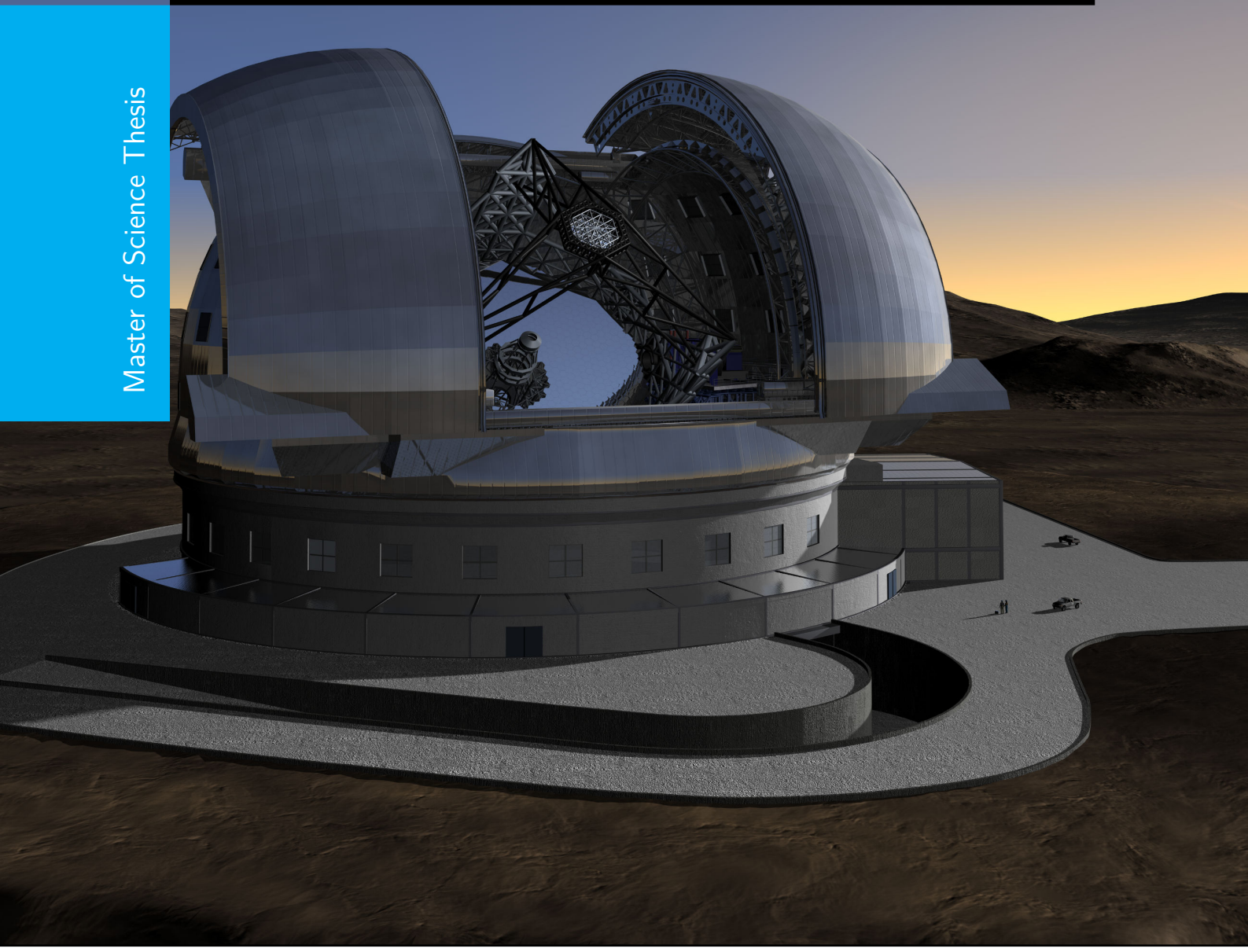
Development of an efficient parallel wavefront reconstructor

With implementation on a GPU

W.A. Klop

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Development of an efficient parallel wavefront reconstructor

With implementation on a GPU

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at Delft University of Technology

W.A. Klop

June 28, 2011

Faculty of Elektrotechniek Wiskunde en Informatica (EWI) · Delft University of Technology

Abstract

In 2018 the European extremely large telescope (E-ELT) a telescope with a diameter of 42 meter build by European Southern Observatory (ESO), is planned to see first light. Such a large telescope requires adaptive optics (AO) systems to compensate for atmospheric turbulence. An essential element of AO control is wavefront reconstruction. Wavefront reconstruction is the process of converting the measured slopes into phases. The slopes are measured by a wavefront sensor consisting out of a grid of separate sensor segments. When N defines the number of sensor segments, the computational effort to calculate the wavefront reconstructor scales with order $O(N^2)$ for standard vector matrix multiply (VMM) methods. Taking into consideration that for the E-ELT, N is in the range of 40.000, makes wavefront reconstruction computational infeasible.

This thesis proposes to use the vast amount of computational power that parallel computing offers to compute the wavefront reconstruction. To make effective use of parallel computing, it is required that a wavefront reconstructor is designed which consist out of a set of independent sub problems. This is achieved by treating every sensor segment as a separated problem, by identifying low order local models and by applying Kalman filtering on neighbouring measurements. The localised approach introduces a linear order in computational complexity and coarse grained parallelism. The tradeoff lies in noise sensitivity. Still the accuracy of the reconstructor does not drop for signal-to-noise (SNR) ratios of 80 dB and higher, when compared to a global Kalman filter. To reduce noise sensitivity further, more measurements per segment could be taken into account. An alternative would be to introduce state exchange in the form of diffusion algorithms. It is demonstrated that diffusion algorithms can give a gain of over 20 dB SNR. This is very promising, however in its current form diffusion algorithms are still computational too complex and future research should resolve this issue before being applicable.

To show that the algorithm tightly fits the parallel computing structure it was implemented on a graphics processing unit (GPU). On the GPU it was feasible to perform a single reconstruction for a grid size of 50×50 in 0.47 millisecond. Extrapolating the results allowed us to reason that 16 modern GPUs or a single future GPU is capable of reconstruction the wavefront of E-ELT sized grids at kilohertz rate.

Table of Contents

List of Figures

List of Tables

Preface & Acknowledgements

The graduation project is the last challenge before achieving your master degree. I conducted my graduation at the Technical University of Delft for the department Delft Center for Systems and Control (DCSC). The document at hand describes the research I have performed during my graduation. As with every research it started with stating a proper thesis question. This was done in collaboration with my supervisor prof.dr.ir. Michel Verhaegen and my daily supervisor dr.ir. Rufus Fraanje. Resulting in the subject: "Selection and adaption of an adaptive optics control algorithm to compensate for atmospheric turbulence, such that it is suitable to be implemented on an array of processing units. The goal is to set the requirements for the European extremely large telescope (E-ELT) application in terms of processing bandwidth and architecture modular scalability". To be more precise, I concerned myself primarily with the computational complexity issue related to the wavefront reconstruction process. This is due to the fact that wavefront reconstruction is the most computational complex task of the control process. To keep my research on track I received frequent supervision of both my supervisors. For that I like to thank both of them.

Delft, University of Technology — W.A. Klop
June 28, 2011

"People asking questions, lost in confusion, well I tell them there's no problem, only solutions."

— *John Lennon* —

Chapter 1

Introduction

Exploring is an essential component of the human character, as long as humans exist we are wondering what the stars in the universe will bring us. To find answers scientists across the world are building a wide variety of instrumentation. Among them are the astronomers of European Southern Observatory (ESO) and the astronomers of the Thirty Meter Telescope (TMT). One of the most important ground based instruments for astronomers is the telescope. Driven by our curiosity we are constructing increasingly larger telescopes. By now we arrived at the era of the extremely large telescopes (ELTs), which is a class of telescopes where the aperture exceeds the 20 meter range. Constructing telescopes from this kind of magnitude comes with numerous challenges. One of these challenges is controlling the Adaptive Optics systems which will be addressed in this document.

1-1 Telescopes

With the invention of the telescope in 1608 [2] it became possible to study the universe in much more detail. In contrast to the only possible way till then: the naked eye. The first telescope existed of an objective and ocular, also referred to as a refractor based telescope. The growth in diameter of the telescopes required another type of design. Isaac Newton introduced in 1668 a design based on a curved mirror also known as a reflective telescope. Despite the theoretical advantages, production technologies held back the introduction of the reflecting telescope till the 18th century. Up till this moment most astronomical telescopes are based on this type of design. Increasing the diameter of the telescopes is driven by two important properties in any optical imaging system: the light collecting power and the angular resolution. Rayleigh defined a criterion that gives an estimation of the angular resolution

$$sin\theta \approx 1.22\frac{\lambda}{\varnothing} \tag{1-1}$$

Where λ is the wavelength of the light emitted by the object that is observed, $\varnothing$ defines the diameter of the aperture and θ gives the angular resolution. The smaller the angular

resolution the more detail can be observed. Eq. (1-1) results in the fact that by increasing $\varnothing$ the angular resolution will improve. Nowadays this will assure that an ELT can deliver an important contribution to our astronomical knowledge. Several of this type of ELTs are planned to be build in the coming decade.

