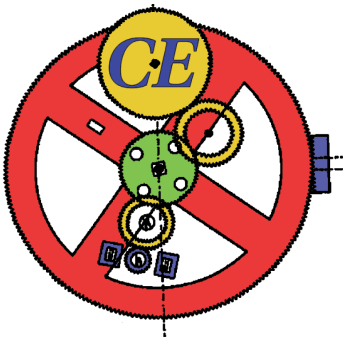

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

Performance Improvement of Optical Algorithms on Multicore Platforms

Ratnakar Madan

CE-MS-2013-16

Abstract

ASML is one of the world's largest suppliers of lithography systems for the semiconductor industry. ASML designs and develops machines that are used to print circuits on silicon wafers, to produce IC chips. These circuits have to be printed with accuracy of upto *2*nm. For this purpose, the machines incorporate several measurement systems. The Parallel Integrated Lens Interferometer At Scanner (PARIS) sensor is responsible for measurement of lens aberrations. The PARIS measurement has a tight timing budget. Currently, the software stack of PARIS runs on a single core PowerPC processor based board and a quadcore SUN M3000 server, which is shared with other components in the machine, making the execution time non-deterministic with a variation of up to 30%. Further, there is a risk that as further enhancements are made, the PARIS software stack will not be able to meet the worst case execution time (WCET) specification. In this thesis we propose a multicore hardware platform such that the execution time of the PARIS software stack is deterministic and is at least reduced to half of the current execution time. This proposal is based on the results of the study conducted on the PARIS software stack to understand the computing needs, the type and the amount of parallelism present in the algorithms. The results show an approximate gain of *9x* for algorithms deployed on the GPU. This results in an approximate *3x* performance gain for the software execution time and *2x* gain in the application performance. However, a GPU based solution requires high investment of time, effort and thus has a high impact on the organization and ASML's product platform. An optimal solution proposed is a platform based on two Intel i7 processors which provides for an approximate *2.4x* performance gain for the software execution time and *1.8x* gain for the entire application with lower impact on the organization and ASML's product platform.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Performance Improvement of Optical Algorithms on Multicore Platforms

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Ratnakar Madan
born in Delhi, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Performance Improvement of Optical Algorithms on Multicore Platforms

by Ratnakar Madan

Abstract

ASML is one of the world's largest suppliers of lithography systems for the semiconductor industry. ASML designs and develops machines that are used to print circuits on silicon wafers, to produce IC chips. These circuits have to be printed with accuracy of upto *2*nm. For this purpose, the machines incorporate several measurement systems. The Parallel Integrated Lens Interferometer At Scanner (PARIS) sensor is responsible for measurement of lens aberrations. The PARIS measurement has a tight timing budget. Currently, the software stack of PARIS runs on a single core PowerPC processor based board and a quadcore SUN M3000 server, which is shared with other components in the machine, making the execution time non-deterministic with a variation of up to 30%. Further, there is a risk that as further enhancements are made, the PARIS software stack will not be able to meet the worst case execution time (WCET) specification. In this thesis we propose a multicore hardware platform such that the execution time of the PARIS software stack is deterministic and is at least reduced to half of the current execution time. This proposal is based on the results of the study conducted on the PARIS software stack to understand the computing needs, the type and the amount of parallelism present in the algorithms. The results show an approximate gain of *9x* for algorithms deployed on the GPU. This results in an approximate *3x* performance gain for the software execution time and *2x* gain in the application performance. However, a GPU based solution requires high investment of time, effort and thus has a high impact on the organization and ASML's product platform. An optimal solution proposed is a platform based on two Intel i7 processors which provides for an approximate *2.4x* performance gain for the software execution time and *1.8x* gain for the entire application with lower impact on the organization and ASML's product platform.

Laboratory : Computer Engineering
Codenumber : CE-MS-2013-16

Committee Members :

Advisor:	Dr. Ir. Zaid Al-Ars, CE, TU Delft
Chairperson:	Prof. Koen Bertels, CE, TU Delft
Member:	Ir. Kees Kotterink, ESD, ASML
Member:	Dr. Ir. Gerard Janssen, WMC, TU Delft

Dedicated to my father whose teachings are a guiding light for me

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

APU Arithmetic Processing Unit

BRAM Block RAM

CCD Charge-Coupled Device

CER Compute Edge Reliability

CLB Configurable Logic Block

CPR Compute Pixel Reliability

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DDR Double Data Rate

DMA Direct Memory Access

DSP Digital Signal Processor

FPGA Field Programmable Gate Array

GPGPU General Purpose GPU

GPU Graphics Processing Unit

HD Horizontal Difference

HDL Hardware Description Language

HER Horizontal Edge Reliability

IC Integrated Circuit

ILIAS Integrated Lens Interferometer At Scanner

IOB Input/Output Block

IP Intellectual Property

LAPACK Linear Algebra Package

LOC Lines Of Code

LUT Look-Up Table

NVVP NVIDIA Visual Profiler

PARIS Parallel ILIAS

PCIe Peripheral Component Interconnect express

PPC PowerPC

RAM Random Access Memory

REMA Reticle Masking

ROI Region Of Interest

RTL Register Transfer Logic

SDRAM Synchronous Dynamic Random Access Memory

SIMD Single Instruction Multiple Data

SM Streaming Multiprocessor

STL Standard Template Library

UV Ultra Violet

VD Vertical Difference

VER Vertical Edge Reliability

VLIW Very Long Instruction Word

WCET Worst Case Execution Time

WS Wafer Stage

ZB Zernike Calculations

ZD Zernike Driver

ZF Zernike Firmware

ZP Zernike Parallel

ZS Zernike Single

Acknowledgements

This Master graduation project would not have been possible without the support of my family and friends and guidance of my supervisors and colleagues whose valuable contributions helped in the completion of this work and to whom I would like to express my heartfelt gratitude.

First of all, I would like to express my deepest gratitude to my supervisor from the University, Dr. Zaid Al-Ars for introducing me to several new concepts before and during this graduation project. His willingness to sit for hours to solve my doubts and a friendly attitude helped me a lot. His critical reviews of my work and encouragement have motivated me to strive to make a plan and follow it rigorously so as to always be on schedule.

I would like to express my sincere indebtedness to my Henk Bodt scholarship and graduation project mentor, Kees Kotterink for firstly, making sure that I get to do the work that I wanted to at ASML and secondly, for his contribution in planning and organizing the project and making sure that I could start the project when I wanted to. I thank him for his expert guidance on my work and scheduling weekly reviews to ensure that I was always clear about my approach and never strayed far from the project goals. He has always been able to take time out from his busy schedule to explain anything I was not clear about for which I am grateful to him.

I am much obliged to Wouter van Heijningen, my supervisor who helped me quickly adapt to the environment and guided me continuously during the project. He shared with me all the minute technical know-how of the project which proved to be very vital and he has been a major contributor to the success of this project.

My colleagues at 7H3; Hans, Diana, Michele, Jori and all the others from the physics group for encouraging me, helping me with my queries and for making my work atmosphere pleasant so that I woke up every day looking forward to going to work.

Finally, I thank my parents for instilling in me the values and traditions of Indian culture. The support extended by my parents, brother and fiance in the times when I had to work continuously and could not give them enough time, is invaluable. Ma, Pa, Bhuji, Soni, this would not have been possible without you.

Ratnakar Madan
Delft, The Netherlands
September 20, 2013

1 Introduction

In the semiconductor industry lithography is an important process that determines the technology node of semiconductor devices. ASML is one of the world's largest suppliers of lithography systems. ASML designs and develops machines that are used in the manufacturing of complex integrated circuits (ICs). These machines accept silicon wafers and print the image of the circuit design on them. A chip is built by stacking up multiple layers of material. Each layer is created using lithography.
In the accelerating world of technology, IC chip manufacturers are required to incorporate more transistors on a chip of the same area to keep pace with Moore's Law [5]. In order to shrink the transistor size with every new generation of IC chips, the lithography machines must be able to print finer features on the wafer. Simultaneously, IC manufacturers require that the lithography machines work faster to produce more wafers per hour, since it translates to a higher IC production rate and correspondingly, an increased profit for them. Consequently, ASML always strives to improve its machine design in terms of accuracy and throughput of wafers.

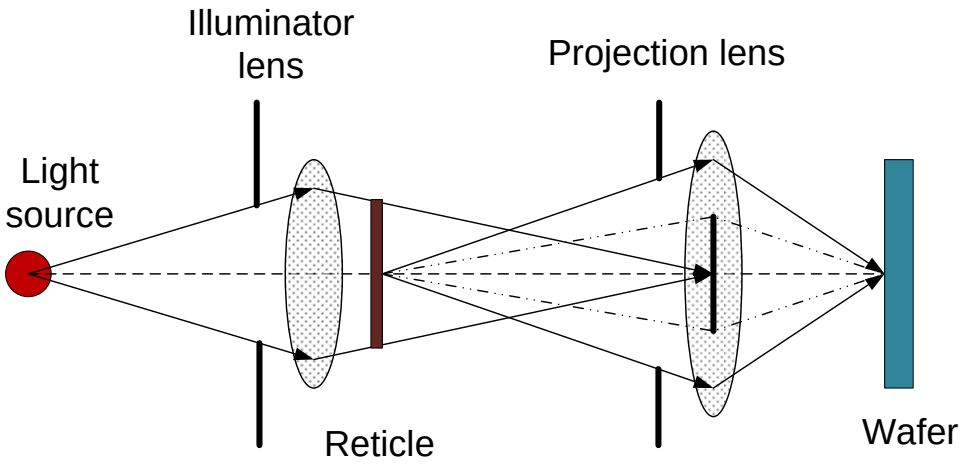

Figure 1.1: An abstract representation of a pattern being printed on a wafer

Explaining in an abstract way, as shown in Fig. 1.1, to print a pattern on the wafer, a beam of laser UV light is passed through the pattern which is to be printed on the wafer, known as reticle. After this, the light passes through a projection lens which projects the pattern on the wafer.
Since the resolution requirement is high and the overlay budget is as low as 2nm for the first few layers, a number of sensors are used to make sure that the requirements of

accuracy are met.
Fig. 1.2 shows how a perfect lens will behave if a light originating from a point source is passed through it. However, in a real lens, small deviations or aberrations occur. This leads to a phase shift of light rays passing through the lens and results in deviated wave front as shown in Fig. 1.3.

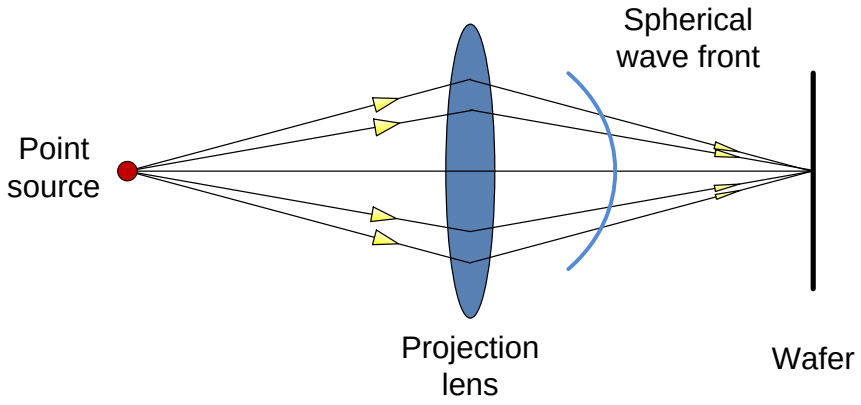

Figure 1.2: A perfect lens producing a spherical wave front

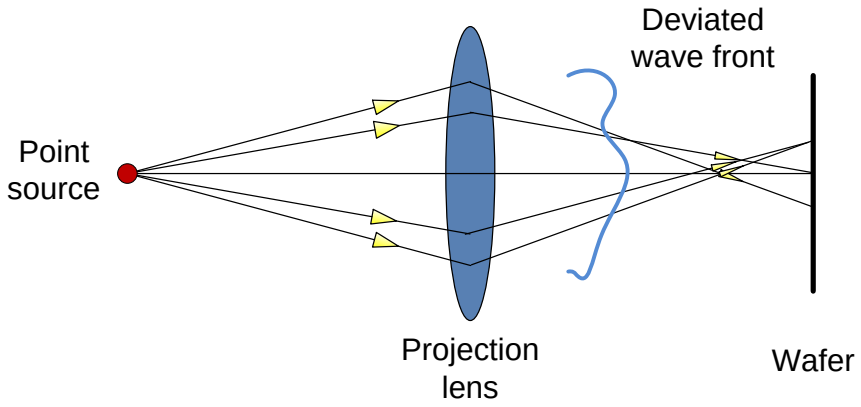

Figure 1.3: A real lens with aberrations producing a deviated wave front

The deviations of the disturbed wave front from the spherical wave front are a measure of the lens aberrations, or errors. These aberrations come partly during the lens manufacturing process, but are mainly due to warm-up of the parts of the lens system during production. The aberrations in turn can be corrected for by tuning the lens settings. If not corrected, the features printed on the wafer are not properly aligned and accurate.

One of the sensors in the machine is capable of measuring the wave front at wafer level before exposing a wafer *lot*[1]. The software stack of this sensor is responsible for controlling the measurement sequence and calculating the measurement results. These results are then used by *Metrology*, which is a client for the sensor software stack to correct the lens settings and minimize the effects of lens aberrations. The sensor measurement results are expressed in Zernike polynomials [6] which are a way to represent the aberrations mathematically.

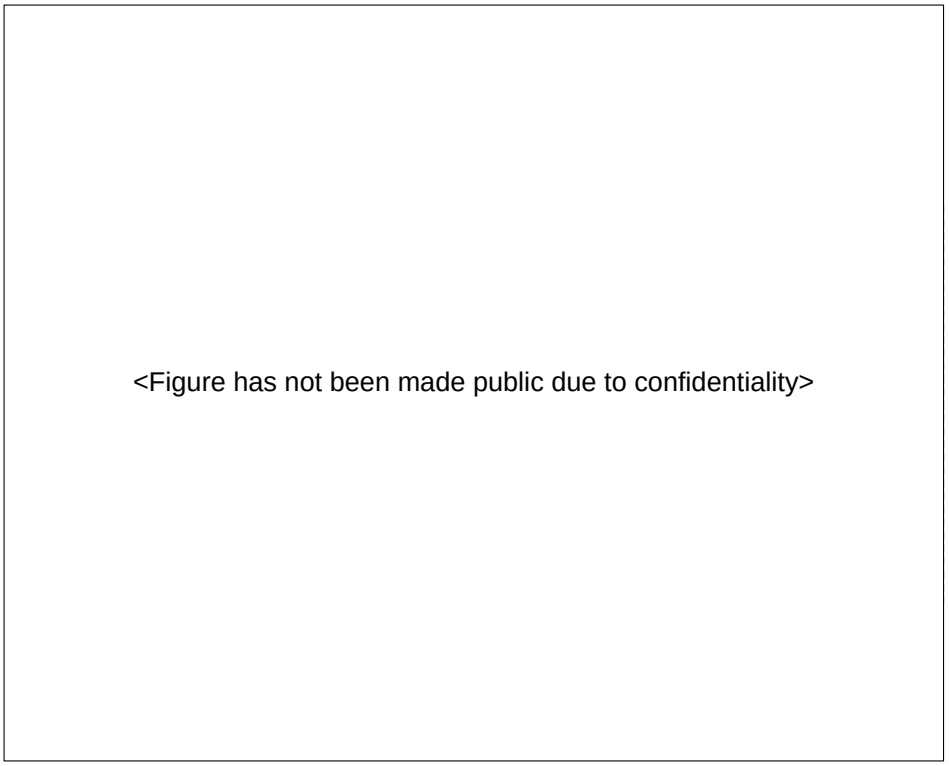

Figure 1.4: Lens aberrations measurement with PARIS sensor

For the next generation system a new sensor, Parallel ILIAS or PARIS sensor has been developed. The PARIS sensor is capable of measuring the lens aberrations before exposing a wafer instead of per lot, as in the case of ILIAS. The PARIS sensor measures seven field points of the wave front in parallel, as shown in Fig. 1.4.

1.1 Problem description

The PARIS sensor measurement and calculation of Zernike polynomial coefficients are an overhead during production of wafers. The throughput of the machine can be increased if this overhead is reduced and hence there is a tight time budget. Currently PARIS measurement is done in 250ms and the Zernike calculation are done in 66ms. Out of

[1] A set of wafers that pass through the machine one after the other

this *66*ms, approximately *40*ms is "post processing software" execution time. Details of the Zernike calculation time are discussed in Section 2.4.
Further improvements and enhancements are regularly being done to the software stack which is always increasing the execution time. Some of the proposed enhancements are to be done on a per-pixel basis and are expected to be computing intensive.
The PARIS sensor software stack currently runs partially on a single core PowerPC based board and partially on a quadcore processor based server by SUN as shown in Fig. 1.4. The SUN server is a shared resource which is also used by other software components of the machine; hence the execution time for the PARIS software stack is unpredictable. There is a risk that software may compete for computing resources in future and it will not be able to meet the Worst Case Execution Time (WCET) specification.

1.2 Motivation

During the past decade, the computing industry has witnessed a paradigm shift from single core to multicore processors. The heat dissipated by high frequency single core processor chips became a problem and increasing the clock frequency for better performance was no longer a viable option. The performance gained by exploiting implicit parallelism was also diminishing. Consequently, the need was felt to move towards multicore processors and exploit explicit parallelism.
A single chip with multiple cores, sharing different architectural components, capable of executing threads in parallel is the industry's answer to keep pace with Moore's law [5]. However, this means a paradigm shift in application development to achieve optimal performance on these new multicore platforms.
The software and algorithm designers have to ensure that enough parallelism exists in the algorithms and the implementation is such that parallel execution is not hindered. A critical problem is establishing which hardware platform should be used to optimally execute the software.

1.3 Project goal

The main objective of this project is to propose a multicore hardware platform that optimally executes the PARIS sensor software stack. The primary requirements of the proposed platform are:

1. Reducing the Zernike calculation time (*41.19*ms) by atleast a factor of two.
2. The execution time of the PARIS sensor software becomes deterministic with milliseconds precision.
3. It has sufficient capacity to handle future enhancements.

Design changes required to optimally execute the PARIS software stack on the multicore platform should be proposed. Moreover, a prototype implementation has to be developed as part of the project, demonstrating the capabilities of the platform and the performance gain that can be achieved by the proposed changes in the PARIS software stack.

1.4 Approach

An initial study will be conducted to understand the PARIS sensor system which includes the functionality of the system, electronic environment of the system, the current hardware platform and the software architecture.
Next, a timing analysis of the software stack will be performed to identify the algorithms with high execution time. A detailed analysis of these algorithms will be done, identifying the reasons which make these algorithms time consuming and strategies to extract parallelism from these algorithms will be proposed.
We then study different hardware platforms under consideration and perform experiments with some of the algorithms of the PARIS sensor software stack on these platforms. An estimation of the performance gain that can be achieved using the proposed parallelization strategies on the multicore platforms will be made. Next we define the comparison metrics and evaluate the hardware platforms on these metrics. Based on the results of this evaluation, an optimal multicore hardware platform will be proposed.
Prototype implementation of the PARIS software stack involved in the Zernike polynomial coefficients will be developed on the proposed multicore platform to demonstrate the capabilities of the platform and the performance will be measured.
The performance on the multicore platform will be evaluated and compared to the initial implementation and to the timing specifications. Causes of any unexpected results will be determined

1.5 Report organization

This section gives an overview of the organization of this thesis. Chapter 2 starts with an explanation of the functionality of the PARIS sensor in detail. It also describes the architecture, components of the software stack, electronics environment of the sensor and the current hardware platform. Then the results of the timing analysis are discussed.
In Chapter 3, the algorithms with high execution time are studied and different strategies to parallelize them are investigated.
In Chapter 4, the different multicore hardware platforms (CPU, GPU and FPGA) under consideration are introduced. The comparison metrics defined to evaluate the hardware platforms are presented and the results of the evaluation are discussed. Chapter 5 provides the details of the prototype implementation including the problems faced, related optimizations done and performance gain achieved.
In Chapter 6 the performance results on different hardware platforms are compared with the baseline. Chapter 7 provides a conclusion to the thesis and recommendations for the future. Appendix A discusses the algorithms used in PARIS software stack and Appendix B presents the reduction algorithm and optimizations which were used in this thesis. Appendix C discusses the calculations and assumptions used in the comparison of performance results.

2 PARIS Sensor

A wafer undergoes two sequences within the ASML lithography machine - *Measure* and *Expose* as shown in Fig. 2.1. In the *Measure* sequence, the wafer position, height and local tilt are measured. On the *Expose* side, aberrations in the projection lens and the amount of light needed are measured after the reticle has been aligned with the wafer. The wafer is then exposed to the image to be printed on it. Due to the requirement of high throughput, both the sequences are expected to be of as small time duration as possible.

The PARIS sensor measurement sequence is part of the *Expose* sequence and is an overhead in production. The client of the PARIS sensor system, the *Metrology* layer, requests the PARIS sensor system to perform a scan and return the Zernike polynomial coefficients. *Metrology* then uses these values to tune the settings of the lens elements in such a way such that the effect of lens aberrations is reduced.
The main motives of introducing PARIS sensor were to increase the accuracy of the pattern printed on the wafer by measuring the lens aberrations on a per-wafer basis and at the same time increase the throughput by reducing the time taken for measurements. While the ILIAS sensor was used to measure lens aberrations per *lot*, PARIS offers fast and accurate aberration measurements that can be done on per-wafer basis which would have not been possible with ILIAS sensor within the time budget.

2.1 Functional overview

A limited functional view and flow of calculations is shown in Fig. 2.2. The details of the sequence of execution and algorithms cannot be made public due to confidentiality.

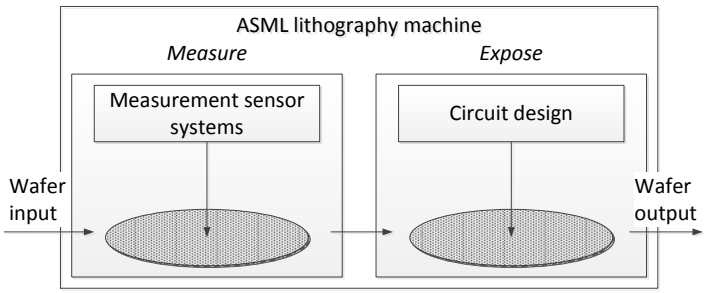

Figure 2.1: *Measure* and *Expose* sequence

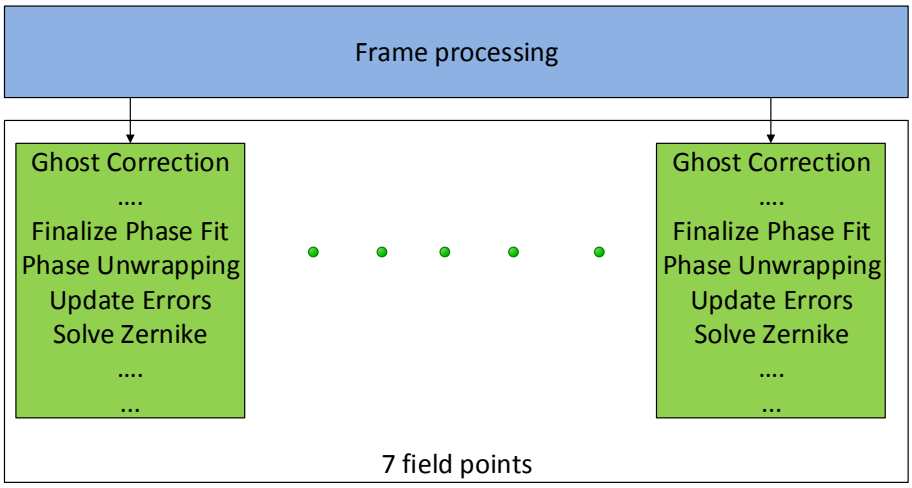

Figure 2.2: Functional overview and flow of calculations

As soon as a frame from the camera is available, there are a number of algorithms which are applied on per-frame basis. These algorithms are a part of the "frame processing" stage. After the frame processing, the frame is split into seven field points and each field points is completely independent from the other and is processed by a separate process. These seven parallel processes are referred to as "PARIS processes" from further onward in this document.

2.2 Software components

The software components involved in the PARIS sensor system are shown in Fig. 2.3. Since most of the components are shared with ILIAS sensor, the relationship is shown as well.

The details of the section cannot be made public due to confidentiality

2.3 Electronic environment

This section introduces the hardware platforms on which the PARIS software stack executes. Fig. 2.4 shows the current electronic environment of the PARIS sensor.

