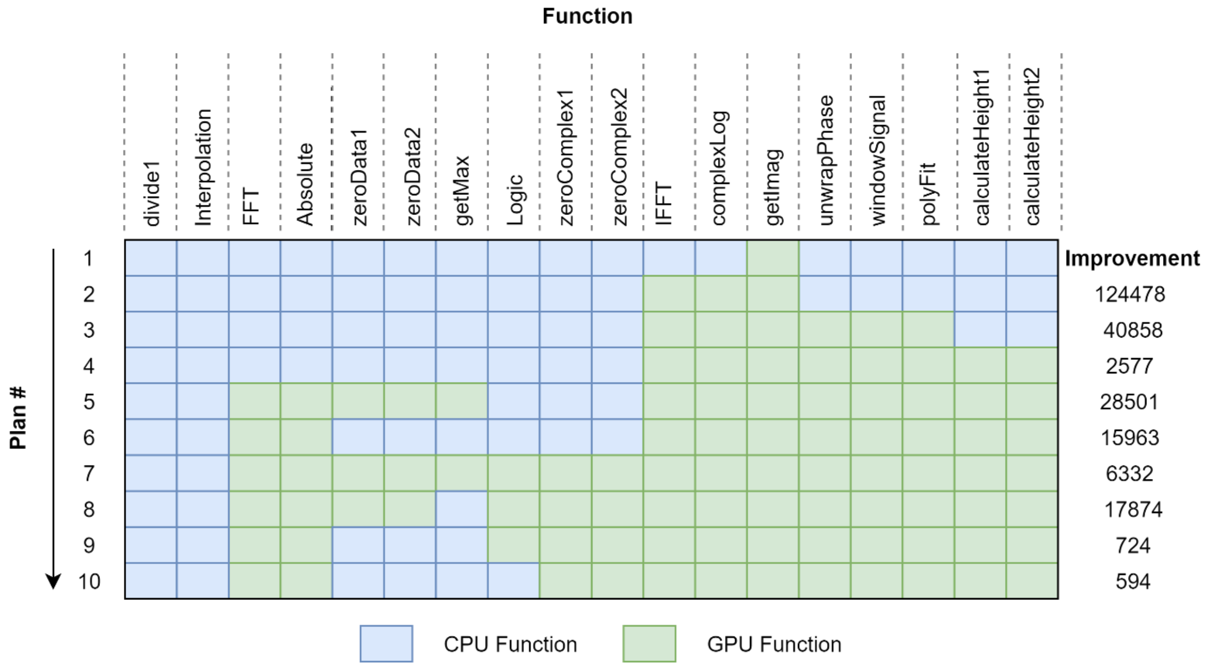


Figure 7-13: *Visual Representation of LSDI Heterogeneous Function Execution on GTX 1070 (not to scale)*

Figure 7-14 shows the selected architecture for each function across all of the ten incremental solutions with each providing more optimisation over the previous. Although each plan is better than the previous, LPG-td sometimes takes multiple iterations before making the best decisions. Just like the search algorithms discussed in the previous chapter, it expands different paths with different priorities, eventually it corrects any mistakes and returns the best solution. For example, solution five found that using the GPU for *zeroData1*, *zeroData2* and *getMax* was best, but this was reversed in solution six. By solution six, the solution was almost there, just two function differ to the final solution (*zeroComplex1* and *zeroComplex2*), but the planner goes through three more solutions getting incrementally better before finding this optimum solution.



Plan #	divide1	Interpolation	FFT	Absolute	zeroData1	zeroData2	getMax	Logic	zeroComplex1	zeroComplex2	IFFT	complexLog	getImag	unwrapPhase	windowSignal	polyFit	calculateHeight1	calculateHeight2	Improvement
1	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟦	🟦	🟦	🟦	🟦	
2	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	124478
3	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	40858
4	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	2577
5	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	28501
6	🟦	🟦	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	15963
7	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	6332
8	🟦	🟦	🟩	🟩	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	17874
9	🟦	🟦	🟩	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	724
10	🟦	🟦	🟩	🟩	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	594

Figure 7-14: *Visual Representation of the different solution paths LPG-td made*

7.4.3 Summary

In summary, LPG-td consistently found the best possible solution for the LSDI signal processing requirements (Figure 7-15). The more powerful the graphics card, the better performance, and therefore allowing some of the shorter functions to also be faster on the GPU than the CPU. Although there is 2^{18} possible permutations for the LSDI, the best solution for each hardware combination was found in under five minutes, with the single architectures only taking seconds. This is a one-time cost and only needs to be recomputed if the signal processing or available hardware changes; a small price to pay for an almost 10% benefit per frame.

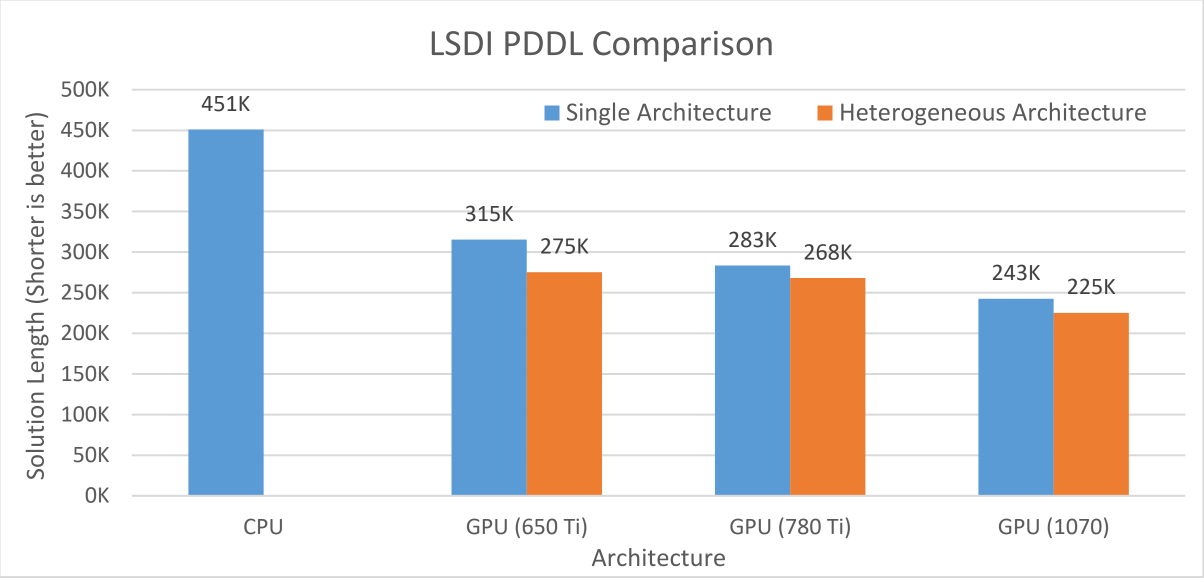


Figure 7-15: *Comparison of PDDL Solutions for the LSDI Signal Processing*

Evaluating for a single GPU, the Nvidia GTX 1070, it is possible to plot the normal distribution of all possible solutions, good or bad. When plotted, all the possible solutions form a standard gaussian distribution, however due to the nature of measuring time, the distribution is slightly right skewed (Figure 7-16). It should be noted that this distribution displays all successful plans for the given set of signal processing requirements with only the necessary memory transfers are included, giving 2^n permutations, where n is the number of signal processing functions. The total number of possible permutations if searching every possible memory transfer that may or may not be required would be $(m^2 + 1)^{n-1}(m^2 + n)$, where m is the number of potential memory transfers, this would return both significantly more and worse permutations. For the LSDI example, there are 14 possible memory transfers that could be required and 18 functions, with an upper limit of 2.17E+41 total paths.

Figure 7-16 shows that when a GPU is available, the CPU alone plan is worse than 80% of the multiple architecture plans, as expected, any use of the GPU for the LSDI should yield better performance. However out of the 2^{18} (262144) possible solutions, only 0.2% of them are better than the GPU only solution. PDDL successfully found the fastest of this small subset of plans, something that would take much longer if randomly selecting a path and evaluating its performance, whist also minimising the memory overhead needed to track which memory changes are required.

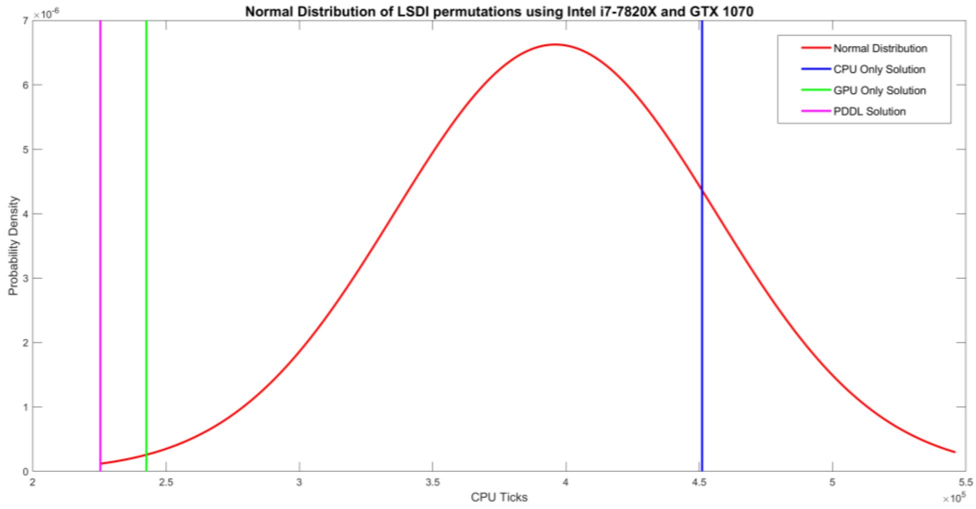


Figure 7-16: Normal Distribution of LSDI permutations using an Intel i7-7820X and Nvidia GTX 1070

7.5 Conclusion

LPD-td has successfully taken domain files generated by OSPW containing the profiling data for both the DRI and LSDI and found the most optimum solution for the given data. PDDL allows the user to find and use the most optimum signal processing without having to manually profile each function to discover if GPU acceleration is beneficial. As shown, profiling a single GPU function sometimes does not demonstrate a benefit, but multiple functions chained together can have a benefit, a task that would take a significant amount of time to manually evaluate all permutations. Within the five minutes it took LPG-td planner to process all the profile data, allows users to achieve performance level near those of bespoke software without having to manually evaluate which architecture best suits which functions, a time consuming process. Furthermore, within that time it enables users to access hardware acceleration, which for the LSDI, provided a 13% performance increase over the fastest single architecture.

8 Discussion

8.1 Summary of Investigations

Initially, relevant literature was reviewed to discover the impact Industry 4.0 is having on high-value manufacturing, and the technologies driving the increase in hardware acceleration taking place in surface and dimensional instrumentation. It was discovered technologies such as Cyber Physical Systems (CPS), the Industrial Internet of Things (IIoT), and the increase and improvement of optical sensor technology are behind the increase of data sizes and processing requirements faced by both academic researchers and industrial engineers. The different types of hardware acceleration being introduced to address this increase in data sizes were assessed to discover the performance benefits of different architectures.

Also reviewed were the types of signal processing algorithms used within surface and dimensional sensors and instrumentation which found that there is significant commonality in the algorithms. However, in most articles analysed the authors are creating these from scratch, when that could be better reinvested in instrument development. The findings from this assessment also identified a significant body of research existing detailing the use of GPGPU processing providing a high level of performance for a relatively low cost.

Additionally, an evaluation of software availability was undertaken to discover the software options that researchers and industry have to choose between. In addition to this, an investigation into the costs associated with developing a bespoke software package were analysed to understand the process required when considering the hiring of a software developer to create a bespoke application. This investigation revealed some of the hidden costs behind software development such as the time required to create documentation, on-going maintenance costs, and additional costs that come from managing and evaluating a project. These costs would explain the reason so many are using generic packages like MATLAB and LabVIEW with poor performance, as the time and financial investment is so high.

Following these investigations, the possibility of creating a software package was explored that contains a pre-built library of signal processing functions used within surface and dimensional instrumentation signal processing, although may have significant benefit to other fields as well. This solution should provide hardware accelerated processing to meet the demands of Industry 4.0, in the form of GPGPU processing, a method technologically and financially viable for most research institutions and manufacturing plants. However, this raised a new problem, the requirement to be able to select the best architecture needed to be smartly determined for each individual function, giving an overall performance benefit by utilising multiple processor types. Therefore, the field of artificial intelligence search and planning was investigated.

This investigation evaluated different uninformed and informed search strategies, these alone were not enough as constraints such as program order and variable usage could not be specified. However, this investigation informed the transition to examine automated planning, more specifically Planning Domain Description Language (PDDL). PDDL enables constraints to be applied to *actions*, pruning the number of possible branches to search through. Effects can also be added to actions, to update the state of the constraints to allow the search to continue towards a goal state. PDDL 2.1 also allows the use of fluents to be able to specify a cost function the planner should optimise for, therefore multiple solutions may be found to the problem, with each solution returning a lower cost than the previous.

From these findings it was clear PDDL would provide a solution to the optimisation problem. The concept of using a Monte Carlo simulation was also investigated, however as the branching factor was directly relating to the number of variables (m^2) it was determined that PDDL provided a better solution to branch pruning. Finally, the use of AI within both software optimisation and manufacturing was surveyed to determine the state of the art techniques currently being used to optimise processes in academia and industry. This study paid specific attention to those using scheduling and process planning techniques when optimising. The study showed significant AI usage for optimisation of software, typically with training data and neural networks to optimise for new, unseen problems. Whereas in manufacturing, the use of AI is much smaller but is increasing, some specific examples of the use in PDDL and similar techniques were discussed to show the benefits AI can provide.

An initial case study, Dispersed Reference Interferometry (DRI), was used for the author to familiarise themself with the software creation process, informed by the findings in the literature review, and also familiarise themself with the signal processing requirements of an optical instrument. This case study lasted around twelve months developing both CPU and GPU algorithms in C++ and CUDA, a user interface in C#, and communication between the two. This development time is likely to be the same for anyone wanting to create bespoke software, and further illustrates why so many accept the reduction in performance that exist in programs such as MATLAB and LabVIEW.

The results from this case study illustrated the possible performance from accelerating signal processing with GPGPU parallelism. However, the DRI image size was one dimensional and even though the GPU processing did provide 7.8x speedup compared to MATLAB, when batching frames together and processing them in 32s or 64s, the GPU could provide 8x performance increase over the CPU and over 43x the performance of MATLAB. Throughout this case study, many lessons were learned, with potential problems being discussed alongside the possible solutions to be evaluated in future versions.

Once the initial learning and application to the case study was completed, a solution to the heterogenous signal processing software package was considered. This examined three main areas of interest, the ability to configure for a user's signal processing needs, the Signal Processing Engine (SPE) that captures and processes the data, and the visualisation of the processed data. During the formulation of the Optical Signal Processing Workspace (OSPW), a second case study was undertaken, Line Scan Dispersive Interferometry (LSDI), to evaluate the performance of the algorithms created on an experimental setup.

To achieve the objectives laid out in chapter 1, many advance C++ techniques were applied such as variadic templates and lambda functions to achieve the polymorphism required at runtime. Multithreading of CPU cores was also utilised to provide acceleration by running tasks such as data capture, inter-process communication and data transfer in the background, while the main thread continues to process incoming data. GPGPU processing was also key objective, and each sequential function was paired with a GPU accelerated counterpart (except CPU bound functions).

Finally, the User Interface (UI) provided an area for users to configure the signal processing using a simple drag-and-drop interface and visualise the data processed using 1D graphs and 2D images. As the UI and the SPE were created in two separate programs, the data was transferred between the two programs via a first-in-first-out (FIFO) inter-process communication method called named pipes.

Contained as part of the SPE is a class that is responsible for execution the algorithms from a queue list. Immediately before the function is called, and again after, a timestamp is taken to be able to evaluate the algorithms performance on a given architecture. During the investigations into these timestamps, of which have a resolution of approximately 285 ns, there was an amount of fluctuation on the results. It was found that due to Operating System (OS) demands consistently changing throughout the signal processing, this affected the time taken to process the algorithm. Although this also did appear in GPU profiling, it was to a much lesser extent.

After an evaluation into averaging calculations such as 95th percentile, mean, and median, the decision was taken to take the median of the results at regular intervals. When a specified number (five) medians were calculated at the same value (within a small tolerance), it was deemed that a fair evaluation of that function's execution time had been captured. This allowed a like-for-like comparison between the CPU and GPU profiling data, the main criteria was the repeatability of a execution time, rather than pinpoint accuracy of the time itself.

The results of the profiling for CPU and GPU were discussed for both the DRI and LSDI, with the GPU only providing benefit for the LSDI, and not the DRI (over a single frame). When the results were re-evaluated for throughput measurement, by batching frames together, both the DRI and LSDI had a marked improvement in performance compared to CPU only execution.

The proposed solution, Optical Signal Processing Workspace (OSPW), was then evaluated against a version without any polymorphism or profiling overhead to investigate the performance lost due to the inherent need for configurability, finding a 2.5% deficit. However, this 2.5% deficit is overshadowed by the significant performance benefits provided compared to previously used MATLAB processing. However, if the signal processing can be distributed across the two architectures to harness the multiple processing core types available, this deficit will be negated.

Finally, using the profiling data as a cost function, PDDL could be introduced to solve the optimisation problem, calculating which architecture is best for each given function to achieve an overall optimum solution. Firstly, the signal processing functions needed to be mapped to PDDL *actions*, prerequisite functions mapped to an action's *predicates*, updating the states of predicates through *effects*, and importing the variables as *types*. Once this mapping was complete, the execution time can be used as the heuristic (cost function) and the LPG-td planner will find the optimum solution.

Solutions were searched for both the DRI and LSDI across several CPU-GPU combinations, with the planner always returning an optimum result within a five-minute window, often quicker. When optimising for a single frame, the planner simply returned a CPU only solution for the DRI due to its small data size. However, for the LSDI, the best solution on an Intel i7-7820X and Nvidia GTX 1070 was a 13% improvement over a single architecture.

In the following section, the research presented will be evaluated against the original objectives laid out in chapter 1 before discussing some of the caveats of the approach and potential solutions to these.

8.2 Conclusions

This sub-section reassesses the work untaken and the results presented in chapters 4 to 7 and compares it against the objectives laid out in chapter 1. The objectives are repeated in bold above each consideration for convenience.

Investigate the requirements of Industry 4.0 and determine the most important features that researchers and industry require for future manufacturing software.

The results of the investigation into the requirements and current implementations of software within surface and dimensional measurements highlight the both the repetition of software development, and the lack of performance from the use of commercial software packages such as MATLAB and LabVIEW. The requirements of software for future manufacturing does not differ between academic researchers or industrial engineers, they may at different levels of the technology readiness scale, but both require software that is configurable, adaptable to their needs, and above all else, provides real-time measurement capabilities.

Developing custom software is an expensive and time consuming process and therefore many have to comprise for sub-optimum performance through generic commercial software, which provides great documentation, ease of use, and high abstraction from programming, but often does not offer the performance required. Industry 4.0 is pushing measurements to be taken in-process to increase production efficiencies, therefore the processing of measurement data needs to be as fast as the production line itself. If it cannot be taken in process it often will not be taken at all, leading to potential failures, defects, and expensive rework.

Examine how hardware acceleration is currently being used in surface and dimensional measurements to process the increasing data sizes and improving signal processing performance.

From the review into hardware acceleration within surface and dimensional signal processing, the use of FPGAs, DSPs and GPUs has steadily increased over the past twenty years. The most discussed of these methods is GPU processing, due to its relatively low price compared to the performance it can provide. GPGPU acceleration has provided significant performance improvements across the literature reviewed in interferometry, structured light and OCT and is extremely likely to be having the same impact in other fields that require real-time processing of data.

Development of an easy-to-use software package for use by researchers and industry to advance the future of manufacturing by prototyping signal processing quickly and efficiently.

Unlike other open-source alternatives, Optical Signal Processing Workspace has been presented as a software package that researchers and industry can use to configure a pre-built signal processing library for their application without the requirement of programming knowledge. Using the configurator, users can create signals and datapoints that can be drag-and-dropped onto functions controlling the data flow between algorithms. Facilitating the configurator with drag-and-drop functionality keeps the interface looking clean and uncluttered. Most functions allow multiple data types and are configurable for n-length and n-dimensional data allowing full configurability by the user.

The configurator creates two configuration files that are then used by the Signal Processing Engine (SPE) to execute the signal processing specified. One file allows the SPE to execute the signal processing serially on the CPU, the other utilises any GPU hardware available by swapping out CPU functions for GPU functions. The configurator automatically configures memory transfers required for the functions selected when processing data using a GPU. Finally, data processed by the SPE is sent to the User Interface (UI) where is can be displayed by graphs and images or saved for post-processing. Further expansion of the UI is discussed in Further Work, section 9.2

Demonstrate the use of AI Planning to find the optimal solution for a given signal processing sequence and hardware combination across a heterogenous, CPU-GPU architecture.

The novel use of AI Planning, specifically PDDL, to optimise signal processing has been demonstrated with results shown in chapter 7 demonstrating a performance increase of 13% for the LSDI over a single architecture implementation without a user having to manually profile each permutation of CPU and GPU functions. This is in addition to the 11x performance gain already achieved in the LSDI case study over previously used MATLAB.