1-2 Adaptive optics in astronomy

To be able to use the full extent of an ELT a problem has to be overcome. When the diameter of the telescope is enlarged beyond approximately 0.2 meter the angular resolution is no longer diffraction limited, but seeing limited, this is caused by turbulence in the earth's atmosphere. Light collected by a ground based telescope has to pass the earth's atmosphere. The light travelling through the atmosphere gets distorted caused by temperature differences, resulting in a blurry image. One solution would be to place the telescope outside the earth's atmosphere as was done with the Hubble telescope. Two major disadvantages that come with this solution is the inability to do rapid maintenance or repair and the extreme costs related to such a telescope. Another critical limitation is the size of the telescope. Bigger telescopes are advantageous for the light collecting power. However travelling to space is still complicated and does not allow for large payloads.

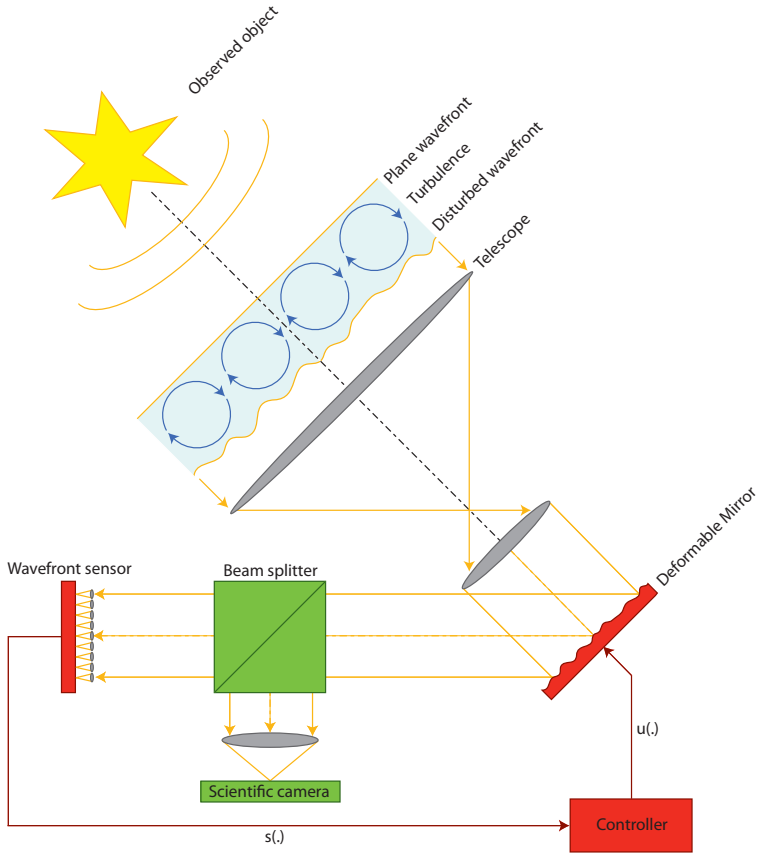


Figure 1-1: Schematic representation of a Adaptive Optics system

A more cost effective option which is often chosen at the moment is adaptive optics (AO). AO is a technology to reduce the effects of wavefront distortions and improving imaging performance. AO achieves this by correcting phase differences originating from atmospheric disturbance. Atmospheric disturbance which can be represented as a state space innovation model of the form

$$\begin{aligned} \xi_{k+1} &= A_d \xi_k + K_d e_k && \text{(1-2)} \\ \phi_k &= C_d \xi_k + e_k && \text{(1-3)} \end{aligned}$$

where ϕ is the phase difference, k is the time index, e is a Gaussian distributed white noise process, A_d is the state transition matrix, K_d is the kalman gain and C_d is the output matrix. Furthermore AO can be divided in three subsystems (Figure 1-1). The wavefront sensor (WFS) measures the phase distortion in the form of gradients $s_x[i,j]$ and $s_y[i,j]$. Where $s_x[i,j]$ and $s_y[i,j]$ define the first differences between adjacent phase points

$$\begin{aligned} s_x[i,j] &= \phi[i+1,j] - \phi[i,j] && \text{(1-4)} \\ s_y[i,j] &= \phi[i,j+1] - \phi[i,j] && \text{(1-5)} \end{aligned}$$

A Shack-Hartman sensor is a commonly applied type of WFS. As each measurement is corrupted by noise the sensor can be modelled as such

$$s_k = G\phi_k + n_k. \tag{1-6}$$

Where G is the phase-to-WFS influence matrix and n is the WFS measurement noise. Leading us to the next subsystem, the controller. In the controller measurements are used to determine the optimal positions of the deformable mirror (DM) actuators. The DM which is the third subsystem, compensates for the measured distortion. Defining H as the DM influence matrix, u as the control inputs and z^{-1} as a discrete shift operator. The following equation represents the correction of the DM actuator

$$\phi_{dm,k} = z^{-1} H u_k. \tag{1-7}$$

Hence, the residual phase error is defined Eq. (1-8). The objective in AO is to minimise this residual error resulting in a reconstruction of the associated wavefront and improving the image quality

$$\phi_{\epsilon,k} = \phi_k + \phi_{dm,k}. \tag{1-8}$$

In [3, p. 110] an optimal expression for the control signals u_k is given such that equation (1-8) is minimised. In addition assume a single sample delay such that u_k can be given by equation (1-9)

$$u_k = (I - H_Q^\dagger H z^{-1})^{-1} H_Q^\dagger \hat{\phi}(k+1|k) \tag{1-9}$$

where $H_Q^\dagger$ is the regularised pseudo inverse of the DM influence matrix H and $\hat{\phi}(k+1|k)$ are the estimated wavefront phases predicted one step ahead. The estimated wavefront phases $\hat{\phi}$ can be derived from the measurements s_k and is referred to as wavefront reconstruction.

1-3 Computational explosion

Not only the mechanical structure increases when building an ELT, but also the control problem. Currently most control algorithms, used in the AO systems of telescopes, use matrix inversions or other computational intensive computations to determine the control signals u_k. Up till the emergence of ELTs the implementation of these methods were not a insurmountable problem. With the increasing size of mirrors also the number of WFS segments and the DM actuators increase. Keeping in mind the limit of approximately 0.2 meter for which telescopes are still diffraction limited. The E-ELT will have a diameter of 42 meter which results in a total amount of actuators N that is expected to be in the order of 10.000 - 40.000. Making an algorithms with a computational complexity of order $O(N^2)$ or higher not acceptable. Already significant amount of research is done to more efficient algorithms. In [4] it is shown that even with these more efficient algorithms it is expected that solving the control problem is still not feasible in real-time, when considering current performance levels and the development of computing power defined by Moore's law. Also in [4] they point out field programmable gate arrays (FPGAs), graphics processing units (GPUs) and even more efficient algorithms as possible research strategies. Therefore there is a need for further research in the development of efficient algorithms.

Each scientific problem can be optimised at several levels, optimising at one level can have an influence on an other level. Note that software and hardware are tightly interconnected, changing the hardware often results in software changes. Even more, preference for specific hardware can also influence the algorithm design or the other way around. Making the total system design from algorithm till hardware a complex interweaved process (Figure 1-2).