SUN M3000, a quadcore, server platform [7] is a shared computing resource in the machine and is used by different components of the machine. The SUN M3000 has a

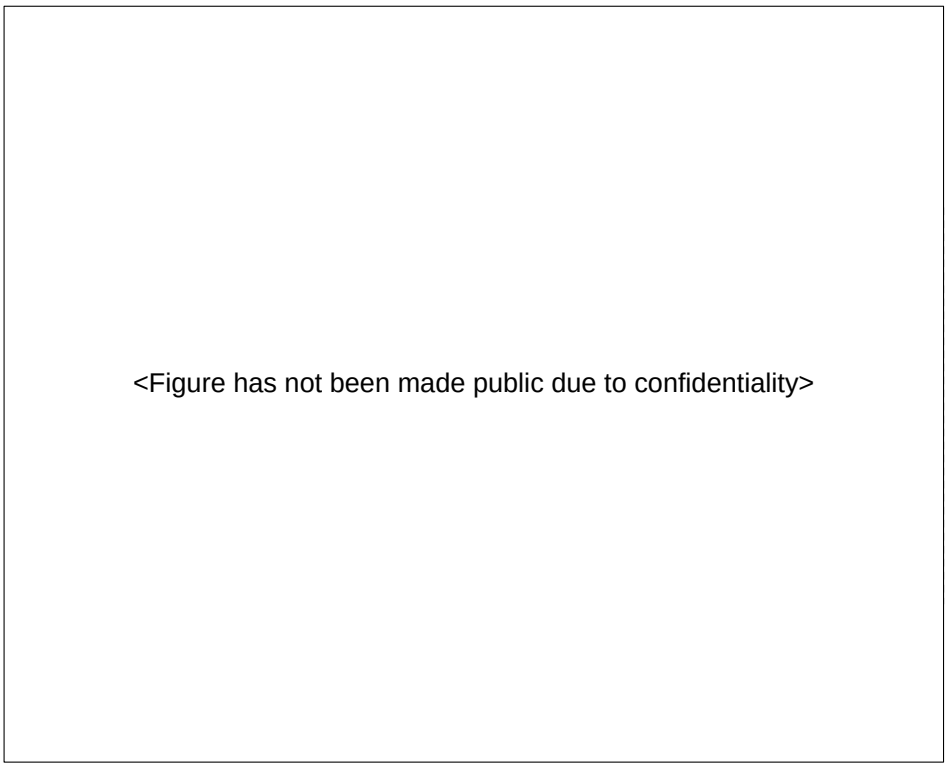

Figure 2.3: PARIS and ILIAS software components

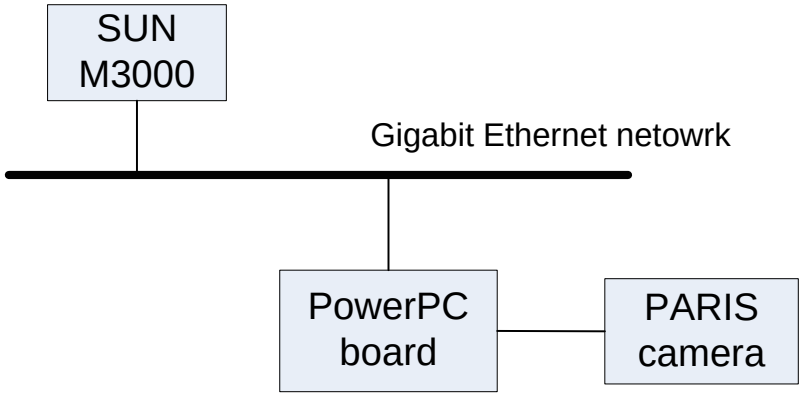

Figure 2.4: PARIS electronic environment

SPARC64 VII+ quad core processor, which allows simultaneous multithreading and is capable of executing two threads per core in parallel. The SUN M3000 runs at a frequency of *2.86*GHz [7].
The driver module for PARIS runs on a PowerPC IBM 750GX microprocessor. It op-

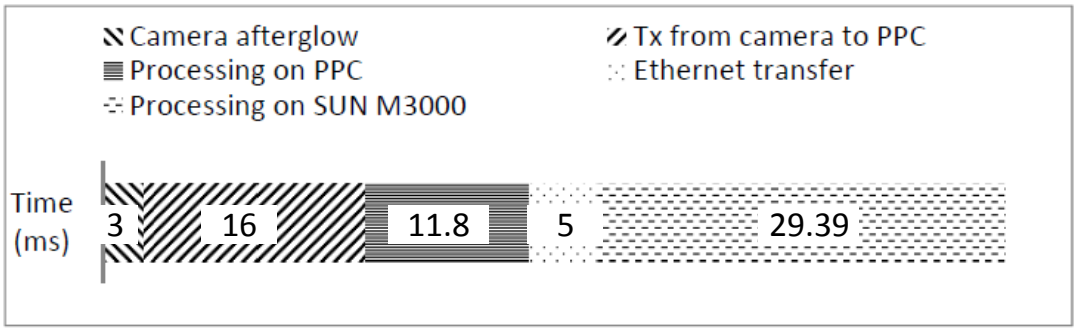

Figure 2.5: Time taken by activities in post processing time budget

erates on *931*MHz frequency. For PARIS, the PPC board is regarded as an advanced frame grabber.
The actual frame size captured by the PARIS sensor camera is calibrated and varies per machine. The PARIS camera sends the frames to the PPC. As soon as PPC receives a frame, it performs signal checks. After this, it sends the frame to SUN M3000 through a Gigabit Ethernet network. This network is shared across the machine.

2.4 Processing time

In this section the time taken by different activities during the time budget is explained. Afterwards, the results of the experiments conducted to measure the execution time are discussed.
The time budget for PARIS includes following activities:

1. Camera afterglow time.
2. Transfer time from the PARIS sensor to PowerPC board
3. Processing time on the PowerPC board
4. Ethernet network transfer time from PowerPC board to SUN M3000
5. Processing time on SUN M3000

The average time taken by the activities included in the post processing budget is shown in Fig. 2.5. The results shown in the figure are from the measurements done on a machine in production context.

The camera afterglow time is fixed (*3*ms) while based on experiments, the transfer time from camera to PPC was found to be *16*ms on average. The processing time on PPC is *11.8*ms on average.
An important fact that must be taken into account is that the measurement of the

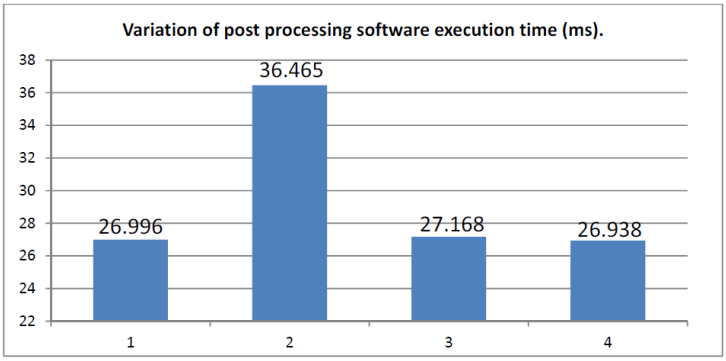

Figure 2.6: Variation of post processing software execution time in four experiments

Ethernet network transfer time is done under minimum load conditions where only the data required for PARIS is being sent over the Ethernet network. As the network is shared across the machine, the performance of the network is not stable and it is not guaranteed that the required bandwidth will be available in production context.
Execution time on SUN M3000 was found to be *29.39*ms on average and the measured worst case execution time was *36.465*ms as can be seen from Fig. 2.6. Since the software executes on SUN M3000 server, which is a shared resource in the machine, the variation in the time measurements is justified as other software components use the SUN M3000 at the same time.
Number of measurements conducted on the machine was limited by access time to the machine. The number of results is insufficient to determine the actual WCET, but it can be concluded from the available results that the execution time on shared SUN M3000 is non-deterministic.
Further, measurements were performed on a dedicated M3000 to get accurate execution time value for the software processing time. On the test bench, execution time was measured to be *28.82*ms on average with a variation of about *3*%. The variation can be attributed to the fact that there are seven processes that are initiated in parallel to work on seven field points. Out of these, each process on average takes *24.56*ms when executed in parallel with all the other six processes.

A detailed investigation showed that there are three algorithms which together contribute *88*% to the average execution time (*24.56*ms) of each process. These modules are “Finalize Phase Fit”,“Phase Unwrapping” and “Solve Zernike”. Rest of the time is taken by frame processing algorithms and other supporting functions used. The average execution time of these algorithms is shown in Fig. 2.7.

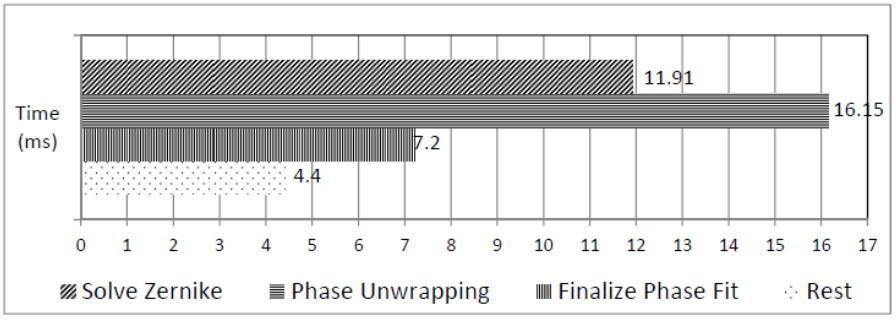

Figure 2.7: Execution time of PARIS algorithms with seven active processes

A detailed study of these algorithms in Chapter 3 identifies the reasons that make these algorithms time consuming. The algorithms are also analyzed for the type and amount of a parallelism and this is also discussed in Chapter 3.

Software Analysis and Parallelism

3

Based on the results of execution time measurement of PARIS sensor system in Chapter 2, this chapter presents a detailed analysis of algorithms that contribute heavily to the post processing execution time. The focus of this analysis is on identifying the parallelism present and proposing different strategies to parallelize these algorithms. In addition, we also analyze the algorithms in terms of branching behavior, precision requirements etc. as this could play an important role in selecting the hardware platform.

The strategy followed is to first focus on the algorithm that takes highest percentage of execution time and then the second highest and so on. In next three sections we discuss these algorithms in detail and present our analysis for each of the algorithms. We conclude this chapter by presenting a summary of our analysis.

To remove the effect of executing seven processes on a four core processor in parallel and to get accurate measurements of the computing need of the algorithms, we disabled six PARIS processes. From our measurements, we identified three algorithms that take *88%* of the software post processing execution time (*18.2*ms when only one PARIS process executes) as shown in Table 3.1. Out of these three algorithms, two are executed one after the other as can be seen in Fig. 3.1. Fig. 3.2 shows the flow of data in between these algorithms.

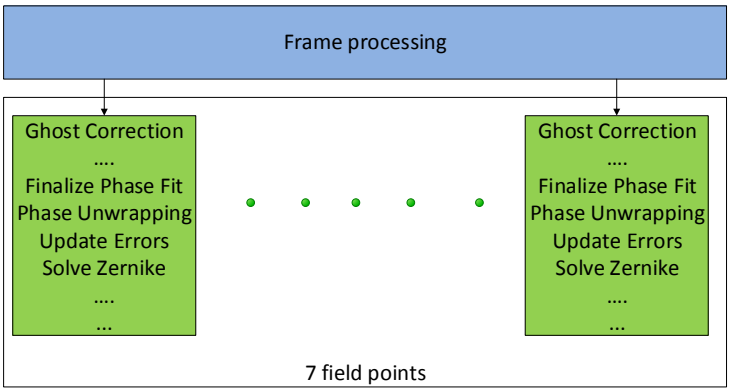

Figure 3.1: Sequence of algorithms

Algorithms	*Baseline execution time (ms)*	*Percentage*
Phase Unwrapping	8.8	48
Solve Zernike	4.6	25
Finalize Phase Fit	2.8	15
Rest	2.0	12

Table 3.1: Execution time of different algorithms when only one PARIS process executes

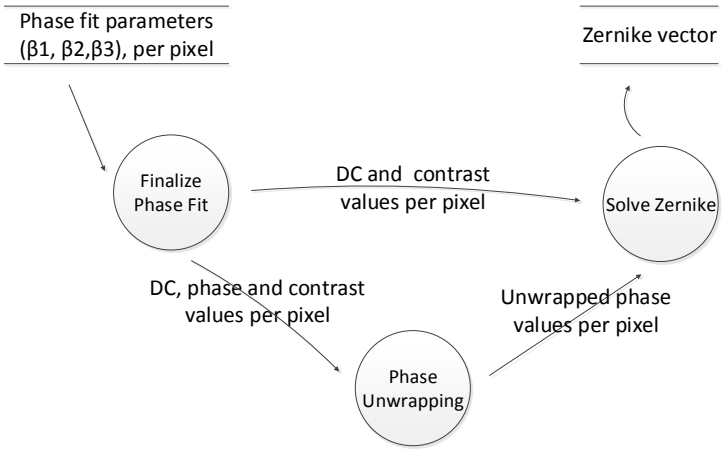

Figure 3.2: Data flow diagram

3.1 Phase unwrapping

From Table 3.1, we can see that phase unwrapping algorithm takes *48%* of the execution time. In this section we first give an introduction of what phase unwrapping is and why is it required in the scan sequence. Then we discuss the algorithm and its implementation. Based on this discussion we present possible strategies to exploit the parallelism in the algorithm. In accordance with our global approach we also analyze precision requirements and branching behavior of the algorithm.

3.1.1 Introduction

It is a common case in interferometry that the phase is calculated by an expression with the arctangent function. This mathematical function returns values that are between -π and +π . Hence the result is given modulo 2π and discontinuities with values near to -π and +π are found in the phase distribution. Unwrapping is the procedure by which these discontinuities are resolved and the result is converted into the desired continuous phase function [9]. This is done by adding or subtracting multiples of 2π to the wrapped phase variation to remove the phase jumps.
Consider an example where the phase varies linearly for a row of pixels in a detector, as shown in Fig. 3.3. When this is demodulated using the arctangent function, it results in wrapped phase with discontinuities and jumps while the physical phase only varies

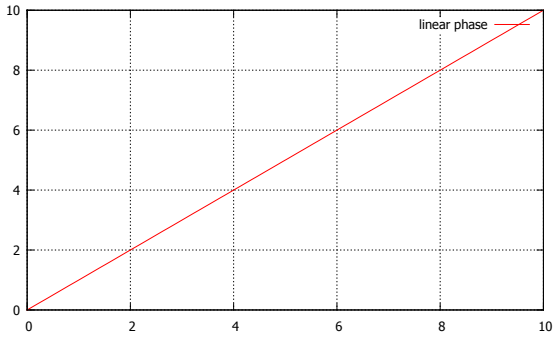

Figure 3.3: Phase varies linearly in a row of pixels

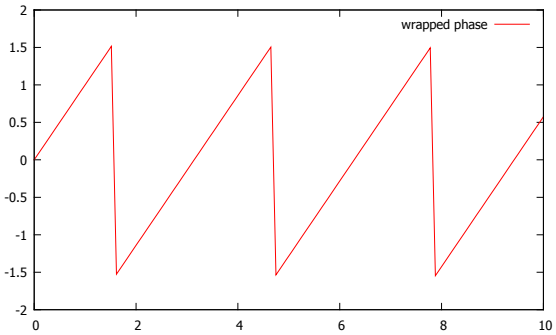

Figure 3.4: Wrapped phase after demodulation

slightly. This effect can be seen in Fig. 3.4.

Thus, essentially a phase unwrapping algorithm must specify a way to detect these discontinuities in the wrapped phase and then remove these discontinuities, resulting in an unwrapped phase. The same concept is used in two dimensional phase unwrapping. Theoretically, phase unwrapping seems a relatively simple problem but, when working with real images which have a lot of noise, procedures have to be designed to deal with many different issues. Discontinuities in the phase wraps, high local variations of signal-to-noise ratios are some of the problems that a phase unwrapping algorithm has to overcome. The algorithm used in PARIS is known as quality guided phase unwrapping.

Quality guided phase unwrapping

In quality guided phase unwrapping algorithm, the unwrapping path is related to the quality of edges. Hence the two main issues in the algorithm are:

1. Choosing the reliability function to determine the quality of each pixel.

2. The design of the unwrapping path.

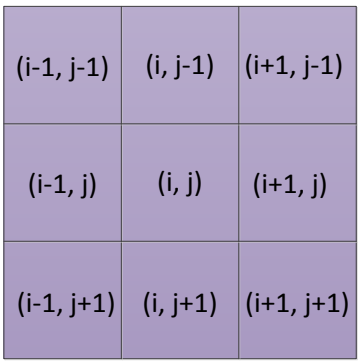

Figure 3.5: A 3 X 3 grid

Reliability function

An edge is an intersection of two pixels that are connected horizontally or vertically. Any pixel can construct an edge with its orthogonal neighboring pixels. Thus the boundary of the grid, in this case is not included in the set of edges. To determine the quality of an edge, first we need to determine the quality of the pixels that constitute the edge.
The criterion to determine the quality of a pixel is based on the phase differences between a pixel and its neighbors. The pixels with low phase difference with respect to their neighbors are considered to be of better quality. By using second differences, a better detection of possible inconsistencies in the phase map can be achieved [9] with minor increase in computations. The calculation of second differences for a pixel in an image is explained with the help of Fig. 3.5 and Eqn. 3.1. To calculate the second difference for a pixel, the phase values of its horizontal and vertical neighbors are required.

$$\begin{aligned} H(i,j) &= \gamma[\phi(i-1,j) - \phi(i,j)] - \gamma[\phi(i,j) - \phi(i+1,j)] \\ V(i,j) &= \gamma[\phi(i,j-1) - \phi(i,j)] - \gamma[\phi(i,j) - \phi(i,j+1)] \end{aligned} \tag{3.1}$$

where $\phi(i,j)$ represents the phase value of (i,j) pixel and $\gamma[]$ operation is explained in Eqn. 3.3.

$$R(i,j) = DC(i,j) * Contrast(i,j) * exp\left(-\frac{16}{4\pi^2}\left(H(i,j)^2 + V(i,j)^2\right)\right) \tag{3.2}$$

where, *H(i,j), V(i,j)* are calculated using Eqn. 3.1.

As in case of first differences, pixels with low second phase difference values are considered to be of better quality. Finally, to determine the reliability of the pixel, DC and contrast values are also used as shown in Eqn. 3.2. Fig. 3.6(A) shows a part of an image whose pixel reliability values are shown.

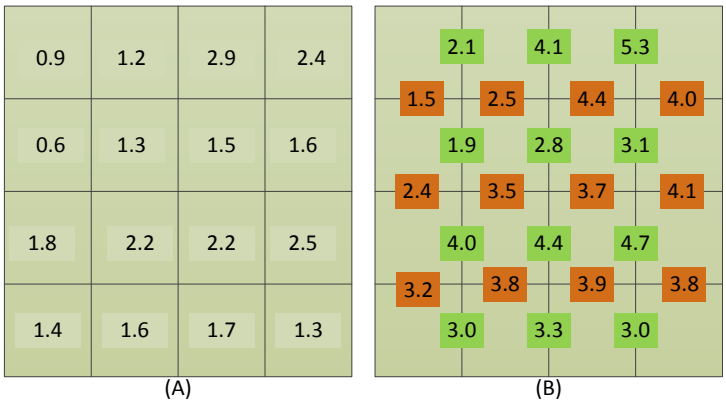

Figure 3.6: Reliability values of pixels (A) and reliability values of edges (B)

The reliability of an edge is computed by adding the reliability values of the pixels that constitute the edge as shown in Fig. 3.6(B) . Finally, reliability of the edges is used in deciding the unwrapping path.

Unwrapping path

Once the reliability of each edge has been computed, the unwrapping path is decided based on the reliability of the edges. The edge with the highest reliability is unwrapped first while the edge with the lowest reliability is unwrapped last.
As can be seen from the flowchart in Fig. 3.7, after computing the reliability of each edge, each pixel is initialized with an invalid group identifier, indicating that it does not belong to any group. Then the pixels constituting the edge with highest reliability are unwrapped to each other and the process of grouping starts. The unwrapping procedure is done by grouping pixels to decrease the effect of low reliability pixels on the unwrapped phase value and to decrease the computation time.

3.1.2 Implementation and analysis

In the current implementation, the algorithm is divided into three functions. First function computes the reliability of each pixel, the second computes reliability of each edge. Then, the process of grouping and unwrapping of pixels starts. The timing measurement of each of these functions is shown in Fig. 3.8.

In this section we briefly explain the implementation of these three functions and then analyze these algorithms for parallelism present, precision requirements etc.

3.1.2.1 Compute Pixel Reliability

As explained in the previous section, the reliability of each pixel is computed using second differences. This is done using a nested loop, traversing the grid in row-major method.

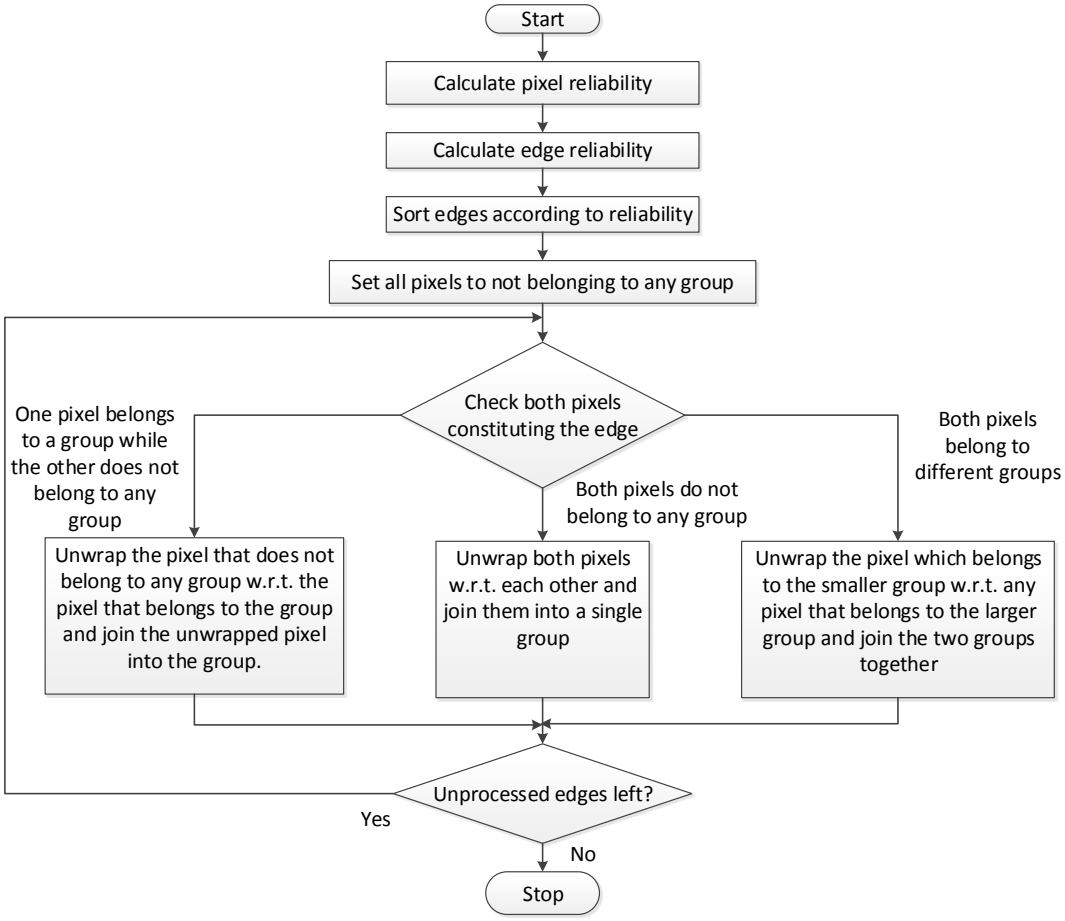

Figure 3.7: Flowchart for phase unwrapping algorithm

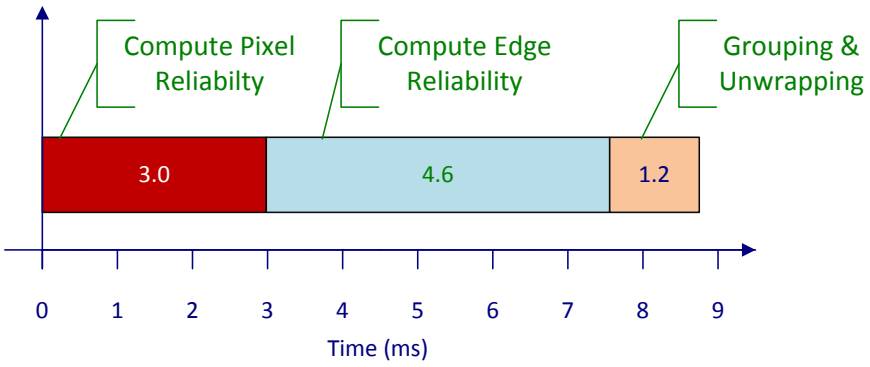

Figure 3.8: Execution time of three functions of phase unwrapping algorithm

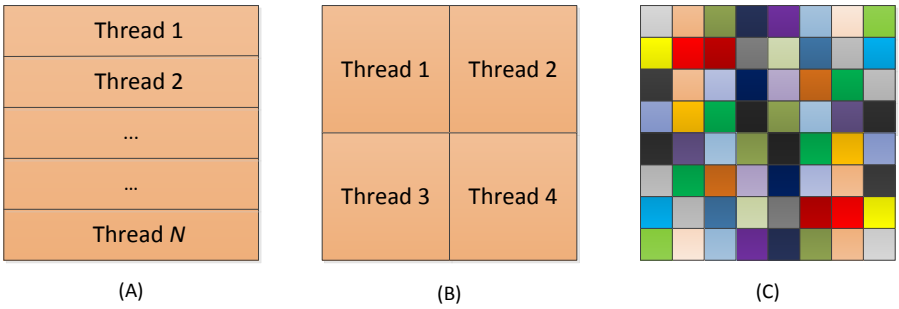

Figure 3.9: Data parallelism, parallel threads process different rows (A), division of data for processing in a grid structure (B) and each pixel is processed by a different thread in parallel (C)

Using the first differences, the horizontal and vertical second differences are computed. They are combined using an exponential function to compute the reliability of each pixel with Eqn. 3.2. The reliability values are then normalized with respect to the highest pixel reliability in the grid.