Having reviewed other informed search techniques and Monte Carlo simulation, the use of PDDL allowed significant pruning of possible permutations (branches) due to the ability to apply constraints to each of the PDDL actions. These constraints and the reduction of branches allowed the search problem to be reduced from $O\left((m^2+n)(m^2+1)^{n-1}\right)$ to closer to $O(2^n)$. Further performance may also be possible with and durative PDDL plans and asynchronous GPU streams, both of which are discussed in the Further Work sections 9.3.2 and 9.4 respectively. Further expansion to different or multiple heuristics is also detailed in the Future Work section 9.3.1.

8.3 Caveats of the Presented Approach

This research has presented a method of optimising the signal processing used within surface and dimensional measurements by using AI planning to select the optimum architecture in a heterogenous system. However, this approach is not without caveats, several of which are discussed here, with potential future solutions to address these deficiencies.

8.3.1 Signal Processing Engine Recompiling

Currently, once OSPW’s configurator has created the configuration files, it modifies the SPE function files to insert the correct function calls into the *CPUConfig* functions. These modifications in a compiled language require the SPE to recompiled before it can be used. Currently, this process is manually achieved by rebuilding the project in Visual Studio, this is far from ideal, especially if this software becomes available for others to use.

It may be beneficial to transfer the compiling of the SPE to GNU Compiler Collection (GCC), a free C compiler that can be accessed through the command line, an important feature for OSPW to automatically initiate the compilation. The benefit to using GCC is its standalone nature, although this require MinGW to run, as GCC is designed for Linux. Whereas the current method of using Visual Studio requires unnecessary additional software, and potentially a license depending on the version used.

8.3.2 Acquisition Devices

Currently, each of the acquisition devices are hard-coded into the SPE, this is not suitable moving forward and requires a more generic approach for future users to be able to use OSPW without needing to manually program their camera SDK. The solution to this is implement the generic API GenICam (European Machine Vision Association), this API has been implemented by a number of machine vision cameras using USB3, GigE and CameraLink.

Utilising this API would allow the SPE to search for cameras connected to the PC, transmit configuration settings, and receive data from the capture device. The user would not need to program anything specific for their camera (unless it was not supported the standard) and would help reduce the compile problem outlined above.

8.3.3 SPE Library Expansion

Currently the library contains a significant number of signal processing functions, these are listed in section 5.3.7, that should cover a wide range of surface and dimensional measurement techniques. There also exists a method to add new functions, that automatically creates configuration XML files and C++ header information discussed in section 5.3.1. However, the signal processing function itself would need to be manually imported from tested, working, C++ code. This may have undesirable consequences if not imported correctly, or is not in a format expected by the import manager.

If a user does not have a programming background, they might not be able to develop a C++ function to import. Or perhaps, a data type that is not currently supported needs to be implemented, or for variables like complex numbers, a different naming convention is required to store the real and complex parts, a problem faced during this research (Table 8-1). Finally, a user may have a bespoke C++ function but no GPU counterpart and, as discussed in chapter 2, there are no adequate solutions that can take MATLAB code, or sequential C++ code and generate an optimised GPU version, especially one that would then easily integrate into OSPW. However, for users with some programming experience a discussion surrounding the release of this research as an open-source package can be found in section 8.4.

Table 8-1: *Complex Variable Naming Conventions Used During This Research*

Implementation	Real	Imaginary
HSFFT	Variable[index].re	Variable[index].im
FFTW	Variable[index][0]	Variable[index][1]
cuFFT	Variable[index].x	Variable[index].y

A more rigorous approach to above problem might be to implement a Dynamic Link Library (DLL) interface to execute functions from precompiled shared libraries. This would also reduce some of the requirements to recompile if the signal processing library exists in DLL files, there would be no need to edit header files for a user's requirement, instead selecting a different variation of the linked library. However, this option was not investigated in this research.

8.3.4 Deterministic Measurement Intervals

In a subset of manufacturing scenarios there is a requirement measurement processing at a consistent frame rate. In an application such as roll-to-roll processing (R2R) the measurand under the measurement apparatus is moving at a constant speed and therefore there is a requirement to take a measurement at pre-determined intervals. However, as previously discussed the execution time of signal processing on an IPC fluctuates unpredictably due to non-related operating system demands (Figure 8-1). These variations are the reason the presented approach uses an averaging algorithm when profiling signal processing algorithms. This averaging method provides a benefit when comparing CPU and GPU execution times but is little help when determining the exact latency of individual measurements.

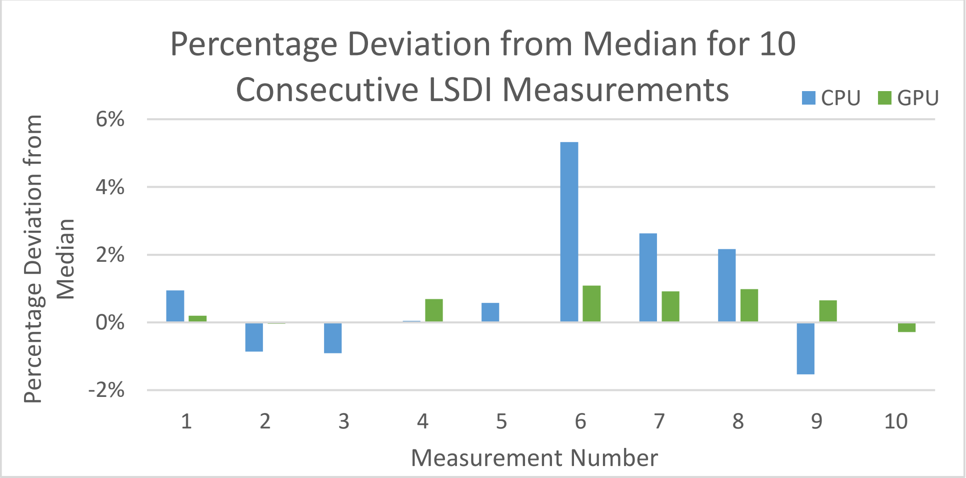


Figure 8-1*: Percentage Deviation from Median for 10 Consecutive LSDI Measurements*

However even though OSPW can not provide determanistic measurement intervals, it does not mean that this research can not provide benefits to those requiring consistant processing frame rates. The above graph illustrates the deviation is no higher than 6% and significantlly less on the GPU. As long as the optimial soluton measurement is faster than the process its measuring, there is still a benefit to optimising. It may be that by optimsing for a heterogenous architecture is the only method that provides a process fast enough to measure the measurand in real-time. All of this excludes the additional benefit provided from having a library of mid-level signal processing functions without requiring the user to program them themselves.

8.3.5 Summary

While there are some limitations to the presented approach, it does provide a significant performance advantage from both the use of C++ and CUDA, and further optimisations from using AI planning. Some of these limitations discussed have had solutions proposed to improve or correct the related issue, and in addition to these, more improvements will be discussed in the future work section, chapter 9.

8.4 Potential Routes to Distribution of OSPW

Distribution of OSPW is limited by the third-party libraries that have been integrated. FFTW is covered by a GPL2 license and ALGLIB uses an MIT license. Neither of these distribution licenses forbid commercial distribution but do require that the source code must distributed along with the program. This would not be an issue as OSPW needs compiling after configurating and for that the source code needs to be modifiable, and therefore open. This also allows user to add extra functionality to OSPW if they have a specific device that they need to include that is not natively supported by GenICam or add new signal processing functionality.

However, trying to commercialise open-source software historically does not work due to the open nature of the code. In the past companies have distributed their software and charged an optional license fee for support to receive a return on their development costs. For OSPW an open-source distribution makes the most sense given the license restrictions, with the potential option to offer a paid support service. This method of distribution is likely to have the greatest impact on the advancement of surface and dimensional measurement signal processing in both industry and academia.

8.5 Closing Statement

This research has presented Optical Signal Processing Workspace (OSPW), a novel approach of using PDDL to optimise a library of signal processing functions used in optical surface and dimensional measurements on a heterogenous system. Using a modern, easy to use, user interface, the signal processing can be configured without prior knowledge of a programming language and can be configured for any data length with up to three dimensions. OSPW provides hardware acceleration using GPGPU though Nvidia's CUDA API with the majority of the library functions having GPU accelerated version. Through the selection of C++ and CUDA, a performance increase of up to 21x was achieved in the DRI case study and 5.45x in the LSDI cast study when compared to MATLAB. With the introduction of AI planning to optimise the signal processing sequence for the heterogenous hardware available, this novel approach provided a further performance increase of 13% for in the LSDI case study.

With some further development work, discussed in the next chapter, this novel approach to providing an adaptable signal processing library that optimises itself will provide significant performance benefit to researchers and industrial engineers. Furthermore, it will save time removing the requirement to learn a programming language and develop their own software, time that can be reinvested in further advance surface and dimensional sensor and instrumentation research.

9 Further Work

9.1 Introduction

In addition to the novel approach of using AI planning to optimise signal processing on a heterogenous system, further work has been identified to improve the presented approach. This chapter introduces on-going research and ideas which aim to further increase the user experience, provide further performance optimisations, and allow users to optimise for multiple heuristics.

9.2 Further Expansion of the User Interface

Chapter 5 described OSPW's user interface and the various visualisation methods available for users to view the data processed by the SPE. Due to the one-dimensional data of the DRI and two-dimensional data of the LSDI, the chance to develop more visualisation techniques unfortunately never materialised. The two main visualisation methods that are absent are two-dimensional and three-dimensional graph plotting.

.NET Framework natively supports drawing two-dimensional graphs using its Chart class; however, it does not support three-dimensional graph plotting. Paid offerings exist from third-party companies such as SciChart, Xceed, and ILNumerics. However, there does appear to be ways of plotting these graphs without the use of paid software, but unlike the commercial offerings, these methods are not optimised and take considerable time to draw, something not useful on a live production line.

Previously discussed ArrayFire does have the capability of drawing three-dimensional graphs using OpenGL and the GPU, however this would have to be part of the SPE and not the UI. There are both benefits and drawbacks to this method. The benefit is that the graphs are rendered on the GPU which is much faster than they can be rendered on the CPU, saving valuable time. The drawback is while the image is rendering, new data is not being processed, reducing overall efficiency. However, if the images cannot be rendered in the UI, and this is the only option, then a reduction in efficiency is preferred over not visualising the data at all. However, if ArrayFire can be implemented correctly, the GPU could be rendering one image, while the data is imported for the next, but this would need careful consideration.

9.3 Modifications to the PDDL Configuration

9.3.1 Introduction of New Heuristics

The notion of different or multiple heuristics was discussed in in section 6.2, with this research concentrating on the execution time (latency) being the heuristic to optimise for. Both storage and network speeds were discussed as possible heuristics but dismissed due to the potential to remove any issues being caused by these parameters by upgrading the IPC equipment at relatively low cost.

While the benefit of batch processing (throughput optimisation) was illustrated in section 6.5, the process to profile the signal processing sequence for multiple batch sizes would be lengthy process for a user, however the ability to find the point of diminishing results for batch size could be useful data for a measurement. As shown for the LSDI, processing more than 32 or more frames simultaneously on the GPU provided a deficit in the overall processing rate, compared to 16 frames.

Finally, the potential for optimising for the lowest power consumption was also discussed in section 6.2. Looking at an illustrative example (Figure 9-1), assuming a constant power draw for the duration of the processing, the GPU uses nearly 4x the energy of a CPU execution. Even when optimising for the lowest latency, the 8.1 J required is still significantly above that required by the sequential CPU sequence.

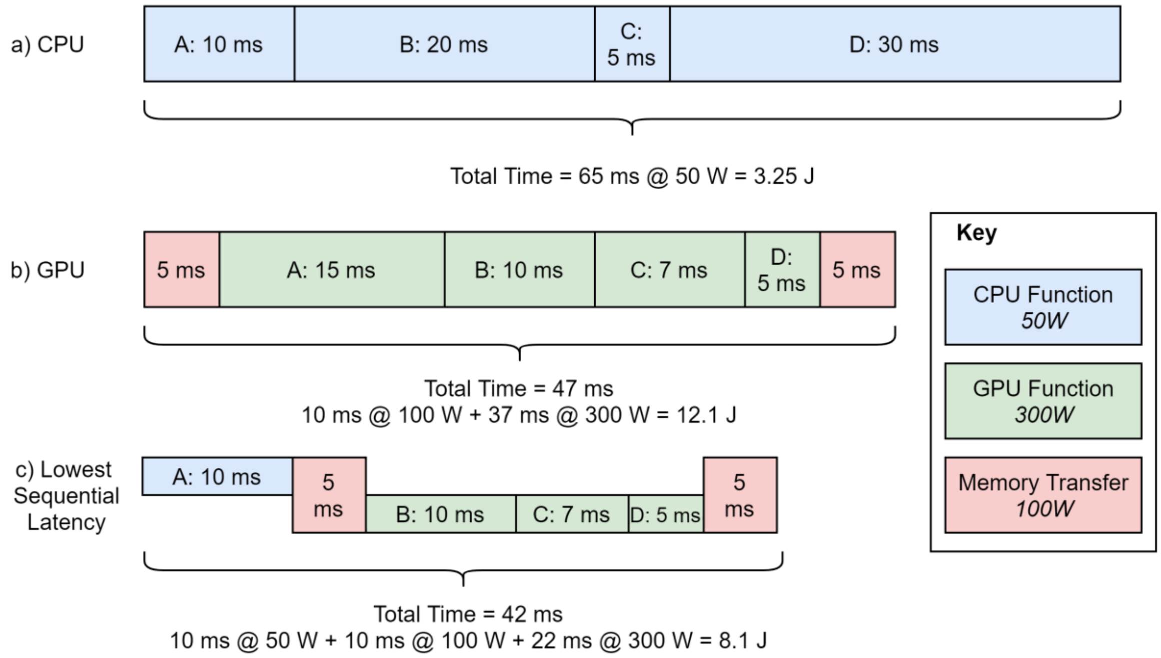


Figure 9-1: *Illustrative Example of Energy Usage – Latency Optimisation*

Therefore, it may safe to assume that if power efficiency is required, a GPU is completely out of the question, however, that is not the case. If for example a function takes one tenth of the time on a GPU compared to a CPU, the 600% increase in power required by the GPU is a more energy efficient over the entire execution. Figure 9-2 illustrates a different example, however, in this example, due to the slow nature of function *D* on the CPU, the CPU sequence consumes 21 J of energy, whereas due to the parallelism nature of the GPU, this function only takes 5 ms, with a total energy consumption of 13.6 J. Furthermore, the GPU is even still providing a latency benefit over the CPU, saving over 350 ms, an 87% reduction.

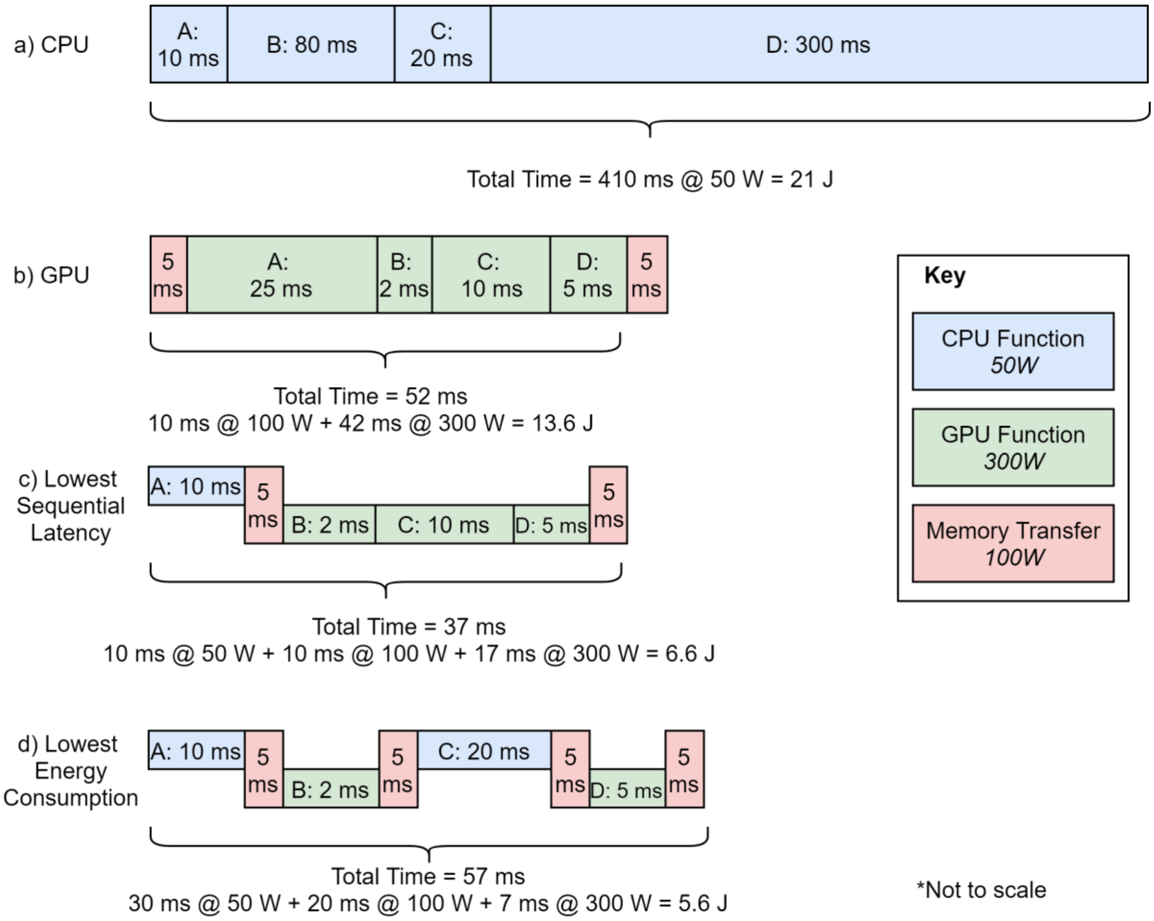


Figure 9-2: *Illustrative Example of Energy Usage – Energy Optimisation*

If, as before, the domain is set to optimise for the lowest latency, Figure 9-2c returns an even faster result which uses even less energy at 6.6 J. However, if instead the domain heuristic is energy consumption, instead of execution time, a more energy efficient, however slower, solution can be found, at just 5.6 J (Figure 9-2d). This illustrates, the potential different heuristics that could be used to optimise a user's signal processing for the parameter that is the most critical for a given application.

9.3.2 PDDL: Durative Actions

Introduced in PDDL 2.1 along with fluents, previously discussed in Chapter 3.3.2.2, were durative actions. In its current setup, PDDL treats every action as instantaneous with a time duration of one, whereas with durative actions, the duration of each action can be specified. This adds a couple of extra properties to each action and a potential benefit to the solution. Firstly, it changes the way preconditions are checked, rather than simply checking whether each of the preconditions are true, the planner requires each of the preconditions to be checked at a particular time interval during the action (Figure 9-3). Secondly, effects are also triggered at time intervals, these intervals are at start, at end and over all.

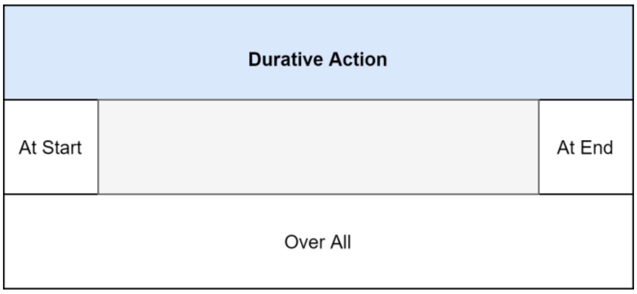
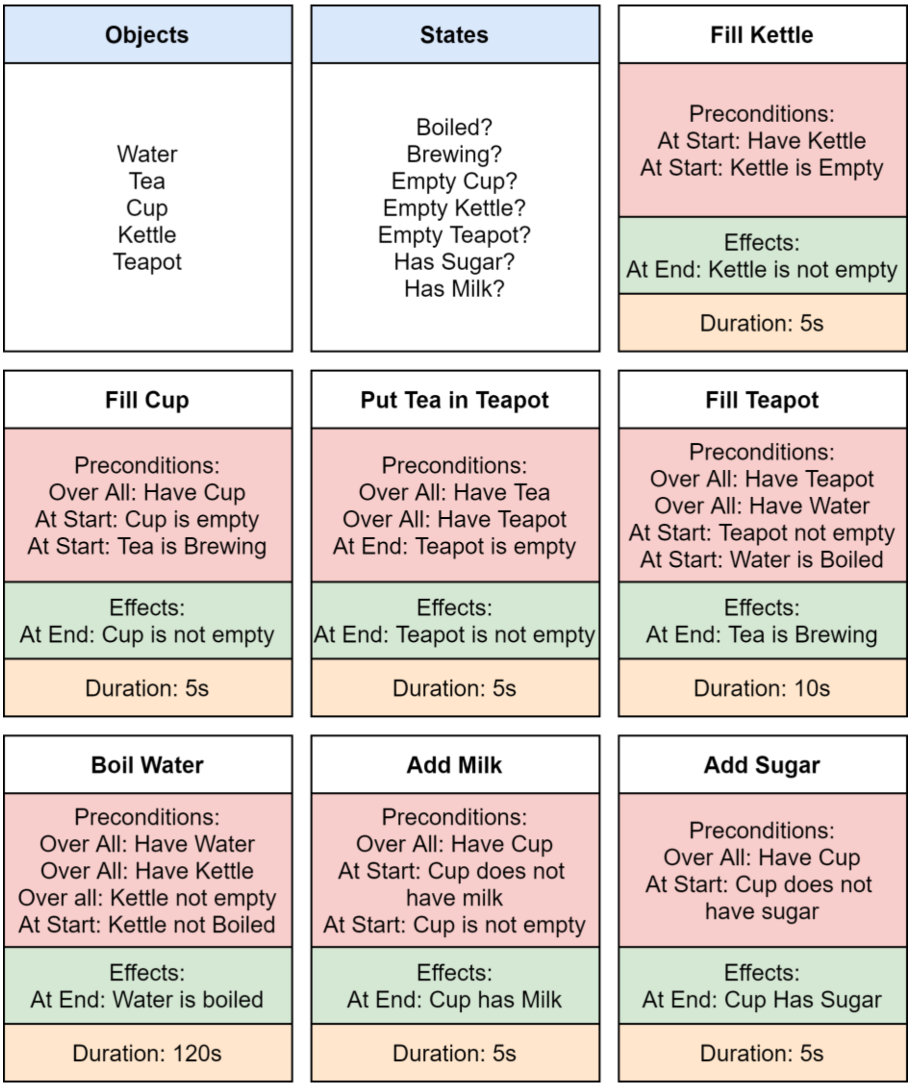


Figure 9-3: *PDDL Durative action time intervals*

From the *Cup of Tea* example laid out in Chapter 3.3.2.1, it can be adapted to utilise durative actions (Figure 9-4). Now, instead of all the actions happening instantaneously, they all have a duration assigned to them. Each action has had its preconditions and effects modified to include the time interval at which they need to be checked or executed. It is important that these are specified correctly, for example, when filling a cup, *haveCup* needs to be true throughout the entire function, therefore no other action can be run concurrently that removes the cup.