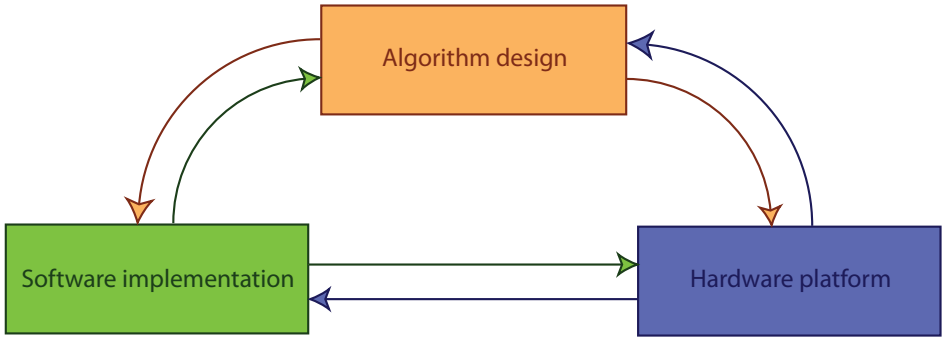


Figure 1-2: Interweaved process of optimising a system design

1-4 Parallel Computing

Usually when a problem has to be solved an algorithm is developed and afterwards the algorithm is implemented as a sequential set of instructions. These instructions are then executed

on a single processing unit. The processing unit executes one instruction at a time, after the first instruction is finished the next is started. This cycle keeps going till the algorithm is completed. The concept is easily comprehensible for programmers and as such it is straightforward to apply, which makes it an attractive option. An alternative approach is parallel computing. This technology uses multiple processing units to work simultaneously on the same problem. To accomplish parallelism the algorithm has to be broken up into several independent sub problems. These sub problems then can be distributed over the multiple processing units. The processing units simultaneously solve the sub problems, where each processing unit solves a sub-problem sequentially. Parallel computing is a well establish field and its primary domain of interest was high-performance computing. This domain of interest is shifting due to physical limitations that prevent further frequency scaling. Frequency scaling was from around 1985 till 2004 the dominant method used by integrated circuit (IC) manufactures to increase the performance. The execution time of a program can be determined by counting the number of instructions and multiplying this by the average instruction runtime. When decreasing the average instruction runtime can be achieved by increasing the clock frequency of the processing unit, this automatically results in a lower execution time of the overall program. For manufacturers reason enough to let frequency scaling be a dominant design parameter for several decades.

Frequency scaling became infeasible to hold as dominant design parameter. Power consumption is the fundamental reason for the ceasing enhance in frequency scaling. The power consumption of an IC is defined as Eq. (1-10) in [5, p. 18].

$$P = \frac{1}{2}CV^2F, \tag{1-10}$$

Where C is the capacitance switched per clock cycle. V is the supply voltage and F is the frequency the IC is driven by. The equation points out the problem, increasing the frequency will also increase the power consumption, if both the capacitance and the voltage are kept equal. Since also the voltage was reduced from 5 volt till 1 volt over the past 20 years some additional headroom was available. Slowly physical limitations were emerging. Burning vast amounts of power meant that temperatures were rising to unacceptable levels.

Gordon E. Moore made the observation that since the invention of the IC, the number of components in IC's doubled every year [6]. Moore also predicted that this trend would carry on, nowadays referred to as Moore's Law. Later on Moore refined the law and changed the period into two years. Till now Moore's law it still valid. From the beginning these new available resources were used to support higher frequencies. Since frequency scaling is no longer the dominant design parameter these components can be used for other purposes. Manufacturers are now aiming at increasingly larger numbers of cores favouring parallel computing.

For some basic understanding in the issues of parallel computing Appendix B could be helpful.

1-5 Research Problem

In [7] a study to fast wavefront reconstruction algorithms was presented. The first important conclusion drawn in [7] as also discussed above, is that parallel devices will become standard, making parallel computing crucial to use the devices to their full extend. Second in order to

efficiently map algorithms to a parallel version it is important that parallelism is addressed as early as the design phase.

Already experiments emphasising on a parallel AO implementation with respect to ELTs were published. In [8] 16 Nvidia 8800 ultra fast GPUs, distributed over 8 dual core HP Opteron computers are used to solve a 64×64 sized AO problem. Applying regular VMM methods they achieve a framerate of 2 kHz. Similar, in [9] 9 boards, containing six FPGAs each controlled by two general purpose computer boards packed with three Intel L7400 dual core processors, solve approximately a 86×86 sized AO problem. In this case a conjugate gradient method is utilised to achieve a framerate of 800 Hz. Both options succeed for the given problem, however they are still relatively small compared to the ELT related AO problem and use impressive hardware setups already.

Therefore the problem considered in this thesis is:

"The development of an adaptive optics control algorithm to compensate for atmospheric turbulence, such that it is suitable to be implemented on an array of processing units. The goal is to set the requirements for the E-ELT application in terms of processing bandwidth and architecture modular scalability"

Where the emphasis will lie on the wavefront reconstruction as this is the most computational complex process within AO control.

1-6 Outline

This document describes the design and evaluation of a linear scaling wavefront reconstructor. The wavefront reconstructor is composed of a set of independent sub problems allowing for a full parallel implementation on for example a GPU.

The thesis is organised as follows. Chapter 2 applies the parallel computing concept by discussing the design of a set of independent local wavefront reconstructors which together solve the wavefront reconstruction for a whole grid. A theoretical evaluation at the end of the chapter shows that the method is computationally very efficient but is relatively sensitive to noise. Chapter 3 therefore proposes to use state exchange in the form of diffusion strategies to reduce the noise sensitivity of the localised reconstructor. Chapter 4 goes into depth on how to implement the designed algorithm on a parallel computing device. The high number of sub problems made the GPU the device of choice. Chapter 5 gives an extended evaluation of the results achieved with the suggested methods including its strengths and weaknesses. Chapter 6 ends the research with conclusions and recommendations based on the findings of the previous chapters.

Chapter 2

Wavefront Reconstructor

A critical performance component of the adaptive optics (AO) system is the control algorithm, since this is what makes the AO system function. Till now there was no real need for computational efficient implementations. This has changed with the upcoming of ELT's. The by far most computationally complex task of the control algorithm is the wavefront reconstructor. Wavefront reconstruction is the process where slopes (s_k) are used to estimate wave front phases ($\hat{\phi}_k$) based on Equation (1-6). For conventional methods it scales as $O(N^2)$, where N is the number of lens lets in the WFS. One can imagine that with 40.000 lens lets this is computational infeasible.

Often the choice for the reconstructor lies in a tradeoff between performance and accuracy. There are already several suggestions for algorithms that are optimised, e.g. the FFT solution proposed in [10] is relatively efficient however it compromises in accuracy. On the other side of the spectrum are the model based predictors which are more accurate [11, 12] but rather computational complex. Recently also a structured Kalman predictor is suggested in [13] which uses sequentially semi separable (SSS) matrices to improve the computational complexity. However, all these algorithms are not designed with parallelism in mind which can make it troublesome to map them to a parallel version (see appendix B and [7]). The underlying cause for this are data dependencies throughout algorithms. With the knowledge that chip manufactures strongly rely on parallel computing to achieve performance enhancements this can be a significant drawback. As described in the parallel computing chapter, the success of a parallel implementation hugely depends on the algorithm under consideration. Consequently it is necessary to start with designing an algorithm with parallelism in mind.

This chapter is organised as follows: Section 2-1 starts with introducing a turbulence model and the global Kalman reconstructor. Section 2-2 proposes an alternative localised Kalman reconstructor to solve the computational complexity issue. Section 2-3 describes the identification procedure for the localised reconstructor. Section 2-4 compares the proposed algorithm with the global Kalman reconstructor in terms of accuracy. Section 2-5 determines the gain in computational complexity.

2-1 Reconstructor based on a global approach

The foundation of the algorithm design is based on a simulation model, which assumes that the spatial behaviour is well described by the von Kármán turbulence model and the temporal evolution is obeying frozen flow behaviour. In [14] a mathematical description of the continuous time von Kármán model is given. The focus for the wavefront reconstruction will lie on discrete time as this is also required by the implementation technology. To be able to discretise assume equidistant sampling for both the temporal as the spatial domain. Let ΔX and ΔY define the resolution for respectively the x-axis and y-axis and similar for the time domain the sampling time is given by ΔT. The phase of the wavefront on time instant $t = k\Delta T$ and located on the spatial coordinate $(i\Delta X, j\Delta Y)$ will be denoted by $\phi(k, i, j)$. Equation (2-1) now specifies the discrete spatial correlation according to the von Kármán model.

$$E(\phi(k,i,j)\phi(k,i-m,j-n)) = c\left(\frac{2\pi r}{L_0}\right)^{5/6} K_{5/6}\left(\frac{2\pi r}{L_0}\right)^{5/6} \tag{2-1}$$

where

$$c = \frac{\Gamma(11/6)}{2^{5/6}\pi^{8/3}}\left(\frac{24}{5}\Gamma(6/5)\right)^{5/6}\left(\frac{L_0}{r_o}\right)^{5/3}, \tag{2-2}$$

$$r = \sqrt{(m\Delta X)^2 + (n\Delta Y)^2} \tag{2-3}$$

and L_0 is the outer scale of the turbulence, r_0 the Fried parameter, $E(.)$ the expectation operator, $\Gamma(.)$ the Gamma function and $K_{5/6}(.)$ the modified Bessel function of the third type of order 5/6. Further, the frozen flow assumption encompasses that the wavefront propagates in time through space with a constant velocity according to

$$\phi(k+1,i,j) = \phi(k, i - v_x\Delta T/\Delta X, j - v_y\Delta T/\Delta Y) \tag{2-4}$$

where v_x and v_y are respectively the velocities in the x- and y- direction. From equation (2-4) it is clear that inconsistencies in the discrete spatial domain can occur if the velocity does not match the propagation in time. So for simplification reasons assume that this is guaranteed such that $(v_x\Delta T/\Delta X, v_y\Delta T/\Delta Y) \in \mathbb{Z}$. By joining equations (2-1) and (2-4) we are left with

$$E(\phi(k,i,j)\phi(k-l,i-m,j-n)) = c\left(\frac{2\pi r}{L_0}\right)^{5/6} K_{5/6}\left(\frac{2\pi r}{L_0}\right)^{5/6} \tag{2-5}$$

where $l = v_x\Delta T/\Delta X + v_y\Delta T/\Delta Y$ and

$$r = \sqrt{(m\Delta X - v_x\Delta T)^2 + (n\Delta Y - v_y\Delta T)^2} \tag{2-6}$$