Parallelism As this algorithm works on the entire grid of pixels and the computations are only dependent on the orthogonal neighboring pixels, a major part of this algorithm is embarrassingly data parallel and all the loop iterations can be performed in parallel. The algorithm can be split into computationally parallel tasks as well where computation of horizontal and vertical second differences can be done in parallel. Hence there could be multiple ways to extract parallelism in this algorithm. Fig. 3.9 and Fig. 3.10 show some of the proposals in which the algorithm can be parallelized.

Fig. 3.9(A) shows that the frame can be split by rows and then for each of the group of rows reliability values can be calculated in parallel. The granularity can be very fine and each row can be processed by a separate thread. The frame can also be split into a grid where each box can be processed by a separate thread as shown in Fig. 3.9(B). The granularity can be as fine as processing each pixel in a separate thread as can be seen in Fig. 3.9(C).

We could also exploit computational parallelism that exists in the algorithm. The horizontal and vertical second differences can be calculated in parallel for the entire grid in separate threads and after that the reliability values can be computed using the results from thread *1* and *2*. This scenario is shown in Fig. 3.10(A).
To combine data and computational parallelism, the computation of reliability values in the previous step can be done in parallel for different parts of the grid as shown in Fig. 3.10(B). All the threads computing the reliability of pixels in different parts of the grid can execute in parallel.

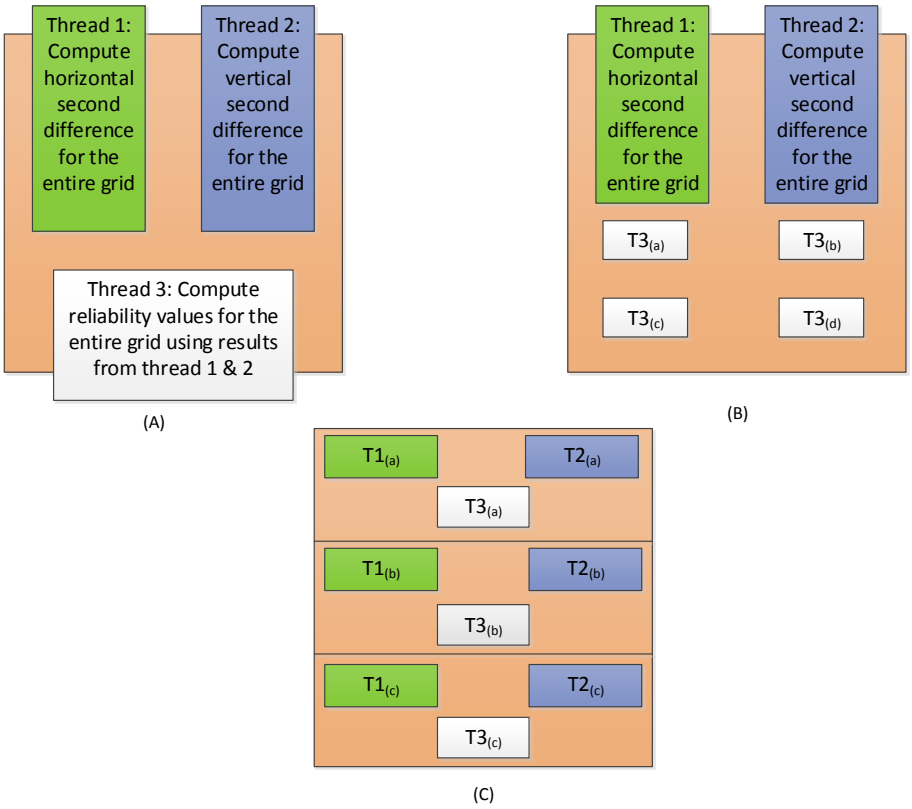

Figure 3.10: Computational parallelism, thread *1* and *2* compute horizontal (HD) and vertical (VD) difference after which thread *3* computes reliability values (A), combination of computational and data parallelism where computation of reliability is data parallel (B) and extension of (B) where HD and VD are also computed in a data parallel way (C).

This can be further extended and the second differences can also be calculated in parallel for different parts of the grid after which reliability can also be calculated in parallel for same parts of the grid as shown in Fig. 3.10(C).

In the same algorithm, pixel with the maximum reliability value is found. The problem is a reduction problem and exhibits coarse grained parallelism. It can be solved using a *parallel divide and conquer approach* to reduce the execution time.
Each thread first finds the pixel with the maximum reliability in its own data set in parallel with other threads and then out of the all local maxima calculated, the global maxima is calculated as shown Fig. 3.11.

Precision requirements In the algorithm, to store the first differences and second differences of phase values of the pixels, double floating point precision variables were

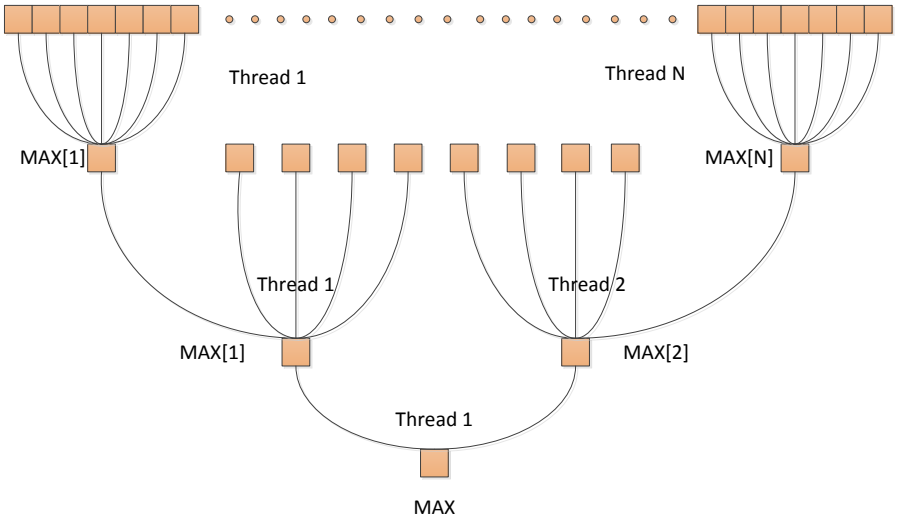

Figure 3.11: A parallel divide and conquer approach to reduction operation

used. In our study we analyzed that this was not required as the phase values itself are stored in single precision.
To compute reliability of pixels, double precision exponential function is used. In our discussions with the physicists, we found that the exponential function was used to get the properties of the exponential curve and the exact values were not necessary. However, the order of reliability values computed was still required to be the same. We concluded that an implementation with single precision was required to keep the order of the reliability values same. Hence instead of a double precision, it is possible to use the single precision exponential function.

Branching behavior In the algorithm, first difference is calculated and Eqn. 3.3 is applied. This constraint introduces *4* branches inside the loop.

$$\begin{aligned} D_1 &= \phi(i,j) - \phi(i+1,j) \\ if abs(D_1) &\geq 0.5, \\ D_1 &= D_1 - floor(D_1 + 0.5) \end{aligned} \tag{3.3}$$

We discussed with the physicists if this could be reduced by removing these constraints from the first differences and applying it on the second differences. This would have reduced the branches to half. However, this change affects the algorithm and potentially the outcome. Functional experiments are needed to qualify such a change.

3.1.2.2 Compute Edge Reliability

To compute the reliability of each edge, again the whole grid is traversed in a row major way with a nested loop, skipping the border pixels. For the inside edges, the reliability

values of the pixels forming the edge are added and the index of the constituting pixels are stored in a structure variable.
After the reliability value of each edge has been calculated, the array of structure is sorted in a descending order of reliability values.

Parallelism The function can be divided into two logical parts:

1. Compute the reliability of all the edges.

2. Sort the edges according to their reliability values.

Computing the reliability of each edge is an embarrassingly parallel algorithm as reliability of each edge depends only on the neighboring pixels. In the implementation, border pixels and edges are skipped and a separate counter is incremented to count the number of edges whose reliability is computed. This makes the iterations of the loop dependent on the previous iteration and hinders the possibility to exploit parallelism in the algorithm.
A possible way to change the algorithm for eliminating this dependency is to use the same counter for edges as used for traversing the pixels. This would imply that the border pixels and edges are not skipped during the traversal. To avoid the effects of this change in the next algorithms, the pixel index and reliability of each border edge should be set to a negative (invalid) value during traversal and this condition must be checked in the next algorithm for invalid edges.
After the above design change in the algorithm, data parallelism can be exploited in this algorithm and Fig. 3.9 is also applicable to this algorithm.
To exploit computational parallelism, reliability values for horizontal and vertical edges can be calculated in parallel in separate threads as shown in Fig. 3.12(A). Further to combine computational and data parallelism the frame can be divided into a grid or by rows and within these parts reliability for horizontal and vertical edges can be calculated in parallel as shown in Fig. 3.12(B).

The second part is sorting of the edges by their reliability values in decreasing order. Sorting operation is, of course not an embarrassingly parallel algorithm. However, a number of approaches have been presented to gain from parallelism in sorting algorithms [10]. We experiment with different parallel approaches to sorting in Chapter 4.

Precision requirements This algorithm does not involve many calculations and all the operations are done using integers. There is nothing here where changing the current precision would reduce the execution time.

Branching behavior From our analysis, we found that there are two conditional branches in the loop whose result can be determined at the compilation time and hence with slight modification to the algorithm, these branches can be removed.

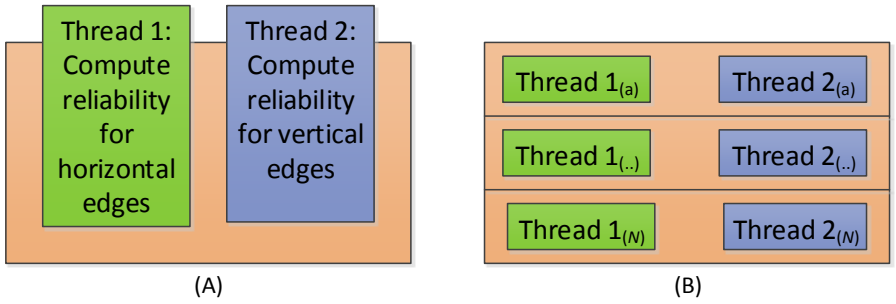

Figure 3.12: (Coarse grained parallelism where two threads compute horizontal (HER) and vertical (VER) edge reliability for the entire grid (A) and fine grained parallelism where n threads compute HER and VER for different parts of the grid (B)

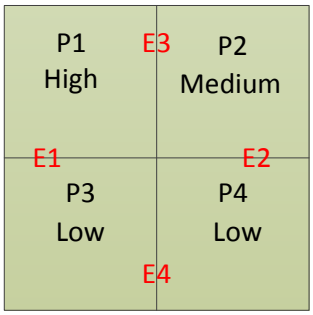

Figure 3.13: A part of the image denoting pixels, edges and their reliability

3.1.2.3 Grouping and Unwrapping

With the reliability of edges calculated, the process of grouping and unwrapping begins. As shown in the flowchart in Fig. 3.7, initially none of the pixels belong to any group. Then, the edges are processed in decreasing order of their reliability values and pixels are grouped and their phase values unwrapped according to the conditions shown in Fig. 3.7.

Parallelism The algorithm here changes the phase value of pixels based on one of the edges that that the pixel has. If two edges which are formed by the same pixel are processed in parallel, the phase value of that pixel would become non-deterministic. Hence it is not possible to exploit parallelism.

For example, if we use two parallel threads to process the sorted list of edges. Then the following case, illustrated with the help of Fig. 3.13 is possible. Since edge E3 would have considerably higher reliability value as compared to edge E2, it is possible that two different threads process edges E3 and E2 at the same time. While thread 1

would be unwrapping pixel P1 and P2 and modify their phase values, thread 2 would be unwrapping pixel P2 and P4 and modifying their pixel values at the same time. This would lead to non-deterministic behavior where phase value of P2 is non-deterministic.

Precision requirements The only calculation done here is the modification of phase value, which is done in single precision and this cannot be changed due to the accuracy requirements. All the other processing in this step of the algorithm is done using integers.

Branching behavior This phase has a lot of single and nested branches because of the nature of the algorithm to classify and group pixels based on their reliability values.

Concluding the analysis of the phase unwrapping algorithm, we would like to say that this algorithm presents an excellent opportunity to reduce execution time by exploiting parallelism. Calculating pixel and edge reliability are embarrassingly parallel while the reduction and sorting operations can be parallelized using a parallel divide and conquer approach. The Grouping and unwrapping procedure is inherently serial and cannot be parallelized but it only takes *1.2*ms and execution time can be reduced by using a faster hardware platform.
Since the precision requirements in phase unwrapping are not critical and single precision calculations suffice, we can efficiently use a broad spectrum of hardware platforms which could further decrease the execution time.
We now discuss the second most time consuming algorithm, the Solve Zernike algorithm.

3.2 Solve Zernike

Solve Zernike algorithm is used to fit a Zernike polynomial curve on the data series. As shown in Fig. 3.2 it computes the Zernike polynomial coefficients with phase, DC and contrast values as inputs.

3.2.1 Introduction

The Solve Zernike function is the largest in terms of LOC out of the three functions analyzed. The function can be divided into three logical steps.

1. Normalize and Remove.
2. Fit Zernike to Data.
3. Calculate Residual and Check Quality.

In this section we explain the algorithm by briefly explaining all the steps.

Normalize and Remove

In this step DC and contrast values are normalized and pixels with very low DC or contrast values are removed from the "valid pixel list". If the total number of rejected pixels is high, the scan is aborted and an error is generated.

Fit Zernike to Data

In this step, Eqn. 3.4 is solved using valid pixels. Here A is the Zernike matrix, G a diagonal matrix with weights, w a vector with measured wave front and z the vector with Zernike coefficients [11].

$$(A^T G A) * z = A^T G * w \tag{3.4}$$

First the right-hand side of this equation is calculated:

$$B = A^T G * w \tag{3.5}$$

It is a straightforward matrix multiplication. In the next step, the left-hand side of Eqn. 3.4 is solved by Cholesky decomposition [12] of the matrix $(A^T GA)$ using the LAPACK numerical library [13]. The result is a vector z with the Zernike coefficients. Next we calculate the residuals using fitted Zernike coefficients.

Calculate Residual and Check Quality

To check the quality of the Zernike fit, the residuals are calculated by subtracting the "calculated wave front values" from the measured wave front values. The residual is calculated over all entries in the vector w using Eqn. 3.6.

$$\text{Residual} = w - A * z \tag{3.6}$$

If the residual for a pixel is higher than a threshold value, then the pixel is added to the "flyer pixel list". If the number of pixels in the list exceeds the allowable limit, then a refit is performed (step 2-3). There is a limit for maximum number of refits which allows for the algorithm to converge. If the residual values are still higher, an error is generated and the scan is aborted.
Fig. 3.14 shows the data flow and dependencies between the three logical steps of the Zernike fit algorithm.

3.2.2 Implementation and analysis

The algorithm is implemented using a lot of sequential steps where the output of a step is input of the next one. The number of iterations of most of the loops is small and not all loops are embarrassingly parallel. In this section we explain different parts of the algorithm from the implementation perspective and present our analysis of these components.

3.2.2.1 Normalize and Remove

Normalizing the DC and contrast values is done in a separate function which takes *700us*. The grid is first divided into regions. The DC and contrast values of the pixels are then normalized. This is done in two separate loops where the first loop is used to calculate the number of pixels and summation of DC and contrast value

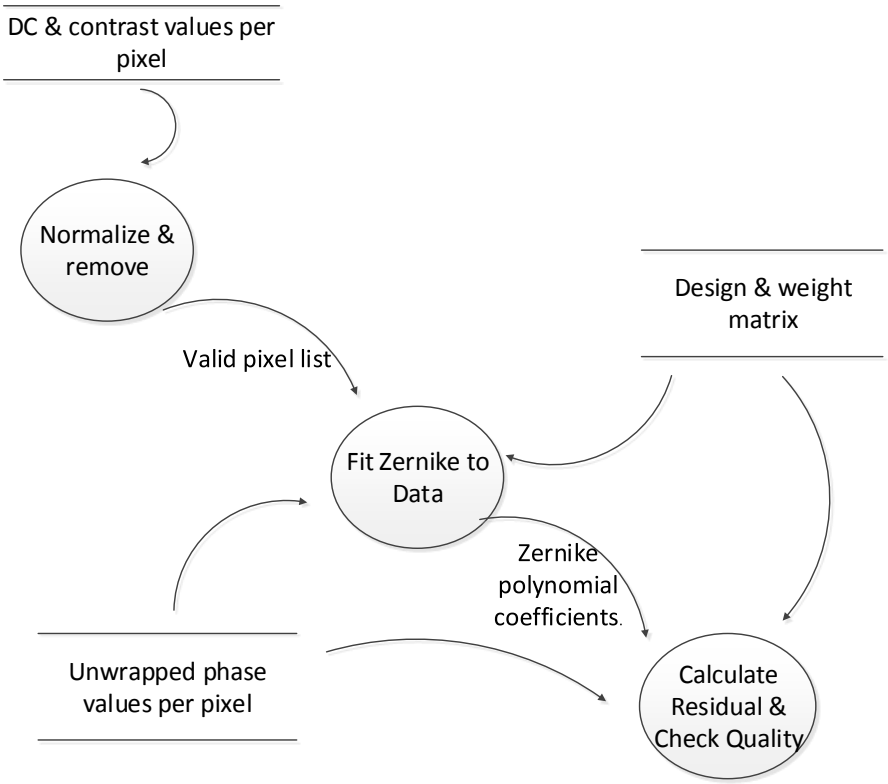

Figure 3.14: Data flow diagram for Solve Zernike algorithm

per-region. The second loop is used for actually normalizing the DC and contrast with respect to the summation values. This process is shown in the flowchart in Fig. 3.15.

Pixels with low DC or contrast values are removed in separate loops for the two shearing directions. Processing in one shear direction is independent of the other shearing direction and can be done in parallel.

Parallelism In the normalization phase, the first loop where the summation is calculated exhibits coarse grained parallelism. The divide and conquer approach can be used to exploit the coarse grained parallelism. In the phase unwrapping algorithm, we proposed a parallel approach in the reduction operation for calculating the maximum value. This is shown in Fig. 3.11. The same approach can be used for computing the summation of all the DC and contrast values.
The second loop which normalizes all the values with respect to the sum is an embarrassingly parallel loop where iterations can be executed independently, in parallel and the approach shown in Fig. 3.9 can be applied here.
During removal phase processing for two shearing directions is done sequentially. However, the operation is independent of the shearing directions and hence the valid pixels

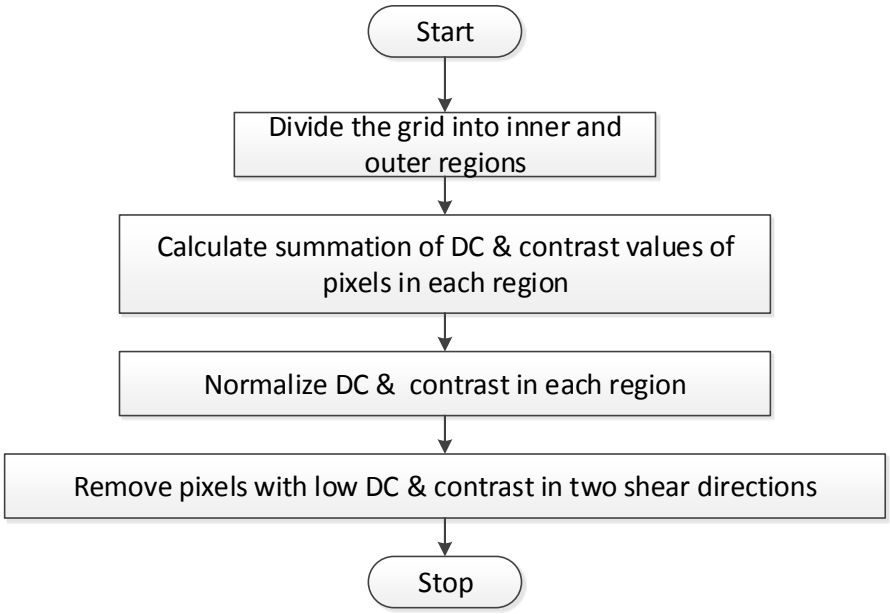

Figure 3.15: Flowchart for Normalize and Remove

list can be changed for both the shearing directions in parallel.

Precision requirements In this function DC and contrast values are used which are stored in single precision, however their sum is stored in double precision and the addition operation hence also becomes a double precision operation. This can be replaced by changing the output variable to a single precision variable.

Branching behavior There are branches in the loop to check for different regions. These are necessary and cannot be reduced.

3.2.2.2 Fit Zernike to Data

In this step, first a loop is used to solve the right side of Eqn. 3.4. According to our measurement, the loop that multiplies the measurement vector (w) with the design matrix (A) takes the longest time in the entire Solve Zernike function.
Next, the equation is solved by using Cholesky decomposition. This is done using an ASML library function. The timing diagram in Fig. 3.16 shows that execution time for this function is low, hence this function is not analyzed further.

Parallelism The loop in which measurement vector (w) is multiplied with the design matrix (A) exhibits coarse grained parallelism which can be exploited. It is again a reduction operation and strategy shown in Fig. 3.17 can be applied here where n

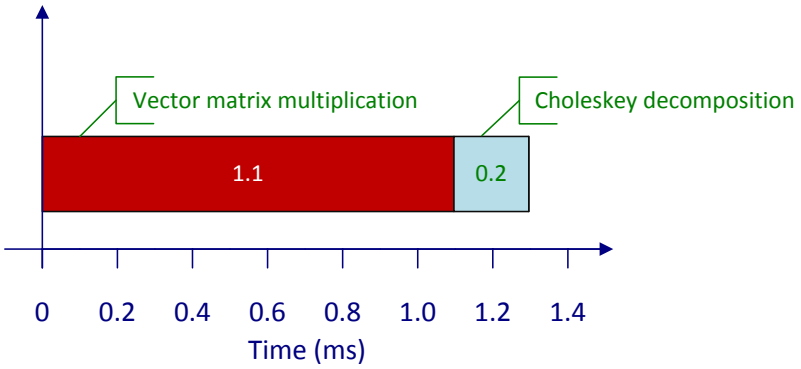

Figure 3.16: Execution time for matrix-vector multiplication and Cholesky decomposition

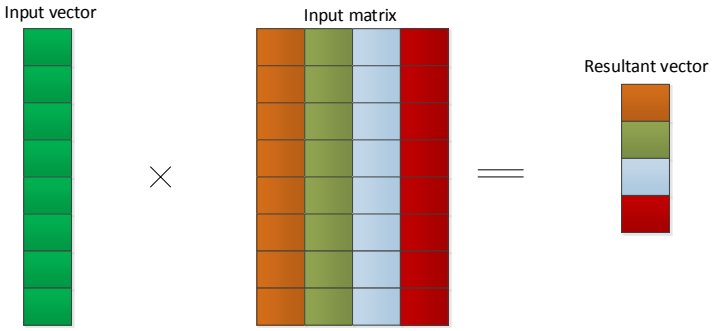

Figure 3.17: Parallel vector-matrix multiplication: Each thread processes a column or a group of columns and computes one or more elements of the resulting vector. Different colors represent different threads processing different columns in the process

threads can calculate different elements of resulting vector in parallel.

Precision requirements The matrix-vector multiplication is done using double precision. Changing the operation to single precision resulted in loss of accuracy and the global requirements of accuracy (0.01nm) were not met. Hence using double precision cannot be avoided here.

Branching behavior There are no unnecessary conditional branches used in the algorithm.

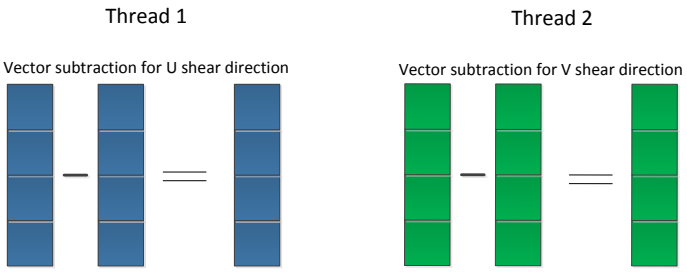

Figure 3.18: Vector subtraction for two shear directions can be done in parallel with two threads

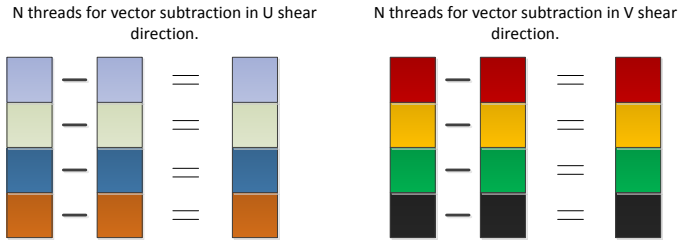

Figure 3.19: Subtraction of two vectors done by 1 thread per vector element for all elements in parallel.

3.2.2.3 Calculate Residual and Check Quality

In this step, the Zernike polynomial coefficient vector (Z) is multiplied to the design matrix (A) as per Eqn. 3.6. In the next step the result is subtracted from the measurement vector using two loops for two shear directions.

Parallelism The matrix vector multiplication operation is a reduction operation. The strategy shown in Fig. 3.17 can also be applied here. The second operation of this algorithm is subtraction of the resultant vector from the measurement vector. Currently, it is done using two separate loops for two shear directions.

These two loops can also execute in parallel as they are independent of each other as shown in Fig. 3.18 .The granularity of parallelism here can go up to per-element operation as shown in Fig. 3.19.