Objects

Water
Tea
Cup
Kettle
Teapot

States

Boiled?
Brewing?
Empty Cup?
Empty Kettle?
Empty Teapot?
Has Sugar?
Has Milk?

Fill Kettle

Preconditions:
At Start: Have Kettle
At Start: Kettle is Empty

Effects:
At End: Kettle is not empty

Duration: 5s

Fill Cup

Preconditions:
Over All: Have Cup
At Start: Cup is empty
At Start: Tea is Brewing

Effects:
At End: Cup is not empty

Duration: 5s

Put Tea in Teapot

Preconditions:
Over All: Have Tea
Over All: Have Teapot
At End: Teapot is empty

Effects:
At End: Teapot is not empty

Duration: 5s

Fill Teapot

Preconditions:
Over All: Have Teapot
Over All: Have Water
At Start: Teapot not empty
At Start: Water is Boiled

Effects:
At End: Tea is Brewing

Duration: 10s

Boil Water

Preconditions:
Over All: Have Water
Over All: Have Kettle
Over all: Kettle not empty
At Start: Kettle not Boiled

Effects:
At End: Water is boiled

Duration: 120s

Add Milk

Preconditions:
Over All: Have Cup
At Start: Cup does not have milk
At Start: Cup is not empty

Effects:
At End: Cup has Milk

Duration: 5s

Add Sugar

Preconditions:
Over All: Have Cup
At Start: Cup does not have sugar

Effects:
At End: Cup Has Sugar

Duration: 5s

Figure 9-4: *Cup of Tea Example (With Durative Actions)*

When applying this to signal processing, each variable required would be checked for its availability *atStart*, any variable that was modified during the function would need to be available *overAll* and the effect flag that indicates that a function has been execution would be applied *atEnd*. This would allow functions that are mutually exclusive to each other to be executed during the same period. Although profiling is not an exact measurement of the time taken to execute each function, instead being a repeatable comparison, therefore, concurrency may not be possible for short functions, but would have benefits for functions that take longer, or have a small standard deviation from the median.

The reason for durative actions not being currently implemented in OSPW is the significant computation required to process the durative data. Not only does the PDDL planner have to find a solution through the actions that meets the goal criteria, there are an infinite number of start times that each action could run at. In the pursuit of computing the optimum plan, the planner investigates multiple paths, as these paths now contain durations, where functions may benefit from running in parallel, the number of paths is exponentially larger. The computation was attempted for the LSDI in Windows 7; however, the planner would return inconsistent solutions and often crash before finishing. LPG-td is available on Linux, but that currently would require having to transfer data and processing the plan on a different operating system.

9.4 Consideration of Asynchronous GPU Streams

In a standard CUDA implementation, one single stream is used, whereby all the kernels and memory transfers are executed by stream zero, like how a single thread is used for basic CPU processing. However, CUDA offers the ability to run multiple concurrent streams, up to sixteen kernels and a memory copy in either direction all at the same time. This concurrency allows significant increases in performance if the application is not too sequential and benefits from multiple streams. Figure 9-5 shows how a serial CUDA implementation could be split into multiple concurrent streams, by executing memory transfers at the same time as kernels, here the kernels themselves are not running in parallel, but up to sixteen kernels can be executed at once. Kernel concurrency is not normally utilised due to data from one kernel often required for subsequent kernels. When increasing the number of concurrent streams, it is also possible to have the GPU executing a kernel and performing two memory transfers while the CPU is computing the fourth, not requiring any data to be transferred between the host and device for the fourth function.

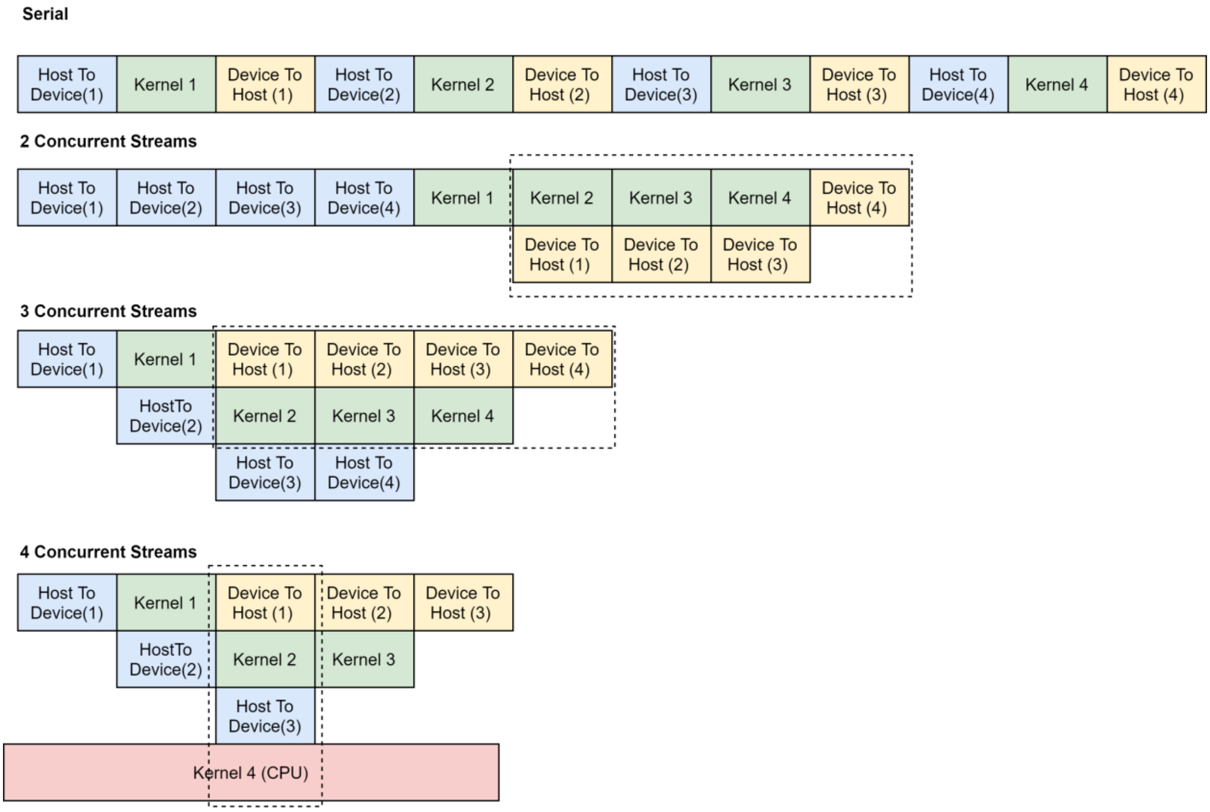


Figure 9-5: Multiple concurrent CUDA Streams

This currently is not implemented as the PDDL current implementation is not able to plan for actions (functions) to execute concurrently. However, having discussed durative actions in the previous section, the problem file can define three simultaneous streams, one for kernels and one for each direction of memory transfer. When a kernel or memory transfer is running, the corresponding stream will be marked as *inUse*, therefore preventing multiple data transfers at once. Finally, if mutually exclusive actions can run concurrently, PDDL will find the fastest solution that satisfies the goal.

The SPE will need modifications to allow the job queue to run concurrent functions either solely on the GPU, or dual execution on the CPU and GPU. It will also need to ensure that if multiple functions are being run concurrently, that all prerequisites to a function have been completed before continuing. Further to the comments made in the previous sub-section, this adds further complication to finding the optimum solution using a PDDL planner, with the number of permutations increasing once again. Each possible stream combination will have to be evaluated to see where performance can be optimised, this exponential problem will further question whether LPG-td is able to process the plan and deliver a solution.

10 References

Aguilar, J., Cordero, J., & Buendía, O. (2018). Specification of the autonomic cycles of learning analytic tasks for a smart classroom. *Journal of Educational Computing Research, 56*(6), 866-891.

Ajila, S. A., & Wu, D. (2007). Empirical study of the effects of open source adoption on software development economics. *Journal of Systems and Software, 80*(9), 1517-1529. doi:https://doi.org/10.1016/j.jss.2007.01.011

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials, 17*(4), 2347-2376. doi:10.1109/COMST.2015.2444095

Albayrak, O. E., Akturk, I., & Ozturk, O. (2013). Improving application behavior on heterogeneous manycore systems through kernel mapping. *Parallel Computing, 39*(12), 867-878. doi:10.1016/j.parco.2013.08.011

Alcácer, V., & Cruz-Machado, V. (2019). Scanning the Industry 4.0: A Literature Review on Technologies for Manufacturing Systems. *Engineering Science and Technology, an International Journal, 22*(3), 899-919. doi:https://doi.org/10.1016/j.jestch.2019.01.006

Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*. Paper presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey.

Ashton, K. (2009). That 'internet of things' thing.

Bailey, D. (2019). *Image Processing Using FPGAs*.

Balser. (2020). Basler racer raL8192-12gm - Line Scan Camera. Retrieved from https://www.baslerweb.com/en/products/cameras/line-scan-cameras/racer/ral8192-12gm/

Boehm, B. W. (1984). Software Engineering Economics. *IEEE Transactions on Software Engineering, SE-10*(1), 4-21. doi:10.1109/TSE.1984.5010193

Boehm, B. W., & Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering, 14*(10), 1462-1477. doi:10.1109/32.6191

Cansalar, C. A., Mavis, E., & Kasnakoglu, C. (2015). *Simulation time analysis of MATLAB/Simulink and LabVIEW for control applications.* Paper presented at the Proceedings of the IEEE International Conference on Industrial Technology.

Cemernek, D., Gursch, H., & Kern, R. (2017, 24-26 July 2017). *Big data as a promoter of industry 4.0: Lessons of the semiconductor industry.* Paper presented at the 2017 IEEE 15th International Conference on Industrial Informatics (INDIN).

Chandra, R. (2001). *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann Publishers.

Cheik Ahamed, A.-K., & Magoulès, F. (2017). Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL. *Advances in Engineering Software, 111*, 32-42. doi:https://doi.org/10.1016/j.advengsoft.2016.10.002

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering, 5*(1), 46-55. doi:10.1109/99.660313

Dastgeer, U., Li, L., & Kessler, C. (2013). Adaptive Implementation Selection in the SkePU Skeleton Programming Library. In C. Wu & A. Cohen (Eds.), *Advanced Parallel Processing Technologies: 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers* (pp. 170-183). Berlin, Heidelberg: Springer Berlin Heidelberg.

Department for Business, E. I. S. (2020). Business population estimates for the UK and regions 2020. Retrieved from https://www.gov.uk/government/statistics/business-population-estimates-2020/business-population-estimates-for-the-uk-and-regions-2020-statistical-release-html#:~:text=UK%20private%20sector.-,At%20the%20start%20of%202020%3A,%C2%A30.7%20trillion%20(16%25)

Desjardins, A. E., Vakoc, B. J., Suter, M. J., Yun, S. H., Tearney, G. J., & Bouma, B. E. (2009). Real-Time FPGA Processing for High-Speed Optical Frequency Domain Imaging. *IEEE Transactions on Medical Imaging, 28*(9), 1468-1472. doi:10.1109/TMI.2009.2017740

Digital, W. (2021). Data Sheet: WD_BLACK SN850 NVMe SSD. Retrieved from https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-black-ssd/data-sheet-wd-black-sn850-nvme-ssd.pdf

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik, 1*(1), 269-271.

Electric, G. (2013). What is the Industrial Internet of Things (IIoT)? Retrieved from https://www.ge.com/digital/blog/what-industrial-internet-things-iiot

European Machine Vision Association. GenICam - EMVA. Retrieved from https://www.emva.org/standards-technology/genicam/

Fang, J., Varbanescu, A. L., & Sips, H. (2011, 13-16 Sept. 2011). *A Comprehensive Performance Comparison of CUDA and OpenCL.* Paper presented at the 2011 International Conference on Parallel Processing.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence, 2*(3-4), 189-208.

Fitzgerald, B. (2006). The Transformation of Open Source Software. *MIS Quarterly, 30*(3), 587-598. doi:10.2307/25148740

Forbes. (2020). Can AMD's Share In The GPU Market Rise 20%? Retrieved from https://www.forbes.com/sites/greatspeculations/2020/03/16/can-amds-share-in-the-gpu-market-rise-20/

Fowers, J., Brown, G., Cooke, P., & Stitt, G. (2012). *A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications*. Paper presented at the Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, Monterey, California, USA. https://doi.org/10.1145/2145694.2145704

Fox, M., & Long, D. (2003). PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*.

Frana, P. L., & Misa, T. J. (2010). An interview with Edsger W. Dijkstra. *Commun. ACM, 53*(8), 41–47. doi:10.1145/1787234.1787249

Freiman, F. R., & Park, R. (1979). *Price software model-version 3: An overview.* Paper presented at the Proceedings, ieee/piny workshop on quantitative software models, ieee catalog no. th0067.

Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE, 93*(2), 216-231.

Gao, W., Haitjema, H., Fang, F. Z., Leach, R. K., Cheung, C. F., Savio, E., & Linares, J. M. (2019). On-machine and in-process surface metrology for precision manufacturing. *CIRP Annals, 68*(2), 843-866. doi:https://doi.org/10.1016/j.cirp.2019.05.005

Gartner. (2021). Gartner Says Worldwide Smartphone Sales Declined 5% in Fourth Quarter of 2020. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2021-02-22-4q20-smartphone-market-share-release

Gartner, I. (2015). Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015 [Press release]. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2015-11-10-gartner-says-6-billion-connected-things-will-be-in-use-in-2016-up-30-percent-from-2015

Geng, J. (2011). Structured-light 3D surface imaging: a tutorial. *Advances in Optics and Photonics, 3*(2), 128-160. doi:10.1364/AOP.3.000128

Georgakopoulos, D., Jayaraman, P. P., Fazia, M., Villari, M., & Ranjan, R. (2016). Internet of Things and Edge Cloud Computing Roadmap for Manufacturing. *IEEE Cloud Computing, 3*(4), 66-73. doi:10.1109/MCC.2016.91

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated planning: theory and practice*. Amsterdam;Boston;: Elsevier/Morgan Kaufmann.

Goodcore. (2020). How Much Does It Cost To Develop Custom Software: What Industry Experts Say. Retrieved from https://www.goodcore.co.uk/blog/cost-to-develop-software/

Greer, C., Burns, M., Wollman, D., & Griffor, E. (2019). Cyber-physical systems and internet of things. In.

Grewe, D., & O'Boyle, M. F. P. (2011) A static task partitioning approach for heterogeneous systems using OpenCL. In*: Vol. 6601 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (pp. 286-305).

Gronle, M., Lyda, W., Wilke, M., Kohler, C., & Osten, W. (2014). itom: an open source metrology, automation, and data evaluation software. *Applied optics, 53*(14), 2974-2982. doi:10.1364/AO.53.002974

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems, 29*(7), 1645-1660. doi:https://doi.org/10.1016/j.future.2013.01.010

HajiRassouliha, A., Taberner, A. J., Nash, M. P., & Nielsen, P. M. F. (2018). Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms. *Signal Processing: Image Communication, 68*, 101-119. doi:https://doi.org/10.1016/j.image.2018.07.007

Hao, G., Takeshi, T., & Idaku, I. (2012). *GPU-based real-time structured light 3D scanner at 500 fps.* Paper presented at the Proc.SPIE.

Horton, I. (2004). *Ivor Horton's beginning ANSI C++: the complete language* (3rd ed.). Berkeley, Calif: Apress.

Huckaby, J., Vassos, S., & Christensen, H. I. (2013, 3-7 Nov. 2013). *Planning with a task modeling framework in manufacturing robotics.* Paper presented at the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems.

Hussain, R. (2016). rafat/hsfft: FFT Implementation. Equally suitable for power of 2 and non-power of 2 input data. Retrieved from https://github.com/rafat/hsfft

Hussain, T., Amin, S., Zabit, U., Kamran, F., Bernal, O. D., & Bosch, T. (2017). *A High Performance Real-Time FGPA-Based Interferometry Sensor Architecture.* Paper presented at the Proceedings - 14th International Conference on Frontiers of Information Technology, FIT 2016.

Intel. (2020). Intel® FPGAs and Programmable Devices - Intel® FPGA. Retrieved from https://www.intel.co.uk/content/www/uk/en/products/programmable.html

Jiang, X., Scott, P. J., Whitehouse, D. J., & Blunt, L. (2007). Paradigm shifts in surface metrology. Part I. Historical philosophy. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science, 463*(2085), 2049-2070. doi:10.1098/rspa.2007.1874

Kádár, B., Lengyel, A., Monostori, L., Suginishi, Y., Pfeiffer, A., & Nonaka, Y. (2010). Enhanced control of complex production structures by tight coupling of the digital and the physical worlds. *CIRP Annals, 59*(1), 437-440. doi:https://doi.org/10.1016/j.cirp.2010.03.123

Kanawaday, A., & Sane, A. (2018). *Machine learning for predictive maintenance of industrial machines using IoT sensor data.* Paper presented at the Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS.

Karasev, P. A., Campbell, D. P., & Richards, M. A. (2007, 17-20 April 2007). *Obtaining a 35x Speedup in 2D Phase Unwrapping Using Commodity Graphics Processors.* Paper presented at the Radar Conference, 2007 IEEE.

Karimi, K., Dickson, N. G., & Hamze, F. (2010). A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*.

Karpiński, M., Khoma, V., Khoma, A., & Więcław, Ł. (2017, 21-23 Sept. 2017). *Segment approximation approach for reconstructing of surface topology.* Paper presented at the 2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS).

Karpinsky, N., Hoke, M., Chen, V., & Zhang, S. (2014). High-resolution, real-time three-dimensional shape measurement on graphics processing unit. *Optical Engineering, 53*(2), 024105-024105. doi:10.1117/1.OE.53.2.024105

Khan, M. N., Hasnain, S. K., & Jamil, M. (2016). *Digital signal processing: a breadth-first approach* (Vol. 1). Aalborg, Denmark: River Publishers.

Khronos. (2019). OpenCL Overview - The Khronos Group Inc. Retrieved from https://www.khronos.org/opencl/

Klöpper, B. (2010). First Steps Towards Distributed Multi-objective Scheduling for Self-optimizing Manufacturing Systems*. *IFAC Proceedings Volumes, 43*(4), 332-337. doi:https://doi.org/10.3182/20100701-2-PT-4011.00057

LabVIEW. (2019). Benefits of Programming Graphically in NI LabVIEW [White paper]. Retrieved from https://www.ni.com/en-gb/innovations/white-papers/13/benefits-of-programming-graphically-in-ni-labview.html

Laney, D. (2001). 3D data management: Controlling data volume, velocity and variety. *META group research note, 6*(70), 1.

Lasi, H., Fettke, P., Kemper, H.-G., Feld, T., & Hoffmann, M. (2014). Industry 4.0. *Business & Information Systems Engineering, 6*(4), 239-242. doi:10.1007/s12599-014-0334-4

Lee, J., Bagheri, B., & Kao, H.-A. (2015). A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters, 3*, 18-23. doi:https://doi.org/10.1016/j.mfglet.2014.12.001

Lee, J., Lapira, E., Bagheri, B., & Kao, H.-a. (2013). Recent advances and trends in predictive manufacturing systems in big data environment. *Manufacturing Letters, 1*(1), 38-41. doi:https://doi.org/10.1016/j.mfglet.2013.09.005

Leo Kumar, S. P. (2017). State of The Art-Intense Review on Artificial Intelligence Systems Application in Process Planning and Manufacturing. *Engineering Applications of Artificial Intelligence, 65*, 294-329. doi:10.1016/j.engappai.2017.08.005

Lerner, J., & Triole, J. (2000). The Simple Economics of Open Source. *National Bureau of Economic Research Working Paper Series, No. 7600*. doi:10.3386w7600

Lewis, T. (1999). The open source acid test. *Computer, 32*(2), 128-127. doi:10.1109/2.745728

Li, J., Bloch, P., Xu, J., Sarunic, M. V., & Shannon, L. (2011). Performance and scalability of Fourier domain optical coherence tomography acceleration using graphics processing units. *Applied optics, 50*(13), 1832-1838. doi:10.1364/AO.50.001832

Li, J., Sarunic, M. V., & Shannon, L. (2011, 1-3 May 2011). *Scalable, High Performance Fourier Domain Optical Coherence Tomography: Why FPGAs and Not GPGPUs.* Paper presented at the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines.