Besides the phase reconstruction from equation (1-6) it is also advantageous to do a prediction due to the fact that in a control loop there is always a delay between the sensor measurement

and the actual control. To be able to perform predictions it is required to have a model of the plant. A common choice would be to model the turbulence statistics by a finite order state-space innovation model. Where the state-space innovation model is given by

$$\begin{aligned} x(k+1) &= Ax(k) + Ke(k) && (2\text{-}7)\\ \phi(k) &= Cx(k) + e(k) && (2\text{-}8)\\ y(k) &= GCx(k) + Ge(k) + n(k) && (2\text{-}9) \end{aligned}$$

where $e(k) \in \mathbb{R}^{N_x N y}$ is a zero-mean white noise stochastic process with variance $E[e(k)e(k)] = R_e$, $x(k) \in \mathbb{R}^{N_m}$ the state vector of the innovation model and the matrices $A \in \mathbb{R}^{N_m \times N_m}$, $C \in \mathbb{R}^{N_x N y \times N_m}$ and $K \in \mathbb{R}^{N_m \times N_x N y}$ are the state-transition matrix, the output matrix and the Kalman gain respectively. The dimensions N_x, N_y and N_m define respectively the grid size in the x-direction, the grid size in the y-direction and the model order. The approach for retrieving this state-space model was introduced in [15]. Stochastic realisation theory [16] was used to determine the state-space matrices deduced from the correlation coefficients $C_\phi(l) := E[\varphi(k)\varphi(k-l)^T]$, where $\varphi(k)$ is a stacked phase vector defined by equation (2-10).

$$\varphi(k) := \begin{bmatrix} \phi(k,1,1) & \dots & \phi(k,1,N_y) & \dots & \phi(k,N_x,N_y) \end{bmatrix}^T \in \mathbb{R}^{N_x N_y} \quad (2\text{-}10)$$

Performing a stochastic realisation requires a singular value decomposition (SVD) and will scale with $O(N_x^3 N_y^3)$. So to reduce the computational complexity and keep the process feasible we can now exploit a structure that is imposed by the frozen flow assumption. In [12] the approach is extensively described resulting in the following global state-space description of a 1-step ahead predictor.

$$\begin{aligned} \hat{e}(k) &= \bar{y}(k) - \bar{C}^y \hat{x}(k|k-1) && (2\text{-}11)\\ \hat{x}(k+1) &= \bar{A}\hat{x}(k|k-1) + \bar{K}_y \hat{e}(k) && (2\text{-}12)\\ \hat{\phi}(k+1|k) &= \bar{C}\hat{x}(k+1|k) && (2\text{-}13) \end{aligned}$$

where $\bar{C}^y = G\bar{C}$, $\bar{D}^y = G\bar{D}$ and the matrices $\bar{A}, \bar{K}, \bar{C}$ and $\bar{D}$ are defined as

$$\begin{aligned} \bar{A} &= \begin{bmatrix} A & & & & \\ C & 0 & & & \\ & I_{N_y} & 0 & & \\ & & \ddots & \ddots & \\ & & & I_{N_y} & 0 \end{bmatrix}, \qquad \bar{K} = \begin{bmatrix} K \\ I_{N_y} \\ 0 \\ \vdots \\ 0 \end{bmatrix},\\ \bar{C} &= \begin{bmatrix} C & & & \\ & I_{N_y} & & \\ & & \ddots & \\ & & & I_{N_y} \end{bmatrix}, \qquad \bar{D} = \begin{bmatrix} I_{N_y} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \end{aligned} \quad (2\text{-}14)$$

and $\bar{K}^y$ is the Kalman gain which is defined as such

$$\bar{K}^y = (\bar{A}\bar{P}(\bar{C}^y))^T + \bar{S}(\bar{C}^y\bar{P}(\bar{C}^y)^T + \bar{R})^{-1} \tag{2-15}$$

and $\bar{P}$ is the stabilising positive definite solution of the DARE

$$\bar{P} = \bar{A}\bar{P}\bar{A}^T - (\bar{A}\bar{P}(\bar{C}^y)^T + \bar{S})(\bar{C}^y\bar{P}(\bar{C}^y)^T + \bar{R})^{-1}(\bar{A}\bar{P}(\bar{C}^y)^T + \bar{S})^T + \bar{Q} \tag{2-16}$$

Let us for this define $\bar{K}e(k)$ as process noise and $v(k)$ as measurement noise were both are assumed to be zero-mean white-noise sequences, then the joint covariance matrix is given by

$$E\left[\begin{bmatrix} v(k) \\ \bar{K}e(k) \end{bmatrix}\begin{bmatrix} v(k)^T & (\bar{K}e(k))^T \end{bmatrix}\right] = \begin{bmatrix} \bar{R}(k) & \bar{S}(k)^T \\ \bar{S}(k) & \bar{Q}(k) \end{bmatrix} \tag{2-17}$$

Such that we have

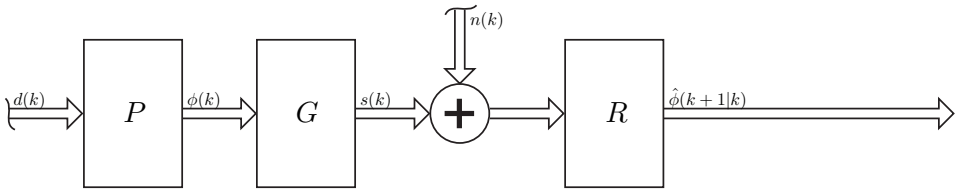
$$\bar{Q} = \bar{K}\bar{R}_e\bar{K}^T \tag{2-18}$$
$$\bar{R} = \bar{D}^y\bar{R}_e(\bar{D}^y)^T + \sigma_v^2 I \tag{2-19}$$
$$\bar{S} = \bar{K}\bar{R}_e(\bar{D}^y)^T \tag{2-20}$$

The reconstructor derived here is a global reconstructor and will be used for comparison throughout the document. Also it will be used as bases for the localised reconstructor which will be show now.

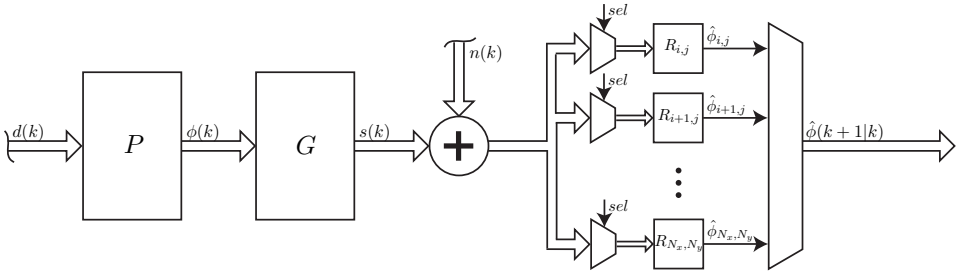
2-2 Reconstructor based on a localised approach

The Kalman reconstructor given by equations (2-11) till (2-13) scales with $O(N_m^2)$. The order N_m of the global model has a strong relation to the grid size and should at least be equal to the number of segments. With the coming of the E-ELT this calculation becomes infeasible such that an alternative method is required. Till this moment the problem of reconstruction was treated as one single problem. Where actually it can also be represented as a set of problems with coupling between the subproblems. By the structure of the WFS the spatial 2D grid of the wavefront is already split into segments, suggesting that each segment can be treated as a subproblem. Figure 2-1 depicts the differences between the approaches, where P represents the atmospheric turbulence, G is the WFS influence matrix, R denotes a reconstructor and *sel* denotes a selection signal. The other variables in the figures adopt the earlier used notation. In (a) a centralised reconstructor can be recognised and is a generalisation of the global Kalman reconstructor. Both (b) and (c) represent a decentralised reconstructor, the difference between them lies in the additional communication between reconstructors in figure (c). In this section a version of (b) is proposed, the main advantage of this approach is the lack of dependencies between reconstructors which makes it an ideal parallel algorithm. An example of approach (c) can be found in Chapter 3. For approach (c) it should be noticed

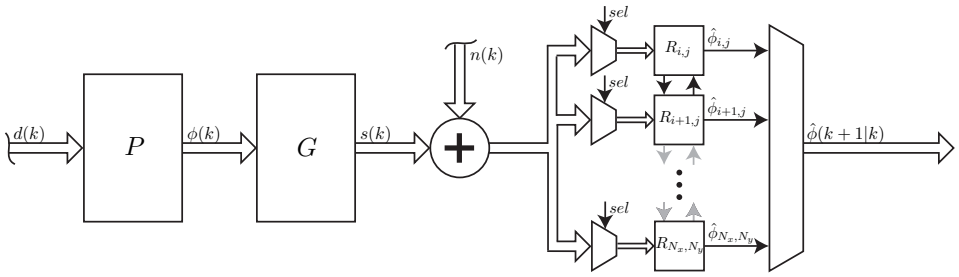
that depending on the form of communication the dependencies can give problems when implementing it in parallel [7].