Precision requirements All calculation in this algorithm are done using double precision and they are necessary to meet the accuracy requirements.

Branching behavior There are no unnecessary branches used in the implementation of this algorithm.

Concluding the analysis of Solve Zernike algorithm, we believe that this algorithm would be difficult to parallelize and would still not give as much gain as can be expected in the previous phase unwrapping algorithm. Instead of two big loops in phase unwrapping, there are many small loops in this algorithm which are sequential. The loops except for one, have lower number of iterations in comparison to loops in phase unwrapping algorithm and are also not embarrassingly parallel.
Now, we analyze the algorithm which takes the least baseline execution time out of the three time consuming algorithms identified.

3.3 Finalize Phase Fit

Finalize phase fit is part of the phase fit algorithm. In the Fast Zernike scan sequence, Phase fitting is done to fit a sine curve to the phase values obtained. The process involves constructing a sine curve which best fits the phase values obtained.

3.3.1 Introduction

Curve fitting is the process of constructing a curve, or mathematical function that has the best fit to a series of data points, possibly subject to constraints. Curve fitting can involve either interpolation, where an exact fit to the data is required, or smoothing, in which a "smooth" function is constructed that approximately fits the data.
The curve fitting algorithm used in the implementation consists of three steps:

Initialize Phase Fit Fitting the function exactly to the first three frames.

Update Phase Fit Recursively applying the rest of the frames to the fit.

Finalize Phase Fit Calculating phase from fit results.

This procedure starts with an exact fit of the first three frames, known as interpolation. There are three unknowns $(\beta_1, \beta_2, \beta_3)$ for the function to be fitted in Eqn. 3.7.

$$\text{Intensity} = \beta_1 + \beta_2 * cos(\phi) + \beta_3 * sin(\phi). \tag{3.7}$$

To start, the first three frames for each shear direction need to be collected in memory. The rest of the frames are processed as soon as they become available. After a frame has been processed it is not needed any more and can in principle be overwritten in memory by a new frame.
Then, for fourth and the fifth frame, a smoothing function is used which approximately fits the data.
Finally after the fifth frame has passed through the smoothing function, the DC, phase and contrast values are calculated in the third step using the Eqn. 3.8.

Algorithm	*Baseline execution time (ms)*
Update Phase Fit	0.24
Finalize Phase Fit	2.8

Table 3.2: Execution time of comparison of phase fit algorithms

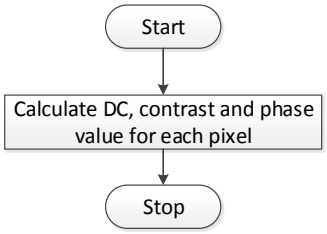

Figure 3.20: Flowchart for Finalize Phase Fit algorithm

$$DC = \beta_1$$

$$Contrast = \left\{ \begin{array}{ll} 0 & if \beta_1 \leq 0 \\ \sqrt{\frac{\beta_2^2+\beta_3^2}{\beta_1}} & if \beta_1 > 0 \end{array} \right\} \tag{3.8}$$

$$Phase = \left\{ \begin{array}{ll} 0 & if \beta_2 and \beta_3 = 0 \\ -atan2(\frac{\beta_3}{\beta_2}) & \end{array} \right\} \tag{3.9}$$

For our analysis, we only need to focus on the processing done on the fifth frame of the second shear direction as it is done during the post processing time budget.

3.3.2 Implementation and analysis

The algorithm is implemented using three functions where each function represents the steps of the algorithm which have been described in the previous section.
Out of the three functions, only Update Phase Fit and Finalize Phase Fit are executed during the post processing time budget. Their execution time is shown in Table 3.2.

Since the execution time of Finalize Phase Fit algorithm is more than ten times the execution time of Update Phase Fit and also contributes *15.38%* (Table 3.1) to the post processing software execution time in our baseline, we focus on analyzing and reducing the execution time of Finalize Phase Fit algorithm.
In this function, the DC, contrast and phase values are calculated for each pixel using the three unknowns which were estimated after the second step of the algorithm. This is done in a loop traversing all pixels in the grid.

While DC values are directly assigned, contrast and phase values are calculated using Eqn. 3.8. Out of all the algorithms that we worked on during this project, this function has the lowest number of (LOC).

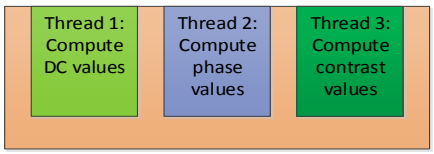

Figure 3.21: Computational parallelism in Finalize Phase Fit algorithm

Parallelism Finalize Phase Fit is an embarrassingly parallel algorithm where calculation of each pixel is completely independent of the other pixels. The algorithm is both embarrassingly data parallel and embarrassingly compute parallel. On each pixel, DC, Phase and contrast can be computed in parallel.
For data parallelism, Fig. 3.9 is also applicable to this algorithm. The possible way to exploit computational parallelism is to calculate DC, phase and contrast values in three separate threads for the entire grid as shown in Fig. 3.21.

Further to combine computational and data parallelism, DC, contrast and phase values can be computed in parallel threads for each pixel. For e.g. each pixel can be processed by *3* threads which each compute the DC, contrast and phase values for that pixel. Total threads that can be used in parallel are three times the number of pixels. However, overhead of creating parallel threads and accessing data from the memory would be the limitation here which is dependent on different hardware platforms.

Precision requirements The computed phase, DC and contrast are stored as single precision values. However, the inputs $\beta_1, \beta_2, \beta_3$ from Eqn. 3.7 are stored as double precision values. The functions used for the calculation of contrast and phase are also double precision functions. There was no defined precision specification for this algorithm and only the precision required in the Zernike polynomial coefficients is defined to be *0.01* nm.
We performed experiments by changing the inputs from double to single precision and also changing the mathematical functions from double to single precision functions. We were able to meet the precision requirements of the final Zernike polynomial coefficients up to *0.0001* nm with the IEEE 754 floating point compliant units.
Hence, it is possible to use floating point calculations in this algorithm with reducing the quality of the result but still meeting the budget.

Branching behavior There are constraints for calculation of phase and contrast as shown in Eqn. 3.8 which introduce branching in the current implementation. We analyzed these branches and concluded that they are indeed required here and cannot be removed.

To conclude, Finalize Phase Fit is an embarrassingly parallel algorithm. It is also the smallest algorithm in terms of lines of code and the execution time is least out of the three

analyzed algorithm. The effort required for parallel implementation for this algorithm would be the least out of the three algorithms.

3.4 Conclusion

After analyzing the three time consuming algorithms, we find that there is sufficient parallelism to exploit and a considerable performance gain on different hardware platforms can be expected.
While Finalize Phase Fit is an embarrassingly parallel algorithm, the Phase Unwrapping algorithm also has some embarrassingly parallel parts. The sorting and reduction problems in Phase Unwrapping can also benefit from a parallel reduction approach.The Phase Unwrapping algorithm also has a step which is sequential in nature and cannot be parallelized. The performance gain here will come from a faster, single hardware core. The Solve Zernike algorithm is complex and has a lot of sequential steps. These steps themselves are loops, with small number of iterations.

Algorithm	*Parallelism*	*Precision required*
Phase Unwrapping	Sufficient parallelism to exploit	Single floating point, with no loss in accuracy
Solve Zernike	Low parallelism, sequential steps	Mostly double precision, single precision in some parts
Finalize Phase Fit	Embarrassingly parallel	Single floating point suffices, double precision preferred for accurate results

Table 3.3: Summary of analysis

Table 3.3 summarizes our analysis for the three algorithms. Since the parallel parts of the algorithms are not continuous and there are some sequential parts in between the algorithms, using an off-chip accelerator would require design choices to be made for algorithm mapping considering the execution time and data transfer time between the processor and the accelerator.
In the next chapter, we focus on different hardware platforms under consideration and discuss how suitable are these analyzed algorithms for execution on multicore and many core hardware platforms and what performance gain could be expected for these algorithms and the entire post processing software stack.

4 Hardware Platforms

Having analyzed the time consuming algorithms in Chapter 3, we now focus on hardware platforms. In this chapter we first discuss about the need of the investigation to find a new multicore hardware platform and expectations from it. Next, we introduce the multicore hardware platforms and different programming models under consideration. Then a discussion is presented on the choice of metrics on which these platforms will be compared. We also rank these metrics according to their importance from the PARIS sensor system development group's perspective.
We compare the hardware platforms under consideration on these metrics. Based on the comparison of hardware platforms on the ranked comparison metrics, we propose a suitable hardware platform and programming tools for the PARIS sensor system.

4.1 Need of a dedicated hardware platform

We introduced the current hardware platform for the PARIS sensor system in Chapter 2 but we now focus on the need of a new hardware platform and some expectations from it. From Fig. 2.6, it is clear that using a shared SUN M3000 for PARIS computations results in variable execution time. Using two separate processing units (PowerPC board and SUN M3000) connected using an Ethernet network which is shared across the machine leads to non-deterministic data transfer time, adding to the variability of the execution time. The combined effect of these two problems leads to a condition where the WCET deadline is not met.
Since the PARIS sensor system is still in development phase and frequent addition of algorithms is done, it is necessary that the execution time of the current algorithms be reduced so that there is enough capacity for future additions.
A solution to this problem could be to use a dedicated hardware platform for the PARIS sensor system which can atleast reduce the execution time of the current algorithms by a factor of two. This solution solves all current problems related to execution time. Since it is a dedicated hardware platform, the execution time can be expected to be deterministic. Using a single platform reduces the amount of data transferred over the Ethernet network. Instead of transferring data for processing (approximately *161*KB per-frame, for ten frames), only the final computed results (approximately *1.75*KB) will be transferred over the Ethernet network. This decreases the load over the network and also the transfer time. This reduction is beneficial not just for the PARIS sensor system but also good for other systems which use the Ethernet network to transfer results of their computation to the central SUN M3000 system.

PARIS sensor application presents parallelism at a high abstraction level where each field point can be processed independently in parallel with other field points. Further, we analyzed that the time consuming algorithms present an opportunity to exploit par-

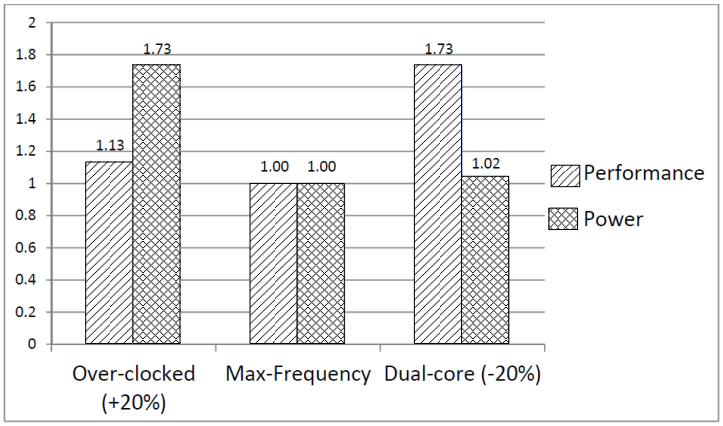

Figure 4.1: Performance increases while frequency decreases in dual-core processor [1]

allelism at a lower abstraction level. Exploiting this parallelism to reduce the execution time is an obvious choice. A platform with at least *seven* cores allows each field point to be processed on a separate core, with no communication required. A platform capable of executing at least *14* threads in parallel allows the processing of each field point to be split up across at least *two* threads.
We now introduce different multicore platforms under consideration for our investigation.

4.2 Hardware platforms under consideration

In the past decade, the computing industry has experienced a paradigm shift from single to multicore processors. This shift was primarily induced by increase in the power consumption of the high frequency single core chips which resulted in high heat dissipation. With increasing the frequency of the chip no longer a viable option, the performance of single core chips was stalling [1].
The multicore design enables two or more cores to run at somewhat slower speeds and at much lower temperatures. The combined throughput of these cores delivers a processing power greater than the maximum available today on single-core processors and at a much lower level of power consumption [1] as shown in Fig. 4.1.

The improvement in performance gained by the use of a multicore processor is highly dependent on the algorithms used and their implementation. In particular, possible gains are limited by the fraction of algorithm that can be run in parallel on multiple cores; this effect is described by the Amdahl's law [14]. In the best case, so-called embarrassingly parallel algorithms may realize speedup factors near the number of cores. Most applications, however, are not accelerated that much. The parallelization of software is a significant on-going topic of research.
With the advent of multicore processors, selecting the optimal hardware platform for the software has become a critical problem. There are no well laid rules to select a hardware platform based on the characteristics of the application.

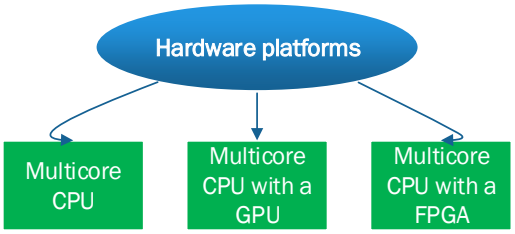

Figure 4.2: Hardware platforms under consideration

In this section we introduce all the hardware platforms under consideration (shown in Fig. 4.2) and discuss their architecture, libraries and standards that are used to program them.

4.2.1 Multicore CPU

A multicore processor implements multiprocessing in a single physical package. Designers may couple cores in a multicore device tightly or loosely. Tightly coupled cores share more resources than loosely coupled cores. For example, cores may or may not share caches and they may implement message passing or shared memory inter-core communication methods.

Multicore systems deliver benefits to multi-threaded programs. The only way to exploit the available CPU cores efficiently is through parallelism. So far, parallelism is mainly being used by operating systems at the process level to provide a seamless multi-tasking, multiprocessing experience. On the application-development side, thread-based concurrent programming is the predominant mechanism for implementing parallelism. In May 2007, Intel fellow, Shekhar Borkar stated that "The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years" [15]. Since the number of cores in a processor is set to increase to keep with Moore's Law, it only makes sense that the software running on these cores takes this into account [16].

Programming Models

To develop multithreaded applications a number of programming models, libraries and standards have been developed. Some of these are OpenMP [17], Pthreads [18] and other proprietary libraries like Intel thread building blocks (Intel TBB) [19]. For this thesis, the OpenMP and Pthreads libraries were considered for implementation. Both the libraries have their own positives and negatives and there are a number of articles comparing the two [20, 21]. Based on the comparison in the articles and analysis of the algorithms in Chapter 3, it was decided that if multicore CPUs are to be used as a platform for PARIS, then OpenMP would be used to exploit the parallelism.
To utilize multiple-cores or processors, it is needed that some parts of the code are

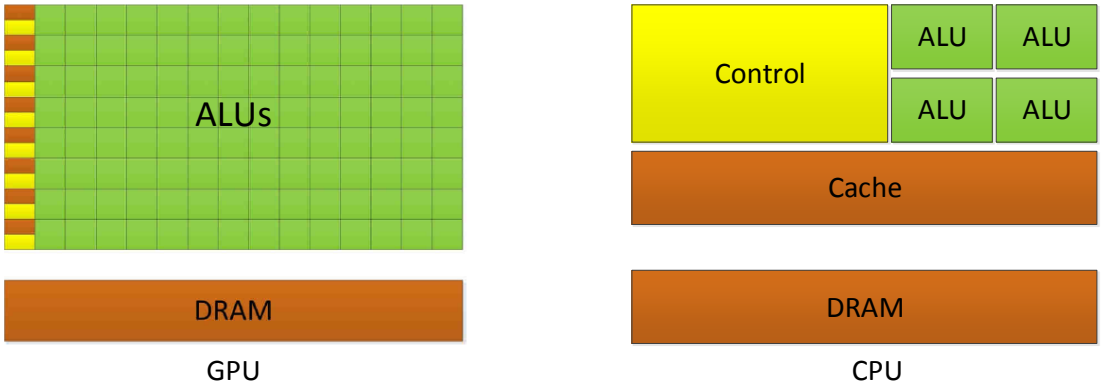

Figure 4.3: Comparison between GPU and CPU design strategy.

written to be executed in parallel. This is usually done by creating multiple threads and assigning each thread to a part of the application to be executed. The OpenMP API covers user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations do not check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. Following is a basic example, to show the usage of an OpenMP API to parallelize a for loop.

```
1 #pragma omp parallel for
2 for(int i = 0;i < 512; i++)
3 {
4       array[i] = array[i] * array[i];
5 }
```

In this example the directive *# pragma omp parallel for* gives a message to the compiler that the content within the brackets can be executed in parallel. The content in the for-loop will then be compiled in such way that it will be executed in parallel.

4.2.2 Graphics Processing Unit

The term GPU was popularized by Nvidia in *1999*, who marketed the GeForce *256* as "the world's first GPU". GPUs were developed with the intention to off load the CPU of tasks typical to graphical, image and video processing and execute them on the GPU. A GPU is tailored for compute intensive, highly parallel operations and for this reason; GPUs have many parallel execution units and lower clock speed. GPUs have faster and advanced memory interfaces as they need to shift around a lot more data than CPUs.

The GPU and CPU design strategy are compared in Fig. 4.3. The CPUs dedicate chip area and transistors to execute one or a few threads as fast as possible by providing

Figure 4.4: CPU - GPU connection via PCIe

an intelligent control unit with speculative, out of order execution, a faster on chip memory for low data access latency etc. The GPUs, on the contrary use the high transistor count to make the workload (as many threads as possible) run as fast as possible.
At the level of abstraction in Fig. 4.3, the GPU looks like sea of computational units with only a few support elements an illustration of the key GPU design goal, which is to maximize floating-point throughput. Since most of the circuitry within each core is dedicated to computation, rather than speculative features meant to enhance single-threaded performance, most of the die area and power consumed by a GPU goes into the application's actual algorithmic work.

GPU computing is not meant to replace CPU computing. Each approach has advantages for certain kinds of software. CPUs are optimized for applications where most of the work is being done by a limited number of threads, especially where the threads exhibit high data locality, a mix of different operations, and a high percentage of conditional branches.
GPU design aims at the other end of the spectrum: applications with multiple threads that are dominated by longer sequences of computational instructions. Over the last few years, GPUs have become much better at thread handling, data caching, virtual memory management, flow control, and other CPU-like features, but the distinction between computationally intensive software and control-flow intensive software is fundamental.
The GPU available for experimentation during this thesis is based on the Fermi architecture [22] by NVIDIA. In a personal computer, a GPU can be present on a video card and connected to the CPU via PCIe as shown in Fig. 4.4 or in certain CPUs, on the CPU die (Intel Integrated Graphics, AMD APUs).

Programming Models

Efforts to exploit the GPU for non-graphical applications have been underway since *2003*. GPUs were programmed using high-level shading languages such as DirectX [23], OpenGL [24]. These early efforts that used graphics APIs for general purpose computing were known as GPGPU programs.
While the GPGPU model demonstrated spectacular performance improvements, it faced several problems. The programmer had to possess intimate knowledge of graphics APIs and GPU architecture which made programming the GPU a very complex task. Basic programming features such as random reads and writes to memory, double precision

calculations, were not supported. This meant that most scientific applications could not be run on the GPU.
To address these problems, NVIDIA introduced CUDA [4] which enabled the GPU to be programmed with a variety of high level languages. Instead of programming dedicated graphics units with graphics APIs, the programmer could now write C programs with CUDA extensions and target a general purpose, massively parallel processor. Different standards have been developed since then to program the GPUs including OpenCL and OpenACC.
An open standard, OpenCL [25] can be used for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs and other processors. It includes a language for writing kernels (functions that execute on OpenCL devices) and APIs that are used to define and then control the platforms. OpenCL can be used to give an application access to a GPU for non-graphical computing.

OpenACC is another open programming standard for parallel computing which has been developed by Cray, Nvidia and PGI [26]. It is designed to simplify programming of heterogeneous CPU/GPU systems [27].
Similar to OpenMP, the programmer can annotate C, C++ source code to identify the areas that should be accelerated using PRAGMA compiler directives and additional functions. Unlike OpenMP, code can be started not only on the CPU, but also on the GPU. This makes programming the GPU easier as the programmer does not have to worry about explicit memory transfers.
However, OpenACC is new and the drivers for the GPU are not yet mature. The ease in programmability comes at a cost of execution time performance [28]. In a real time system like PARIS, where performance is a high priority, OpenACC, in its current state of development cannot be used.
An important capability which was required to implement PARIS sensor system algorithms on a GPU was the asynchronous execution on CPU and GPU, such that data transfer overhead between the two could be minimized. This is not yet efficiently implemented in OpenCL [29]. Hence it is proposed that if GPUs are chosen as a hardware platform for the PARIS sensor system, then CUDA would be used for development. An implication of using CUDA is that we become dependent on NVIDA as a sole developer of GPUs that support CUDA. Until OpenCL implements asynchronous execution features efficiently, this is the only possible choice.
In NVIDIA's CUDA software platform, the computational elements of algorithms which execute on the GPU are known as *kernels*. These kernels can blend with the existing code and only the code executing on the GPU and related CUDA API calls need to be added. An application or library function may consist of one or more kernels. Kernels can be written in the C language extended with additional keywords to express parallelism directly rather than through the usual looping constructs.
Once compiled, kernels consist of many *threads* that execute the same operation in parallel: one thread is like an iteration of a loop. Multiple threads are grouped into *thread blocks*. All of the threads in a thread block can cooperate and share memory as shown in Fig. 4.5.
Thread blocks can coordinate the use of global shared memory among them but may

execute in any order, concurrently or sequentially. We present a simple example of how a CUDA kernel which finds the square of the elements in an array is called.

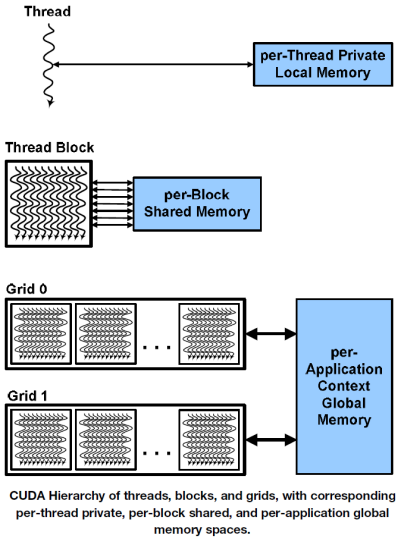

Figure 4.5: CUDA thread hierarchy. [2]

```
1 int elements = 512; // number of elements
2 int threads = 32; // number of threads, to be defined by the programmer
3 float *C_array, *G_array; // pointers for array on CPU and GPU
4 ..... // allocate memory on CPU and GPU
5 ..... // transfer array elements to GPU
6 cudaSqKernel<<elements/threads, threads>>(G_array) // CUDA kernel called
7 ..... // transfer result from GPU to CPU
8 ..... // free memory
```

```
1 // This example assumes that the array elements have been transferred
      explicitly to the GPU memory.
2 __global__ void cudaSqKernel (float *array)
3 {
4   int i = blockIdx.x * blockDim.x + threadIdx.x; // indexer defined with
        internal CUDA variables
5   array[i] = array[i] * array[i]; // square operation
6 }
```

4.2.3 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. There are several manufacturers that produce FPGAs. A couple of these are Xilinx and Altera. Each of these manufacturers have their own architectural implementation, but the basics are the same. This thesis will refer to the architectural implementation of Xilinx, because

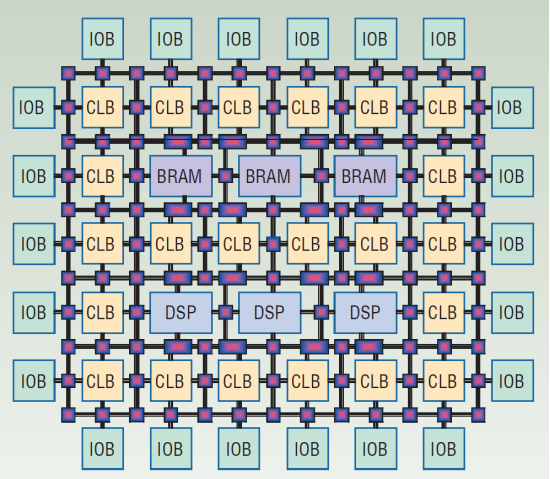

Figure 4.6: FPGA internal structure based on the Xilinx architecture style [3]

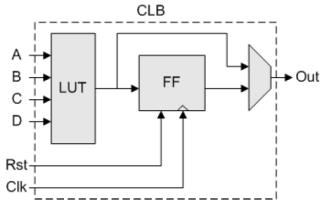

Figure 4.7: Internal structure of a CLB [3]

the Xilinx Vivado, discussed in this section, is used in this thesis for experiments. The basics of an FPGA architecture are shown in Fig. 4.6 [3].

An FPGA is a semiconductor device consisting of configurable logic blocks (CLBs), interconnects, and input output blocks (IOBs) that allow implementing complex digital circuits. On the outside of the FPGA the IOBs form a ring for connection I/O pins that are situated on the exterior of the FPGA. Inside this ring lies a rectangular array of logic blocks.As shown in Fig. 4.7, a typical FPGA logic block consists of a lookup table (LUT) and a flip-flop. Modern FPGA devices also include higher-level functionality such as Digital Signal Processing (DSP), high-speed IOBs, embedded memories (BRAM) and embedded processors. The programmable interconnect wires are required to connect CLBs to other CLBs and CLBs to IOBs.
Modern FPGAs consist of tens of thousands of CLBs and a large programmable interconnection network.