Li, X., Xiao, S., Zhou, Q., Ni, K., & Wang, X. (2019). *A real-time distance measurement data processing platform for multi-axis grating interferometry type optical encoders.* Paper presented at the Proceedings of SPIE - The International Society for Optical Engineering.

Lischner, R. (2013). *Exploring C++ 11* (Second;2; ed.). Berkeley, CA: Apress.

Lu, Y. (2017). Industry 4.0: A survey on technologies, applications and open research issues. *Journal of Industrial Information Integration, 6*, 1-10. doi:https://doi.org/10.1016/j.jii.2017.04.005

Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K., & Melonakos, J. (2012). *ArrayFire: a GPU acceleration platform.* Paper presented at the SPIE Defense, Security, and Sensing.

Martin, H., & Jiang, X. (2013). Dispersed reference interferometry. *CIRP Annals - Manufacturing Technology, 62*(1), 551-554. doi:10.1016/j.cirp.2013.03.104

Martin, H., Kumar, P., Henning, A., & Jiang, X. (2020). Extended sub-surface imaging in industrial OCT using 'non-diffracting'Bessel beams. *CIRP Annals, 69*(1), 493-496.

MATLAB. (2020). MATLAB vs. Python: Top Reasons to Choose MATLAB. Retrieved from https://uk.mathworks.com/products/matlab/matlab-vs-python.html

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., . . . Wilkins, D. (1998). PDDL-the planning domain definition language. In: Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational ....

McKeown, P., Wills-Moren, W., & Read, R. (1987). *In-Situ Metrology And Machine Based Interferometry For Shape Determination* (Vol. 0802): SPIE.

Memeti, S., Pllana, S., Binotto, A., Kołodziej, J., & Brandic, I. (2018). *A Review of Machine Learning and Meta-heuristic Methods for Scheduling Parallel Computing Systems*. Paper presented at the Proceedings of the International Conference on Learning and Optimization Algorithms: Theory and Applications, Rabat, Morocco. https://doi.org/10.1145/3230905.3230906

Microsoft. (2016). Ellipsis and Variadic Templates. Retrieved from https://docs.microsoft.com/en-us/cpp/cpp/ellipses-and-variadic-templates?view=vs-2019

Microsoft. (2018). QueryPerformanceCounter function. Retrieved from https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter

Microsoft. (2019, 05/06/2019). Unions. Retrieved from https://docs.microsoft.com/en-us/cpp/cpp/unions?view=vs-2019

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., . . . Ueda, K. (2016). Cyber-physical systems in manufacturing. *CIRP Annals, 65*(2), 621-641. doi:https://doi.org/10.1016/j.cirp.2016.06.005

Montrym, J. S., Baum, D. R., Dignam, D. L., & Migdal, C. J. (1997). *InfiniteReality: A real-time graphics system.* Paper presented at the Proceedings of the 24th annual conference on Computer graphics and interactive techniques.

Morgan, L., & Finnegan, P. (2007). *Benefits and Drawbacks of Open Source Software: An Exploratory Study of Secondary Software Firms*, Boston, MA.

Muhamedsalih, H. (2013). *Investigation of wavelength scanning interferometry for embedded metrology.* (PhD). Huddersfield,

Muhamedsalih, H., Jiang, X., & Gao, F. (2011, 30/6/2011). *Acceleration computing process in wavelength scanning interferometry*.

Nvidia. (2019). CUDA Zone | NVIDIA Developer. Retrieved from https://developer.nvidia.com/cuda-zone

Nvidia. (2021). Nvidia RTX A6000 Datasheet. Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web%20(1).pdf

Obitko, M., & Jirkovský, V. (2015). *Big data semantics in industry 4.0.* Paper presented at the International conference on industrial applications of holonic and multi-agent systems.

Obitko, M., Jirkovský, V., & Bezdíček, J. (2013). Big data challenges in industrial automation. In *Industrial Applications of Holonic and Multi-Agent Systems* (pp. 305-316): Springer.

Pacholik, A., Muller, M., Fengler, W., Machleidt, T., & Franke, K. H. (2011, 2011). *GPU vs FPGA: Example Application on White Light Interferometry*.

Parkinson, S., Longstaff, A. P., Fletcher, S., Crampton, A., & Gregory, P. (2012). Automatic planning for machine tool calibration: A case study. *Expert Syst. Appl., 39*(13), 11367–11377. doi:10.1016/j.eswa.2012.03.054

Paulsen, G. Y., Feinberg, J., Cai, X., Nordmoen, B., & Dahle, H. P. (2016, 12-16 June 2016). *Matlab2cpp: A Matlab-to-C++ code translator.* Paper presented at the 2016 11th System of Systems Engineering Conference (SoSE).

PayScale. (2021). Average Software Engineer Salary in United Kingdom. Retrieved from https://www.payscale.com/research/UK/Job=Software_Engineer/Salary

Pednault, E. P. (1989). ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. *Kr, 89*, 324-332.

Peng, Y., Xue, Y., & Gao, R. (2015). An optical Fourier Transform spectrometer based on the Michelson interferometer with angle difference between two mirrors. *Optik, 126*(9-10), 1002-1005. doi:10.1016/j.ijleo.2015.01.003

Pisching, M. A., Junqueira, F., Filho, D. J. S., & Miyagi, P. E. (2015). *Service Composition in the Cloud-Based Manufacturing Focused on the Industry 4.0*, Cham.

Platforms, G. I. (2012). The rise of industrial big data. *GE Intelligent Platforms*.

Pryor, G., Lucey, B., Maddipatla, S., McClanahan, C., Melonakos, J., Venugopalakrishnan, V., . . . Malcolm, J. (2011). *High-level GPU computing with Jacket for MATLAB and C/C++.* Paper presented at the Proceedings of SPIE - The International Society for Optical Engineering.

Purde, A., Meixner, A., Schweizer, H., Zeh, T., & Koch, A. (2004, 18-20 May 2004). *Pixel shader based real-time image processing for surface metrology.* Paper presented at the Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE.

Putnam, L. H. (1978). A General Empirical Solution to the Macro Software Sizing and Estimating Problem. *IEEE Transactions on Software Engineering, SE-4*(4), 345-361. doi:10.1109/TSE.1978.231521

Ralston, T., Mayen, J., Marks, D., & Boppart, S. (2004). *Real-time digital design for an optical coherence tomography acquisition and processing system* (Vol. 5324): SPIE.

Rasakanthan, J., Sugden, K., & Tomlins, P. H. (2011). Processing and rendering of Fourier domain optical coherence tomography images at a line rate over 524 kHz using a graphics processing unit. *Journal of Biomedical Optics, 16*(2), 020505-020505-020503. doi:10.1117/1.3548153

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd, International ed.). London;Boston, [Mass.];: Pearson.

Sanchez, M., Exposito, E., & Aguilar, J. (2020). Autonomic computing in manufacturing process coordination in industry 4.0 context. *Journal of Industrial Information Integration, 19*, 100159. doi:https://doi.org/10.1016/j.jii.2020.100159

Sanderson, C. (2010). Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments.

Schneider, M., Fey, D., Kapusi, D., & Machleidt, T. (2011). Performance comparison of designated preprocessing white light interferometry algorithms on emerging multi- and many-core architectures. *Procedia Computer Science, 4*, 2037-2046. doi:https://doi.org/10.1016/j.procs.2011.04.222

Scholz, T., Rosenberger, M., & Notni, G. (2019). *Massively Parallel Implementation of a Fast Resource Efficient White Light Interferometry Algorithm.* Paper presented at the 2018 International Conference on Digital Image Computing: Techniques and Applications, DICTA 2018.

Strauß, P., Schmitz, M., Wöstmann, R., & Deuse, J. (2019). *Enabling of Predictive Maintenance in the Brownfield through Low-Cost Sensors, an IIoT-Architecture and Machine Learning.* Paper presented at the Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018.

Sun, H.-W., Zhou, M., & Wolf, C. (2001). *A methodology for software development cost analysis in information-based manufacturing.* Paper presented at the Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164).

Sussman, G. J. (1973). *A computational model of skill acquisition.* Massachusetts Institute of Technology,

Takaya, Y. (2014). In-Process and On-Machine Measurement of Machining Accuracy for Process and Product Quality Management: A Review. *International Journal of Automation Technology, 8*(1), 4-19. doi:10.20965/ijat.2014.p0004

Tang, D. (2016). *Investigation of Line-Scan Dispersive Interferometry for In-Line Surface Metrology.* (Doctoral). University of Huddersfield, Retrieved from http://eprints.hud.ac.uk/id/eprint/29153/

Tanuska, P., Spendla, L., Kebisek, M., Duris, R., & Stremy, M. (2021). Smart Anomaly Detection and Prediction for Assembly Process Maintenance in Compliance with Industry 4.0. *Sensors, 21*(7), 2376. Retrieved from https://www.mdpi.com/1424-8220/21/7/2376

Tašner, T., Lovrec, D., Tašner, F., & Edler, J. (2012). Comparison of LabVIEW and MATLAB for Scientific Research. *Annals of the Faculty of Engineering Hunedoara-International Journal of Engineering, 10*(3).

Thames, L., & Schaefer, D. (2016). Software-defined Cloud Manufacturing for Industry 4.0. *Procedia CIRP, 52*, 12-17. doi:https://doi.org/10.1016/j.procir.2016.07.041

TinyXML. (2015). TinyXml Main Page. Retrieved from http://www.grinninglizard.com/tinyxml/

Tomczewski, S., Pakula, A., Van Erps, J., Thienpont, H., & Salbut, L. (2013). Low-coherence interferometry with polynomial interpolation on Compute Unified Device Architecture-enabled graphics processing units. *Optical Engineering, 52*(9), 094105-094105. doi:10.1117/1.OE.52.9.094105

Trojanowski, M., Kraszewski, M., Strakowski, M., & Pluciński, J. (2014). *Parallel multithread computing for spectroscopic analysis in optical coherence tomography*.

Vacharanukul, K., & Mekid, S. (2005). In-process dimensional inspection sensors. *Measurement, 38*(3), 204-218. doi:https://doi.org/10.1016/j.measurement.2005.07.009

Wally, B., Vyskocil, J., Novak, P., Huemer, C., Sindelar, R., Kadera, P., . . . Wimmer, M. (2019). *Production planning with IEC 62264 and PDDL.* Paper presented at the IEEE International Conference on Industrial Informatics (INDIN).

Wang, J., Zhang, W., Shi, Y., Duan, S., & Liu, J. (2018). Industrial big data analytics: challenges, methodologies, and applications. *arXiv preprint arXiv:1807.01016*.

Wang, L., Hofer, B., Guggenheim, J. A., & Považay, B. (2012). Graphics processing unit-based dispersion encoded full-range frequency-domain optical coherence tomography. *Journal of Biomedical Optics, 17*(7), 077007-077007. doi:10.1117/1.JBO.17.7.077007

Wang, Y., Oh, C. M., Oliveira, M. C., Islam, M. S., Ortega, A., & Park, B. H. (2012). GPU accelerated real-time multi-functional spectral-domain optical coherence tomography system at 1300nm. *Optics Express, 20*(14), 14797-14813. doi:10.1364/OE.20.014797

Wang, Z., & O'Boyle, M. F. P. (2009). Mapping parallelism to multi-cores: A machine learning based approach. *ACM SIGPLAN Notices, 44*(4), 75-84. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-70350608629&partnerID=40&md5=06ee7f8ff5f85e1168329699666099ac

Watanabe, Y., Maeno, S., Aoshima, K., Hasegawa, H., & Koseki, H. (2010). Real-time processing for full-range Fourier-domain optical-coherence tomography with zero-filling interpolation using multiple graphic processing units. *Applied optics, 49*(25), 4756-4762. doi:10.1364/AO.49.004756

Weill, R., Spur, G., & Eversheim, W. (1982). Survey of computer-aided process planning systems. *CIRP Annals, 31*(2), 539-551.

Whitehouse, D. J. (2011). *Handbook of surface and nanometrology* (2nd ed.). Boca Raton, FL: CRC Press.

Williamson, J. (2016). *Dispersed reference interferometry for on-machine metrology.* University of Huddersfield,

Williamson, J., Martin, H., & Jiang, X. (2016). High resolution position measurement from dispersed reference interferometry using template matching. *Optics Express, 24*(9), 10103-10114. doi:10.1364/OE.24.010103

Wolpert, D., Kempes, C., Stadler, P. F., & Grochow, J. A. (2019). *The energetics of computing in life and machines*: SFI Press.

Wolverton, R. W. (1974). The Cost of Developing Large-Scale Software. *IEEE Transactions on Computers, C-23*(6), 615-636. doi:10.1109/T-C.1974.224002

Xilinx. (2020). Xilinx - Adaptable. Intelligent. Retrieved from https://www.xilinx.com/

Xu, D., Huang, Y., & Kang, J. U. (2014a). GPU-accelerated non-uniform fast Fourier transform-based compressive sensing spectral domain optical coherence tomography. *Optics Express, 22*(12), 14871-14884. doi:10.1364/OE.22.014871

Xu, D., Huang, Y., & Kang, J. U. (2014b). Real-time compressive sensing spectral domain optical coherence tomography. *Optics Letters, 39*(1), 76-79. doi:10.1364/OL.39.000076

Yan, Y., Grossman, M., & Sarkar, V. (2009). *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA*.

Yasuno, Y., Madjarova, V. D., Makita, S., Akiba, M., Morosawa, A., Chong, C., . . . Yatagai, T. (2005). Three-dimensional and high-speed swept-source optical coherence tomography for in vivo investigation of human anterior eye segments. *Optics Express, 13*(26), 10652-10664. doi:10.1364/OPEX.13.010652

Zhang, Y., Zhang, G., Wang, J., Sun, S., Si, S., & Yang, T. (2015). Real-time information capturing and integration framework of the internet of manufacturing things. *International Journal of Computer Integrated Manufacturing, 28*(8), 811-822. doi:10.1080/0951192X.2014.900874

Zhong, H., Tang, J., & Zhang, S. (2015). Phase quality map based on local multi-unwrapped results for two-dimensional phase unwrapping. *Applied optics, 54*(4), 739-745. doi:10.1364/AO.54.000739

Zhou, T., Tang, D., Zhu, H., & Wang, L. (2020). Reinforcement Learning with Composite Rewards for Production Scheduling in a Smart Factory. *IEEE Access*.

11 Appendices

11.1 Sample of Function Files

11.1.1 getDerivative

```
<Function>
   <Ref>13</Ref>
   <Name>getDerivative</Name>
   <Serial>true</Serial>
   <Parallel>true</Parallel>
   <GPU>true</GPU>
   <Description>Calculate the derivative of a signal</Description>
      <Parameter>
         <ParameterName>Input</ParameterName>
         <Type>Type1</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
         <MemoryCopy>Input</MemoryCopy>
      </Parameter>
      <Parameter>
         <ParameterName>Output</ParameterName>
         <Type>Type2</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>false</ReadOnly>
         <MemoryCopy>Output</MemoryCopy>
      </Parameter>
      <Parameter>
         <ParameterName>Width</ParameterName>
         <Type>int</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
      </Parameter>
      <Parameter>
         <ParameterName>Height</ParameterName>
         <Type>int</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
      </Parameter>
      <RuntimeOptions>byte,byte,int,int</RuntimeOptions>
      <RuntimeOptions>byte,int,int,int</RuntimeOptions>
      <RuntimeOptions>byte,float,int,int</RuntimeOptions>
      <RuntimeOptions>byte,double,int,int</RuntimeOptions>
      <RuntimeOptions>int,int,int,int</RuntimeOptions>
      <RuntimeOptions>int,float,int,int</RuntimeOptions>
      <RuntimeOptions>int,double,int,int</RuntimeOptions>
      <RuntimeOptions>float,float,int,int</RuntimeOptions>
      <RuntimeOptions>float,double,int,int</RuntimeOptions>
      <RuntimeOptions>double,double,int,int</RuntimeOptions>
</Function>
```

11.1.2 Autoconvolution

```
<Function>
   <Ref>6</Ref>
   <Name>autoConvolution</Name>
   <Serial>true</Serial>
   <Parallel>true</Parallel>
   <GPU>true</GPU>
      <Parameter>
         <ParameterName>Input</ParameterName>
         <Type>Type1</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
         <MemoryCopy>Input</MemoryCopy>
      </Parameter>
      <Parameter>
         <ParameterName>Output</ParameterName>
         <Type>Type1</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>false</ReadOnly>
         <MemoryCopy>Output</MemoryCopy>
      </Parameter>
      <Parameter>
         <ParameterName>Width</ParameterName>
         <Type>int</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
      </Parameter>
      <Parameter>
         <ParameterName>Height</ParameterName>
         <Type>int</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
      </Parameter>
      <Parameter>
         <ParameterName>ResultID</ParameterName>
         <Type>Type1</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>false</ReadOnly>
         <MemoryCopy>Output</MemoryCopy>
      </Parameter>
      <Parameter>
         <ParameterName>Maximum</ParameterName>
         <Type>bool</Type>
         <Category>DataPoint</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
      </Parameter>
      <RuntimeOptions>int,int,int,int,int,bool</RuntimeOptions>
      <RuntimeOptions>float,float,int,int,float,bool</RuntimeOptions>
      <RuntimeOptions>double,double,int,int,double,bool</RuntimeOptions>
</Function>
```

11.1.3 CopyFromGPU

```
<Function>
   <Ref>25</Ref>
   <Name>copyFromGPU</Name>
   <Serial>false</Serial>
   <Parallel>false</Parallel>
   <GPU>true</GPU>
      <Parameter>
         <ParameterName>S</ParameterName>
         <Type>Signal</Type>
         <Category>Object</Category>
         <Description>No Description Available</Description>
         <ReadOnly>true</ReadOnly>
         <Hidden>true</Hidden>
         <HiddenParameter>F.Signals[0]</HiddenParameter>
      </Parameter>
      <Parameter>
         <ParameterName>CPU_Input</ParameterName>
         <Type>Type1</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>false</ReadOnly>
      </Parameter>
      <Parameter>
         <ParameterName>GPU_Output</ParameterName>
         <Type>Type1</Type>
         <Category>Signal</Category>
         <Description>No Description Available</Description>
         <ReadOnly>false</ReadOnly>
      </Parameter>
      <RuntimeOptions>int</RuntimeOptions>
      <RuntimeOptions>float</RuntimeOptions>
      <RuntimeOptions>double</RuntimeOptions>
</Function>
```

11.2 Configuration Files

11.2.1 DRI

11.2.1.1 CPU

```
<configurationProperties>
  <Signal>
    <ID>0</ID>
    <Name>Camera</Name>
    <Type>Byte</Type>
    <Length>16384</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>1</ID>
    <Name>Interferogram</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>2</ID>
    <Name>Derivative</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>3</ID>
    <Name>FilteredDeriv</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>4</ID>
    <Name>Autoconv</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <DataPoint>
    <ID>0</ID>
    <Name>SignalWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>8192</Value>
  </DataPoint>
  <DataPoint>
    <ID>1</ID>
    <Name>CameraWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>16384</Value>
  </DataPoint>
  <DataPoint>
    <ID>2</ID>
    <Name>SignalHeight</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>1</Value>
  </DataPoint>
```

```
<DataPoint>
    <ID>3</ID>
    <Name>Alignment</Name>
    <Type>Bool</Type>
    <ReadOnly>true</ReadOnly>
    <Value>true</Value>
</DataPoint>
  <DataPoint>
    <ID>4</ID>
    <Name>SmoothWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>101</Value>
  </DataPoint>
  <DataPoint>
    <ID>5</ID>
    <Name>SmoothDepth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>2</Value>
  </DataPoint>
  <DataPoint>
    <ID>6</ID>
    <Name>ACResult</Name>
    <Type>Float</Type>
    <ReadOnly>false</ReadOnly>
    <Value>0</Value>
  </DataPoint>
  <DataPoint>
    <ID>7</ID>
    <Name>SignalDepth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>1</Value>
  </DataPoint>
  <Function>
    <FunctionConfig>
      <Name>Basler_GigE</Name>
      <Ref>0</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>0</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals />
    <DataPoints>1,2,7</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>getData</Name>
      <Ref>2</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>1</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>0</Signals>
    <DataPoints>1,2</DataPoints>
  </Function>
```

```
   <Function>
    <FunctionConfig>
      <Name>convertBytes</Name>
      <Ref>8</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>2</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>0,1</Signals>
    <DataPoints>3,0,2</DataPoints>
  </Function>
  <Function>
  <Function>
    <FunctionConfig>
      <Name>getDerivative</Name>
      <Ref>13</Ref>
      <Description>Calculate the derivative
of a signal</Description>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>3</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>1,2</Signals>
    <DataPoints>0,2</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>filter</Name>
      <Ref>19</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>4</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>2,3</Signals>
    <DataPoints>4,5,0,2</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>autoConvolution</Name>
      <Ref>5</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>12</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>3,4</Signals>
    <DataPoints>0,2,6,3</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>outputArray</Name>
      <Ref>3</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>6</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>1</Signals>
    <DataPoints />
  </Function>
 <Function>
    <FunctionConfig>
      <Name>outputArray</Name>
      <Ref>3</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>7</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>4</Signals>
    <DataPoints />
  </Function>
</configurationProperties>
```