(a) centralised reconstructor



(b) isolated decentralised reconstructor



(c) collaborating decentralised reconstructor

Figure 2-1: block diagrams of respectively a centralised reconstructor, a isolated decentralised reconstructor and a collaborating decentralised reconstructor

In the localised approach each subproblem determines a wavefront phase point separately, solely based on local information. This neglects some of the coupling between segments, focusing more on exploiting the temporal information to reconstruct the wavefront. By giving each problem also access to some spatial information the coupling between the segments can be partially restored. The spatial information is represented by measurements of the WFS. This immediately suggest that the more measurements are used the more spatial knowledge is available. Such that we are in the search for a set of local models. Let a set of local state-space models be defined as follows.

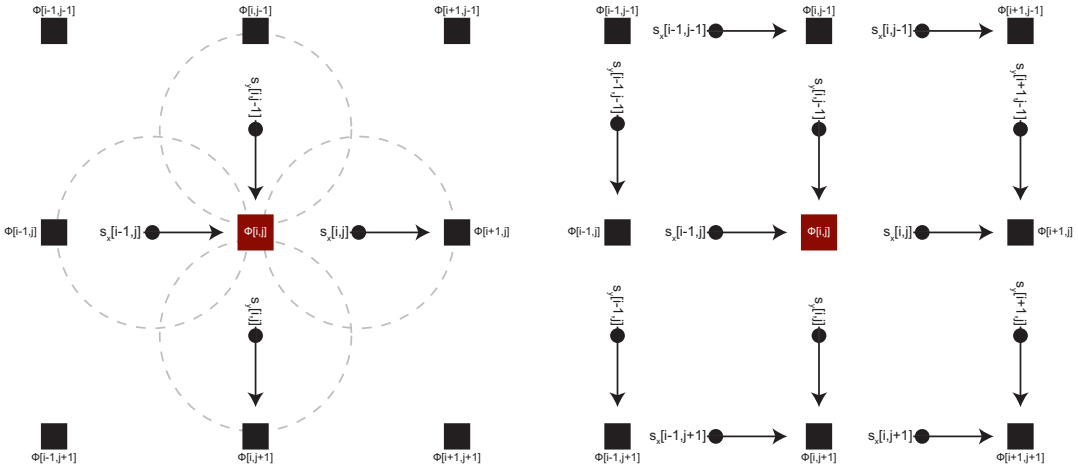
$$x^l_{(i,j)}(k+1) = A^l_{(i,j)}x^l_{(i,j)}(k) + K^l_{(i,j)}e^l_{(i,j)}(k) \tag{2-21}$$

$$\phi_{(i,j)}(k) = C^l_{\phi(i,j)}x^l_{(i,j)}(k) + e^l_{(i,j)}(k) \tag{2-22}$$

$$y^l_{(i,j)}(k) = C^l_{(i,j)}x^l_{(i,j)}(k) + e^{yl}_{(i,j)}(k) + n^l(k) \tag{2-23}$$

To be able to explicitly define the local models we have to choose which measurements from the spatial domain are considered. The accuracy of the final reconstruction and prediction will strongly depend on the number of measurements per local reconstructor that are taken into account, as additional measurements will likely improve reconstruction accuracy. However there is a tradeoff, the increase in measurements will have a negative impact on the computational complexity. So the best way is to choose as few measurements as possible that still achieve acceptable accuracy.

Beside the amount of measurements there is the question which measurements are significant. It is for example likely that measurements in the proximity of a certain grid point will have a high correlation with the phase related to that specific grid point, while this correlation diminishes over distance. The previous statements suggest that there is an optimal choice in selecting the right measurements. For now determining this optimal set of measurements will lead too far. Since determining the optimal set without considering structure is an NP hard problem. Such that based on sensor knowledge modelled by equation (1-4) two possible measurement sets are selected as depicted in Figure 2-2. The selections are chosen such that the difference in the number of measurements can be demonstrated in both accuracy and performance. The figures show a partial section of a sensor grid where dotted circles in the left figure represent sensor segments, the arrows represent a specific measurement in a either the x or y direction and the squares define the wavefront phase points.



(a) measurements level 1 (b) measurements level 2

Figure 2-2: Selection of measurements for a local model, two possible selections are chosen to demonstrate the difference in the number of measurements taken into account.

We can mathematically define the measurements for the level 1 selection as the set

$$y^l_{(i,j)}(k) = [s_x(i-1,j), s_x(i,j), s_y(i,j-1), s_y(i,j)]^T \tag{2-24}$$

and for level 2 measurements

$$\begin{aligned} y^l_{(i,j)}(k) &= [s_x(i-1,j-1), s_x(i,j-1), s_x(i-1,j), s_x(i,j), s_x(i-1,j+1) \\ &, \quad s_x(i,j+1), s_y(i-1,j-1), s_y(i,j-1), s_y(i+1,j-1), s_y(i-1,j) \\ &, \quad s_y(i,j), s_y(i+1,j)]^T \end{aligned} \tag{2-25}$$

The local models can be derived from the global models. In the global model the wavefront phases $\phi(k)$ are stacked in the output matrices. So let α define the index of segment (i,j) related to a locations in the global matrices. Similar let β be a vector containing the indices of the set of measurements. Resulting in the local estimators

$$\hat{e}^l_{(i,j)}(k) = y^l_{(i,j)}(k) - C^l_{(i,j)}\hat{x}^l_{(i,j)}(k|k-1) \tag{2-26}$$

$$\hat{x}^l_{(i,j)}(k+1|k) = A^l_{(i,j)}\hat{x}^l_{(i,j)}(k|k-1) + K^{yl}_{(i,j)}\hat{e}^l_{(i,j)}(k) \tag{2-27}$$

$$\hat{\phi}_{(i,j)}(k+1|k) = C^l_{\phi(i,j)}\hat{x}^l_{(i,j)}(k) \tag{2-28}$$

where the matrices are defined as in equations (2-29) till (2-33). For these equations Matlab like notation is adopted to define.

$$A^l_{(i,j)} = \bar{A} \tag{2-29}$$

$$C^l_{\phi(i,j)} = \bar{C}(\alpha, 1:N_m) \tag{2-30}$$

$$C^l_{(i,j)} = \bar{C}^y(\beta, 1:N_m) \tag{2-31}$$

$$D^l_{(i,j)} = \bar{D}^y(\beta, 1:N_u) \tag{2-32}$$

$$K^l_{(i,j)} = \bar{K} \tag{2-33}$$

Where $\alpha = (j-1)Nx+i$, $\beta = [(j-1)Nx+i-1, (j-1)Nx+i, \dots]^T$ and $K^{yl}_{(i,j)}$ is the Kalman gain is defined as such

$$K^{yl}_{(i,j)} = (A^l_{(i,j)}P^l_{(i,j)}(C^{yl}_{(i,j)}))^T + S^l_{(i,j)})(C^{yl}_{(i,j)}P^l_{(i,j)}(C^{yl}_{(i,j)})^T + R^l_{(i,j)})^{-1} \tag{2-34}$$

and $P^l_{(i,j)}$ is the stabilising positive definite solution of the DARE

$$\begin{aligned} P^l_{(i,j)} &= A^l_{(i,j)}P^l_{(i,j)}{A^l_{(i,j)}}^T - (A^l_{(i,j)}P^l_{(i,j)}(C^{yl}_{(i,j)})^T + S^l_{(i,j)})(C^{yl}_{(i,j)}P^l_{(i,j)}(C^{yl}_{(i,j)})^T + R^l_{(i,j)})^{-1} \\ &\times (A^l_{(i,j)}P^l_{(i,j)}(C^{yl}_{(i,j)})^T + S^l_{(i,j)})^T + Q^l_{(i,j)} \end{aligned} \tag{2-35}$$

where

$$Q^l_{(i,j)} = K^l_{(i,j)}R^l_e{K^l_{(i,j)}}^T \tag{2-36}$$

$$R^l_{(i,j)} = D^{yl}_{(i,j)}R^l_e(D^{yl}_{(i,j)})^T + \sigma_v^2 I \tag{2-37}$$

$$S^l_{(i,j)} = K^l_{(i,j)}R^l_e(D^{yl}_{(i,j)})^T \tag{2-38}$$

Equations (2-26) till (2-27) describe local models. There is only a strong drawback, the order N_m of a single local model is in the same range as the global model. Further, note that equations (2-26) till (2-28) have to be evaluated $N_x N_y$ times. So the only gain is that we are now able to implement the reconstruction process in parallel at the cost of a tremendous increase in computational complexity. To overcome this we should reduce the order N_m, in section 2-3 a possible strategy is given.