FPGAs can be reprogrammed many times to perform a different function. For programming the FPGA, code written in a Hardware Description Language (HDL) is used, for example VHDL [30]. With the inheritance of speed and parallelism from a hardware

solution, FPGA-based co-processors are used to execute compute intensive tasks while maintaining the flexibility of a software programmable solution.
For the PARIS development team, it was necessary to keep abstraction level high and not to focus on hardware development. The team that develops PARIS has little or no knowledge about hardware development languages and FPGAs. Migrating from current PARIS software code to VHDL would also be time consuming. Using FPGAs was only possible with a tool which could translate C code to VHDL or if, using VHDL development with FPGAs was the only possible option and no other hardware platforms were found suitable.

Xilinx Vivado Design Suite

The Vivado Design Suite is a system-centric design environment that accelerates IP development and integration. It enables designs to be created easily, meet timing more quickly, and automate the developer's preferred design flow [31].

Vivado accelerates design implementation and verification by enabling C, C++ specifications to be directly synthesized into VHDL or Verilog, after exploring a multitude of micro-architectures based on design requirements. Functional simulation can be performed in C, providing order of magnitude acceleration over VHDL or Verilog simulation. This provides a faster and more robust way of delivering quality designs [31].

The keys benefits of Vivado are:

- Integrated Eclipse C Development Tool environment to specify, compile, simulate, and debug C/C++/SystemC
- Automatic extraction of parallelism at instruction level and task level
- Automated verification through co-simulation with original C-based test bench

We did experiments with the Vivado and started with the Finalize Phase Fit algorithm. The reasons for choosing this algorithm were:

- Smallest algorithm in terms of LOC.
- Embarrassingly parallel.

In our experiments, we were not able to directly generate the parallel implementation of the algorithm because:

- The algorithm uses arctangent function of the math library. The current version of Vivado (2012.2) does not support it.
- The C to VHDL translator fails to parallelize the algorithm if there is a mathematical operation being performed inside a conditional branch which is in the parallel algorithm.

Although there were workarounds to these problems, implementing these workarounds in PARIS software stack would be a laborious task. Besides a number of math library functions, other functions such as sorting and reduction which are directly available with Standard Template Library (STL) are not yet supported by Vivado.

These functions are needed in abundance in the measurement systems group because of the nature and application of the algorithms.
Hence, after a discussion with the PARIS development team representative, we concluded that in the current level of maturity, it is not possible to use tools like Vivado. However there is definite potential for future development in such technologies.

Since traditional HDL development would have a high organizational impact and would require too much effort for the PARIS development team and Vivado cannot be used, we do not consider the possibility of using a FPGA based hardware platform in further analysis.

4.3 Comparison metrics

To compare the hardware platforms under consideration for the PARIS sensor system, we defined and ranked a set of comparison metrics. These comparison metrics include not only quantifiable metrics such as application speedup, precision requirements, cost etc. but also include on non-quantifiable metrics (within the scope of this thesis) which are important for the PARIS development team. We present these metrics in order of their importance.

1. *Application speedup*
   Application speedup was considered to be a highest priority metric. Speeding up the PARIS sensor application on the dedicated platform would mean that there is enough computational budget for future enhancements and addition of computation algorithms to the sensor application.

2. *Execution time variation*
   In Chapter 2, we showed with the measurements on the machine that using shared computing resources for PARIS resulted in high variation of execution time. This is a major problem in a real-time system such as PARIS. Hence execution time variation is a very important metric and it is expected from the proposed hardware platform that the execution time is deterministic within milliseconds precision.

3. *Precision requirements*
   We know that the final precision required in Zernike polynomial coefficients is *0.001*nm. In Chapter 3, we discussed that some of the algorithms in the PARIS software stack require calculations to be performed in double precision. Hence, it is important that the proposed hardware platform is able to provide support for double precision calculations without a high penalty in the execution time performance.

4. *Development time and cost*
   The time and cost needed to make the changes in the software are also an issue of concern. Since the PARIS sensor system is still in its early days, constant modifications are made to the algorithms. Sometimes, new algorithms are added to the execution sequence. The costs incurred and time spent in maintenance and iterative development is thus also an important metric.

5. *Match with competencies*
   The current team working on the PARIS sensor has a good understanding of the PARIS sensor system. It is important that the team is able to work with the new hardware platform without too much effort needed to understand the complexities of the new platform and development models.

6. *Maturity of tools*
   The platform proposed for the PARIS sensor system should be well supported and the tools used for development should be mature. Mature tools with good support will reduce the development time required. Thus, it is important that the tools used for PARIS sensor software are developer friendly and mature.

7. *Manufacturers and Vendors*
   A preference for any organization is that it should not be dependent on one manufacturer for the hardware platform. Since the development efforts based on the platform will have associated cost and time, we would not want to be in a situation where the hardware platform is no longer supported/available by the manufacturer while ASML still needs it for their machines. Thus we would like to have clear knowledge about the lifetime of the platform, road-map for the future etc.

8. *Cost of platform and tools*
   ASML is a low volume manufacturing company and the PARIS system is a small part of the manufacturing cost. The actual cost of the hardware platform (within comparable limits) is of least importance to ASML out of the concerned metrics.

4.4 Hardware platform comparison

In this section, we present the comparison of the hardware platforms based on the metrics defined in the previous section. The details and results of the experiments conducted to measure the quantifiable metrics are also presented within the comparison.

1. *Application speedup*
   Since Application speedup is a high priority metric, we conducted a number of experiments to estimate and predict the speedup that would be possible by deploying the PARIS sensor application on different hardware platforms. The goal of these experiments was not only identifying a particular hardware platform but, finding out how the application speedup scales up with increasing the number of cores/ threads etc. and compare it with theoretical estimates. We share some of the results in this section.
   To do these experiments, we separated the three most time consuming algorithms from rest of the application which were identified in Chapter 3. These in total constitute *88*% of the execution time and provide sufficient representation of the post processing software. Separating these algorithms would make the experimentation less time consuming.
   Since the seven PARIS processes are completely independent of each other and require no communication in between them, we decided to experiment with only

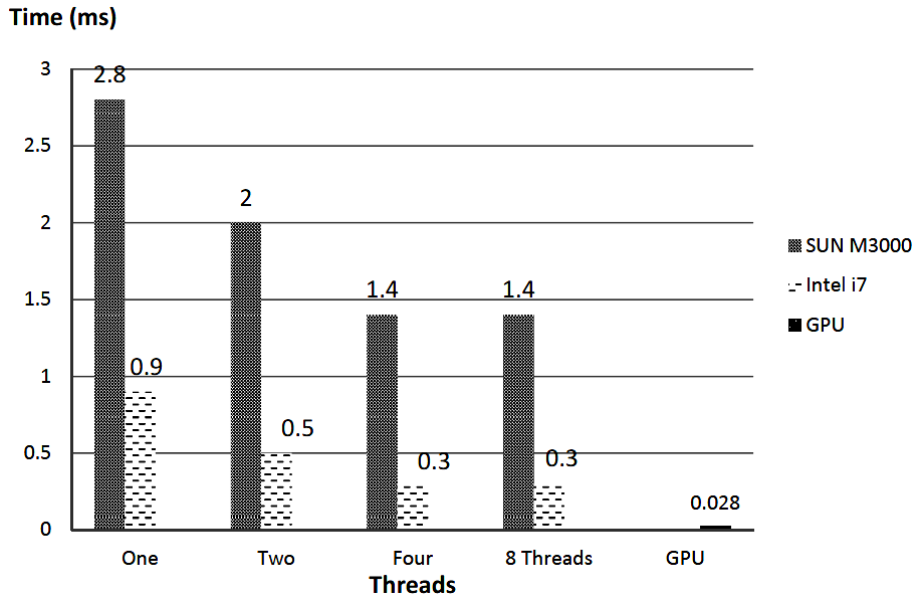

Figure 4.8: Finalize Phase Fit algorithm on different hardware platforms.

one PARIS process at this stage. If all seven PARIS processes would execute, the overhead of executing seven processes on a platform with less number of cores would dominate. To get the real computation time for the algorithms and estimate the number of cores required for optimal execution, this was necessary.
To consider the possibility of exploiting parallelism, within one PARIS process, the above step was necessary because of the limited number of cores available in the experimental multicore CPU platform available. In Chapter 6 we also do experiments with two and four PARIS processes executing on the multicore CPU.

Experiment 1: Embarrassingly parallel algorithm
The first experiment we conducted was with the Finalize Phase Fit algorithm as it represents the category of embarrassingly parallel, computation intensive algorithms. For the CPU, the implementation was straightforward and required only the insertion of OpenMP pragmas to experiment with parallelism. The GPU implementation required the use of CUDA APIs to first transfer the input array to GPU and then let the CUDA cores process the data in parallel and then transfer the results back to CPU.

The results of this experiment are shown in the graph in Fig. 4.8. The baseline execution time for the algorithm on SUN M3000 is *2.8*ms. With two threads, the execution time reduces to *2*ms. On Intel i7, the single thread implementation is about three times faster than the implementation on SUN M3000.

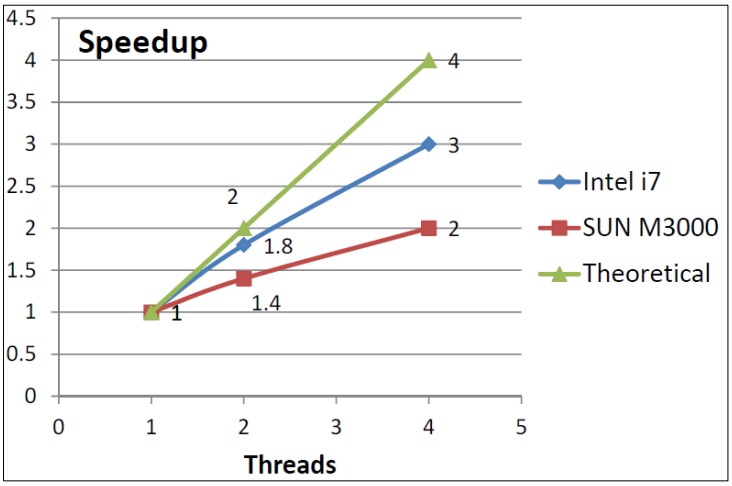

Figure 4.9: Speedup of Finalize Phase Fit algorithm vs. number of threads.

This can be attributed to improved architecture of i7 with faster data access and on chip accelerators for mathematical operations.
With *4* threads, the performance further improves. However, as shown in Fig. 4.9 the core utilization drops, and the application speedup does not scale up linearly with increase in number of threads, in spite of the algorithm being embarrassingly parallel.This can be attributed to high number of cache misses and cache flushing when multiple threads want to have different data in the cache at the same time. The optimal core utilization is achieved with one or two threads for both SUN M3000 and Intel i7. With this experiment, we conclude that Intel i7 is better in terms of speedup and core utilization when compared to SUN M3000.

On the GPU, we get a speedup of around *100x* in the execution time. However, this does not include the time spent in transferring data from CPU to GPU and vice versa. If we add that up, the total time becomes 300μs and the speedup reduces to about *9x* when compared to the baseline on SUN M3000. This gives us an indication that transferring of data between the CPU and GPU via the PCIe bus, using the CUDA API is a bottleneck and would require optimization. More details about the GPU implementation and data transfer time are shared in Chapter 5.

Experiment 2: Partially parallel, sorting algorithm

We decided to experiment with a partially parallel, sorting algorithm which is part of the Phase Unwrapping algorithm 3.1.2.2 and is used to sort an array of structures. Currently the *qsort* function of the STL library [32] is used.

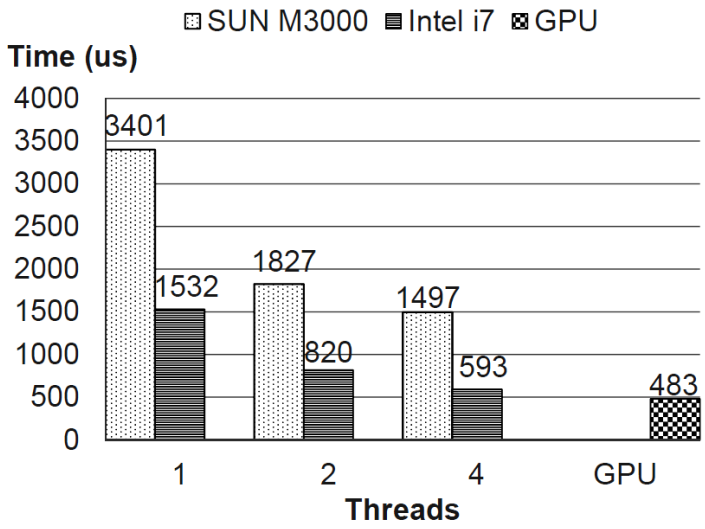

Figure 4.10: Sorting implementation with different threads on experimental hardware platforms

Although sorting is not an embarrassingly parallel algorithm, a number of parallel approaches to sorting exist. For sorting on multicore CPUs, we experimented with a combination of OpenMP and *qsort* implementation. The results from our best implementation are shown in Fig. 4.10. For sorting operations on the GPU, we were able to find a number of papers that established the fact the sorting operation shows performance gain on GPUs [33–35]. However, all the research work focused on large data sets (1- 10 million elements or more). To the best of our knowledge, we could find limited mention and no reference to any research work on estimating the performance of sorting algorithms on GPUs with small data sets.

On the GPU, with modifications to the structure being sorted, we were able to use the Thrust library [36]. Details of these modifications are discussed in Chapter 5. The results of the experiments on sorting algorithm are shown in Fig. 4.10. While the baseline execution time is 3401μs, the execution time of our OpenMP parallel implementation is 820μs with two threads on Intel i7, a speedup of more than $4x$. For our GPU implementation using the Thrust library, the execution time is 483μs, a speedup of more than $7x$. With the results of these experiments, we conclude that a parallel approach to sorting is possible and provides definite gains. The general line of thought suggests that GPUs are only suitable for embarrassingly parallel algorithms, or for partially parallel algorithms that work on huge data sets. From our experiments, we show that even for small data set (~ 32000 elements), a partially parallel operation like sorting does show considerable performance improvement.

Implementation	*Execution time (ms)*	*Speedup*
Baseline	18.60	
Seven threads on SUN M3000	17.8	1.05x
Seven threads on quad-core Intel i7	9.5	1.97x

Table 4.1: Execution time and speedup of seven field point implementation on CPUs

Experiment 3: Finalize Phase Fit and Phase Unwrapping on multicore CPUs

In this experiment, we worked with seven field points instead of one. This was done to understand how the performance scales up on CPU when seven field points are processed in parallel. We implemented Finalize Phase Fit and Phase Unwrapping algorithms on the CPU. In Chapter 5 we discuss the GPU implementation of these two algorithms for seven field points. This experiment serves as a benchmark to compare the GPU and CPU performance for these two algorithms for seven field points.
The results of this experiment are discussed in Table 4.1. The baseline time for these two algorithms for seven field points has been estimated using calculations shown in Appendix C.

Having analyzed the algorithms in previous chapter and based on the results of the experiments presented in this chapter we can estimate the PARIS performance on CPUs. On a *4* core intel i7, we expect a speedup of approximately *2x* in the post processing software execution based on the results discussed in Table 4.1. Extrapolating these results, we expect a *3-4x* speedup in the post processing software if the application is deployed on a similar platform as the experimental CPU platform but with *8* cores instead of *4*.
From our experiments we found that considerable effort would be required to deploy the application on the GPU, but that effort spent would also provide considerable performance gain. We also conclude from our experiments with the GPU the following:

(a) Communication overhead between CPU and GPU dominates and hence only those functions whose execution time on GPU and the time required for communication is less than the execution time on the CPU, should be mapped to GPU.

(b) To avoid communication over PCIe, preferably, only functions that execute one after the other in continuation should be mapped on the GPU.

(c) Not every algorithm maps well on the GPU. Control-flow intensive, nonparallel algorithms generally do not show double digit performance gain on the

GPU and hence should only be mapped if they fall between two algorithms that show considerable performance gain on the GPU.

(d) Mapping the algorithms on the GPU might require restructuring the data structures to optimize the performance. Memory hierarchy of the GPU should be exploited for higher gain.

Based on the results of the experiments we expect a speedup of around *10x* for the algorithms mapped on the GPU. This would mean a *5-7x* speedup for the entire post processing software when compared to average baseline ($\sim$*40*ms).

Summarizing this discussion on execution time speedup, we expect to see a *3-4*x speedup on a octa-core CPU and *5-7*x speedup on the GPU. The speedup achieved on the GPU will highly depend on the deployment strategy and optimization of data transfers between CPU and GPU and vice versa.

2. *Execution time variation*
   During our experiments on the dedicated hardware platforms, we observed variation in execution time within milliseconds precision. According to the guidelines, this is acceptable. The dominating reason for this variation is the non-real time operating system being used. Addition of GPU as an accelerator does not cause any noticeable change in the variation of the execution time.

3. *Precision requirements*
   CPUs and modern day GPUs both provide support for double precision arithmetic and hence both the platforms are able to meet the precision requirements. On GPUs however, double precision arithmetic is considerably slower than single precision arithmetic as shown in Fig. 4.11. Hence, care should be taken to avoid double precision arithmetic wherever possible for better performance on GPUs.

4. *Development time and cost*
   Development on multicore CPUs would require the use of OpenMP to exploit parallelism while using GPUs implies using CUDA APIs and explicit memory transfers. Although, GPU development cost will be somewhat higher, we do not expect much difference in development time. There are only three time consuming algorithms out of which only two have sufficient parallelism to be mapped onto the GPU which is already being done in this thesis. However, maintenance might become an issue as addition of new algorithms would require further analysis to decide if they should be mapped on the GPU or the multicore CPU. Thus, we expect development time and cost to be higher for GPU based hardware platform.

5. *Match with the competencies*
   Both OpenMP and CUDA would be new for the group. During our experiments, we found that OpenMP was easier to get acquainted with while CUDA took more time. Although, this is an opinion but, we expect the same for the group working for PARIS sensor software development.

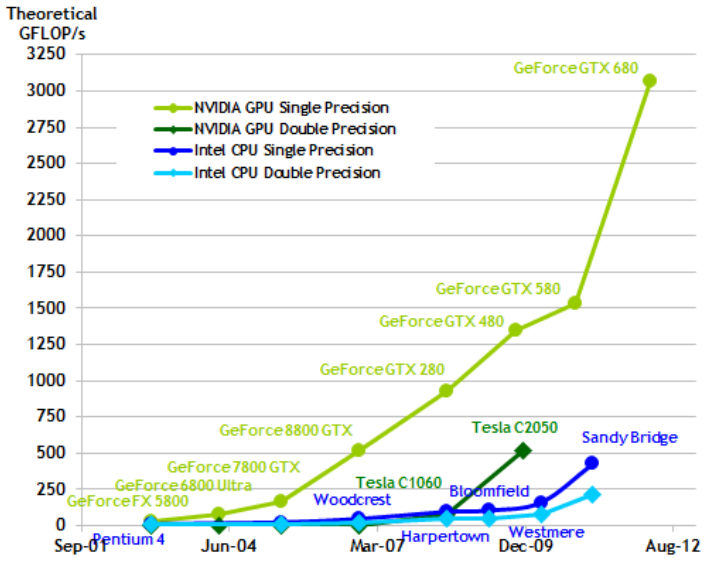

Figure 4.11: Theoretical floating point operations per second of various CPUs and GPUs. [4]

6. *Maturity of the tools*
   OpenMP has been in use for more than *15* years while CUDA is still less than *10* years old. However, in the past few years CUDA has been extensively used to exploit the powerful GPUs for general/scientific computing. While the GNU debugger is capable of debugging OpenMP programs, tools provided by NVIDIA have to be used for development, debugging and optimizing CUDA code. These tools are advanced, mature and provide a graphical, interactive interface to the developers. For both the programming models, OpenMP and CUDA, extensive support is available online on public forums. NVIDIA also has a strong presence in online forums and can also provide dedicated support for the tools if required. Therefore, we find no major difference between the two hardware platforms in terms of maturity of tools, on the other hand, if present trends of interest [37] continue, we expect to see much more active development in GPU programming using CUDA.

7. *Manufacturers and vendors*
   There are a number of manufacturers for multicore CPU like Intel, AMD, IBM etc. and there is a wide variety of options to choose from. OpenMP being an open, portable standard, does not reduce the platform choice in multicore CPU domain. There are also a number of programming models for GPUs. However, currently, we can only use CUDA because of the specific needs of our application. Hence we are limited to using only NVIDIA GPUs. This is one of the major drawbacks for opting for a GPU based solution. However, NVIDIA's GPU architecture roadmap in Fig. 4.12 is a positive. NVIDIA has been able to meet the targets that it has

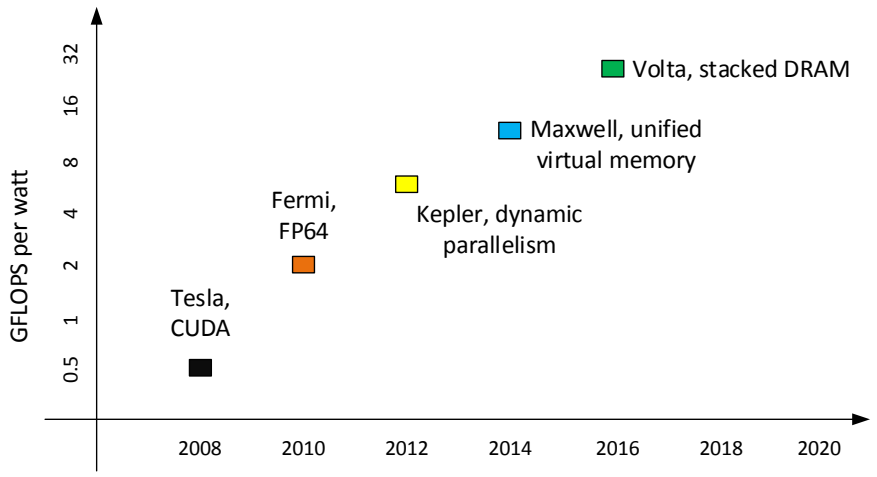

Figure 4.12: NVIDIA architecture development roadmap

set in the past with Tesla and Fermi architectures. The new Kepler architecture is also out in the market and as we see in Chapter 5, it would provide further acceleration for our GPU implementation. The architecture specification of the Volta, set to be released in *2016* have already been presented by NVIDIA and they plan to wait till *2020* before releasing the next architecture. NVIDIA's support and backward compatibility for it's earliest CUDA GPUs makes it easier to choose NVIDIA GPUs.
Besides CUDA, OpenCL and OpenAcc discussed in section 4.2.2 are two open standard for GPU development that are promising. If they include the capability of asynchronous execution on CPU and GPU in future, it will be possible to use them for the PARIS sensor system. This would allow to consider other GPU manufacturer, ATI.

8. *Cost of platform and tools*
   A GPU based platform would always need a CPU. The GPU acts as a "co-processor" to the CPU. In our case we would require the same multicore CPU in both the platform choices. Hence, the cost of a GPU based platform would be higher. However, the GPU used for experiments only costs *$350*USD [38] and therefore the platform cost difference is insignificant. There might be however other hardware costs associated with using GPUs, such as using extra cooling solutions, etc.. Study of these costs is beyond the scope of this thesis and left as future work.

In Table 4.2, we summarize the comparison of hardware platforms.

Metric	*Proposed multicore CPU*	*Proposed multicore CPU with GPU*
Application speedup	*3-4x* [a]	*5-7x* [b]
Execution time variation	Deterministic with milliseconds precision	Deterministic with milliseconds precision
Precision	No effect on performance	Performance decreases if double precision is used
Development time and cost	Low	Medium
Match with competencies	High	Medium
Maturity of tools	High	High [c]
Manufacturers and vendors	Intel, IBM, AMD etc.	NVIDIA, ATI [d]
Cost of platform and tools	Low	Slightly higher

Table 4.2: Summary of hardware platform comparison

[a] Octa-core CPU
[b] With an efficient deployment strategy reducing the data transfer time
[c] Besides CUDA, OpenCl and OpenAcc are promising technologies
[d] ATI GPUs can only be used in future if OpenCL or OpenAcc is used

4.5 Proposed hardware platform

Based on the analysis of the PARIS post processing software, comparison of hardware platforms and discussion with PARIS software development group, we decided to investigate the performance of the PARIS software stack on a GPU based hardware platform. With this prototype we will try to achieve the estimated performance gain (Table 4.2) practically when all seven detector ROIs are processed in parallel.
While the multicore CPU performance estimation is easier, GPU performance is a difficult task without actually implementing a prototype. With seven parallel processes running, GPU performance estimation becomes even more difficult. This prototype will help us estimate the performance of the PARIS software stack on the GPU based hardware platform.
Going for a GPU based solution in this thesis would help the sensor development teams to understand the capability of the hardware platform which would not have been possible for them in their normal work schedule. The GPU based hardware platform, if it is able to meet the estimated performance gain would provide a long term solution for the PARIS sensor software stack and also create an option to exploit for other sensor development teams in future.
An important research contribution of this prototype implementation would be an estimation of the performance of different types of algorithms (embarrassingly, partially parallel) on GPUs if they work on small data sets and have real time requirements.

In the next chapter we discuss the prototype implementation on the GPU, the optimizations made and their performance gain.

Prototype Implementation and Optimizations

5

In this chapter we discuss the prototype implementation developed. We present the optimizations made and the performance gain achieved in various steps, leading to the final prototype implementation. We analyze the implementation in terms of resource utilization of critical parts and discuss how we propose to improve it. Limitations due to hardware platform and possibilities for further speeding up the algorithm are also discussed wherever possible.
After discussing the implementation steps, we present the performance details of the prototype implementation.