11.2.1.2 GPU

```
<configurationProperties>
  <Signal>
    <ID>0</ID>
    <Name>Camera</Name>
    <Type>Byte</Type>
    <Length>16384</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>1</ID>
    <Name>Interferogram</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>2</ID>
    <Name>Derivative</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>3</ID>
    <Name>FilteredDeriv</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <Signal>
    <ID>4</ID>
    <Name>Autoconv</Name>
    <Type>Float</Type>
    <Length>8192</Length>
    <Width>1</Width>
    <Depth>1</Depth>
  </Signal>
  <DataPoint>
    <ID>0</ID>
    <Name>SignalWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>8192</Value>
  </DataPoint>
  <DataPoint>
    <ID>1</ID>
    <Name>CameraWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>16384</Value>
  </DataPoint>
  <DataPoint>
    <ID>2</ID>
    <Name>SignalHeight</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>1</Value>
  </DataPoint>
  <DataPoint>
    <ID>3</ID>
    <Name>Alignment</Name>
    <Type>Bool</Type>
    <ReadOnly>true</ReadOnly>
    <Value>true</Value>
  </DataPoint>
 <DataPoint>
    <ID>4</ID>
    <Name>SmoothWidth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>101</Value>
  </DataPoint>
  <DataPoint>
    <ID>5</ID>
    <Name>SmoothDepth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>2</Value>
  </DataPoint>
  <DataPoint>
    <ID>6</ID>
    <Name>ACResult</Name>
    <Type>Float</Type>
    <ReadOnly>false</ReadOnly>
    <Value>0</Value>
  </DataPoint>
  <DataPoint>
    <ID>7</ID>
    <Name>SignalDepth</Name>
    <Type>Int</Type>
    <ReadOnly>true</ReadOnly>
    <Value>1</Value>
  </DataPoint>
  <Function>
    <FunctionConfig>
      <Name>Basler_GigE</Name>
      <Ref>0</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>0</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals />
    <DataPoints>1,2,7</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>getData</Name>
      <Ref>2</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>1</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>0</Signals>
    <DataPoints>1,2</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
      <Name>Interferogram_copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>2</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>0</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>convertBytes</Name>
      <Ref>8</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>3</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>0,1</Signals>
    <DataPoints>3,0,2</DataPoints>
  </Function>
  <Function>
    <FunctionConfig>
     <Name>Interferogram_copyFromGPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>4</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>1</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>Interferogram_copyTomGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>5</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>1</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>getDerivative</Name>
      <Ref>13</Ref>
      <Description>Calculate the derivative
of a signal</Description>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>6</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>1,2</Signals>
    <DataPoints>0,2</DataPoints>
  </Function>
 <Function>
    <FunctionConfig>
      <Name>Derivative_copyFromGPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>7</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>2</Signals>
    <DataPoints />
  </Function>
```

```
<Function>
  <FunctionConfig>
    <Name>Derivative_copyToGPU</Name>
    <Ref>24</Ref>
    <GPU>true</GPU>
    <Hidden>false</Hidden>
  </FunctionConfig>
  <ID>8</ID>
  <Type>0</Type>
  <Platform>2</Platform>
  <Thread>0</Thread>
  <Signals>2</Signals>
  <DataPoints />
</Function>
<Function>
  <FunctionConfig>
    <Name>smoothData</Name>
    <Ref>19</Ref>
    <GPU>true</GPU>
    <Hidden>false</Hidden>
  </FunctionConfig>
  <ID>9</ID>
  <Type>0</Type>
  <Platform>2</Platform>
  <Thread>0</Thread>
  <Signals>2,3</Signals>
  <DataPoints>4,5,0,2</DataPoints>
</Function>
<Function>
  <FunctionConfig>
   <Name>FilteredDeriv_copyFromGPU</Name>
    <Ref>25</Ref>
    <GPU>true</GPU>
    <Hidden>false</Hidden>
  </FunctionConfig>
  <ID>10</ID>
  <Type>0</Type>
  <Platform>2</Platform>
  <Thread>0</Thread>
  <Signals>3</Signals>
  <DataPoints />
</Function>
<Function>
  <FunctionConfig>
    <Name>FilteredDeriv_copyToGPU</Name>
    <Ref>24</Ref>
    <GPU>true</GPU>
    <Hidden>false</Hidden>
  </FunctionConfig>
  <ID>11</ID>
  <Type>0</Type>
  <Platform>2</Platform>
  <Thread>0</Thread>
  <Signals>3</Signals>
  <DataPoints />
</Function>
<Function>
  <FunctionConfig>
    <Name>autoConvolution</Name>
    <Ref>6</Ref>
    <GPU>true</GPU>
    <Hidden>false</Hidden>
  </FunctionConfig>
  <ID>12</ID>
  <Type>0</Type>
  <Platform>2</Platform>
  <Thread>0</Thread>
  <Signals>3,4</Signals>
  <DataPoints>0,2,6,3</DataPoints>
</Function>
```

```
 <Function>
    <FunctionConfig>
<Name>autoConvolution_copyFromGPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>13</ID>
    <Type>0</Type>
    <Platform>2</Platform>
    <Thread>0</Thread>
    <Signals>4</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>outputArray</Name>
      <Ref>3</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>14</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>1</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>outputArray</Name>
      <Ref>3</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>15</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals>4</Signals>
    <DataPoints />
  </Function>
  <Function>
    <FunctionConfig>
      <Name>sendData</Name>
      <Ref>5</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
    </FunctionConfig>
    <ID>16</ID>
    <Type>0</Type>
    <Platform>0</Platform>
    <Thread>0</Thread>
    <Signals />
    <DataPoints />
  </Function>
</configurationProperties>
```

11.2.2 LSDI

11.2.2.1 CPU

```
<configurationProperties>
<Signal>
   <ID>0</ID>
   <Name>Lambda</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>1</ID>
   <Name>interSampleX</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>2</ID>
   <Name>Background</Name>
   <Type>byte</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>3</ID>
   <Name>Background_f</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>4</ID>
   <Name>interSampleX_c</Name>
   <Type>float</Type>
   <Length>1326</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>5</ID>
   <Name>Signal</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>6</ID>
   <Name>sampleYt1</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>7</ID>
   <Name>interSampleYt1</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>8</ID>
   <Name>fftResult</Name>
   <Type>Complex</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>9</ID>
   <Name>fftResult_Real</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>10</ID>
   <Name>MaxID</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>11</ID>
   <Name>MaxValue</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>12</ID>
   <Name>LowCutFreq</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>13</ID>
   <Name>HighCutFreq</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>14</ID>
   <Name>ifftResult</Name>
   <Type>Complex</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>15</ID>
   <Name>Phase</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>16</ID>
   <Name>unwrapedPhase</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
```

```
<Signal>
      <ID>17</ID>
      <Name>phase_final_c</Name>
      <Type>float</Type>
      <Length>1326</Length>
      <Width>480</Width>
      <Depth>1</Depth>
</Signal>
<Signal>
   <ID>18</ID>
   <Name>Polynomial</Name>
   <Type>float</Type>
   <Length>2</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>19</ID>
   <Name>getHeight</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>20</ID>
   <Name>getHeight2</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<DataPoint>
   <ID>0</ID>
   <Name>Lambda_A</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>-0.0000032722</Value>
</DataPoint>
<DataPoint>
   <ID>1</ID>
   <Name>Lambda_B</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.083694</Value>
</DataPoint>
<DataPoint>
   <ID>2</ID>
   <Name>Lambda_C</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>604.99</Value>
</DataPoint>
<DataPoint>
   <ID>3</ID>
   <Name>Width</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>640</Value>
</DataPoint>
<DataPoint>
   <ID>4</ID>
   <Name>SamplingPoints</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1424</Value>
</DataPoint>
<DataPoint>
   <ID>5</ID>
   <Name>Height</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>480</Value>
</DataPoint>
<DataPoint>
   <ID>6</ID>
   <Name>pointSpacing</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.007</Value>
</DataPoint>
<DataPoint>
   <ID>7</ID>
   <Name>lCutOff</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.3</Value>
</DataPoint>
<DataPoint>
   <ID>8</ID>
   <Name>hCutOff</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0</Value>
</DataPoint>
<DataPoint>
   <ID>9</ID>
   <Name>window_Start</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>49</Value>
</DataPoint>
<DataPoint>
   <ID>10</ID>
   <Name>SamplingPoints2</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1326</Value>
</DataPoint>
<DataPoint>
   <ID>11</ID>
   <Name>Area</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>307200</Value>
</DataPoint>
<DataPoint>
   <ID>12</ID>
   <Name>FFT_Forward</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<DataPoint>
   <ID>13</ID>
   <Name>HalfSamplingPoints</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>712</Value>
</DataPoint>
<DataPoint>
   <ID>14</ID>
   <Name>zero_Start</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0</Value>
</DataPoint>
<DataPoint>
   <ID>15</ID>
   <Name>zero_End</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2</Value>
</DataPoint>
<DataPoint>
   <ID>16</ID>
   <Name>FFT_Backward</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>-1</Value>
</DataPoint>
```

```
<DataPoint>
   <ID>17</ID>
   <Name>polynomialOrder</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2</Value>
</DataPoint>
<DataPoint>
   <ID>18</ID>
   <Name>SamplingCenter</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>711</Value>
</DataPoint>
<DataPoint>
   <ID>19</ID>
   <Name>scaleChange</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1000</Value>
</DataPoint>
<DataPoint>
   <ID>20</ID>
   <Name>saveFolder</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>D:\Platform\</Value>
</DataPoint>
<DataPoint>
   <ID>21</ID>
   <Name>flipBMP</Name>
   <Type>bool</Type>
   <ReadOnly>true</ReadOnly>
   <Value>true</Value>
</DataPoint>
<DataPoint>
   <ID>22</ID>
   <Name>FileName1</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>getHeight1</Value>
</DataPoint>
<DataPoint>
   <ID>23</ID>
   <Name>FileName2</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>getHeight2</Value>
</DataPoint>
   DataPoint>
   <ID>24</ID>
   <Name>importBackground</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>background.bmp</Value>
</DataPoint>
<DataPoint>
   <ID>25</ID>
   <Name>importSignal</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>measurement47nm.bmp</Value>
</DataPoint>
<DataPoint>
   <ID>26</ID>
   <Name>singleHeight</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<DataPoint>
   <ID>27</ID>
   <Name>cameraOffset</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2596</Value>
</DataPoint>
<DataPoint>
   <ID>28</ID>
   <Name>cameraRowOffset</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>8</Value>
</DataPoint>
<DataPoint>
   <ID>29</ID>
   <Name>cameraFileLocation</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>D:\\FringeOrder\\LSDI.cxf</Value>
</DataPoint>
<DataPoint>
   <ID>30</ID>
   <Name>cameraChannel</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<Function>
   <FunctionConfig>
      <Name>importBMP</Name>
      <Ref>45</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>0</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>2</Signals>
   <DataPoints>20,24</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>createLamda</Name>
      <Ref>48</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>1</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>0</Signals>
   <DataPoints>0,1,2,3</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>linspace</Name>
      <Ref>49</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>2</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,0</Signals>
   <DataPoints>3,4</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>GaussianFilter</Name>
      <Ref>50</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>3</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>2,3</Signals>
   <DataPoints>3,5,6,7,8</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>windowSignal</Name>
      <Ref>38</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>4</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,4</Signals>
   <DataPoints>9,10,4,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>startCamera</Name>
      <Ref>52</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>6</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints>29,30</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>getSignal</Name>
      <Ref>53</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>5</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints>5,3,27,28,14</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>divide</Name>
      <Ref>30</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>7</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5,3,6</Signals>
   <DataPoints>3,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Interpolation</Name>
      <Ref>44</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>8</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>0,6,1,7</Signals>
   <DataPoints>3,5,4</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>FFT</Name>
      <Ref>41</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>9</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>7,8</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide2</Name>
      <Ref>32</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>10</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,8</Signals>
   <DataPoints>4,3,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Absolute</Name>
      <Ref>26</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>11</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,9</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Zero_Start</Name>
      <Ref>39</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>12</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints>14,15,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Zero_End</Name>
      <Ref>39</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>13</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints>13,4,4,5</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>getMax</Name>
      <Ref>34</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>14</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9,10,11</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Logic</Name>
      <Ref>43</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>15</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,12,13</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Subtract</Name>
      <Ref>35</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>16</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,12,12</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Add</Name>
      <Ref>36</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>17</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,13,13</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>zeroComplex_Start</Name>
      <Ref>40</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>18</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,12</Signals>
   <DataPoints>14,4,5</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>zeroComplex_End</Name>
      <Ref>40</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>19</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,13</Signals>
   <DataPoints>4,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>IFFT</Name>
      <Ref>42</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>20</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,14</Signals>
   <DataPoints>4,5,16</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>divide3</Name>
      <Ref>32</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>21</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,14</Signals>
   <DataPoints>4,3,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>complexLog</Name>
      <Ref>29</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>22</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,14</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>getImag</Name>
      <Ref>33</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>23</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,15</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>unWrapPhase</Name>
      <Ref>37</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>24</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>15,16</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>windowSignal</Name>
      <Ref>38</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>25</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>16,17</Signals>
   <DataPoints>9,10,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>polyFit</Name>
      <Ref>51</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>26</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>4,17,18</Signals>
   <DataPoints>17,10,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>calcHeight1</Name>
      <Ref>27</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>27</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>18,19</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>calcHeight2</Name>
      <Ref>28</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>28</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,15,19,20</Signals>
   <DataPoints>18,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide4</Name>
      <Ref>31</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>29</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19,19</Signals>
   <DataPoints>19,5,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide5</Name>
      <Ref>31</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>30</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20,20</Signals>
   <DataPoints>19,5,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>saveCSV</Name>
      <Ref>47</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>31</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints>20,22,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>saveCSV</Name>
      <Ref>47</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>32</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20</Signals>
   <DataPoints>20,23,5</DataPoints>
</Function>
</configurationProperties>
```

11.2.2.2 GPU

```
<Signal>
   <ID>0</ID>
   <Name>Lambda</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>1</ID>
   <Name>interSampleX</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>2</ID>
   <Name>Background</Name>
   <Type>byte</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>3</ID>
   <Name>Background_f</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>4</ID>
   <Name>interSampleX_c</Name>
   <Type>float</Type>
   <Length>1326</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>5</ID>
   <Name>Signal</Name>
   <Type>byte</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>6</ID>
   <Name>sampleYt1</Name>
   <Type>float</Type>
   <Length>640</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>7</ID>
   <Name>interSampleYt1</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>8</ID>
   <Name>fftResult</Name>
   <Type>Complex</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>9</ID>
   <Name>fftResult_Real</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>10</ID>
   <Name>MaxID</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>11</ID>
   <Name>MaxValue</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>12</ID>
   <Name>LowCutFreq</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>13</ID>
   <Name>HighCutFreq</Name>
   <Type>int</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>14</ID>
   <Name>ifftResult</Name>
   <Type>Complex</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>15</ID>
   <Name>Phase</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>16</ID>
   <Name>unwrapedPhase</Name>
   <Type>float</Type>
   <Length>1424</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>17</ID>
   <Name>phase_final_c</Name>
   <Type>float</Type>
   <Length>1326</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
```

```
<Signal>
   <ID>18</ID>
   <Name>Polynomial</Name>
   <Type>float</Type>
   <Length>2</Length>
   <Width>480</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>19</ID>
   <Name>getHeight</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<Signal>
   <ID>20</ID>
   <Name>getHeight2</Name>
   <Type>float</Type>
   <Length>480</Length>
   <Width>1</Width>
   <Depth>1</Depth>
</Signal>
<DataPoint>
   <ID>0</ID>
   <Name>Lambda_A</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>-0.0000032722</Value>
</DataPoint>
<DataPoint>
   <ID>1</ID>
   <Name>Lambda_B</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.083694</Value>
</DataPoint>
<DataPoint>
   <ID>2</ID>
   <Name>Lambda_C</Name>
   <Type>Double</Type>
   <ReadOnly>true</ReadOnly>
   <Value>604.99</Value>
</DataPoint>
<DataPoint>
   <ID>3</ID>
   <Name>Width</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>640</Value>
</DataPoint>
<DataPoint>
   <ID>4</ID>
   <Name>SamplingPoints</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1424</Value>
</DataPoint>
<DataPoint>
   <ID>5</ID>
   <Name>Height</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>480</Value>
</DataPoint>
<DataPoint>
   <ID>6</ID>
   <Name>pointSpacing</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.007</Value>
</DataPoint>
<DataPoint>
   <ID>7</ID>
   <Name>lCutOff</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0.3</Value>
</DataPoint>
<DataPoint>
   <ID>8</ID>
   <Name>hCutOff</Name>
   <Type>float</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0</Value>
</DataPoint>
<DataPoint>
   <ID>9</ID>
   <Name>window_Start</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>49</Value>
</DataPoint>
<DataPoint>
   <ID>10</ID>
   <Name>SamplingPoints2</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1326</Value>
</DataPoint>
<DataPoint>
   <ID>11</ID>
   <Name>Area</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>307200</Value>
</DataPoint>
<DataPoint>
   <ID>12</ID>
   <Name>FFT_Forward</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<DataPoint>
   <ID>13</ID>
   <Name>HalfSamplingPoints</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>712</Value>
</DataPoint>
<DataPoint>
   <ID>14</ID>
   <Name>zero_Start</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>0</Value>
</DataPoint>
<DataPoint>
   <ID>15</ID>
   <Name>zero_End</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2</Value>
</DataPoint>
<DataPoint>
   <ID>16</ID>
   <Name>FFT_Backward</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>-1</Value>
</DataPoint>
```

```xml
<DataPoint>
   <ID>17</ID>
   <Name>polynomialOrder</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2</Value>
</DataPoint>
<DataPoint>
   <ID>18</ID>
   <Name>SamplingCenter</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>711</Value>
</DataPoint>
<DataPoint>
   <ID>19</ID>
   <Name>scaleChange</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1000</Value>
</DataPoint>
<DataPoint>
   <ID>20</ID>
   <Name>saveFolder</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>E:\Platform\</Value>
</DataPoint>
<DataPoint>
   <ID>21</ID>
   <Name>flipBMP</Name>
   <Type>bool</Type>
   <ReadOnly>true</ReadOnly>
   <Value>true</Value>
</DataPoint>
<DataPoint>
   <ID>22</ID>
   <Name>FileName1</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>getHeight1</Value>
</DataPoint>
<DataPoint>
   <ID>23</ID>
   <Name>FileName2</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>getHeight2</Value>
</DataPoint>
<DataPoint>
   <ID>24</ID>
   <Name>importBackground</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>background.bmp</Value>
</DataPoint>
<DataPoint>
   <ID>25</ID>
   <Name>importSignal</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>measurement47nm.bmp</Value>
</DataPoint>
<DataPoint>
   <ID>26</ID>
   <Name>singleHeight</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<DataPoint>
   <ID>27</ID>
   <Name>cameraOffset</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>2596</Value>
</DataPoint>
<DataPoint>
   <ID>28</ID>
   <Name>cameraRowOffset</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>8</Value>
</DataPoint>
<DataPoint>
   <ID>29</ID>
   <Name>cameraFileLocation</Name>
   <Type>string</Type>
   <ReadOnly>true</ReadOnly>
   <Value>D:\\FringeOrder\\LSDI.cxf</Value>
</DataPoint>
<DataPoint>
   <ID>30</ID>
   <Name>cameraChannel</Name>
   <Type>int</Type>
   <ReadOnly>true</ReadOnly>
   <Value>1</Value>
</DataPoint>
<Function>
   <FunctionConfig>
      <Name>importBMP</Name>
      <Ref>45</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>0</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>2</Signals>
   <DataPoints>20,24</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>createLamda</Name>
      <Ref>48</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>1</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>0</Signals>
   <DataPoints>0,1,2,3</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>linspace</Name>
      <Ref>49</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>2</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,0</Signals>
   <DataPoints>3,4</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>GaussianFilter</Name>
      <Ref>50</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>3</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>2,3</Signals>
   <DataPoints>3,5,6,7,8</DataPoints>
```

```
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>4</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>3</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>5</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>windowSignal</Name>
      <Ref>38</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>6</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,4</Signals>
   <DataPoints>9,10,4,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>7</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>4</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>startCamera</Name>
      <Ref>52</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>8</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints>29,30</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>importBMP</Name>
      <Ref>45</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>0</ID>
   <Type>1</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints>20,25</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>10</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>divide</Name>
      <Ref>30</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>11</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5,3,6</Signals>
   <DataPoints>3,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>12</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>6</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Interpolation</Name>
      <Ref>44</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>13</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>0,6,1,7</Signals>
   <DataPoints>3,5,4</DataPoints>
</Function>
```

```xml
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>14</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>7</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>FFT</Name>
      <Ref>41</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>15</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>7,8</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>16</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>17</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide2</Name>
      <Ref>32</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>18</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,8</Signals>
   <DataPoints>4,3,5</DataPoints>
</Function>
```

```xml
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>19</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>20</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Absolute</Name>
      <Ref>26</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>21</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,9</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>22</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>23</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>Zero_Start</Name>
      <Ref>39</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>24</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints>14,15,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>25</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>26</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Zero_End</Name>
      <Ref>39</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>27</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints>13,4,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>28</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>29</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>getMax</Name>
      <Ref>34</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>30</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>9,10,11</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>31</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>32</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>11</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>33</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>Logic</Name>
      <Ref>43</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>34</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,12,13</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>35</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>12</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>36</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>13</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>37</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>38</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>12</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Subtract</Name>
      <Ref>35</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>39</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,12,12</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>40</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>12</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>41</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>42</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>13</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Add</Name>
      <Ref>36</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>43</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>10,13,13</Signals>
   <DataPoints>5</DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>44</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>13</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>45</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>46</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>12</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>zeroComplex_Start</Name>
      <Ref>40</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>47</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,12</Signals>
   <DataPoints>14,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>48</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>49</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>50</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>13</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>zeroComplex_End</Name>
      <Ref>40</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>51</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,13</Signals>
   <DataPoints>4,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>52</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>53</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>IFFT</Name>
      <Ref>42</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>54</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>8,14</Signals>
   <DataPoints>4,5,16</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>55</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>56</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>divide3</Name>
      <Ref>32</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>57</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,14</Signals>
   <DataPoints>4,3,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>58</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>59</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>complexLog</Name>
      <Ref>29</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>60</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,14</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToCPU</Name>
      <Ref>55</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>61</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyComplexToGPU</Name>
      <Ref>54</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>62</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>getImag</Name>
      <Ref>33</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>63</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>14,15</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
```

```xml
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>64</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>15</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>65</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>15</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>unWrapPhase</Name>
      <Ref>37</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>66</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>15,16</Signals>
   <DataPoints>4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>67</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>16</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>68</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>16</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>windowSignal</Name>
      <Ref>38</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>69</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>16,17</Signals>
   <DataPoints>9,10,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>70</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>17</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>71</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>17</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>polyFit</Name>
      <Ref>51</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>72</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>4,17,18</Signals>
   <DataPoints>17,10,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>73</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>18</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>74</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>18</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>calcHeight1</Name>
      <Ref>27</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>75</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>18,19</Signals>
   <DataPoints>5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>76</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>77</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>15</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>78</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>calcHeight2</Name>
      <Ref>28</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>79</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>1,15,19,20</Signals>
   <DataPoints>18,4,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>80</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>81</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide4</Name>
      <Ref>31</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>82</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19,19</Signals>
   <DataPoints>19,5,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>83</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints></DataPoints>
</Function>
```

```
<Function>
   <FunctionConfig>
      <Name>copyToGPU</Name>
      <Ref>24</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>84</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>Divide5</Name>
      <Ref>31</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>85</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20,20</Signals>
   <DataPoints>19,5,26</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>copyToCPU</Name>
      <Ref>25</Ref>
      <GPU>true</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>86</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20</Signals>
   <DataPoints></DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>saveBMP</Name>
      <Ref>46</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>87</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>5</Signals>
   <DataPoints>20,3,5,21</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>saveCSV</Name>
      <Ref>47</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>88</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>19</Signals>
   <DataPoints>20,22,5</DataPoints>
</Function>
<Function>
   <FunctionConfig>
      <Name>saveCSV</Name>
      <Ref>47</Ref>
      <GPU>false</GPU>
      <Hidden>false</Hidden>
   </FunctionConfig>
   <ID>89</ID>
   <Type>0</Type>
   <Platform>0</Platform>
   <Thread>0</Thread>
   <Signals>20</Signals>
   <DataPoints>20,23,5</DataPoints>
</Function>

</configurationProperties>
```