2-3 Identification local predictors

To make the local models applicable a crucial step still has to be performed, a reduction in model size. Model order reduction methods would solve this complication, but instead of applying order reduction methods to the derived models it is more effective to apply identification methods. This is because of the following, the global model has to be identified first which is computationally very complex as it scales as $O(N^3)$, so by identifying the local models directly we can skip the global identification process and an order reduction. This will make the process a lot more efficient. Identification processes as e.g the subspace model identification described in [17, sec. 9.6] can be used. With subspace identification it is possible to estimate a state space model based on input and output data. As the turbulence model does not have a deterministic input the obvious choice would be to estimate an output only model. By reformatting Equations (2-26) till (2-27) a model structure is obtained that is suitable for subspace identification. Such that the problem can be formulated as given $\left\{y^l_{(i,j)}(k), \phi(k,i,j)\right\}_{i=1,j=1}^{N_x,N_y}$ estimate

$$x^{id}_{(i,j)}(k+1) = A^{id}_{(i,j)} x^{id}_{(i,j)}(k) + K^{id}_{(i,j)} e_{(i,j)}(k) \tag{2-39}$$

$$y^{id}_{(i,j)}(k) = C^{id}_{(i,j)} x_{(i,j)}(k) + G_{(i,j)} e_{(i,j)}(k) + n_{(i,j)}(k) \tag{2-40}$$

where the identification output data vector is defined as equation (2-41). Notable is that also the wavefront phases $\phi(k)$, are required, however these are not measured directly. Since the identification is an offline procedure it is allowed to use more computational complex methods to predetermine the wavefront phases using alternative model less or white box model based methods.

$$y^{id}_{(i,j)}(k) = \left[\ \phi_{(i,j)}(k) \mid y_{(i,j)}(k)\ \right]^T \tag{2-41}$$

Denote that currently the measurements and the wavefront phase are stacked in the output equation. Resulting that $C^{id}_{(i,j)}$ is the stacking of $[C^l_{\phi(i,j)} C^{yl}_{(i,j)}]^T$ with $C^l_{\phi(i,j)} \in \mathbb{R}^{1\times 1}$ and $C^{yl}_{(i,j)} \in \mathbb{R}^{N_u \times 1}$. Since only the slopes are measured, after identification a partitioning should be made to be able to determine the correct Kalman gain. Leading us to the next procedure which is simulating the model with the output vector $y^{id}_{(i,j)}$ as input. From this we obtain an estimated output vector $\hat{y}^{id}_{(i,j)}$, which we can exploit to derive the joint covariance matrix related to the model defined by equations 2-39 and 2-40.

$$\begin{aligned} Re^{id} &= cov(y^{id}_{(i,j)} - \hat{y}^{id}_{(i,j)}) = \begin{bmatrix} Re^{id}_{11} & Re^{id}_{12} \\ Re^{id}_{21} & Re^{id}_{22} \end{bmatrix} \\ Q^{id}_{(i,j)} &= K^{id}_{(i,j)} Re^{id} (K^{id}_{(i,j)})^T \\ S^{id}_{(i,j)} &= K^{id}_{(i,j)} [Re^{id}_{12}; Re^{id}_{22}] \\ R^{id}_{(i,j)} &= Re^{id}_{22} \end{aligned} \tag{2-42}$$

Where the block covariance matrices are $Re^{id}_{11} \in \mathbb{R}^{1\times 1}$, $Re^{id}_{12} \in \mathbb{R}^{1\times N_u}$, $Re^{id}_{21} \in \mathbb{R}^{N_u\times 1}$ and $Re^{id}_{22} \in \mathbb{R}^{N_u\times N_u}$. With this information we can derive the Kalman gain for the one step ahead predictor solely based on the measurements.

$$K^{yid}_{(i,j)} = (A^{id}_{(i,j)} P^{id}_{(i,j)} (C^{yid}_{(i,j)}))^T + S^{id}_{(i,j)})(C^{yid}_{(i,j)} P^{id}_{(i,j)} (C^{yid}_{(i,j)})^T + R^{id}_{(i,j)})^{-1} \tag{2-43}$$

and $P^{id}_{(i,j)}$ is the stabilising positive definite solution of the DARE

$$\begin{aligned} P^{id}_{(i,j)} &= A^{id}_{(i,j)} P^{id}_{(i,j)} {A^{id}_{(i,j)}}^T - (A^{id}_{(i,j)} P^{id}_{(i,j)} (C^{yid}_{(i,j)})^T + S^{id}_{(i,j)})(C^{yid}_{(i,j)} P^{id}_{(i,j)} (C^{yl}_{(i,j)})^T + R^{id}_{(i,j)})^{-1} \\ &\times (A^{id}_{(i,j)} P^{id}_{(i,j)} (C^{yid}_{(i,j)})^T + S^{id}_{(i,j)})^T + Q^{id}_{(i,j)} \end{aligned} \tag{2-44}$$

where the state space matrices are identified up to a similarity transformation such that they approximate the earlier stated optimal local models. The identified models approximate and are not equal to the earlier derived models, this due to the identification procedure and model order reduction.

$$A^{l}_{(i,j)} \approx T_{(i,j)} A^{id}_{(i,j)} T^{-1}_{(i,j)} \tag{2-45}$$

$$K^{yl}_{(i,j)} \approx T_{(i,j)} K^{yid}_{(i,j)} \tag{2-46}$$

$$C^{l}_{\phi(i,j)} \approx C^{id}_{\phi(i,j)} T^{-1}_{(i,j)} \tag{2-47}$$

$$C^{yl}_{(i,j)} \approx C^{yid}_{(i,j)} T^{-1}_{(i,j)} \tag{2-48}$$

Both the identification and the reconstruction obtained are fully independent between segments and can be implemented in parallel. The model order of the local identified models will strongly influence the computational complexity as we will see later on. During identification it is critical to reduce the model order as much as possible. Algorithm 1 depicts an iterative implementation of the reconstruction process at a single time step. Notice that the algorithm consists of a double for loop without spatial dependencies between iterations.

Algorithm 1 $Reconstruct(s_x, s_y)$

$\phi \leftarrow zeros(N_x, N_y)$
for $j = 1$ **to** N_y **do**
  for $i = 1$ **to** N_x **do**
    $y^l_{(i,j)} \leftarrow [s_x(\beta); s_y(\beta)]$
    $\hat{x}_{(i,j)}(k+1|k) = [A^l_{(i,j)} - K^{yl}_{(i,j)} C^l_{(i,j)}]\hat{x}_{(i,j)}(k|k-1) + K^{yl}_{(i,j)} y^l_{(i,j)}$
    $\phi_{(i,j)}(k+1|k) = C^l_{\phi(i,j)} \hat{x}_{(i,j)}(k+1|k)$
  end for
end for
return ϕ

2-4 Accuracy analysis

Validating the accuracy of the designed algorithm should help in determining the usefulness of the method. In chapter 5 an extended analysis will be given. Where in this section solely a fixed grid of $(N_x \times N_y) = (8 \times 8)$ is considered. The validation is based on a von Kármán turbulence model. For the von Kármán turbulence model the following parameters are defined, the turbulence outer-diameter L_0 is set to $10m$, the Fried parameter r_0 is set to $0.2m$ and the wind velocities v_x and v_y are respectively set to $20\,\mathrm{m/s}$ and $0\,\mathrm{m/s}$. The local models vary in order between 5 and 12 depending on location in the grid and the sensor noise. Figure 2-3 shows the results based on only the nearest measurements (level 1) for both the derived local models as the identified version. Whereas Figure 2-4 shows it for the level 2 measurements.