5.1 Deployment decisions

Based on the analysis of algorithms in Chapter 3 and architecture study of hardware platforms in Chapter 4 we discussed and compared a number of deployment strategies. At an algorithmic level, we could eliminate most of the approaches based on the conclusions drawn from experiments on the GPU in Chapter 4. We discuss two strategies here which are similar at an algorithmic level but different on a system level. We could not conclude which strategy would give better system performance based on logic, previous experiments and knowledge about GPU capabilities. We implement both the strategies and compare the system execution time to conclude which is better for the system.

Fig. 5.1 shows how algorithms could be mapped on the CPU-GPU hardware platform for a single field point. To process seven field points on the platform efficiently and execute algorithms asynchronously, we make use of "CUDA streams". For an optimal execution, we expect an execution time line similar to one shown in Fig. 5.2. The time line shows how two field points will be processed on the new platform. The figure can easily be extended for seven field points. The "can do processing" phase in each field point processing can be used to execute algorithms on CPU in parallel to GPU execution. This will allow us to utilize the precious CPU time efficiently.
In the GPU available for experimentation, it is not possible to process different field points in parallel. We discuss about this limitation further in the chapter.
Strategy 1 is shown in Fig. 5.1 on the proposed multicore hardware platform and is also discussed next.

Strategy 1

1. The Finalize Phase Fit algorithm, being embarrassingly parallel is deployed on the GPU.

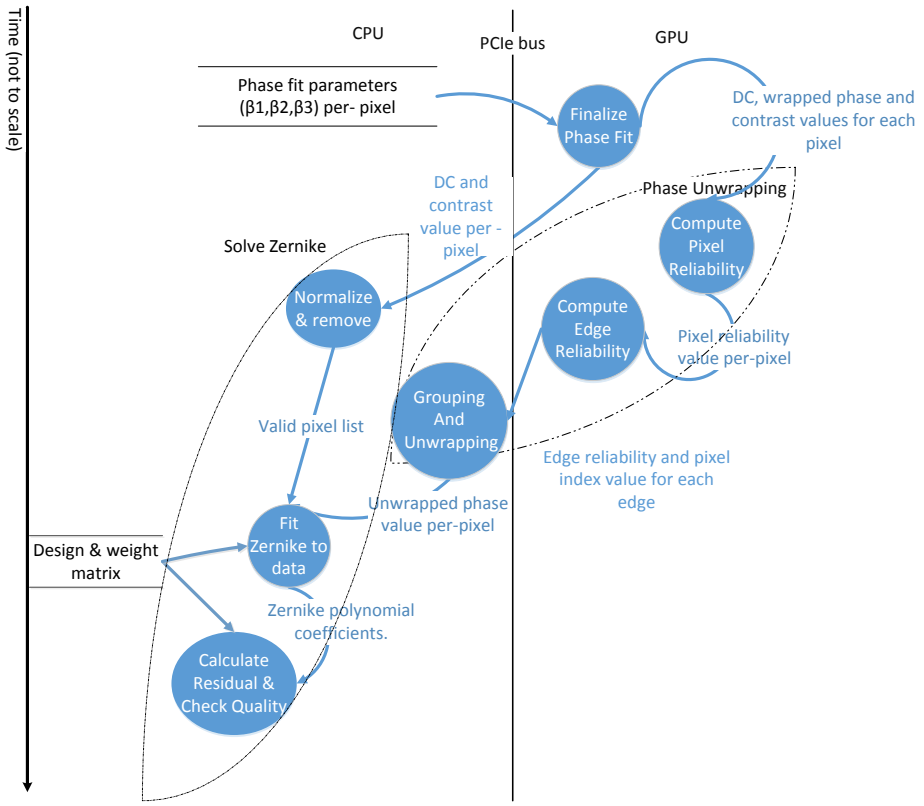

Figure 5.1: Data flow between algorithms deployed on the proposed multicore platform

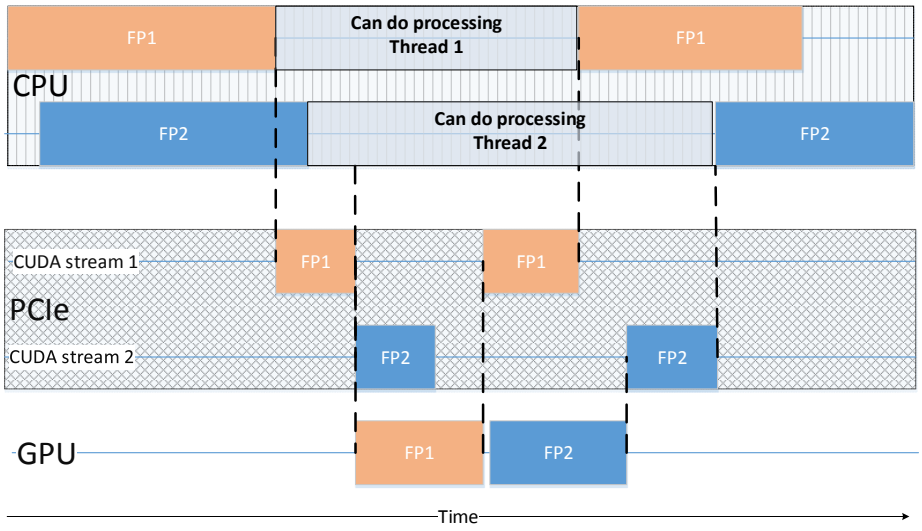

Figure 5.2: Time line showing processing of *2* field points on the proposed platform

2. Embarrassingly parallel and partially parallel parts of Phase Unwrapping algorithm are deployed on the GPU to reduce their execution time considerably. The non-parallel part of Phase Unwrapping algorithm (Grouping and Unwrapping) is deployed on the CPU.

3. OpenMP is used with Solve Zernike algorithm and it is deployed on the multicore CPU to reduce the execution time of the loops in Solve Zernike algorithm.

4. The "Normalize and Remove" algorithm , discussed in Section 3.2.2.1 is a part of the Solve Zernike algorithm. It is independent of Phase Unwrapping results and can execute after Finalize Phase Fit algorithm. Hence it can be processed in parallel on the CPU while parts of Phase Unwrapping are executing on the GPU. However, this might increase the GPU execution time.

5. Since the rest of the algorithms take only *12*% of the time (Table 3.1), these are deployed as it is on the multicore CPU.

Strategy 2

As discussed earlier, on an algorithmic level, both Strategy 1 and Strategy 2 are the same and the platform chosen (CPU or GPU) is the same for all the algorithms in both the strategies. Strategy 2 differs from Strategy 1 in terms of data transfers.
In Strategy 2, we transfer the results of the Finalize Phase Fit algorithm to the CPU together with the results of the Phase Unwrapping algorithm. This eliminates the possibility of executing the "Normalize and Remove" algorithm on the CPU in parallel to GPU execution. However, this might reduce the GPU execution time by allowing for higher overlap of execution and communication.

We implement both the strategies to conclude which gives a better system performance.

5.2 Implementation

Due to limitation of time during this project, it was decided that only the algorithms that are mapped on to GPU will be developed as a proof of concept. For algorithms mapped on the CPU, we will extrapolate the results of the experiments on CPU in the previous Chapter to provide an estimation of the final speedup.

Approach

We decided to start our prototype implementation based on the code written for experiments performed and build up on it in an iterative manner. The prototype implementation was developed in the following steps.

1. We first reduced the execution time and data transfer time for single field point implementation for Finalize Phase Algorithm.

2. Then, we enabled processing of the seven field points on the GPU. We optimized this by pipelining the access to GPU and allowing asynchronous execution on CPU.

3. For Phase Unwrapping algorithm:

   (a) We deployed Calculate Pixel Reliability algorithm discussed in Section 3.1.2.1 on the GPU for seven field points.

   (b) Next, we deployed the algorithm to find the pixel with maximum reliability on the GPU for seven field points. This was the first non-embarrassingly parallel algorithm being deployed on the GPU in our implementation. We modified the "parallel reduction" approach discussed in Section 3.1.2.1 to better utilize the computational units and memory hierarchy of the GPU. We discuss these modification later in this section and compare the results.

   (c) Afterwards, we deployed Compute Edge Reliability algorithm on the GPU. We removed the divergent branches inside the loop as discussed in Section 3.1.2.2 to suit the architecture of GPU core.

   (d) Finally, we deployed an optimized parallel sorting algorithm to sort the structure on the GPU.

5.2.1 Finalize Phase Fit

During our experiments with the Finalize Phase Fit algorithm in Chapter 4 we observed that although the execution time reduced by approximately *100* times, the data transfer time between CPU and GPU and vice versa was the bottleneck and needed to be reduced.

Reducing the execution time The *100x* reduction in the execution of the basic GPU implementation was when the calculations were done using double precision arithmetic. From our experiments, we concluded that we do meet the final precision requirement of *0.01*nm with single precision calculations. However we do affect the accuracy of the results of Finalize Phase algorithm and the execution time only decreases to *21*μs as shown in Table 5.1.

Implementation	*Execution time (μs)*	*Speedup*
Baseline	2800	
Double precision on GPU	28	100x
Single precision on GPU	21	133.3x
Double precision on GPU with 48KB L1 cache	16	175x

Table 5.1: Execution time and speedup of Finalize Phase Fit algorithm

To further reduce this execution time and still perform calculations in double precision to maintain the accuracy, we decided to use the fast, on-chip memory. The on-chip memory in a GPU is user/application configurable and is divided into two parts:

- A Level *1* cache.
- Shared memory

The difference between the two is that while the contents of the shared memory are handled explicitly by the code, the L1 cache is automatically managed. The total on-chip memory can be configured as *48*KB of shared memory and *16*KB of L1 cache and vice versa.

To ease the programming and yet benefit from the fast memory we configured the on-chip memory as *48*KB of L1 cache and rest to be used as shared memory. After this change, the execution time for double precision calculation for Finalize Phase Fit decreased to *16*μs. The speedup achieved for the execution time of this algorithm is shown in Table 5.1.
We also decided to keep this configuration of on-chip memory for all the algorithms where we do not need to share data in between parallel threads.

Reducing the data transfer time We know that transferring more data in a single API is more efficient than transferring less data and enables higher utilization of the PCIe bus [39]. Concluding from the report [29] we know that transferring *1* byte of data takes the same amount of time (*23*μs average) as *1* kilobyte if *CUDA API* shown in listing below is used.

```
1 cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, enum
      cudaMemcpyKind kind);
2 // dst   - Destination memory address
3 // src   - Source memory address
4 // count - Size in bytes to copy
5 // kind  - Type of transfer
```

From this we conclude that the transferring data via the PCIe bus using CUDA API has very high latency.
In the experimental platform available, the GPU is connected to the CPU using a *16* lane, second generation PCIe bus (PCIe *2.0 * 16*). This bus gives a maximum theoretical throughput of *500*MB/s per lane. Thus the maximum throughput that can be achieved using this bus is calculates using Eqns. 5.1 and 5.2.

$$\text{Maximum throughput } = 16 * 500\text{MB/s} = 8\text{GB/s} \tag{5.1}$$

Since *8/10* bit encoding is used, *20*% of the bandwidth is gone. Hence the effective theoretical throughput that can be achieved is shown in Eqn. 5.2.

$$\text{Effective throughput } = 80\% \, of \, 8\text{GB/s } = 6.4\text{GB/s} \tag{5.2}$$

In our measurements we found that the average achieved throughput was *2.67*GB/s. We did the following changes to reduce the data transfer time by trying to achieve the peak effective throughput.

1. *Optimization 1: Use single data transfer API*
   To reduce the transfer time we did experiments and found that transferring data using a single API call takes less time than transferring the same amount of data in small chunks using multiple API calls. The exact difference depends on the amount of data sent and the number of API calls used.
   In our case, we require approximately *367*KB of data to be transferred from CPU to GPU per field point. We observed an acceleration of *1.48x* in data transfer time when we transferred data using a single API call for one field point. The comparison of absolute values is shown in Table 5.2. After this optimization, the achieved throughput increased from *2.67*GB/s to *3.98*GB/s an improvement of *49*%. The PCIe bandwidth utilization can be computed by using the throughput achieved and peak effective throughput improved from Eqn. 5.2. The PCIe utilization increased from *41.7*% to *62.1*%.

Implementation	*Transfer time (μs)*	*Speedup*	*Throughput (GB/s)*	*Utilization*
Multiple API	137		2.67	41.7%
Single API	92	1.48x	3.98	62.1%

Table 5.2: Optimization 1: Data transfer time from CPU to GPU, speedup, throughput and bandwidth utilization

   To do this optimization, we had to change the input and the output data structure. The three separate input arrays ($\beta_1, \beta_2, \beta_3$, described in Eqn 3.8) were packed in one structure array so that they could be transferred using single API call. Similarly, the three output arrays (phase, DC, contrast) were packed in a single structure array.

2. *Optimization 2: Use "pinned" memory*
   To further reduce the transfer time, we performed one more optimization. Instead of allocating memory directly on the *host* (CPU) side, we used the CUDA API (*CudaMallocHost*) to allocate "pinned" memory on the CPU as shown in the code listing below.

```
1 cudaError_t cudaMallocHost (void ** ptr, size_t size);
2 // ptr - Pointer to allocated host memory
3 // size - Requested allocation size in bytes
```

   cudaMallocHost allocates host (CPU) memory that is page-locked and accessible to the device (GPU). Since the memory can be accessed directly (using DMA) by the device, it can be read or written with much higher bandwidth than pageable

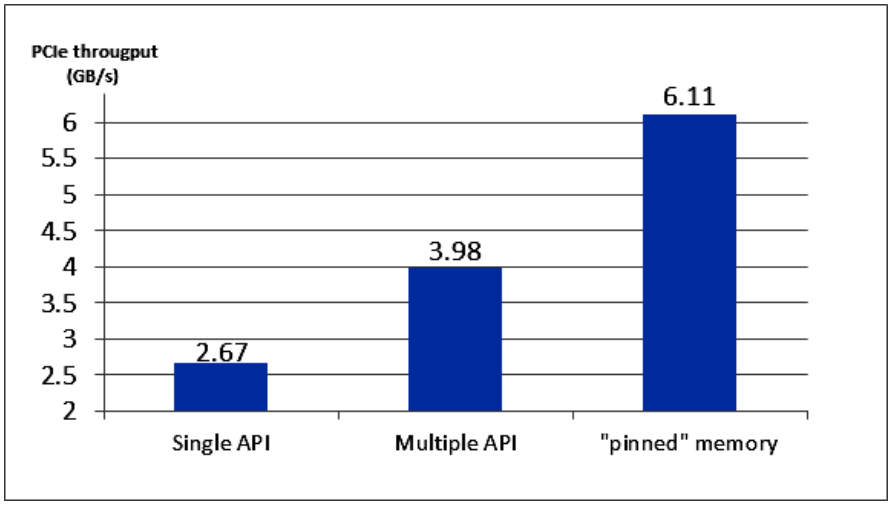

Figure 5.3: PCIe throughput achieved in *3* implementations

memory obtained with functions such as *malloc*. To transfer data for which memory is allocated using *cudaMallocHost*, call to *cudaMemcpyAsync*, shown in listing below has to be used.

```
1 cudaError_t cudaMemcpyAsync (void* dst, const void* src, size_t count,
    enum cudaMemcpyKind kind, cudaStream_t  stream = 0)
```

cudaMemcpyAsync is asynchronous with respect to the CPU, so the call may return before the copy is complete.
After this optimization, the transfer time for one field point further accelerated by *1.53x* when compared to transfer time after optimization *1* as shown in Table 5.3. The bandwidth utilization is now *95.56*% which is considerably high. Fig. 5.3 compares the throughput of baseline implementation with multiple API calls, implementation using single API call and implementation using single asynchronous API call.

The copy can optionally be associated to a stream by passing a non-zero stream argument. If the stream is non-zero, the copy may overlap with operations in other streams. This is a crucial benefit of using "pinned memory" and CUDA streams. Another major benefit of this optimization was that it allowed asynchronous execution on CPU and GPU and transfer of data between GPU and CPU at the same time. This will be beneficial when all seven field points are deployed on the hardware platform. It will enable transferring of data for one field point while the other is being processed on the GPU. This is further discussed in next section.

Implementation	*Transfer time (μs)*	*Step speedup*	*Cumulative speedup*	*Throughput (GB/s)*	*Utilization*
Multiple API	137			2.67	41.7%
Single API	92	1.48x	1.48x	3.98	62.1%
Using "pinned" memory	60	1.533x	2.28x	6.11	95.56%

Table 5.3: Optimization 2: Data transfer time from CPU to GPU, speedup and throughput

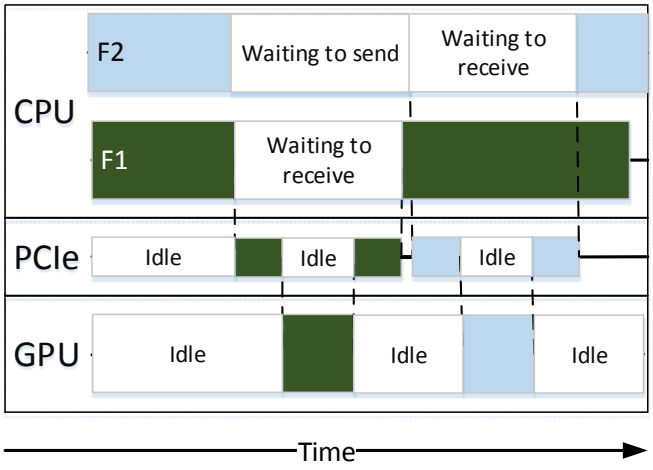

Figure 5.4: Time line showing processing of *2* field points on GPU without the use of streams.

5.2.2 Seven field points on the GPU

After having optimized data transfer at a field point level in previous optimization, we decided to develop the implementation where all seven field points are deployed on the GPU. Our initial implementation used the GPU in a sequential, serial way. This resulted in a linear extrapolation of execution time from one to seven field points for the Finalize Phase Fit algorithm on the GPU and lot of idle time on CPU as shown in Fig. 5.4.

It can be seen from Fig. 5.4 that their are lot of "idle" spaces on all the resources (CPU, GPU and the PCIe bus). The results of this basic implementation are shown in Table 5.4.

Using "pinned" memory allowed us to use CUDA streams. Using streams provides us with following benefits.

1. Asynchronous execution on CPU and GPU. This means that if an algorithm is executing on the CPU using one data set, another algorithm on the GPU could

Field points	*Total (execution + transfer) time (µs)*
Single	60 + 16 + 35 = 111
Seven	7 * (Single) = 777

Table 5.4: Total (execution and transfer) time for Finalize Phase Fit algorithm

Implementation	*Total (execution + transfer) time (µs)*	*Increase*
Single field point	111	
Seven field points without streams	777	7
Seven field points with streams	660	5.94

Table 5.5: Benefit of using CUDA streams with seven field points for Finalize Phase Fit algorithm

execute using a different data set.

2. We could pipeline the usage of the GPU. For example while data for field point *'2'* is being transferred from CPU to GPU, processing for field point *'1'* can be done on the GPU.

Both these benefits are crucial for the PARIS software. Due to the first benefit, we can execute " Normalize and Remove" algorithm 3.2.2.1 on the CPU while other algorithms execute on the GPU. Since in this implementation, we only develop the algorithms that are deployed on the GPU the performance improvement due to first optimization cannot be measured.

In our experiments using streams we found that the total time, instead of increasing linearly, increased only by *5.94* times for seven field points as shown in table 5.5.

Fig. 5.5 shows the screen shot of the results of our experiment. The screen shots

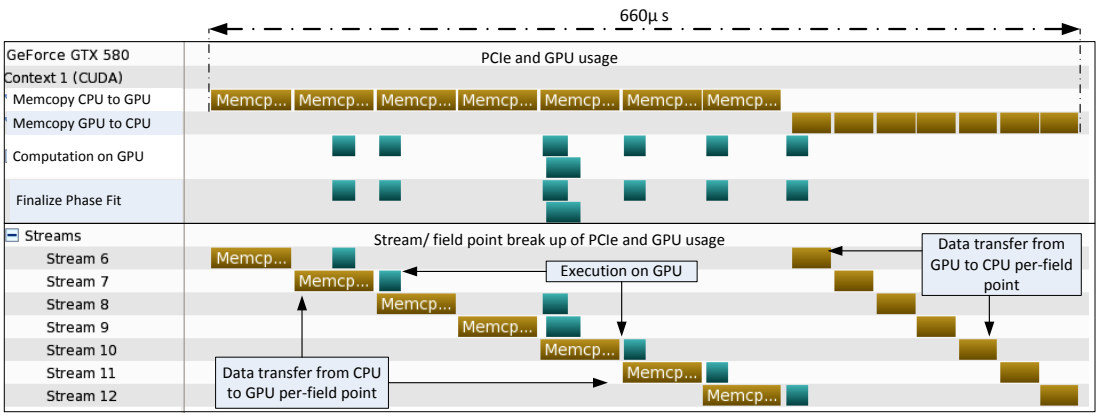

Figure 5.5: Execution of Finalize Phase Fit on GPU using streams for seven field points.

used in this thesis to show the execution of the algorithms have been taken using the NVIDIA Visual Profiler (NVVP). From the figure it is clear that although, the "idle spaces" have reduced when compared to Fig. 5.4, there are still many idle spaces while the data is being transferred using CUDA streams. This space will be utilized and idle time would decrease as we add more algorithms in our implementation on the GPU.
We estimate that as we add more algorithms on the GPU, this optimization of using CUDA streams will provide higher performance gain. During the time when data transfer for field point two occurs, more algorithms for field point one can execute on the GPU without adding to the total executing time.
A limitation of the GPU available for experiments was that it only had one "copy engine" and one "compute engine". Consequently, only data transfer in one direction could overlap with execution on the GPU as shown in Fig. 5.5 . It is possible to overlap data transfer in both directions with execution on the GPU. This is possible in the latest, more expensive GPUs from NVIDIA which have two "copy engines" and one "compute engine".
For example, in the time line shown in Fig. 5.5, if the transfer of results from GPU to CPU could overlap then the total GPU time will decrease from 660μs to approximately 460μs, an improvement of around 30%.
After successfully creating a foundation of deploying seven field points on the GPU with the simplest Finalize Phase Fit algorithm, we moved on to deploying the more complex Phase Unwrapping algorithm on the GPU.

5.2.3 Phase Unwrapping

We discussed the Phase Unwrapping algorithm in Section 3.1. In this section we discuss the implementation of parts of the Phase Unwrapping algorithm on the GPU one by one.

Compute Pixel Reliability

The Compute Pixel Reliability algorithm was divided into two parts for GPU implementation. These are:

1. Calculate pixel reliability
2. Find pixel with maximum reliability

Calculate pixel reliability

Deploying calculate pixel reliability algorithm on the GPU was easy and similar to deploying Finalize Phase Fit algorithm. In this algorithm also, we do per-pixel processing and each pixel is processed by a separate thread on the GPU. However, we do need the phase value of the neighboring pixels to process each pixel.
Following modifications were done in the algorithm to optimally deploy it on the GPU:

1. Modification in data access because of the modifications done to the data structure in optimization *1* of Finalize Phase Fit algorithm.

Implementation	*Execution time (μs)*	*Speedup*
Double precision exponential function	30	
Single precision exponential function	16	1.875x
Single precision exponential function with 48KB L1 cache	10	3x

Table 5.6: Performance improvement in GPU implementation of calculate pixel reliability algorithm

2. Based on the study of the accuracy required for exponential function in Section 3.1.2.1, we changed the double precision exponential function to hardware accelerated (intrinsic) exponential function provided the CUDA API. This is not compatible with IEEE 754 floating point standard but satisfies the requirements of the algorithm.

Because of the optimization above, the execution time of calculate pixel reliability is reduced to *16*μs. Further, after changing the configuration of on-chip memory, the execution time decreased to *10*μs. The absolute value of execution time and the speedup is presented in Table 5.6. We do not compare the execution time with the baseline implementation because the baseline implementation of Compute Pixel Reliability includes the computation of pixel with maximum reliability value in the same loop. In our implementation, we do this in a separate step to efficiently utilize the GPU for both the algorithms which have different type of parallelism. Hence we compare the execution time with the baseline by adding the execution time for finding the pixel with maximum reliability in the next section.

The screen shot of the time line of the implementation of Finalize Phase Fit and calculate pixel reliability for seven field points is shown in Fig. 5.6. As can be seen the execution of both the algorithms still overlaps the data transfer over the PCIe. Although the execution time of the both the algorithms is *26*μs per field point, it does not add to the total execution time on the GPU and it is still *660*μs. A clear benefit of using streams which can be further increased as we deploy the next algorithm.

Find pixel with maximum reliability

In this section we discuss the the algorithm to find the pixel with maximum reliability. This problem is a reduction operation and we follow a parallel approach as discussed in Section 3.1.2.1. However to optimize the performance on the GPU architecture we modify our approach of accessing the elements of the array. We discuss these modifications in this section. Our first implementation on the GPU is pictorially depicted in Fig. 5.7.