11.3 DRI PDDL Files

11.3.1 Sample Domain File

```
(define (domain DRI)
    (:requirements :strips :fluents)
    (:functions
        (total-cost)
    )
    (:predicates
        ;Variables
        (Camera ?sig)           ;0
        (Interferogram ?sig)    ;1
        (Derivative ?sig)       ;2
        (FilteredDeriv ?sig)    ;3
        (Autoconv ?sig)         ;4

        ;Function completion
        (done-getSignal ?sig)
        (done-convertBytes ?sig)
        (done-getDerivative ?sig)
        (done-filterDerivative ?sig)
        (done-autoConvolution ?sig)
        (done-outputData1 ?sig)
        (done-outputData2 ?sig)

        ;Misc Parameters
        (CPU ?dev)
        (GPU ?dev)
        (data-onHost ?sig)
        (data-onDevice ?sig)

        (device-inUse ?dev)
    )

    ;##########################Memory Copy##################################

    (:action Camera_ToGPU
        :parameters(?cpu ?gpu ?sig0)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            (GPU ?gpu)
            ;Check Signals
            (Camera ?sig0)
            ;Check Memory Location
            (data-onHost ?sig0)
        )
        :effect(and
            (data-onDevice ?sig0)
            (increase (total-cost) 151)
        )
    )
     (:action Interferogram_ToGPU
        :parameters(?cpu ?gpu ?sig1)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            (GPU ?gpu)
            ;Check Signals
            (Interferogram ?sig1)
            ;Check Memory Location
            (data-onHost ?sig1)
        )
        :effect(and
            (data-onDevice ?sig1)
            (increase (total-cost) 95)
        )
    )
```

```
(:action Derivative_ToGPU
    :parameters(?cpu ?gpu ?sig2)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Derivative ?sig2)
        ;Check Memory Location
        (data-onHost ?sig2)
    )
    :effect(and
        (data-onDevice ?sig2)
        (increase (total-cost) 93)
    )
)

(:action FilteredDeriv_ToGPU
    :parameters(?cpu ?gpu ?sig3)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (FilteredDeriv ?sig3)
        ;Check Memory Location
        (data-onHost ?sig3)
    )
    :effect(and
        (data-onDevice ?sig3)
        (increase (total-cost) 92)
    )
)
 (:action Interferogram_ToCPU
    :parameters(?cpu ?gpu ?sig1)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Interferogram ?sig1)
        ;Check Memory Location
        (data-onDevice ?sig1)
    )
    :effect(and
        (data-onHost ?sig1)
        (increase (total-cost) 125)
    )
)

(:action Derivative_ToCPU
    :parameters(?cpu ?gpu ?sig2)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Derivative ?sig2)
        ;Check Memory Location
        (data-onDevice ?sig2)
    )
    :effect(and
        (data-onHost ?sig2)
        (increase (total-cost) 121)
    )
)
```

```
(:action FilteredDeriv_ToCPU
        :parameters(?cpu ?gpu ?sig3)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            (GPU ?gpu)
            ;Check Signals
            (FilteredDeriv ?sig3)
            ;Check Memory Location
            (data-onDevice ?sig3)

        )
        :effect(and
            (data-onHost ?sig3)
            (increase (total-cost) 134)
        )
    )
     (:action Autoconv_ToCPU
        :parameters(?cpu ?gpu ?sig4)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            (GPU ?gpu)
            ;Check Signals
            (Autoconv ?sig4)
            ;Check Memory Location
            (data-onDevice ?sig4)
        )
        :effect(and
            (data-onHost ?sig4)
            (increase (total-cost) 127)
        )
    )

    ;################## CPU Functions #############################

    (:action getSignal
        :parameters(?cpu ?sig0)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (Camera ?sig0)
            ;Check Function Status
            (not(done-getSignal ?sig0))
        )
        :effect(and
            (data-onHost ?sig0)
            (not(data-onDevice ?sig0))
            (done-getSignal ?sig0)
            (increase (total-cost) 4152)
        )
    )
 (:action convertBytes
        :parameters(?cpu ?sig0 ?sig1)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (Camera ?sig0)
            (Interferogram ?sig1)
            ;Check Function Status
            (done-getSignal ?sig0) (not(done-convertBytes ?sig1))
            ;Check Memory Location
            (data-onHost ?sig0)
        )
        :effect(and
            (data-onHost ?sig1)
            (not(data-onDevice ?sig1))
            (done-convertBytes ?sig1)
            (increase (total-cost) 16)
        )
    )
 (:action getDerivative
    :parameters(?cpu ?sig1 ?sig2)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (Interferogram ?sig1)
        (Derivative ?sig2)
        ;Check Function Status
        (done-convertBytes ?sig1) (not(done-getDerivative ?sig2))
        ;Check Memory Location
        (data-onHost ?sig1)
    )
    :effect(and
        (data-onHost ?sig2)
        (not(data-onDevice ?sig2))
        (done-getDerivative ?sig2)
        (increase (total-cost) 8)
    )
)
 (:action filterDerivative
    :parameters(?cpu ?sig2 ?sig3)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (Derivative ?sig2)
        (FilteredDeriv ?sig3)
        ;Check Function Status
        (done-getDerivative ?sig2) (not(done-filterDerivative ?sig3))
        ;Check Memory Location
        (data-onHost ?sig2)
    )
    :effect(and
        (data-onHost ?sig3)
        (not(data-onDevice ?sig3))
        (done-filterDerivative ?sig3)
        (increase (total-cost) 322)
    )
)

(:action autoConvolution
    :parameters(?cpu ?sig3 ?sig4)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (FilteredDeriv ?sig3)
        (Autoconv ?sig4)
        ;Check Function Status
        (done-filterDerivative ?sig3) (not(done-autoConvolution ?sig4))
        ;Check Memory Location
        (data-onHost ?sig3)
    )
    :effect(and
        (data-onHost ?sig4)
        (not(data-onDevice ?sig4))
        (done-autoConvolution ?sig4)
        (increase (total-cost) 878)
    )
)
```

```
 (:action outputData1
    :parameters(?cpu ?sig1)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (Interferogram ?sig1)
        ;Check Function Status
        (done-convertBytes ?sig1) (not(done-outputData1 ?sig1))
        ;Check Memory Location
        (data-onHost ?sig1)
    )
    :effect(and
        (done-outputData1 ?sig1)
        (increase (total-cost) 8)
    )
)

(:action outputData2
    :parameters(?cpu ?sig4)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (Autoconv ?sig4)
        ;Check Function Status
        (done-autoConvolution ?sig4) (not(done-outputData2 ?sig4))
        ;Check Memory Location
        (data-onHost ?sig4)
    )
    :effect(and
        (done-outputData2 ?sig4)
        (increase (total-cost) 6)
    )
)
;################## KERNEL Functions #############################

(:action convertBytes_Kernel
    :parameters(?gpu ?sig0 ?sig1)
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (Camera ?sig0)
        (Interferogram ?sig1)
        ;Check Function Status
        (done-getSignal ?sig0) (not(done-convertBytes ?sig1))
        ;Check Memory Location
        (data-onDevice ?sig0)
    )
    :effect(and
        (data-onDevice ?sig1)
        (not(data-onHost ?sig1))
        (done-convertBytes ?sig1)
        (increase (total-cost) 117)
    )
)
```

```
   (:action getDerivative_Kernel
        :parameters(?gpu ?sig1 ?sig2)
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (Interferogram ?sig1)
            (Derivative ?sig2)
            ;Check Function Status
            (done-convertBytes ?sig1) (not(done-getDerivative ?sig2))
            ;Check Memory Location
            (data-onDevice ?sig1)
        )
        :effect(and
            (data-onDevice ?sig2)
            (not(data-onHost ?sig2))
            (done-getDerivative ?sig2)
            (increase (total-cost) 113)
        )
    )
    (:action filterDerivative_Kernel
        :parameters(?gpu ?sig2 ?sig3)
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (Derivative ?sig2)
            (FilteredDeriv ?sig3)
            ;Check Function Status
            (done-getDerivative ?sig2) (not(done-filterDerivative ?sig3))
            ;Check Memory Location
            (data-onDevice ?sig2)
        )
        :effect(and
            (data-onDevice ?sig3)
            (not(data-onHost ?sig3))
            (done-filterDerivative ?sig3)
            (increase (total-cost) 639)
        )
    )

    (:action autoConvolution_Kernel
        :parameters(?gpu ?sig3 ?sig4)
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (FilteredDeriv ?sig3)
            (Autoconv ?sig4)
            ;Check Function Status
            (done-filterDerivative ?sig3) (not(done-autoConvolution ?sig4))
            ;Check Memory Location
            (data-onDevice ?sig3)
        )
        :effect(and
            (data-onDevice ?sig4)
            (not(data-onHost ?sig4))
            (done-autoConvolution ?sig4)
            (increase (total-cost) 1168)
        )
    )
)
```

11.3.2 Problem File

```
(define (problem DRI-p1)
        (:domain DRI)
        (:objects _Camera
                      _Interferogram
                      _Derivative
                      _FilteredDeriv
                      _Autoconv
                      _CPU
                      _GPU)
        (:init
        ;Init Signals
        (Camera _Camera)
        (Interferogram _Interferogrm
        (Derivative _Derivative)
        (FilteredDeriv _FilteredDeriv)
        (Autoconv _Autoconv)

        (CPU _CPU)
        (GPU _GPU)

        ;Init Durative Paramters
        (= (total-cost) 0))

        (:goal (and (done-outputData1 _Interferogram)
                          (done-outputData2 _Autoconv)))
        (:metric minimize (total-cost))
)
```

11.3.3 Solution Files

11.3.3.1 CPU

```
; Version LPG-td-1.4
; Seed 115643919
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p CPU/ -out CPU
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 1307.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CONVERTBYTES CPU CAMERA INTERFEROGRAM) [1]
2:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
2:   (GETDERIVATIVE CPU INTERFEROGRAM DERIVATIVE) [1]
3:   (FILTERDERIVATIVE CPU DERIVATIVE FILTEREDDERIV) [1]
4:   (AUTOCONVOLUTION CPU FILTEREDDERIV AUTOCONV) [1]
5:   (OUTPUTDATA2 CPU AUTOCONV) [1]
```

11.3.3.2 GTX 520

```
; Version LPG-td-1.4
; Seed 45288303
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 520/ -out 520
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.00
; Parsing time 0.01
; Mutex time 0.00
; MetricValue 9973.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CAMERA_TOGPU CPU GPU CAMERA) [1]
2:   (CONVERTBYTES_KERNEL GPU CAMERA INTERFEROGRAM) [1]
3:   (INTERFEROGRAM_TOCPU CPU GPU INTERFEROGRAM) [1]
3:   (GETDERIVATIVE_KERNEL GPU INTERFEROGRAM DERIVATIVE) [1]
4:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
4:   (FILTERDERIVATIVE_KERNEL GPU DERIVATIVE FILTEREDDERIV) [1]
5:   (AUTOCONVOLUTION_KERNEL GPU FILTEREDDERIV AUTOCONV) [1]
6:   (AUTOCONV_TOCPU CPU GPU AUTOCONV) [1]
7:   (OUTPUTDATA2 CPU AUTOCONV) [1]
```

11.3.3.3 GTX 650Ti

```
; Version LPG-td-1.4
; Seed 32093033
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 650/ -out 650
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 2367.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CAMERA_TOGPU CPU GPU CAMERA) [1]
2:   (CONVERTBYTES_KERNEL GPU CAMERA INTERFEROGRAM) [1]
3:   (INTERFEROGRAM_TOCPU CPU GPU INTERFEROGRAM) [1]
3:   (GETDERIVATIVE_KERNEL GPU INTERFEROGRAM DERIVATIVE) [1]
4:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
4:   (FILTERDERIVATIVE_KERNEL GPU DERIVATIVE FILTEREDDERIV) [1]
5:   (AUTOCONVOLUTION_KERNEL GPU FILTEREDDERIV AUTOCONV) [1]
6:   (AUTOCONV_TOCPU CPU GPU AUTOCONV) [1]
7:   (OUTPUTDATA2 CPU AUTOCONV) [1]
```

11.3.3.4 GTX 780Ti

```
; Version LPG-td-1.4
; Seed 19134684
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 780/ -out 780
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 6564.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CAMERA_TOGPU CPU GPU CAMERA) [1]
2:   (CONVERTBYTES_KERNEL GPU CAMERA INTERFEROGRAM) [1]
3:   (INTERFEROGRAM_TOCPU CPU GPU INTERFEROGRAM) [1]
3:   (GETDERIVATIVE_KERNEL GPU INTERFEROGRAM DERIVATIVE) [1]
4:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
4:   (FILTERDERIVATIVE_KERNEL GPU DERIVATIVE FILTEREDDERIV) [1]
5:   (AUTOCONVOLUTION_KERNEL GPU FILTEREDDERIV AUTOCONV) [1]
6:   (AUTOCONV_TOCPU CPU GPU AUTOCONV) [1]
7:   (OUTPUTDATA2 CPU AUTOCONV) [1]
```

11.3.3.5 GTX 1070

```
; Version LPG-td-1.4
; Seed 117664417
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 1070/ -out 1070
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 2663.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CAMERA_TOGPU CPU GPU CAMERA) [1]
2:   (CONVERTBYTES_KERNEL GPU CAMERA INTERFEROGRAM) [1]
3:   (INTERFEROGRAM_TOCPU CPU GPU INTERFEROGRAM) [1]
3:   (GETDERIVATIVE_KERNEL GPU INTERFEROGRAM DERIVATIVE) [1]
4:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
4:   (FILTERDERIVATIVE_KERNEL GPU DERIVATIVE FILTEREDDERIV) [1]
5:   (AUTOCONVOLUTION_KERNEL GPU FILTEREDDERIV AUTOCONV) [1]
6:   (AUTOCONV_TOCPU CPU GPU AUTOCONV) [1]
7:   (OUTPUTDATA2 CPU AUTOCONV) [1]
```

11.3.3.6 Heterogeneous CPU and GPU

```
; Version LPG-td-1.4
; Seed 97952378
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p C1070/ -out C1070
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.00
; Search time 0.00
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 1307.00

0:   (GETSIGNAL CPU CAMERA) [1]
1:   (CONVERTBYTES CPU CAMERA INTERFEROGRAM) [1]
2:   (OUTPUTDATA1 CPU INTERFEROGRAM) [1]
2:   (GETDERIVATIVE CPU INTERFEROGRAM DERIVATIVE) [1]
3:   (FILTERDERIVATIVE CPU DERIVATIVE FILTEREDDERIV) [1]
4:   (AUTOCONVOLUTION CPU FILTEREDDERIV AUTOCONV) [1]
5:   (OUTPUTDATA2 CPU AUTOCONV) [1]

(All heterogeneous plans are the same due to the CPU being the fastest architecture.
```