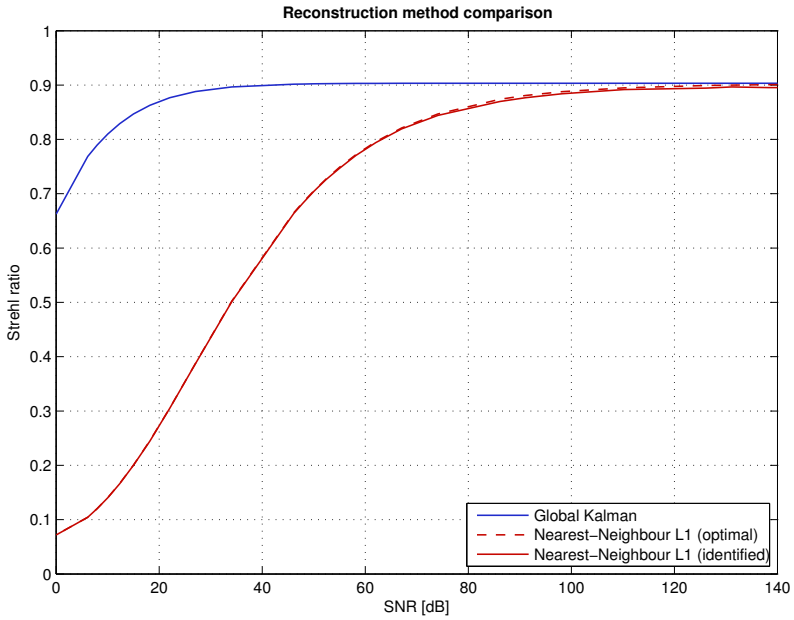


Figure 2-3: Prediction error in terms of strehl versus SNR for various wavefront reconstructors. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

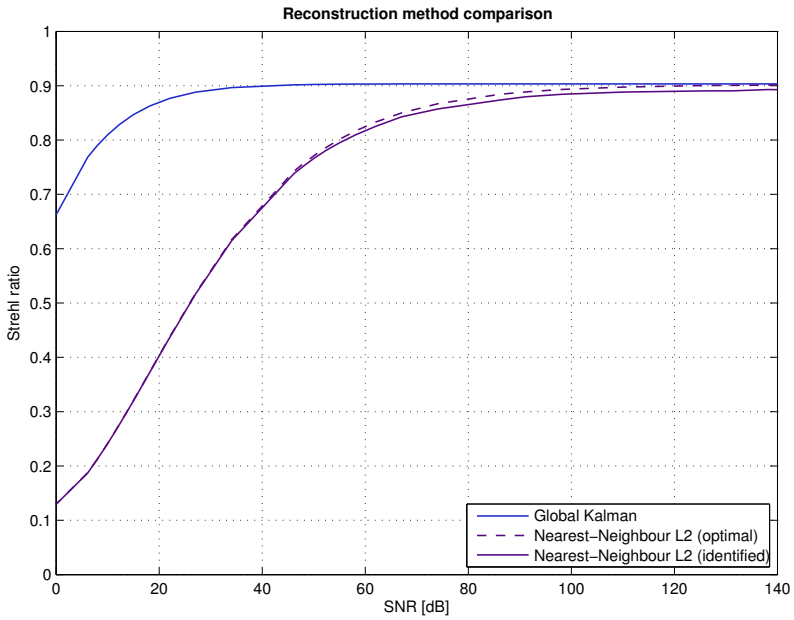


Figure 2-4: Prediction error in terms of strehl versus SNR for various wavefront reconstructors. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

What was expected is that the localised algorithm is outperformed by the global algorithm especially for high noise ratios. Promising is the fact for high signal to noise ratios from and above approximately 80 dB there is only a slight drop in accuracy compared to the global model. Below the denoted SNR the localised identification drops fast to zero and looses the ability to reconstruct correctly. The expectation that more measurements would lead to better accuracy is confirmed by the simulation. Especially in the border region where the reconstruction accuracy start to drop a clear shift is visible to lower signal-to-noise ratios. If for example the 20 dB SNR level is considered then for the level 1 case a strehl ratio of 0.28 is obtained while for the level 2 case a strehl ratio of 0.4 is obtained. Although it is not a significant improvement the trend will persist if even more measurements would be involved. Notice also the small difference between the optimal localised reconstruction and the identified localised reconstruction. Explanations for the difference can be found in the order reduction and the identification process. Which lead to deviations between the identified model and the derived model.

2-5 Computational complexity

Usually parallelism is applied in cases where the process under consideration is rather computationally complex. In these cases parallel computing is able to reduce the time required to complete the process. For the analysis of the wavefront reconstruction algorithm we should first define under which conditions the analysis is performed. We will assume that the aperture is square, such that the number of segments can be defined as $N = N_x N_y$. Also the border models will be treated equal to the other models with relation to the number of inputs. Such that N_u defines the number of inputs. Similar, the order of the models N_m is assumed to

be equal for all models. The described algorithm defines a local subproblem for each segment. Resulting in N subproblems, ensuring a linear scale in order with relation to the number of segments. Each subproblem is defined by equations (2-26) till (2-28), which can always be rewritten to equations (2-49) till (2-50). This form will be more efficient as $A^l_{(i,j)} - K^{yl}_{(i,j)} C^l_{(i,j)}$ can be precomputed.

$$\hat{x}^l_{(i,j)}(k+1|k) = \tilde{A}^l_{(i,j)} \hat{x}^l_{(i,j)}(k|k-1) + K^{yl}_{(i,j)} y^l_{(i,j)}(k)) \tag{2-49}$$

$$\hat{\phi}_{(i,j)}(k+1|k) = C^l_{\phi(i,j)} \hat{x}^l_{(i,j)}(k+1|k) \tag{2-50}$$

where $\tilde{A}^l_{(i,j)}$ is given by

$$\tilde{A}^l_{(i,j)} = A^l_{(i,j)} - K^{yl}_{(i,j)} C^l_{(i,j)} \tag{2-51}$$

These equations consist out of a set of standard vector matrix multiply (VMM). To be able to analyse the computational complexity we should first define the size of all the parameters, See Table 2-1.

Parameter	Height	Width
$[A^l_{(i,j)} - K^{yl}_{(i,j)} C^l_{(i,j)}]$	N_m	N_m
$K^{yl}_{(i,j)}$	N_m	N_u
$C^l_{\phi(i,j)}$	1	N_m
$x^l_{(i,j)}(k\|k-1)$	N_m	1
$y^l_{(i,j)}(k)$	N_u	1

Table 2-1: Matrix and vector size overview of local reconstruction parameters.

Now we are able to determine the computational complexity for each subproblem.

Equation	**Complexity**
$[A^l_{(i,j)} - K^{yl}_{(i,j)} C^l_{(i,j)}] \hat{x}^l_{(i,j)}(k\|k-1) + K^{yl}_{(i,j)} y^l_{(i,j)}(k)$	$2N_m^2 + 2N_u N_m + N_m$
$C^l_{\phi(i,j)} \hat{x}^l_{(i,j)}(k+1\|k)$	$2N_m$
Total	$2N_m^2 + 2N_u N_m + 3N_m$

Table 2-2: Computational complexity analysis from the equations belonging to local subproblems.

Combining the information leaves us with an order of

$$O(N(2N_m^2 + 2N_u N_m + 3N_m)) \tag{2-52}$$

Notice that N_m is a critical parameter, e.g if this is the same size as the original model the same amount of operations compared to the global model has to be performed N times resulting in a tremendous increase in computational complexity. Figure 2-5 shows the computational

complexity of the localised reconstructor related to the grid size for models with a order $N_m = 12$.

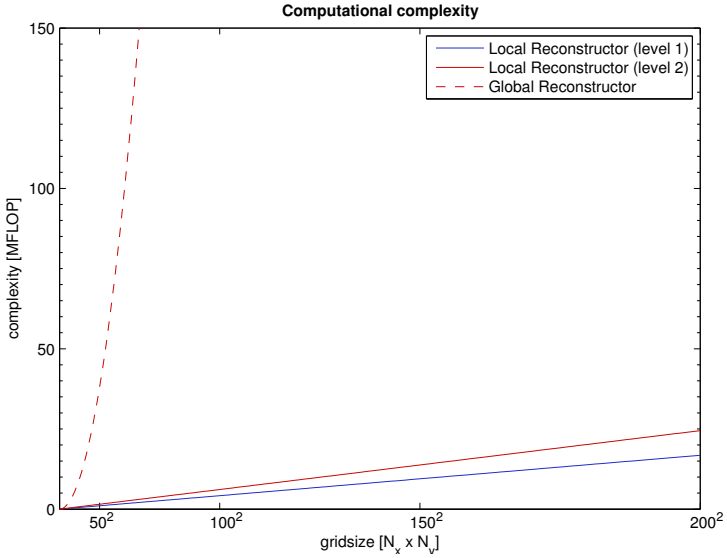


Figure 2-5: Relation of the computational complexity to the grid size, for the local models an of order $N_m = 12$ and respectively $N_u = 4$ and $N_u = 12$ number of inputs are used.