The kernel code for this implementaion is presented in Section B.1 of Appendix B. The execution time for this implementation on the GPU was *80*μs. On analysis

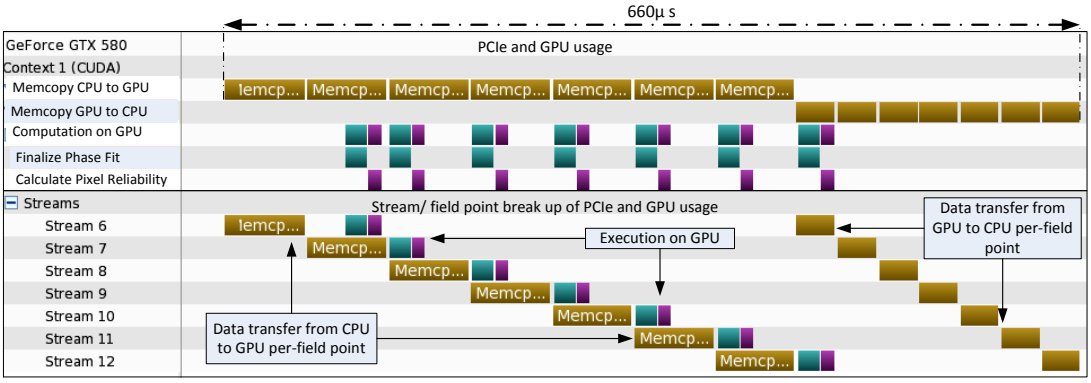

Figure 5.6: Execution of calculate pixel reliability and Finalize Phase Fit on GPU using streams

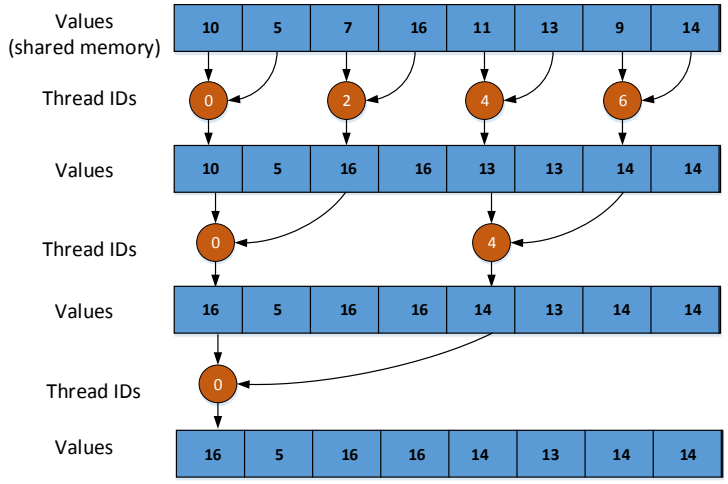

Figure 5.7: Pictorial representation of our first implementation of the algorithm to find pixel with maximum reliability on the GPU

we found that it suffered from problems which have been discussed in Section B.1 and could be improved.
The pictorial representation of the improved implementation is shown in Fig. 5.8 and the execution time reduction is compared in Table 5.7. The *kernel* code for this implementation has been presented in Section B.2.

Comparison of execution time of calculate pixel reliability and finding pixel with maximum reliability in the baseline implementation and in our implementation is shown in Table 5.8.

After the pixel with maximum reliability has been found, rest of the pixel reliability

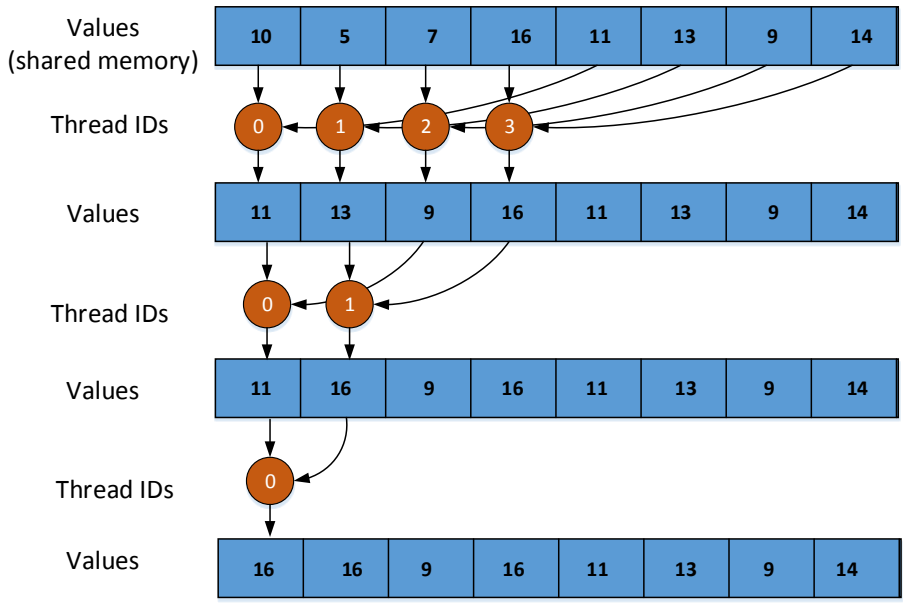

Figure 5.8: Pictorial representation of our improved implementation of the algorithm to find pixel with maximum reliability on the GPU

Implementation	*Execution time (μs)*	*Speedup*
Basic	80	
Improved	5	16x

Table 5.7: Performance comparison of two implementations on GPU of algorithm to find pixel with maximum reliability

values are normalized w.r.t the highest reliability value. This again, is an embarrassingly parallel operation and takes approximately 5μs on the GPU as compared to 800μs on the SUN M3000.

Fig. 5.9 shows the time line of the implementation when the following algorithms are deployed on the GPU.

1. Finalize Phase Fit
2. Compute Pixel Reliability

Implementation	*Execution time (μs)*	*Speedup*
Baseline on SUN M3000	2200	
GPU	15	146.66x

Table 5.8: Acceleration of Compute Pixel Reliability algorithm on the GPU

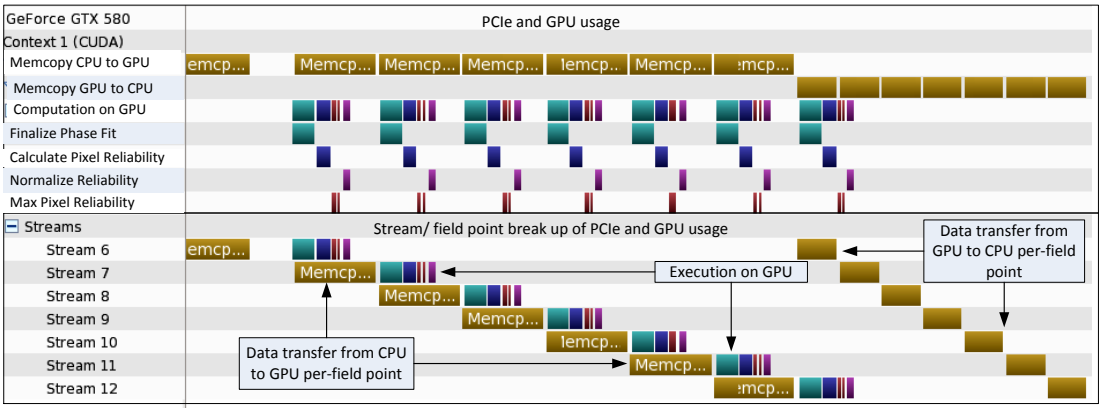

Figure 5.9: Time line showing execution of Finalize Phase Fit and all parts of CPR on the GPU.

(a) Find reliability of all the pixels.
(b) Find the pixel with maximum reliability.

3. Normalize all pixel reliability values w.r.t pixel with the maximum reliability.

The total execution time is still *660*μs as the execution of all the algorithms is still overlapped by data transfer. In the next section we discuss the implementation of Compute Edge Reliability algorithm.

Compute Edge Reliability

The algorithm can be divided into two logical parts:

1. Calculate the reliability of all the edges.
2. Sort the edges according to their reliability values.

Calculate reliability of all edges

The algorithm to calculate reliability of all the edges is an embarrassingly parallel algorithm as discussed in Section 3.1.2.2. However, we had to change the implementation so that the loop iterations could become independent and the algorithm could be efficiently deployed on a parallel architecture. After the change, the algorithm was deployed on the GPU and our implementation on the GPU showed a speedup of *109*x in the execution time performance as shown in Table 5.9. Further, on re-configuring the on-chip memory, the execution time decreased to *7*μs, improving the speedup to *171*x. The total execution time is still *660us* as the execution still overlaps with data transfer.

This algorithm results in new data (reliability value of edges and a list which stores the constituent pixel ids of each edge). This data has to be sorted in decreasing order and sent back to the CPU. We discuss the implementation of parallel sorting algorithm in next section.

Implementation	*Execution time (μs)*	*Cumulative speedup*
Baseline	1200	
GPU	11	109x
48KB L1 cache on GPU	7	171x

Table 5.9: Performance improvement in Calculate Edge Reliability algorithm

Implementation	*Execution time (μs)*	*Speedup*
Baseline	3401	
On GPU using Thrust library	483	∼7x
custom implementation on GPU	223	∼15x

Table 5.10: Performance improvement in sorting of edges

Sort the edges according to their reliability value
For sorting on the GPU, a number of a libraries are available including the well known Thrust [36] and CUDPP [40] etc.. In our experiment in Section 4.4, we used the Thrust library to sort the structure shown in listing below based on the reliability value.

The experimental result had shown a performance improvement of around *7x* over the baseline as shown in Fig. 4.10 . However, when we tried to implement sorting using Thrust library for seven field points, the performance degraded and it affected the performance of other algorithms as well.
After analysis we found that the Thrust library is not compatible with CUDA streams and hence whenever a sorting call was made, the asynchronous stream operations stalled and sorting algorithm would execute.
This increased the execution time from *660*μs to *5*ms. On further research, we found, to the best of our knowledge none of the libraries offered a sorting function compatible with CUDA streams. This meant that we had to develop our own sorting implementation on the GPU.
We found some sorting implementations provided by NVIDIA [41,42]. These implementations were already optimized for GPU execution but had to be tailored to our specific problem and according to the GPU available for experimentation. For example, the merge sort implementation in [42] was implemented for sorting integer values and we had to modify it to sort structure values based on a integer key. We had to change the data structure which stored the reliability of the edge and the index of pixels constituting an edge.
After the modification and using the sorting implementation from [42], the execution time of sorting algorithm was measured to be *223*μs per field point. This is an approximate improvement of *2*x when compared to experimental results of Chapter 4 which were based on development using the Thrust library. The comparison of baseline, experiments using the Thrust library and our final sorting implementation are shown in Table 5.10 and in Fig. 5.10.

However, the sum of data transfer time and execution time of the algorithms implemented on the GPU increases from *660*μs to *1936*μs. This is because of the addition of

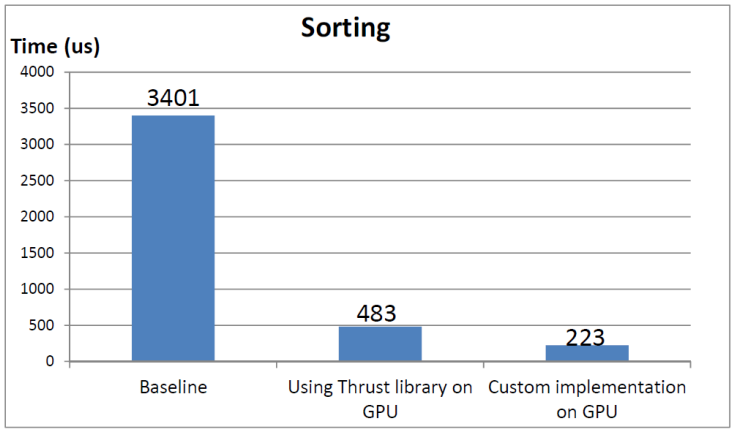

Figure 5.10: Execution time of sorting in different implementations

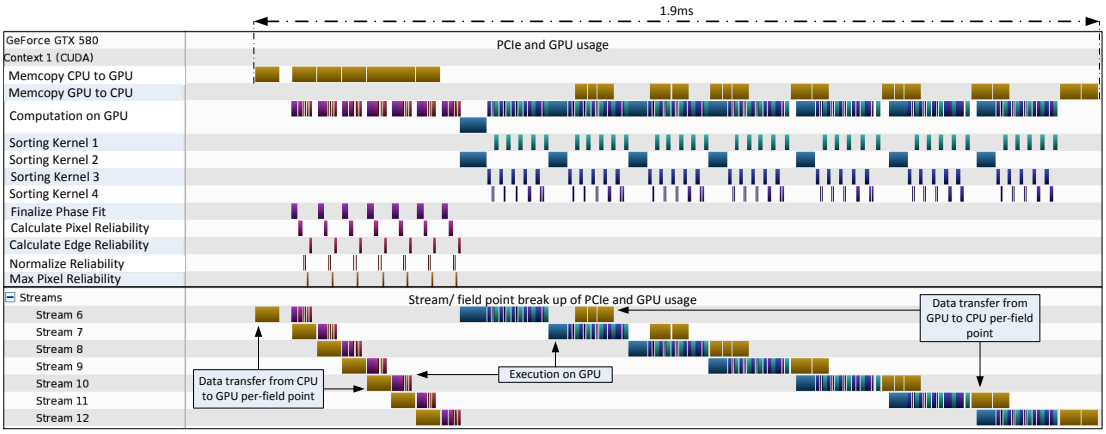

Figure 5.11: Time line representing execution of all algorithms deployed on the GPU and data transfer

the sorting algorithm to the execution sequence and transfer of the sorting results from GPU to CPU.
The time line representing the execution of all the algorithms deployed on the GPU and the data transfers is shown in Fig. 5.11.

Alternative Approach
In the implementation represented in Fig. 5.11, the phase, DC and contrast values are transferred back to the CPU after sorting the edge reliability values. Another approach could be to send these values back to CPU before sorting operation on the GPU. This would enable execution of "Normalize and Remove" algorithm of Solve Zernike on the CPU in parallel (asynchronously) to execution on the GPU. This is the Strategy 1 which we discussed in Section 5.1 of this chapter.

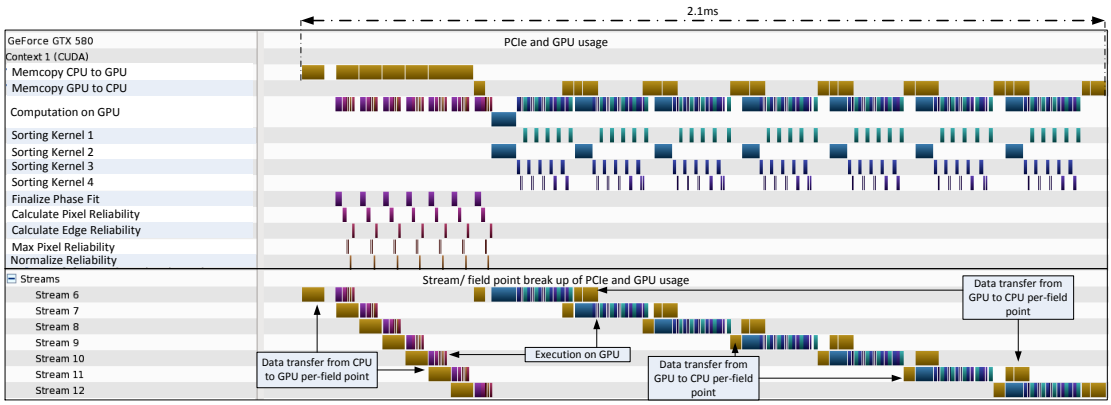

Figure 5.12: Alternative approach: time line representing execution of all algorithms deployed on the GPU and data transfer

The time line representation of this approach is shown in Fig. 5.12. The GPU execution time of this approach is *2140* μs, which is *204* μs higher when compared to our previous approach (Strategy 2 discussed in Section 5.1) . However, the benefit of utilizing the precious CPU time compensates for this increase. This approach results in lower GPU performance but is capable of higher "system performance". The "Normalize and Remove" algorithm which takes *700* μs per field point in the baseline can execute on the CPU while the processing is still being done on the GPU.

5.3 Conclusion

Fig. 5.11 shows that data transfers from CPU to GPU and vice versa for all field points, except for one in each direction are overlapping with the execution. Thus we hide the "latency" of transferring data. We could have also reduced the execution time of algorithms by a more detailed analysis of the results provided by the NVVP and doing more optimization with the algorithms.
However, as we saw from the start of this Chapter, execution time acceleration was never an issue with the GPU. Even the non-embarrassingly parallel algorithm like sorting showed an approximate *15*x performance gain in comparison to the baseline while the embarrassingly parallel algorithms showed a performance improvement of over *100*x in all cases. The baseline execution time and the execution time of algorithms on the GPU is shown in Fig. 5.13. The scale of the graph is logarithmic. It can be seen from the figure that the least performance gain is in the sorting algorithm. This was expected as sorting is not an embarrassingly parallel algorithm.
Another important conclusion that can be drawn from Fig. 5.13 is that although the order of execution time is same for all the algorithms under consideration on the CPU, it is not the case on the GPU. The sorting algorithm takes $\sim$ *14*x the time taken by second highest time consuming algorithm on the GPU (Finalize Phase Fit). This stresses the

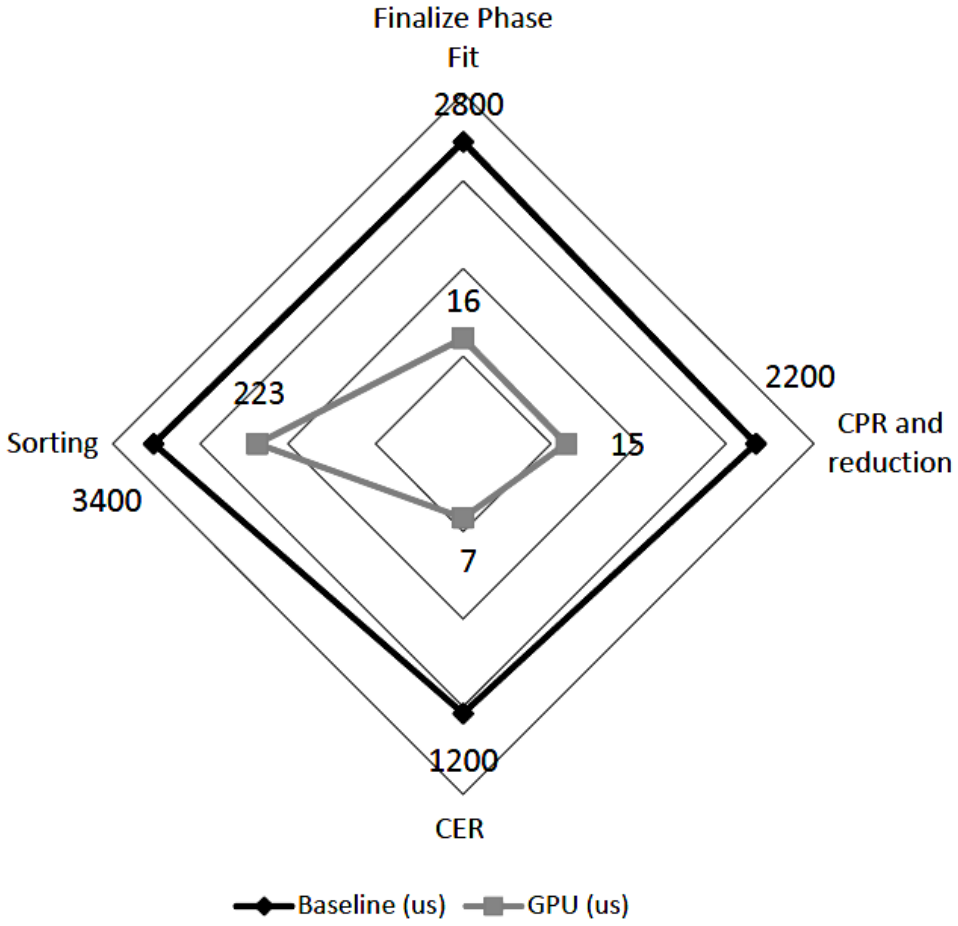

Figure 5.13: Execution time (μs) comparison of algorithms deployed on GPU

need of suitability of the algorithm for massively parallel architectures such as that of a GPU.

The real issues were data transfer latency, GPU being inefficiently utilized and CPU being idle. We were able to reduce and hide data transfer time completely with execution of algorithms. The problem of CPU being idle was also solved. However, we could only solve the problem of inefficient GPU utilization partially.
As can be seen from Fig. 5.11 and 5.12 there are still idle spaces in processing of each field point on the GPU. These spaces are caused by limitation of the GPU available for the experiments. The GPU is based on the Fermi architecture which does not support overlapping Kernel execution for multiple streams on the GPU. Only the last Kernel of a stream in an execution sequence can overlap with the first kernel of another stream. Support for overlapping execution of Kernels from different streams is provided in the latest Kepler architecture from NVIDIA. Kepler provides *32* hardware managed connec-

tions between the CPU and the GPU while in Fermi there is only *1* and all software streams are multiplexed into it. The occupancy achieved by sorting kernels is in the range of *55%* to *69%*. Thus we could achieved higher accuracy and higher performance if sorting on different field points could execute in parallel on the GPU.
We expect that on a Kepler based GPU, the idle space in processing of each field point would decrease drastically which would have a positive impact on the execution time performance on the GPU.

After deploying sorting algorithm successfully on the GPU and comparing the output values with the baseline implementation, we have successfully completed our prototype implementation. In the next chapter we compare our GPU implementation performance with the baseline, the estimation made in Chapter 4 and other models which we describe in Chapter 6.

System Performance Comparison

6

In Chapter 5 we discussed the steps that led to our prototype implementation. In this chapter we discuss and compare system performance of different hardware platforms composed using CPUs and GPUs. We discuss the performance improvement for each of these platforms compared with the baseline performance shown in Fig. 6.1.

6.1 SUN M3000 server with a GPU

In this section we discuss the performance of a platform consisting of the GPU and the current shared SUN M3000 platform. Fig. 6.1 shows the baseline performance of the PARIS post processing software stack. From the calculations shown in Appendix C, we know that the Phase Unwrapping and Finalize Phase Fit (part of processing on SUN M3000) algorithms take approximately *18.6*ms in the baseline execution time. Our prototype implementation using Strategy *1* on the GPU of the same algorithms takes *2.1*ms as discussed in Section 5.3.
To estimate the system performance of PARIS on the SUN M3000 - GPU platform we assume the following:

1. The GPU is connected to the SUN M3000 platform using PCIe 2.0.

2. The data transfer time from CPU to GPU and vice versa remains the same for this platform as for the prototype implementation platform in Chapter 5.

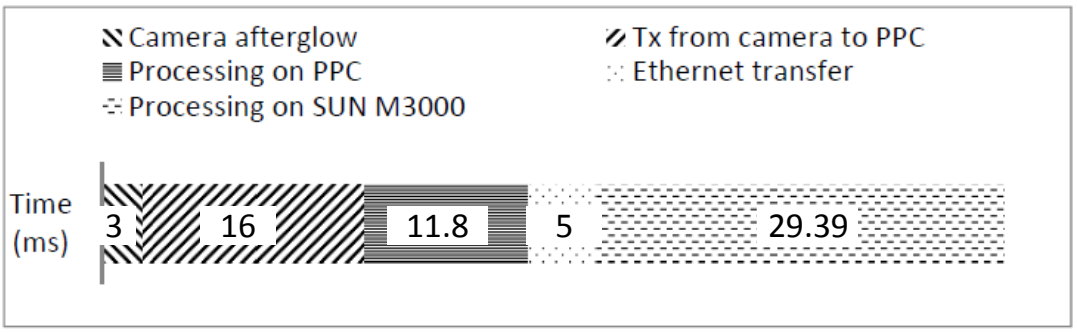

Figure 6.1: Average time taken by activities during post processing time budget on SUN M3000 and PowerPC based platform

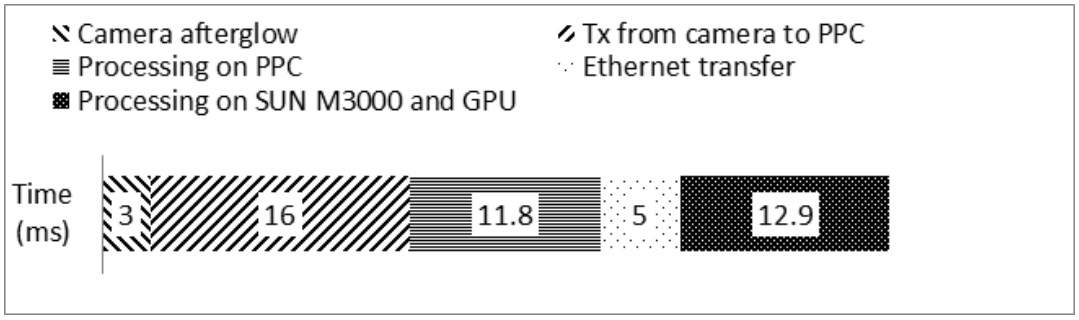

Figure 6.2: Average time taken by activities during post processing time budget on GPU, SUN M3000 and PowerPC based platform

3. The algorithms deployed on the GPU in the prototype implementation of Chapter 5 are deployed on the GPU in this hypothetical implementation as well and have the same execution time.

4. Rest of the algorithms are deployed on SUN M3000 and PowerPC as in the baseline implementation and have the same execution time (approx. *10.8*ms from calculations in Appendix C).

Considering the above assumptions, the system performance on the GPU and SUN M3000 platform can be estimated. Thus the baseline execution time of *29.39*ms on SUN M3000 is now replaced with *10.8*ms on SUN M3000 and *2.1*ms on GPU, adding to *12.9*ms as shown in Fig. 6.2.

However, this platform is not practically feasible because CUDA drivers are not yet available for Solaris operating system. Moreover, this platform does not solve the issue of variability in the execution time as the SUN M3000 and the shared Ethernet network are still used in the system.