11.4 LSDI PDDL Files

11.4.1 Sample Domain File

```
(define (domain LSDI)
    (:requirements :strips :fluents)
    (:functions
        (total-cost)
    )
    (:predicates
        ;Variables
        (Background ?sig)
        (Signal ?sig)
        (Background_f ?sig)
        (Lamda ?sig)
        (sampleYt1 ?sig)
        (Polynomial ?sig)
        (MaxID ?sig)
        (MaxValue ?sig)
        (LowCutFreq ?sig)
        (HighCutFreq ?sig)
        (getHeight ?sig)
        (getHeight2 ?sig)
        (interSampleX ?sig)
        (interSampleYt1 ?sig)
        (fftResult_Real ?sig)
        (Phase ?sig)
        (unwrapedPhase ?sig)
        (interSampleX_c ?sig)
        (phase_final_c ?sig)
        (fftResult ?sig)
        (ifftResult ?sig)

        ;Function completion
        (done-divide1 ?sig)
        (done-interpolation ?sig)
        (done-fft ?sig)
        (done-absolute ?sig)
        (done-zeroData1 ?sig)
        (done-zeroData2 ?sig)
        (done-getMax ?sig)
        (done-Logic ?sig)
        (done-zeroComplexData1 ?sig)
        (done-zeroComplexData2 ?sig)
        (done-ifft ?sig)
        (done-complexLog ?sig)
        (done-getImag ?sig)
        (done-unwrapPhase ?sig)
        (done-windowSignal ?sig)
        (done-polyFit ?sig)
        (done-calculateHeight1 ?sig)
        (done-calculateHeight2 ?sig)
        (done-divide4 ?sig)
        (done-divide5 ?sig)

;Misc Parameters
        (CPU ?dev)
        (GPU ?dev)

        (data-onHost ?sig)
        (data-onDevice ?sig)

        (device-inUse ?dev)
    )
```

```
;##########################Memory Copy###################################

(:action Signal_toGPU
    :parameters(?cpu ?gpu ?sig1)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Signal ?sig1)
        ;Check Memory Location
        (data-onHost ?sig1)
        (not(data-onDevice ?sig1))
    )
    :effect(and
        (data-onDevice ?sig1)
        (increase (total-cost) 875)
    )
)

(:action interSampleYt1_toGPU
    :parameters(?cpu ?gpu ?sig6)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (interSampleYt1 ?sig6)
        ;Check Memory Location
        (data-onHost ?sig6)
        (not(data-onDevice ?sig6))
    )
    :effect(and
        (data-onDevice ?sig6)
        (increase (total-cost) 3092)
    )
)

(:action fftResult_toGPU
    :parameters(?cpu ?gpu ?sig7)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (fftResult ?sig7)
        ;Check Memory Location
        (data-onHost ?sig7)
        (not(data-onDevice ?sig7))
    )
    :effect(and
        (data-onDevice ?sig7)
        (increase (total-cost) 13714)
    )
)

(:action fftResult_Real_toGPU
    :parameters(?cpu ?gpu ?sig8)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (fftResult_Real ?sig8)
        ;Check Memory Location
        (data-onHost ?sig8)
        (not(data-onDevice ?sig8))
    )
    :effect(and
        (data-onDevice ?sig8)
        (increase (total-cost) 3844)
    )
)
```

```
(:action LowCutFreq_toGPU
    :parameters(?cpu ?gpu ?sig11)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (LowCutFreq ?sig11)
        ;Check Memory Location
        (data-onHost ?sig11)
        (not(data-onDevice ?sig11))
    )
    :effect(and
        (data-onDevice ?sig11)
        (increase (total-cost) 491)
    )
)
 (:action MaxID_toGPU
    :parameters(?cpu ?gpu ?sig9)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (MaxID ?sig9)
        ;Check Memory Location
        (data-onHost ?sig9)
        (not(data-onDevice ?sig9))
    )
    :effect(and
        (data-onDevice ?sig9)
        (increase (total-cost) 389)
    )
)

(:action HighCutFreq_toGPU
    :parameters(?cpu ?gpu ?sig12)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (HighCutFreq ?sig12)
        ;Check Memory Location
        (data-onHost ?sig12)
        (not(data-onDevice ?sig12))
    )
    :effect(and
        (data-onDevice ?sig12)
        (increase (total-cost) 461)
    )
)

(:action ifftResult_toGPU
    :parameters(?cpu ?gpu ?sig13)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (ifftResult ?sig13)
        ;Check Memory Location
        (data-onHost ?sig13)
        (not(data-onDevice ?sig13))
    )
    :effect(and
        (data-onDevice ?sig13)
        (increase (total-cost) 11031)
    )
)
```

```
 (:action Phase_toGPU
    :parameters(?cpu ?gpu ?sig14)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Phase ?sig14)
        ;Check Memory Location
        (data-onHost ?sig14)
        (not(data-onDevice ?sig14))
    )
    :effect(and
        (data-onDevice ?sig14)
        (increase (total-cost) 3274)
    )
)

(:action unwrapedPhase_toGPU
    :parameters(?cpu ?gpu ?sig15)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (unwrapedPhase ?sig15)
        ;Check Memory Location
        (data-onHost ?sig15)
        (not(data-onDevice ?sig15))
    )
    :effect(and
        (data-onDevice ?sig15)
        (increase (total-cost) 3037)
    )
)

(:action phase_final_c_toGPU
    :parameters(?cpu ?gpu ?sig16)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (phase_final_c ?sig16)
        ;Check Memory Location
        (data-onHost ?sig16)
        (not(data-onDevice ?sig16))
    )
    :effect(and
        (data-onDevice ?sig16)
        (increase (total-cost) 3087)
    )
)

 (:action Polynomial_toGPU
    :parameters(?cpu ?gpu ?sig18)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Polynomial ?sig18)
        ;Check Memory Location
        (data-onHost ?sig18)
        (not(data-onDevice ?sig18))
    )
    :effect(and
        (data-onDevice ?sig18)
        (increase (total-cost) 357)
    )
)
```

```
(:action getHeight_toGPU
       :parameters(?cpu ?gpu ?sig19)
       :precondition(and
           ;Check Hardware Status
           (CPU ?cpu)
           (GPU ?gpu)
           ;Check Signals
           (getHeight ?sig19)
           ;Check Memory Location
           (data-onHost ?sig19)
           (not(data-onDevice ?sig19))
       )
       :effect(and
           (data-onDevice ?sig19)
           (increase (total-cost) 360)
       )
   )

   (:action sampleYt1_toCPU
       :parameters(?cpu ?gpu ?sig3)
       :precondition(and
           ;Check Hardware Status
           (CPU ?cpu)
           (GPU ?gpu)
           ;Check Signals
           (sampleYt1 ?sig3)
           ;Check Memory Location
           (data-onDevice ?sig3)
           (not(data-onHost ?sig3))
       )
       :effect(and
           (data-onHost ?sig3)
           (increase (total-cost) 1393)
       )
   )

    (:action fftResult_toCPU
       :parameters(?cpu ?gpu ?sig7)
       :precondition(and
           ;Check Hardware Status
           (CPU ?cpu)
           (GPU ?gpu)
           ;Check Signals
           (fftResult ?sig7)
           ;Check Memory Location
           (data-onDevice ?sig7)
           (not(data-onHost ?sig7))
       )
       :effect(and
           (data-onHost ?sig7)
           (increase (total-cost) 12685)
       )
   )

   (:action fftResult_Real_toCPU
       :parameters(?cpu ?gpu ?sig8)
       :precondition(and
           ;Check Hardware Status
           (CPU ?cpu)
           (GPU ?gpu)
           ;Check Signals
           (fftResult_Real ?sig8)
           ;Check Memory Location
           (data-onDevice ?sig8)
           (not(data-onHost ?sig8))
       )
       :effect(and
           (data-onHost ?sig8)
           (increase (total-cost) 3559)
       )
   )
```

```
(:action LowCutFreq_toCPU
     :parameters(?cpu ?gpu ?sig11)
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         (GPU ?gpu)
         ;Check Signals
         (LowCutFreq ?sig11)
         ;Check Memory Location
         (data-onDevice ?sig11)
         (not(data-onHost ?sig11))
     )
     :effect(and
         (data-onHost ?sig11)
         (increase (total-cost) 465)
     )
 )

  (:action MaxID_toCPU
     :parameters(?cpu ?gpu ?sig9)
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         (GPU ?gpu)
         ;Check Signals
         (MaxID ?sig9)
         ;Check Memory Location
         (data-onDevice ?sig9)
         (not(data-onHost ?sig9))
     )
     :effect(and
         (data-onHost ?sig9)
         (increase (total-cost) 665)
     )
 )

 (:action MaxValue_toCPU
     :parameters(?cpu ?gpu ?sig10)
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         (GPU ?gpu)
         ;Check Signals
         (MaxID ?sig10)
         ;Check Memory Location
         (data-onDevice ?sig10)
         (not(data-onHost ?sig10))
     )
     :effect(and
         (data-onHost ?sig10)
         (increase (total-cost) 476)
     )
 )

 (:action HighCutFreq_toCPU
     :parameters(?cpu ?gpu ?sig12)
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         (GPU ?gpu)
         ;Check Signals
         (HighCutFreq ?sig12)
         ;Check Memory Location
         (data-onDevice ?sig12)
         (not(data-onHost ?sig12))
     )
     :effect(and
         (data-onHost ?sig12)
         (increase (total-cost) 460)
     )
 )
 (:action ifftResult_toCPU
    :parameters(?cpu ?gpu ?sig13)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (ifftResult ?sig13)
        ;Check Memory Location
        (data-onDevice ?sig13)
        (not(data-onHost ?sig13))
    )
    :effect(and
        (data-onHost ?sig13)
        (increase (total-cost) 10523)
    )
)

(:action Phase_toCPU
    :parameters(?cpu ?gpu ?sig14)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Phase ?sig14)
        ;Check Memory Location
        (data-onDevice ?sig14)
        (not(data-onHost ?sig14))
    )
    :effect(and
        (data-onHost ?sig14)
        (increase (total-cost) 2700)
    )
)

(:action unwrapedPhase_toCPU
    :parameters(?cpu ?gpu ?sig15)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (unwrapedPhase ?sig15)
        ;Check Memory Location
        (data-onDevice ?sig15)
        (not(data-onHost ?sig15))
    )
    :effect(and
        (data-onHost ?sig15)
        (increase (total-cost) 2916)
    )
)

 (:action phase_final_c_toCPU
    :parameters(?cpu ?gpu ?sig16)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (phase_final_c ?sig16)
        ;Check Memory Location
        (data-onDevice ?sig16)
        (not(data-onHost ?sig16))
    )
    :effect(and
        (data-onHost ?sig16)
        (increase (total-cost) 2667)
    )
)
```

```
(:action Polynomial_toCPU
    :parameters(?cpu ?gpu ?sig18)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (Polynomial ?sig18)
        ;Check Memory Location
        (data-onDevice ?sig18)
        (not(data-onHost ?sig18))
    )
    :effect(and
        (data-onHost ?sig18)
        (increase (total-cost) 569)
    )
)

(:action getHeight_toCPU
    :parameters(?cpu ?gpu ?sig19)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (GPU ?gpu)
        ;Check Signals
        (getHeight ?sig19)
        ;Check Memory Location
        (data-onDevice ?sig19)
        (not(data-onHost ?sig19))
    )
    :effect(and
        (data-onHost ?sig19)
        (increase (total-cost) 418)
    )
)

 (:action getHeight2_toCPU
        :parameters(?cpu ?gpu ?sig20)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            (GPU ?gpu)
            ;Check Signals
            (getHeight2 ?sig20)
            ;Check Memory Location
            (data-onDevice ?sig20)
            (not(data-onHost ?sig20))
        )
        :effect(and
            (data-onHost ?sig20)
            (increase (total-cost) 422)
        )
    )
```

```
;################## CPU Functions #############################

    (:action divide1
        :parameters(?cpu ?sig1 ?sig2 ?sig3)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (Signal ?sig1)
            (Background_f ?sig2)
            (sampleYt1 ?sig3)
            ;Check Function Status
            (not(done-divide1 ?sig3))
            ;Check Memory Location
            (data-onHost ?sig1) (data-onHost ?sig2)
        )
        :effect(and
            (data-onHost ?sig3)
            (not(data-onDevice ?sig3))
            (done-divide1 ?sig3)
            (increase (total-cost) 779)
        )
    )

     (:action interpolation
        :parameters(?cpu ?sig4 ?sig3 ?sig5 ?sig6)
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (Lamda ?sig4) (sampleYt1 ?sig3)
                (interSampleX ?sig5) (interSampleYt1 ?sig6)
            ;Check Function Status
            (done-divide1 ?sig3) (not(done-interpolation ?sig6))
            ;Check Memory Location
            (data-onHost ?sig5) (data-onHost ?sig4) (data-onHost ?sig3)
        )
        :effect(and
            (data-onHost ?sig6)
            (not(data-onDevice ?sig6))
            (done-interpolation ?sig6)
            (increase (total-cost) 174914)
        )
    )

    (:action fft
        :parameters(?cpu ?sig6 ?sig7 )
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (interSampleYt1 ?sig6) (fftResult ?sig7)
            ;Check Function Status
            (done-interpolation ?sig6) (not(done-fft ?sig7))
            ;Check Memory Location
            (data-onHost ?sig6)
        )
        :effect(and
            (data-onHost ?sig7)
            (not(data-onDevice ?sig7))
            (done-fft ?sig7)
            (increase (total-cost) 57899)
        )
    )
```

```
 (:action absolute
    :parameters(?cpu ?sig7 ?sig8 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (fftResult ?sig7) (fftResult_Real ?sig8)
        ;Check Function Status
        (done-fft ?sig7) (not(done-absolute ?sig8))
        ;Check Memory Location
        (data-onHost ?sig7)
    )
    :effect(and
        (data-onHost ?sig8)
        (not(data-onDevice ?sig8))
        (done-absolute ?sig8)
        (increase (total-cost) 8926)
    )
)

(:action zeroData1
    :parameters(?cpu ?sig8)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (fftResult_Real ?sig8)
        ;Check Function Status
        (done-absolute ?sig8) (not(done-zeroData1 ?sig8))
        ;Check Memory Location
        (data-onHost ?sig8)
    )
    :effect(and
        (data-onHost ?sig8)
        (not(data-onDevice ?sig8))
        (done-zeroData1 ?sig8)
        (increase (total-cost) 18)
    )
)

 (:action zeroData2
    :parameters(?cpu ?sig8)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (fftResult_Real ?sig8)
        ;Check Function Status
        (done-absolute ?sig8) (not(done-zeroData2 ?sig8))
        ;Check Memory Location
        (data-onHost ?sig8)
    )
    :effect(and
        (data-onHost ?sig8)
        (not(data-onDevice ?sig8))
        (done-zeroData2 ?sig8)
        (increase (total-cost) 240)
    )
)
```

```
(:action getMax
     :parameters(?cpu ?sig8 ?sig9 ?sig10 )
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         ;Check Signals
         (fftResult_Real ?sig8) (maxID ?sig9) (maxValue ?sig10)
         ;Check Function Status
         (done-zeroData1 ?sig8) (done-zeroData2 ?sig8)
             (not(done-getMax ?sig9)) (not(done-getMax ?sig10))
         ;Check Memory Location
         (data-onHost ?sig8)
     )
     :effect(and
         (data-onHost ?sig9)
         (not(data-onDevice ?sig9))
         (data-onHost ?sig10)
         (not(data-onDevice ?sig10))
         (done-getMax ?sig9)
         (done-getMax ?sig10)
         (increase (total-cost) 5512)
     )
 )

  (:action Logic
     :parameters(?cpu ?sig9 ?sig11 ?sig12 )
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         ;Check Signals
         (maxID ?sig9) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
         ;Check Function Status
         (done-getMax ?sig9) (not(done-Logic ?sig9))
         ;Check Memory Location
         (data-onHost ?sig9)
     )
     :effect(and
         (data-onHost ?sig11)
         (not(data-onDevice ?sig11))
         (data-onHost ?sig12)
         (not(data-onDevice ?sig12))
         (done-Logic ?sig9)
         (done-Logic ?sig11)
         (done-Logic ?sig12)
         (increase (total-cost) 18)
     )
 )

 (:action zeroComplexData1
     :parameters(?cpu ?sig7 ?sig11 ?sig12)
     :precondition(and
         ;Check Hardware Status
         (CPU ?cpu)
         ;Check Signals
         (fftResult ?sig7) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
         ;Check Function Status
         (done-fft ?sig7) (done-Logic ?sig11)
             (done-Logic ?sig12) (not(done-zeroComplexData1 ?sig7))
         ;Check Memory Location
         (data-onHost ?sig7) (data-onHost ?sig11) (data-onHost ?sig12)
     )
     :effect(and
         (data-onHost ?sig7)
         (not(data-onDevice ?sig7))
         (done-zeroComplexData1 ?sig7)
         (increase (total-cost) 73)
     )
 )
 (:action zeroComplexData2
    :parameters(?cpu ?sig7 ?sig11 ?sig12)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (fftResult ?sig7) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
        ;Check Function Status
        (done-fft ?sig7) (done-Logic ?sig11)
         (done-Logic ?sig12) (not(done-zeroComplexData2 ?sig7))
        ;Check Memory Location
        (data-onHost ?sig7) (data-onHost ?sig11) (data-onHost ?sig12)
    )
    :effect(and
        (data-onHost ?sig7)
        (not(data-onDevice ?sig7))
        (done-zeroComplexData2 ?sig7)
        (increase (total-cost) 937)
    )
)
(:action ifft
    :parameters(?cpu ?sig7 ?sig13 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (fftResult ?sig7) (ifftResult ?sig13)
        ;Check Function Status
        (done-zeroComplexData1 ?sig7)
            (done-zeroComplexData2 ?sig7) (not(done-ifft ?sig13))
        ;Check Memory Location
        (data-onHost ?sig7)
    )
    :effect(and
        (data-onHost ?sig13)
        (not(data-onDevice ?sig13))
        (done-ifft ?sig13)
        (increase (total-cost) 49653)
    )
)

 (:action complexLog
    :parameters(?cpu ?sig13 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (ifftResult ?sig13)
        ;Check Function Status
        (done-ifft ?sig13) (not(done-complexLog ?sig13))
        ;Check Memory Location
        (data-onHost ?sig13)
    )
    :effect(and
        (data-onHost ?sig13)
        (not(data-onDevice ?sig13))
        (done-complexLog ?sig13)
        (increase (total-cost) 79875)
    )
)
```

```
(:action getImag
        :parameters(?cpu ?sig13 ?sig14 )
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (ifftResult ?sig13) (Phase ?sig14)
            ;Check Function Status
            (done-complexLog ?sig13) (not(done-getImag ?sig14))
            ;Check Memory Location
            (data-onHost ?sig13)
        )
        :effect(and
            (data-onHost ?sig14)
            (not(data-onDevice ?sig14))
            (done-getImag ?sig14)
            (increase (total-cost) 1779)
        )
    )

     (:action unwrapPhase
        :parameters(?cpu ?sig14 ?sig15 )
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (Phase ?sig14) (unwrapedPhase ?sig15)
            ;Check Function Status
            (done-getImag ?sig14) (not(done-unwrapPhase ?sig15))
            ;Check Memory Location
            (data-onHost ?sig14)
        )
        :effect(and
            (data-onHost ?sig15)
            (not(data-onDevice ?sig15))
            (done-unwrapPhase ?sig15)
            (increase (total-cost) 5802)
        )
    )

    (:action windowSignal
        :parameters(?cpu ?sig15 ?sig16 )
        :precondition(and
            ;Check Hardware Status
            (CPU ?cpu)
            ;Check Signals
            (unwrapedPhase ?sig15) (phase_final_c ?sig16)
            ;Check Function Status
            (done-unwrapPhase ?sig15) (not(done-windowSignal ?sig16))
            ;Check Memory Location
            (data-onHost ?sig15)
        )
        :effect(and
            (data-onHost ?sig16)
            (not(data-onDevice ?sig16))
            (done-windowSignal ?sig16)
            (increase (total-cost) 1263)
        )
    )
```

```
 (:action polyFit
    :parameters(?cpu ?sig17 ?sig16 ?sig18 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (interSampleX_c ?sig17) (phase_final_c ?sig16) (Polynomial ?sig18)
        ;Check Function Status
        (done-windowSignal ?sig16) (not(done-polyFit ?sig18))
        ;Check Memory Location
        (data-onHost ?sig16) (data-onHost ?sig17)
    )
    :effect(and
        (data-onHost ?sig18)
        (not(data-onDevice ?sig18))
        (done-polyFit ?sig18)
        (increase (total-cost) 62974)
    )
)

(:action calculateHeight1
    :parameters(?cpu ?sig18 ?sig19 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (Polynomial ?sig18) (getHeight ?sig19)
        ;Check Function Status
        (done-polyFit ?sig18) (not(done-calculateHeight1 ?sig19))
        ;Check Memory Location
        (data-onHost ?sig18)
    )
    :effect(and
        (data-onHost ?sig19)
        (not(data-onDevice ?sig19))
        (done-calculateHeight1 ?sig19)
        (increase (total-cost) 4)
    )
)

 (:action calculateHeight2
    :parameters(?cpu ?sig5 ?sig14 ?sig19 ?sig20 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        ;Check Signals
        (interSampleX ?sig5) (Phase ?sig14)
            (getHeight ?sig19) (getHeight2 ?sig20)
        ;Check Function Status
        (done-getImag ?sig14) (done-calculateHeight1 ?sig19)
            (not(done-calculateHeight2 ?sig20))
        ;Check Memory Location
        (data-onHost ?sig5) (data-onHost ?sig14) (data-onHost ?sig19)
    )
    :effect(and
        (data-onHost ?sig20)
        (not(data-onDevice ?sig20))
        (done-calculateHeight2 ?sig20)
        (increase (total-cost) 195)
    )
)
```

```
(:action divide4
    :parameters(?cpu ?sig19 )
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (not(device-inUse ?cpu))
        ;Check Signals
        (getHeight ?sig19)
        ;Check Function Status
        (done-calculateHeight1 ?sig19) (not(done-divide4 ?sig19))
        ;Check Memory Location
        (data-onHost ?sig19)
    )
    :effect(and
    (data-onHost ?sig19)
        (not(data-onDevice ?sig19))
        (done-divide4 ?sig19)
        (increase (total-cost) 2)
    )
)

 (:action divide5
    :parameters(?cpu ?sig20)
    :precondition(and
        ;Check Hardware Status
        (CPU ?cpu)
        (not(device-inUse ?cpu))
        ;Check Signals
        (getHeight2 ?sig20)
        ;Check Function Status
        (done-calculateHeight2 ?sig20) (not(done-divide5 ?sig20))
        ;Check Memory Location
        (data-onHost ?sig20)
    )
    :effect(and
        (data-onHost ?sig20)
        (not(data-onDevice ?sig20))
        (done-divide5 ?sig20)
        (increase (total-cost) 1)
    )
)

;################# Kernel Functions ############################

(:action divide1_Kernel
    :parameters(?gpu ?sig1 ?sig2 ?sig3)
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (Signal ?sig1)
        (Background_f ?sig2)
        (sampleYt1 ?sig3)
        ;Check Function Status
        (not(done-divide1 ?sig3))
        ;Check Memory Location
        (data-onDevice ?sig1) (data-onDevice ?sig2)
    )
    :effect(and
        (not(data-onHost ?sig3))
        (data-onDevice ?sig3)
        (done-divide1 ?sig3)
        (increase (total-cost) 310)
    )
)
```

```
 (:action fft_Kernel
    :parameters(?gpu ?sig6 ?sig7 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (interSampleYt1 ?sig6) (fftResult ?sig7)
        ;Check Function Status
        (done-interpolation ?sig6) (not(done-fft ?sig7))
        ;Check Memory Location
        (data-onDevice ?sig6)
    )
    :effect(and
        (not(data-onHost ?sig7))
        (data-onDevice ?sig7)
        (done-fft ?sig7)
        (increase (total-cost) 1540)
    )
)

(:action absolute_Kernel
    :parameters(?gpu ?sig7 ?sig8 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (fftResult ?sig7) (fftResult_Real ?sig8)
        ;Check Function Status
        (done-fft ?sig7) (not(done-absolute ?sig8))
        ;Check Memory Location
        (data-onDevice ?sig7)
    )
    :effect(and
        (not(data-onHost ?sig8))
        (data-onDevice ?sig8)
        (done-absolute ?sig8)
        (increase (total-cost) 788)
    )
)

 (:action zeroData1_Kernel
    :parameters(?gpu ?sig8)
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (fftResult_Real ?sig8)
        ;Check Function Status
        (done-absolute ?sig8) (not(done-zeroData1 ?sig8))
        ;Check Memory Location
        (data-onDevice ?sig8)
    )
    :effect(and
        (not(data-onHost ?sig8))
        (data-onDevice ?sig8)
        (done-zeroData1 ?sig8)
        (increase (total-cost) 508)
    )
)
```

```
(:action zeroData2_Kernel
        :parameters(?gpu ?sig8)
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (fftResult_Real ?sig8)
            ;Check Function Status
            (done-absolute ?sig8) (not(done-zeroData2 ?sig8))
            ;Check Memory Location
            (data-onDevice ?sig8)
        )
        :effect(and
            (not(data-onHost ?sig8))
            (data-onDevice ?sig8)
            (done-zeroData2 ?sig8)
            (increase (total-cost) 474)
        )
    )

     (:action getMax_Kernel
        :parameters(?gpu ?sig8 ?sig9 ?sig10 )
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (fftResult_Real ?sig8) (maxID ?sig9) (maxValue ?sig10)
            ;Check Function Status
            (done-zeroData1 ?sig8) (done-zeroData2 ?sig8)
                (not(done-getMax ?sig9)) (not(done-getMax ?sig10))
            ;Check Memory Location
            (data-onDevice ?sig8)
        )
        :effect(and
            (not(data-onHost ?sig9))
            (data-onDevice ?sig9)
            (not(data-onHost ?sig10))
            (data-onDevice ?sig10)
            (done-getMax ?sig9)
            (done-getMax ?sig10)
            (increase (total-cost) 23756)
        )
    )

    (:action Logic_Kernel
        :parameters(?gpu ?sig9 ?sig11 ?sig12 )
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (maxID ?sig9) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
            ;Check Function Status
            (done-getMax ?sig9) (not(done-Logic ?sig9))
            ;Check Memory Location
            (data-onDevice ?sig9)
        )
        :effect(and
            (data-onDevice ?sig11)
            (not(data-onHost ?sig11))
            (data-onDevice ?sig12)
            (not(data-onHost ?sig12))
            (done-Logic ?sig9)
            (done-Logic ?sig11)
            (done-Logic ?sig12)
            (increase (total-cost) 1237)
        )
    )
```

```
 (:action zeroComplexData1_Kernel
    :parameters(?gpu ?sig7 ?sig11 ?sig12 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (fftResult ?sig7) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
        ;Check Function Status
        (done-fft ?sig7) (done-Logic ?sig11)
            (done-Logic ?sig12) (not(done-zeroComplexData1 ?sig7))
        ;Check Memory Location
        (data-onDevice ?sig7) (data-onDevice ?sig11)
            (data-onDevice ?sig12)
    )
    :effect(and
        (not(data-onHost ?sig7))
        (data-onDevice ?sig7)
        (done-zeroComplexData1 ?sig7)
        (increase (total-cost) 469)
    )
)

(:action zeroComplexData2_Kernel
    :parameters(?gpu ?sig7 ?sig11 ?sig12 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (fftResult ?sig7) (LowCutFreq ?sig11) (HighCutFreq ?sig12)
        ;Check Function Status
        (done-fft ?sig7) (done-Logic ?sig11)
            (done-Logic ?sig12) (not(done-zeroComplexData2 ?sig7))
        ;Check Memory Location
        (data-onDevice ?sig7) (data-onDevice ?sig11)
            (data-onDevice ?sig12)
    )
    :effect(and
        (not(data-onHost ?sig7))
        (data-onDevice ?sig7)
        (done-zeroComplexData2 ?sig7)
        (increase (total-cost) 449)
    )
)

 (:action ifft_Kernel
    :parameters(?gpu ?sig7 ?sig13 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (fftResult ?sig7) (ifftResult ?sig13)
        ;Check Function Status
        (done-zeroComplexData1 ?sig7)
            (done-zeroComplexData2 ?sig7) (not(done-ifft ?sig13))
        ;Check Memory Location
        (data-onDevice ?sig7)
    )
    :effect(and
        (not(data-onHost ?sig13))
        (data-onDevice ?sig13)
        (done-ifft ?sig13)
        (increase (total-cost) 1710)
    )
)
```

```
(:action complexLog_Kernel
     :parameters(?gpu ?sig13 )
     :precondition(and
         ;Check Hardware Status
         (GPU ?gpu)
         ;Check Signals
         (ifftResult ?sig13)
         ;Check Function Status
         (done-ifft ?sig13) (not(done-complexLog ?sig13))
         ;Check Memory Location
         (data-onDevice ?sig13)
     )
     :effect(and
         (not(data-onHost ?sig13))
         (data-onDevice ?sig13)
         (done-complexLog ?sig13)
         (increase (total-cost) 572)
     )
 )

  (:action getImag_Kernel
     :parameters(?gpu ?sig13 ?sig14 )
     :precondition(and
         ;Check Hardware Status
         (GPU ?gpu)
         ;Check Signals
         (ifftResult ?sig13) (Phase ?sig14)
         ;Check Function Status
         (done-complexLog ?sig13) (not(done-getImag ?sig14))
         ;Check Memory Location
         (data-onDevice ?sig13)
     )
     :effect(and
         (not(data-onHost ?sig14))
         (data-onDevice ?sig14)
         (done-getImag ?sig14)
         (increase (total-cost) 469)
     )
 )

 (:action unwrapPhase_Kernel
     :parameters(?gpu ?sig14 ?sig15 )
     :precondition(and
         ;Check Hardware Status
         (GPU ?gpu)
         ;Check Signals
         (Phase ?sig14) (unwrapedPhase ?sig15)
         ;Check Function Status
         (done-getImag ?sig14) (not(done-unwrapPhase ?sig15))
         ;Check Memory Location
         (data-onDevice ?sig14)
     )
     :effect(and
         (not(data-onHost ?sig15))
         (data-onDevice ?sig15)
         (done-unwrapPhase ?sig15)
         (increase (total-cost) 6456)
     )
 )
 (:action windowSignal_Kernel
    :parameters(?gpu ?sig15 ?sig16 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (unwrapedPhase ?sig15) (phase_final_c ?sig16)
        ;Check Function Status
        (done-unwrapPhase ?sig15) (not(done-windowSignal ?sig16))
        ;Check Memory Location
        (data-onDevice ?sig15)
    )
    :effect(and
        (not(data-onHost ?sig16))
        (data-onDevice ?sig16)
        (done-windowSignal ?sig16)
        (increase (total-cost) 320)
    )
)

(:action polyFit_Kernel
    :parameters(?gpu ?sig17 ?sig16 ?sig18 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (interSampleX_c ?sig17) (phase_final_c ?sig16) (Polynomial ?sig18)
        ;Check Function Status
        (done-windowSignal ?sig16) (not(done-polyFit ?sig18))
        ;Check Memory Location
        (data-onDevice ?sig16) (data-onDevice ?sig17)
    )
    :effect(and
        (not(data-onHost ?sig18))
        (data-onDevice ?sig18)
        (done-polyFit ?sig18)
        (increase (total-cost) 22218)
    )
)

 (:action calculateHeight1_Kernel
    :parameters(?gpu ?sig18 ?sig19 )
    :precondition(and
        ;Check Hardware Status
        (GPU ?gpu)
        ;Check Signals
        (Polynomial ?sig18) (getHeight ?sig19)
        ;Check Function Status
        (done-polyFit ?sig18) (not(done-calculateHeight1 ?sig19))
        ;Check Memory Location
        (data-onDevice ?sig18)
    )
    :effect(and
        (not(data-onHost ?sig19))
        (data-onDevice ?sig19)
        (done-calculateHeight1 ?sig19)
        (increase (total-cost) 309)
    )
)
```

```
(:action calculateHeight2_Kernel
        :parameters(?gpu ?sig5 ?sig14 ?sig19 ?sig20 )
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            ;Check Signals
            (interSampleX ?sig5) (Phase ?sig14)
                (getHeight ?sig19) (getHeight2 ?sig20)
            ;Check Function Status
            (done-getImag ?sig14) (done-calculateHeight1 ?sig19)
                (not(done-calculateHeight2 ?sig20))
            ;Check Memory Location
            (data-onDevice ?sig5) (data-onDevice ?sig14)
                (data-onDevice ?sig19)
        )
        :effect(and
            (not(data-onHost ?sig20))
            (data-onDevice ?sig20)
            (done-calculateHeight2 ?sig20)
            (increase (total-cost) 320)
        )
    )
)
   (:action divide4_Kernel
        :parameters(?gpu ?sig19 )
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            (not(device-inUse ?gpu))
            ;Check Signals
            (getHeight ?sig19)
            ;Check Function Status
            (done-calculateHeight1 ?sig19) (not(done-divide4 ?sig19))
            ;Check Memory Location
            (data-onDevice ?sig19)
        )
        :effect(and
            (not(data-onHost ?sig19))
            (data-onDevice ?sig19)
            (done-divide4 ?sig19)
            (increase (total-cost) 301)
        )
    )

    (:action divide5_Kernel
        :parameters(?gpu ?sig20 )
        :precondition(and
            ;Check Hardware Status
            (GPU ?gpu)
            (not(device-inUse ?gpu))
            ;Check Signals
            (getHeight2 ?sig20)
            ;Check Function Status
            (done-calculateHeight2 ?sig20) (not(done-divide5 ?sig20))
            ;Check Memory Location
            (data-onDevice ?sig20)
        )
        :effect(and
            (not(data-onHost ?sig20))
            (data-onDevice ?sig20)
            (done-divide5 ?sig20)
            (increase (total-cost) 293)
        )
    )
)
```

11.4.2 Problem File

```
(define (problem LSDI-p1)
    (:domain LSDI)
       (:objects _Background _Signal _Background_f _Lamda _sampleYt1
                      _Polynomial _MaxID _MaxValue _LowCutFreq _HighCutFreq
                      _getHeight _getHeight2 _interSampleX _interSampleYt1
                      _fftResult_Real _Phase _unwrapedPhase _interSampleX_c
                      _phase_final_c _fftResult _ifftResult _CPU _GPU)

    (:init
        ;Init Signals
        (Background _Background)
        (Signal _Signal)
        (Background_f _Background_f)
        (Lamda _Lamda)
        (sampleYt1 _sampleYt1)
        (Polynomial _Polynomial)
        (MaxID _MaxID)
        (MaxValue _MaxValue)
        (LowCutFreq _LowCutFreq)
        (HighCutFreq _HighCutFreq)
        (getHeight _getHeight)
        (getHeight2 _getHeight2)
        (interSampleX _interSampleX)
        (interSampleYt1 _interSampleYt1)
        (fftResult_Real _fftResult_Real)
        (Phase _Phase)
        (unwrapedPhase _unwrapedPhase)
        (interSampleX_c _interSampleX_c)
        (phase_final_c _phase_final_c)
        (fftResult _fftResult)
        (ifftResult _ifftResult)
        (CPU _CPU)
        (GPU _GPU)

        (data-onHost _Signal)
        (data-onHost _Background_f)
        (data-onDevice _Background_f)
        (data-onHost _Lamda)
        (data-onDevice _Lamda)
        (data-onHost _interSampleX)
        (data-onDevice _interSampleX)
        (data-onHost _LowCutFreq)
        (data-onHost _HighCutFreq)
        (data-onHost _interSampleX_c)
        (data-onDevice _interSampleX_c)
        ;Init Durative Paramters
        (= (total-cost) 0))

    (:goal (and (done-calculateHeight1 _getHeight)
                      (done-calculateHeight2 _getHeight2)
                      (data-onHost _getHeight) (data-onHost _getHeight2)))
    (:metric minimize (total-cost))
)
```

11.4.3 Solution Files

11.4.3.1 CPU

```
; Version LPG-td-1.4
; Seed 48473465
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 50 -p CPU/ -out CPU -cputime 30
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 451059.00

0:   (DIVIDE1 CPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
1:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
2:   (FFT CPU INTERSAMPLEYT1 FFTRESULT) [1]
3:   (ABSOLUTE CPU FFTRESULT FFTRESULT_REAL) [1]
4:   (ZERODATA1 CPU FFTRESULT_REAL) [1]
5:   (ZERODATA2 CPU FFTRESULT_REAL) [1]
6:   (GETMAX CPU FFTRESULT_REAL MAXID MAXVALUE) [1]
7:   (LOGIC CPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
8:   (ZEROCOMPLEXDATA1 CPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
9:   (ZEROCOMPLEXDATA2 CPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
10:   (IFFT CPU FFTRESULT IFFTRESULT) [1]
11:   (COMPLEXLOG CPU IFFTRESULT) [1]
12:   (GETIMAG CPU IFFTRESULT PHASE) [1]
13:   (UNWRAPPHASE CPU PHASE UNWRAPEDPHASE) [1]
14:   (WINDOWSIGNAL CPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
15:   (POLYFIT CPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
16:   (CALCULATEHEIGHT1 CPU POLYNOMIAL GETHEIGHT) [1]
17:   (CALCULATEHEIGHT2 CPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
```

11.4.3.2 GTX 650Ti

```
; Version LPG-td-1.4
; Seed 82056988
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 650/ -out 650 -cputime 30
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.02
; Search time 0.01
; Parsing time 0.01
; Mutex time 0.00
; MetricValue 315345.00

0:   (SIGNAL_TOGPU CPU GPU SIGNAL) [1]
1:   (DIVIDE1_KERNEL GPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
2:   (SAMPLEYT1_TOCPU CPU GPU SAMPLEYT1) [1]
3:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
4:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
5:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
6:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
7:   (ZERODATA2_KERNEL GPU FFTRESULT_REAL) [1]
8:   (ZERODATA1_KERNEL GPU FFTRESULT_REAL) [1]
9:   (GETMAX_KERNEL GPU FFTRESULT_REAL MAXID MAXVALUE) [1]
10:   (LOGIC_KERNEL GPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (UNWRAPPHASE_KERNEL GPU PHASE UNWRAPEDPHASE) [1]
17:   (WINDOWSIGNAL_KERNEL GPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
18:   (POLYFIT_KERNEL GPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
19:   (CALCULATEHEIGHT1_KERNEL GPU POLYNOMIAL GETHEIGHT) [1]
20:   (CALCULATEHEIGHT2_KERNEL GPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
20:   (GETHEIGHT_TOCPU CPU GPU GETHEIGHT) [1]
21:   (GETHEIGHT2_TOCPU CPU GPU GETHEIGHT2) [1]
```

11.4.3.3 GTX 780Ti

```
; Version LPG-td-1.4
; Seed 112100601
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 780/ -out 780 -cputime 30
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.02
; Search time 0.01
; Parsing time 0.01
; Mutex time 0.00
; MetricValue 283388.00

0:   (SIGNAL_TOGPU CPU GPU SIGNAL) [1]
1:   (DIVIDE1_KERNEL GPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
2:   (SAMPLEYT1_TOCPU CPU GPU SAMPLEYT1) [1]
3:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
4:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
5:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
6:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
7:   (ZERODATA2_KERNEL GPU FFTRESULT_REAL) [1]
8:   (ZERODATA1_KERNEL GPU FFTRESULT_REAL) [1]
9:   (GETMAX_KERNEL GPU FFTRESULT_REAL MAXID MAXVALUE) [1]
10:   (LOGIC_KERNEL GPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (UNWRAPPHASE_KERNEL GPU PHASE UNWRAPEDPHASE) [1]
17:   (WINDOWSIGNAL_KERNEL GPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
18:   (POLYFIT_KERNEL GPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
19:   (CALCULATEHEIGHT1_KERNEL GPU POLYNOMIAL GETHEIGHT) [1]
20:   (CALCULATEHEIGHT2_KERNEL GPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
20:   (GETHEIGHT_TOCPU CPU GPU GETHEIGHT) [1]
21:   (GETHEIGHT2_TOCPU CPU GPU GETHEIGHT2) [1]
```

11.4.3.4 GTX 1070

```
; Version LPG-td-1.4
; Seed 14150209
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 10 -p 1070/ -out 1070 -cputime 30
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.01
; Search time 0.01
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 242646.00

0:   (SIGNAL_TOGPU CPU GPU SIGNAL) [1]
1:   (DIVIDE1_KERNEL GPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
2:   (SAMPLEYT1_TOCPU CPU GPU SAMPLEYT1) [1]
3:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
4:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
5:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
6:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
7:   (ZERODATA2_KERNEL GPU FFTRESULT_REAL) [1]
8:   (ZERODATA1_KERNEL GPU FFTRESULT_REAL) [1]
9:   (GETMAX_KERNEL GPU FFTRESULT_REAL MAXID MAXVALUE) [1]
10:   (LOGIC_KERNEL GPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (UNWRAPPHASE_KERNEL GPU PHASE UNWRAPEDPHASE) [1]
17:   (WINDOWSIGNAL_KERNEL GPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
18:   (POLYFIT_KERNEL GPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
19:   (CALCULATEHEIGHT1_KERNEL GPU POLYNOMIAL GETHEIGHT) [1]
20:   (CALCULATEHEIGHT2_KERNEL GPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
20:   (GETHEIGHT_TOCPU CPU GPU GETHEIGHT) [1]
21:   (GETHEIGHT2_TOCPU CPU GPU GETHEIGHT2) [1]
```

11.4.3.5 Heterogeneous CPU and GTX 650Ti

```
; Version LPG-td-1.4
; Seed 44203424
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 50 -p C650/ -out C650 -cputime 300
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 0.56
; Search time 0.54
; Parsing time 0.02
; Mutex time 0.00
; MetricValue 275466.00

0:   (DIVIDE1 CPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
1:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
2:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
3:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
4:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
5:   (FFTRESULT_REAL_TOCPU CPU GPU FFTRESULT_REAL) [1]
6:   (ZERODATA2 CPU FFTRESULT_REAL) [1]
7:   (ZERODATA1 CPU FFTRESULT_REAL) [1]
8:   (GETMAX CPU FFTRESULT_REAL MAXID MAXVALUE) [1]
9:   (LOGIC CPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
10:   (HIGHCUTFREQ_TOGPU CPU GPU HIGHCUTFREQ) [1]
10:   (LOWCUTFREQ_TOGPU CPU GPU LOWCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (PHASE_TOCPU CPU GPU PHASE) [1]
17:   (UNWRAPPHASE CPU PHASE UNWRAPEDPHASE) [1]
18:   (WINDOWSIGNAL CPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
19:   (POLYFIT CPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
20:   (CALCULATEHEIGHT1 CPU POLYNOMIAL GETHEIGHT) [1]
21:   (CALCULATEHEIGHT2 CPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
```

11.4.3.6 Heterogeneous CPU and GTX 780Ti

```
; Version LPG-td-1.4
; Seed 88305982
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 50 -p C780/ -out C780 -cputime 300
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 2.63
; Search time 2.62
; Parsing time 0.01
; Mutex time 0.00
; MetricValue 268044.00

0:   (DIVIDE1 CPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
1:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
2:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
3:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
4:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
5:   (FFTRESULT_REAL_TOCPU CPU GPU FFTRESULT_REAL) [1]
6:   (ZERODATA1 CPU FFTRESULT_REAL) [1]
7:   (ZERODATA2 CPU FFTRESULT_REAL) [1]
8:   (GETMAX CPU FFTRESULT_REAL MAXID MAXVALUE) [1]
9:   (LOGIC CPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
10:   (HIGHCUTFREQ_TOGPU CPU GPU HIGHCUTFREQ) [1]
10:   (LOWCUTFREQ_TOGPU CPU GPU LOWCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (PHASE_TOCPU CPU GPU PHASE) [1]
17:   (UNWRAPPHASE CPU PHASE UNWRAPEDPHASE) [1]
18:   (WINDOWSIGNAL CPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
19:   (POLYFIT CPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
20:   (CALCULATEHEIGHT1 CPU POLYNOMIAL GETHEIGHT) [1]
21:   (CALCULATEHEIGHT2 CPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
```

11.4.3.7 Heterogeneous CPU and GTX 1070

```
; Version LPG-td-1.4
; Seed 4375665
; Command line: ./lpg-td -o Domain.pddl -f Problem.pddl -n 50 -p C1070/ -out C1070 -cputime
300
; Problem Problem.pddl
; Actions having STRIPS duration
; Time 1.42
; Search time 1.42
; Parsing time 0.00
; Mutex time 0.00
; MetricValue 225403.00

0:   (DIVIDE1 CPU SIGNAL BACKGROUND_F SAMPLEYT1) [1]
1:   (INTERPOLATION CPU LAMDA SAMPLEYT1 INTERSAMPLEX INTERSAMPLEYT1) [1]
2:   (INTERSAMPLEYT1_TOGPU CPU GPU INTERSAMPLEYT1) [1]
3:   (FFT_KERNEL GPU INTERSAMPLEYT1 FFTRESULT) [1]
4:   (ABSOLUTE_KERNEL GPU FFTRESULT FFTRESULT_REAL) [1]
5:   (FFTRESULT_REAL_TOCPU CPU GPU FFTRESULT_REAL) [1]
6:   (ZERODATA1 CPU FFTRESULT_REAL) [1]
7:   (ZERODATA2 CPU FFTRESULT_REAL) [1]
8:   (GETMAX CPU FFTRESULT_REAL MAXID MAXVALUE) [1]
9:   (LOGIC CPU MAXID LOWCUTFREQ HIGHCUTFREQ) [1]
10:   (LOWCUTFREQ_TOGPU CPU GPU LOWCUTFREQ) [1]
10:   (HIGHCUTFREQ_TOGPU CPU GPU HIGHCUTFREQ) [1]
11:   (ZEROCOMPLEXDATA2_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
12:   (ZEROCOMPLEXDATA1_KERNEL GPU FFTRESULT LOWCUTFREQ HIGHCUTFREQ) [1]
13:   (IFFT_KERNEL GPU FFTRESULT IFFTRESULT) [1]
14:   (COMPLEXLOG_KERNEL GPU IFFTRESULT) [1]
15:   (GETIMAG_KERNEL GPU IFFTRESULT PHASE) [1]
16:   (UNWRAPPHASE_KERNEL GPU PHASE UNWRAPEDPHASE) [1]
17:   (WINDOWSIGNAL_KERNEL GPU UNWRAPEDPHASE PHASE_FINAL_C) [1]
18:   (POLYFIT_KERNEL GPU INTERSAMPLEX_C PHASE_FINAL_C POLYNOMIAL) [1]
19:   (CALCULATEHEIGHT1_KERNEL GPU POLYNOMIAL GETHEIGHT) [1]
20:   (GETHEIGHT_TOCPU CPU GPU GETHEIGHT) [1]
20:   (CALCULATEHEIGHT2_KERNEL GPU INTERSAMPLEX PHASE GETHEIGHT GETHEIGHT2) [1]
21:   (GETHEIGHT2_TOCPU CPU GPU GETHEIGHT2) [1]
```