6.2 Intel i7 processor with GPU

In this section we estimate the performance on a platform consisting of a GPU and a dedicated quadcore Intel i7.
We assume that the same algorithms are deployed on the GPU as in the implementation of Chapter 5 and execute in *2.1*ms. For algorithms that were deployed on the SUN M3000 and took *10.8*ms, we extrapolate the results of experiments performed on Intel i7 in Chapter 4. Based on the results discussed in Table 4.1 we estimate a *2*x performance gain. In this model, we also assume that the algorithms which were earlier deployed on PowerPC and took *11.8*ms to execute, are now deployed on Intel i7. We make an assumption of a minimum *2*x performance gain for them as well.
Considering the above performance gain, the expected execution time line of the PARIS post processing is shown in Fig. 6.3. The software execution time adds up to *13.4*ms

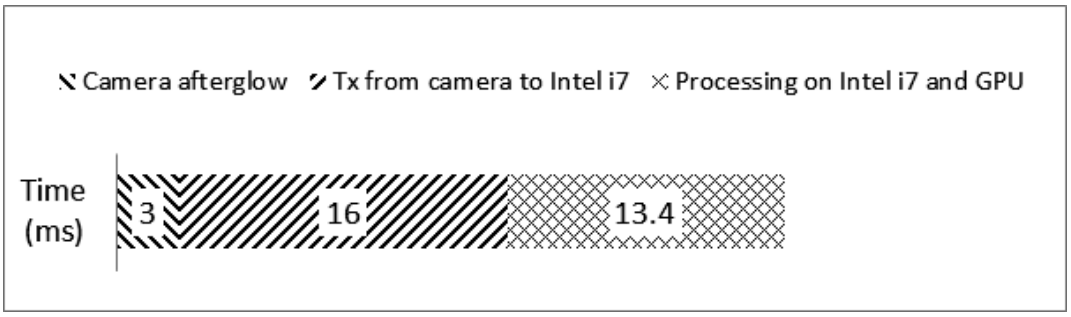

Figure 6.3: Average time taken by activities during post processing time budget on GPU and Intel i7 based platform

as shown in the figure.

The performance achieved is shown in Table 6.1. This platform also ensures deterministic execution time with milliseconds precision as discussed in experiments of Chapter 4. Since there is no shared Ethernet network used to transfer data, this system is expected to have a deterministic performance.

6.3 Intel i7 processor

In this section we estimate the performance on a quadcore Intel i7 which is capable of executing eight threads concurrently. We assume that the seven threads process seven field points.
We extrapolate the results of experiments discussed in Chapter 4 for the entire software stack which was executing on SUN M3000 earlier. Thus, we expect that the execution time will decrease from *29.39*ms to approximately *14.7*ms . We assume the same reduction of *2*x for algorithms that were previously deployed on the PowerPC reducing their execution time from *11.8*ms to *5.9*ms.
Using these extrapolated results and the assumption, the expected performance is shown in Fig. 6.4 and compared to baseline performance in Table 6.1.

6.4 Two Intel i7 processors

In this section we estimate the PARIS performance on a platform with two Intel i7 processors. Fig. 6.5 shows the two Intel i7 processors based platform. In this platform we assume the following:

1. The data transfer time between both the CPUs is same as data transfer time between a CPU and GPU of our prototype implementation.

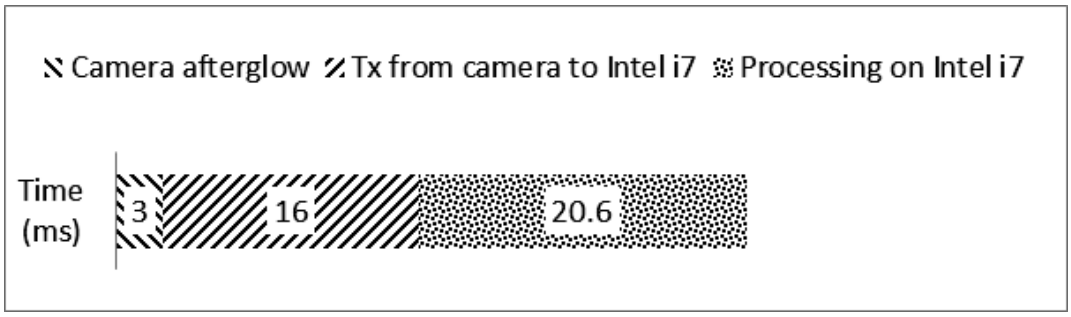

Figure 6.4: Average time taken by activities during post processing time budget on Intel i7 based platform

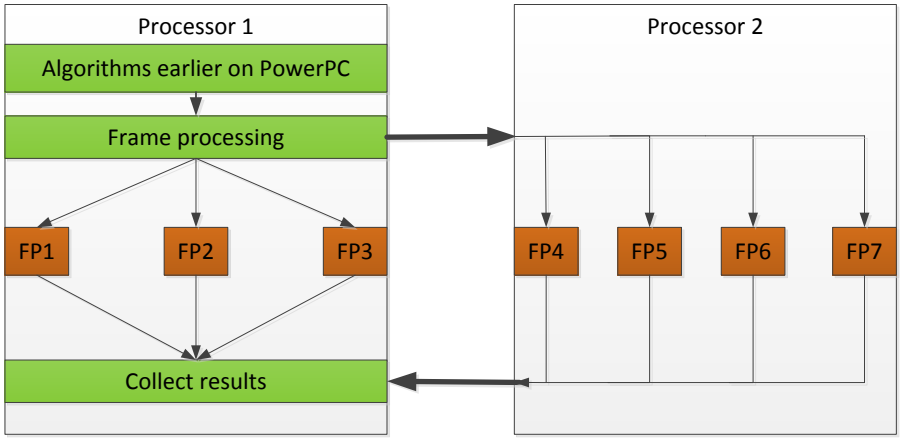

Figure 6.5: Processing on two processor platform

2. We assume that four field points are processed on one Intel i7 while the other processes three field points and the algorithms that were earlier deployed on PowerPC. Each field point is processed by two threads which allows to parallelize the algorithms according to the strategies discussed in Chapter 3.

3. Since *4* field points are processed on an i7 processor in parallel with *3* field points on another i7 processor, we assume that the upper bound on execution time is provided by the timing measurement of *4* field point processing.

From our experiments on Intel i7 we found that processing *4* field points using eight threads takes *6.8*ms for Finalize Phase Fit and Phase Unwrapping algorithms. This is a speedup of approximately *2.75*x when compared to the baseline on SUN M3000 which has been estimated to be approx. *18.6*ms in Appendix C. We extrapolate this gain to Solve Zernike algorithm as well. Since we did not implement other algorithms using

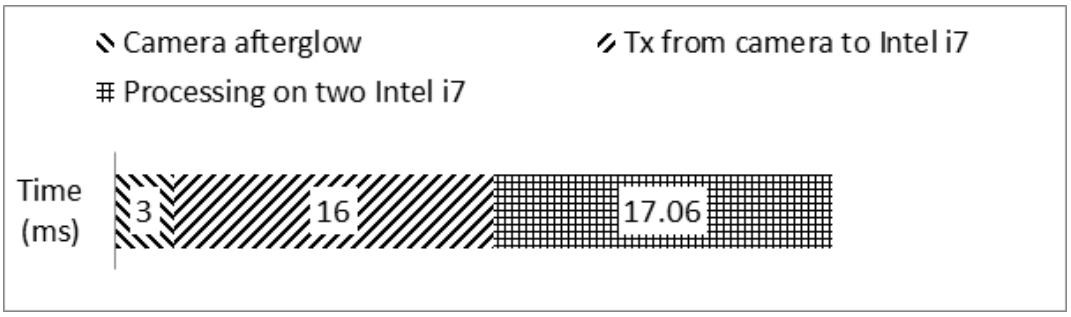

Figure 6.6: Average time taken by activities during post processing time budget on a platform with two Intel i7 processors

Platform	*Post processing time (ms)*	*Software execution time (ms)*	*Post processing speedup*	*Software execution speedup*
Baseline SUN M3000	65.19	41.19		
Shared SUN M3000 with GPU	48.7	24.7	1.33x	1.66x
Intel i7 with GPU	32.4	13.4	2.01x	3.07x
Intel i7	39.6	20.6	1.64x	1.99x
Two Intel i7	36.06	17.06	1.80x	2.41x

Table 6.1: System performance comparison on discussed hardware platforms

OpenMP, we estimate a single thread performance gain of *2*x as in previous cases for these algorithms. Thus, three algorithms under consideration which took *88*% of the baseline have a *2.75x* performance gain while the rest of the algorithms which take *12*% of the execution time in the baseline are assumed to have a minimum *2x* performance gain.
We also assume that the algorithms which were earlier on PowerPC have a performance gain of *2*x as in previous cases. The estimated time line is shown in Fig. 6.6. The performance improvement at a system level is compared with other implementations in Table 6.1.

6.5 Conclusion

In this chapter we estimated the system performance on different hardware platform models based on experiments, prototype implementation, assumptions etc. The comparison is shown in Table 6.1.

Out of the discussed platforms, two (Intel i7 processor with GPU and two Intel i7

processors) platforms meet the performance goals set in Section 1.3 of this thesis. The performance estimated on these platforms can further be improved by using some of the suggestions discussed in Chapter 7.

7 Conclusion and Future Work

Having compared the prototype solution developed with other possible solutions in the previous chapter, we now present the conclusion to this thesis. We also discuss some of the future work that could be done to further improve the performance of the PARIS sensor system. This also includes some technologies to look out for in the near future and some suggestions to the software development team.

7.1 Conclusions

In this thesis we analyzed the PARIS sensor software stack and found that there was a good opportunity to exploit parallelism in the algorithms to reduce the execution time. The current hardware platform however, is not capable of this because of the limited number of cores. We also observed that since a shared computing platform and Ethernet network are used, the execution time is non-deterministic.
To solve these problems, we considered three hardware platforms (CPU, GPU and FPGA). Based on the analysis of software and comparison of hardware platforms, we proposed that both multicore CPU and GPU based platforms will be able to meet the performance requirement to reduce the execution time to *50%* of the current execution time. We did not do detailed performance analysis with FPGAs due to reasons discussed in Chapter 4.
In Chapter 6 we discussed four hardware platform models based on CPUs and GPUs and found that two of them will be able to meet the performance expectations that we had from the new hardware platform. Each of the hardware platforms will provide for different performance improvement as shown in Table 6.1. However, each of them would require different investment of time and money and would have a different impact on the organization as discussed in Section 4.4. Higher performance would come at a higher cost and impact. Therefore it depends on the concerned department to choose a hardware platform according to the performance requirement and other aspects as well.
Fig. 7.1 shows a graph that compares the expected performance of different hardware platforms and the impact they are expected to have on the organization and their product platform. While the performance refers to the execution time speedup, impact can be thought of in terms of cost of development, effort required, compatibility with the machine hardware etc.

As can be seen from Fig. 7.1, the GPU based solutions have high impact. This is in accordance with our discussion in Section 4.4 where we compared different hardware platforms. However, the combination of Intel i7 and GPU also gives the best performance among the compared hardware platforms. Hardware platform based on two Intel i7 processors is expected to have the second highest performance but second lowest impact and

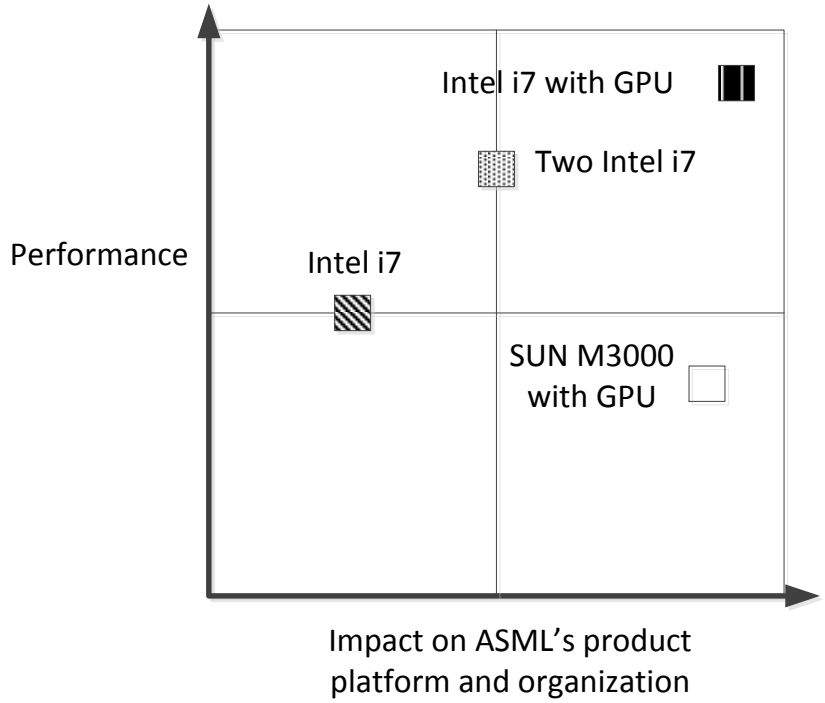

Figure 7.1: Performance vs. impact for different hardware platforms

hence falls in the optimal region. The other two platforms (SUN M3000 with GPU and Intel i7) do not meet the criteria of reducing the execution time by half. Considering the current performance requirements and the PARIS team development skill set, platform consisting of two Intel i7 processors provides an optimal solution.
Adapting to the GPU and Intel i7 based solution will have an high impact on the organization and will require considerable effort. However, to obtain that extra performance, GPUs are the way to go ahead. The development cost and impact on the machine will be high initially, but once GPUs are integrated into ASML machines, they will provide for a long term solution for the computing needs. Not only for PARIS, but the expertise and experience gained after implementing a GPU based solution will promote it's use in other components of the machine and reduce development cost and time on the GPUs.

Concluding, investing in GPUs for PARIS will take high initial effort but the long term gain that the GPUs might provide could more than compensate for this first big step.

7.2 Future work

In this section, we discuss the recommendations which could be useful for the PARIS sensor development team to further improve the performance. Due to lack of time and limitations in the scope of this thesis, we were not able to quantify these improvements. We divide this section into three parts.

1. Modifications to the current software stack
2. Suggestions for the development team
3. Technologies to look out for

These are further discussed in this section.

Modifications to the current software stack

In this section we describe some of the changes to the software stack that would result in a reduction of execution time and the effort required to make these changes.

1. *Use of shared memory instead one process "pushing" data*
   In the current implementation, when the division of processing seven field points in separate processes occurs, then data is pushed to all the seven processes sequentially. This results in low overlap of execution for the seven processes and thus higher execution time. If shared memory is used, all the PARIS processes can access the data after it has been written by the frame processing step. This will eliminate the need of copying data and reduce the execution time.

2. *Change from a process to a thread based implementation*
   In the current implementation, the seven field points are processed in seven separate "processes". A "process" is an operating system level entity. The other option available is to change from "process" implementation to a "thread" based implementation. "Threads" are supported at hardware level and hence will provide for higher performance, avoiding the expensive context switching between processes and other overheads.
   This change would come at a cost of making all the code and libraries thread safe and testing the code. Considering the size of the code, this might be too much work for performance gain and it is advised to do this modification only as the last resort to improve performance.

Suggestions for development team

1. *Feedback to the algorithm development team*
   Currently, people with a physics background develop the algorithms that are used for the PARIS sensor. These engineers may or may not have knowledge of where the computing world is heading. With the advent of parallel architectures and multicore processors becoming common in the industry, it is necessary that the software development team gives necessary feedback to the physics engineers to develop more parallel algorithms with low divergent branches.

2. *Evaluate use of double precision calculations*
   During our study of time consuming algorithms we found that double precision calculations had been used in all the algorithms without checking if there was a need for it. At some of the places we could change it to single precision but at some places, such a change would need to qualify tests which we could not do during this thesis. It is advisable to check if using double precision calculations is a requirement to obtain the desired accuracy in the results. This would give a performance gain depending on the platform (higher gain on GPUs and moderate gain on CPUs).

Technologies to look out for

1. *Single chip with integrated graphics*
   Some of the desktop processors from Intel, AMD etc. come with an integrated graphics card on chip with the main multicore CPU. Such a hardware platform would reduce/remove the overhead of copying data between CPU and GPU memory. Present day CPUs with integrated GPUs can be programmed with OpenCL but this is a new development and the drivers are still in the alpha stage. Secondly, to the best of our knowledge, no octacore CPU yet comes with an integrated GPU. This is something to keep an eye for, octa or more core CPUs with integrated on chip graphics card that could be programmed using OpenCL or OpenACC.

2. *Development of OpenCL, OpenACC*
   In this thesis we were limited to using CUDA because of performance benefits. However OpenCL and OpenACC are developing fast and are expected to provide better performance in comparison to what they offer now. Both of them would allow us to use other GPUs and remove the dependency on NVIDIA as sole supplier for GPUs. OpenACC is also expected to reduce development cost with some performance trade-off.

3. *Kepler architecture from NVIDIA*
   As discussed in Chapter 5, the Kepler architecture from NVIDIA provides features enabling parallel Kernel execution across CUDA streams. This would further reduce the GPU time. The gain from this cannot be easily approximated without actually doing experiments on a Kepler based GPU.

Bibliography

[1] P. E. Ross, "Why cpu frequency stalled," Apr 2013, http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled.

[2] N. Leischner, V. Osipov, and P. Sanders, "Fermi architecture white paper," 2009.

[3] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "Guest editors' introduction: High-performance reconfigurable computing," *Computer*, vol. 40, no. 3, pp. 23–27, 2007.

[4] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2013.

[5] G. E. Moore, "Readings in computer architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming more components onto integrated circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[6] von F. Zernike, "Beugungstheorie des schneidenver-fahrens und seiner verbesserten form, der phasenkontrastmethode," *Physica*, vol. 1, no. 712, pp. 689 – 704, 1934. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031891434802595

[7] "Sun m3000 datasheet," Mar. 2013, http://www.oracle.com/us/products/servers-storage/035939.pdf.

[8] "Ibm powerpc 750gx datasheet," Mar. 2013, https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/4D86B2273E8218CE87256E660058763D/$file/750GX_ds9-2-05.pdf.

[9] M. A. Herráez, D. R. Burton, M. J. Lalor, and M. A. Gdeisat, "Fast two-dimensional phase-unwrapping algorithm based on sorting by reliability following a noncontinuous path," *Appl. Opt.*, vol. 41, no. 35, pp. 7437–7444, Dec 2002. [Online]. Available: http://ao.osa.org/abstract.cfm?URI=ao-41-35-7437

[10] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '11. New York, NY, USA: ACM, 2011, pp. 203–204. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048207

[11] "Eds parallel ilias eds part i: Measurement sequence and data processing," ASML Internal documentation, Tech. Rep.

[12] "Cholesky decomposition," Mar. 2013, http://en.wikipedia.org/wiki/Cholesky_decomposition.

[13] "Lapack library webpage," Mar. 2013, http://www.netlib.org/lapack/.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[15] T. Holwerda, "Intel: Software needs to heed moore's law," Apr 2013, http://www.osnews.com/story/17983/Intel_Software_Needs_to_Heed_Moore_s_Law.

[16] B. Schauer, "Multicore processors–a necessity," *ProQuest Discovery Guides1–14*, 2008. [Online]. Available: http://www.csa.com/discoveryguides/multicore/review.pdf

[17] O. ARB, "The openmp api specification for parallel programming," Apr 2013, http://openmp.org/wp/.

[18] B. Barney, "Posix threads programming," Apr 2013, https://computing.llnl.gov/tutorials/pthreads/.

[19] "Intel threading building blocks," Apr 2013, http://software.intel.com/en-us/intel-tbb.

[20] B. Kuhn, P. Petersen, and E. OToole, "Openmp versus threading in c/c++," *Concurrency: Pract. Exper*, vol. 12, pp. 1165–1176, 2000.

[21] A. Binstock, "Threading models for high-performance computing: Pthreads or openmp?" Apr 2013, http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp.

[22] D. Patterson, "The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges," *NVIDIA Whitepaper*, 2009.

[23] K. Gray, *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Pr, 2003.

[24] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] A. Munshi *et al.*, "The opencl specification," *Khronos OpenCL Working Group*, vol. 1, pp. l1–15, 2009.

[26] "The openacc application programming interface," June 2013, http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf.

[27] K. O'Flaherty, "Nvidia, cray, pgi, and caps launch openacc programming standard for parallel computing." [Online]. Available: http://www.theinquirer.net/inquirer/news/2124878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing

[28] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[29] "Digital control for wafer stage 450mm -a computational perspective," ASML Internal documentation, Tech. Rep.

[30] P. J. Ashenden, *The designer's guide to VHDL*. Morgan Kaufmann, 2010, vol. 3.

[31] T. Feist, "Vivado Design Suite," Xilinx Inc., Tech. Rep., June 2012.

[32] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[33] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, "High performance comparison-based sorting algorithm on many-core gpus," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–10.

[34] N. Leischner, V. Osipov, and P. Sanders, "Gpu sample sort," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–10.

[35] A. Greb and G. Zachmann, "Gpu-abisort: optimal parallel sorting on stream architectures," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, pp. 10 pp.–.

[36] J. Hoberock and N. Bell, "Thrust: A parallel template library," *Online at http://thrust.googlecode.com*, 2010.

[37] "Cuda and openmp search trends," Jun 2013, http://www.google.com/trends/explore#q=CUDA%2C%20OpenMP&cmpt=q.

[38] "Video card benchmarks," Jun 2013, http://www.videocardbenchmark.net/gpu.php?gpu=GeForce+GTX+580.

[39] "Memory transfer overhead," Apr. 2013, http://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html.

[40] "Cuda data-parallel primitives library," Apr. 2013, http://cudpp.googlecode.com/svn/tags/2.0/doc/html/index.html.

[41] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[42] C. Nvidia, "Sdk code samples."

List of definitions

Diffraction The diffraction phenomenon is described as the apparent bending of waves around small obstacles and the spreading out of waves past small openings.

Focal plane The focal plane represents the area in a camera where light is focused.

Grating A grating is an optical component with a periodic structure, which splits and diffracts light into several beams traveling in different directions.

Interference In physics, interference is a phenomenon in which two waves superimpose to form a resultant wave of greater or lower amplitude.

Lens aberrations In an ideal optical system, all rays of light from a point in the object plane would converge to the same point in the image plane, forming a clear image. The influences which cause different rays to converge to different points are called aberrations.

Wave front In physics, a wave front is the locus of points having the same phase: a line or curve in *2D*, or a surface for a wave propagating in *3D*.

Description of Algorithms

A

A.1 Description of Algorithms

The details of this section cannot be made public due to confidentiality

Algorithms for Reduction on GPU

B

This appendix discusses the reduction algorithm used to find the pixel with maximum reliability.

B.1 Basic algorithm

The *kernel* code for our first implementation is shown in the listing below. This *kernel* is recursively invoked until the problem of finding the pixel with maximum reliability is reduced to only *1* pixel.

```
1  __global__ voidreduce1(int* g_idata, int* g_odata)
2  {
3    extern __shared__ int sdata[];
4    // each thread loads one element from global to shared memory
5    unsigned int tid = threadIdx.x;
6    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
7    sdata[tid] = g_idata[i];
8    __syncthreads();
9
10   // do reduction in shared memory
11   for(unsigned int s = 1; s < blockDim.x; s *= 2)
12   {
13     if(tid % (2 * s) == 0)
14     {
15       sdata[tid] = sdata[tid + s] > sdata[tid] ? sdata[tid + s] : sdata[tid
             ];
16     }
17     __syncthreads();
18   }
19   // write result for this block to global memory
20   if(tid == 0) g_odata[blockIdx.x] = sdata[0];
21 }
```

The *kernel* code listing showed above suffers from following problems which lead to higher execution time.

- The modulus (%) operator used for interleaved memory access is very slow.
- The divergent branch inside the loop makes threads divergent and reduces the performance.

B.2 Improved algorithm

The *kernel* code of our improved algorithm is shown in the listing below.

```
1  // perform first level of reduction,
2  // reading from global memory, writing to shared memory
3
4  unsigned int tid = threadIdx.x;
5  unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
6  sdata[tid] = g_idata[i + blockDim.x] > g_idata[i] ? g_idata[i + blockDim.x]
       : g_idata[i];
7  __syncthreads();
8
9  // reverse loop and thread id based sequential addressing as shown in Fig.
       5.8
10 for(unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
11 {
12   if (tid < s)
13   {
14     sdata[tid] = sdata[tid + s] > sdata[tid] ? sdata[tid + s] : sdata[tid];
15   }
16   __syncthreads();
17 }
```

Calculations for Execution Time

C

This appendix discusses the calculations and estimates used to approximate the execution time for Finalize Phase Fit and Phase Unwrapping algorithms on different hardware platforms.

C.1 Estimation of baseline performance

To estimate the time taken by these two algorithms in the baseline, we use the following steps.

- From Table 3.1 we know that Finalize Phase Fit and Phase Unwrapping algorithm together take approximately *63.3%* of the execution time for one field point.
- We assume that they take the same percentage of execution time when seven field points are computed.
- From Fig. 2.5 we know that baseline time for processing seven field points on SUN M3000 is *29.39*ms.
- Now, time taken by these two algorithms can be estimated as shown in Eqn. C.1.

$$T_{7,\text{Finalize Phase Fit, Phase Unwrapping}} = 63.3\% \text{ of } 29.39\text{ms} \;=\; \text{approx. } 18.60\text{ms} \qquad \text{(C.1)}$$

Thus, Finalize Phase Fit and Phase Unwrapping take *18.60*ms in the baseline. This implies that the time taken by rest of the algorithms in the baseline can be computed as shown in Eqn. C.2.

$$T_{7,\text{Rest of the algorithms}} = (29.39 - 18.60)\text{ms} \;= \text{approx. } 10.80\text{ms} \qquad \text{(C.2)}$$