The global reconstructor denoted by the dotted red line is out of range for a grid size of above 2500 due to the non-linear order. Where both the level 1 and level 2 localised reconstructors shown by respectively the solid red and blue line are linear in order. The linear order is vital for the E-ELT application as it proofs scalability property of the algorithm. Already mentioned was the effect of the number of measurements on the computational complexity, which is emphasised by figure Figure 2-5. The figure shows that for an E-ELT sized telescope almost twice the amount of calculations is required for calculating the level 2 case in comparison to the level 1 case. What makes selecting the optimal number of measurements a crucial design parameter.

2-6 Reconstructor Comparison

At the moment applying Kalman filtering for wavefront reconstruction is primarily discussed in literature. In practise still more conventional methods like linear least squares (LLS) and auto regressive (AR) predictors are used. Interesting would be to see how these reconstructors compare to the Kalman based wavefront reconstructors. The LLS reconstructor is given by

$$\hat{\phi}(k) = (G^T G)^\dagger G^T s(k) \tag{2-53}$$

Note that a LLS reconstructor does not perform prediction, such that the accuracy will strongly depend on the sample delay. In contrast the AR-1 one step ahead predictor will provide a minimum variance prediction of $\phi(k+1)$ based on the measurement $s(k)$. The AR-1 predictor is defined by

$$\hat{\phi}(k+1|k) = H_1 s(k) \tag{2-54}$$

where H_1 is

$$H_1 = \begin{bmatrix} CA & CK & 0 \end{bmatrix} \begin{bmatrix} P & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & R \end{bmatrix} \begin{bmatrix} C^T G^T \\ D^T G^T \\ I \end{bmatrix} \left(\begin{bmatrix} GC & GD & I \end{bmatrix} \begin{bmatrix} P & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & R \end{bmatrix} \begin{bmatrix} C^T G^T \\ D^T G^T \\ I \end{bmatrix} \right)^{-1} \tag{2-55}$$

and $P = E[x(k)x(k)^T]$ is the state covariance and $R = E[v(k)v(k)^T]$ is the noise covariance. In figure 2-6 the reconstructors are evaluated with respect to accuracy.

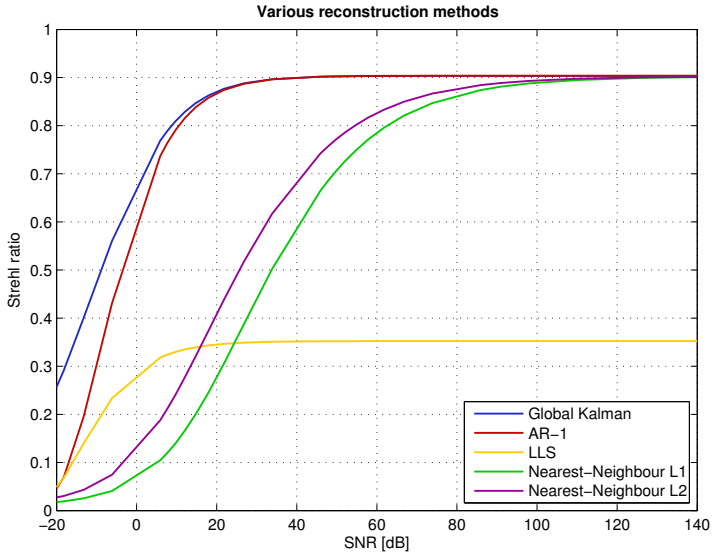


Figure 2-6: Reconstruction error in terms of strehl versus SNR for various wavefront reconstructors. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

The relative low accuracy of the LLS reconstructor is due to the sample delay. Would there be a reduction in turbulence variation or would the framerate be increased than the accuracy would be higher. This is due to the fact that the reconstruction will then be less dependent on prediction. We further observe that the AR-1 predictor performs comparable, the AR-1 predictor is slightly more sensitive to noise which can be explained by the fact that the Kalman predictor is better in averaging the effect of noise. The last significant observation is that both local reconstructs are the most sensitive for noise. This is due to the low number of measurements, the more measurements are available the better the noise can be averaged. This is supported by the fact that the level 2 reconstructor is less noise sensitive compared to the level 1 reconstructor.

At the beginning it was already mentioned that a tradeoff has to be made between computational complexity and accuracy. Also for the designed algorithm this tradeoff is present. Comparing the tradeoff of the various methods will reveal whether the localised reconstructor

achieves a good tradeoff. It was shown that the localised reconstructor is rather noise sensitive, so to make a fair comparison noise sensitivity should be taken into account. From which it is possible conclude that an integration of the accuracy over the SNR will be an adequate accuracy measure.

$$\Omega = \int_a^b e^{-||\phi-\hat{\phi}||_2^2} d\sigma \tag{2-56}$$

In the case of the reconstructors the region of interest with respect to SNR lies between $a = -20dB$ and $b = 140dB$. Leaving us with determining the computational complexity of the AR-1 predictor and the LLS reconstructor. Observe that by precomputing the H_1 matrix and the pseudo inverse of G the complexity of both the AR and the LLS reconstructor is determined by a single matrix vector multiplication of size $N \times N_u$. By applying the sensor structure knowledge it is possible to redefine N_u as $N_u = N - N^{1/2}$. For both reconstructors this will give an order of $O(4N^2 + 4N^{3/2})$. Figure 2-7 shows the tradeoff made by the various reconstructors.

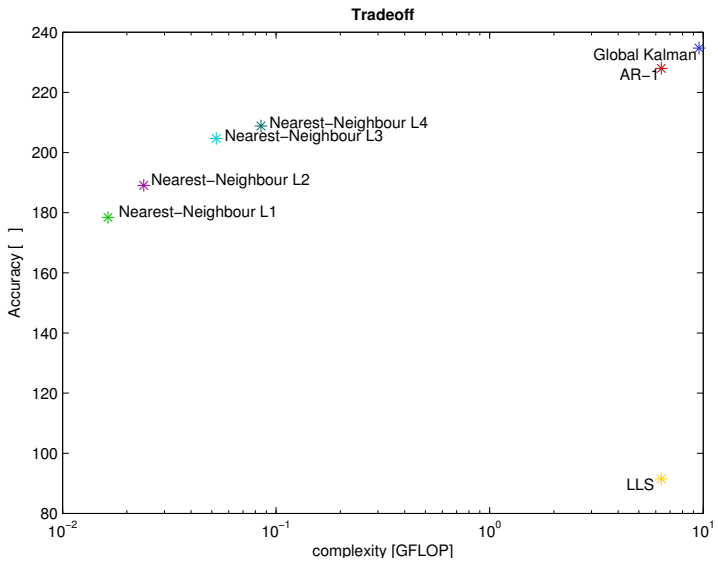


Figure 2-7: Accuracy versus computational complexity tradeoff comparison for various types of wavefront reconstructors.

Note that the computational complexity is given in a logarithmic scale. Once noticed it is clear that the localised reconstructors are computational very efficient, compared to the other methods. The price for this is paid in accuracy, but if this is actually weighted against the computational complexity it is possible to conclude that the tradeoff is done rather effective. An accuracy loss of a factor 1.3 in favour of a complexity reduction with a factor of approximately 600. Still it might be interested to improve the noise sensitivity of the algorithm. Since fainter objects result noisier measurements improvement of noise sensitivity would increase the applicability as also fainter objects can be observed. Chapter 3 proposes to introduce state exchange to improve noise sensitivity.

Chapter 3

Collaboration strategies

The distributed wavefront reconstruction algorithm derived before has a lot of resemblance with distributed Kalman filtering. Especially when considering the local optimal models without model order reduction. In its current form each local segment estimates the state of the system independently, making them completely isolated from each other. The major advantage is that this is computational and communication wise very efficient. Then again, incorporating state information from other segments could improve the state estimate which would improve the reconstruction accuracy. This is due to the fact that state estimates from your neighbours contain more information then the measurements can provide. The states are contains also all previous information which makes it a valuable source of information.

In the literature several proposals have been made to let the different filters collaborate in distributed Kalman problems. In [18] for example a fusion centre is used to combine the estimates. Obvious disadvantage is that in this step the computational burden will lie on a central processing unit. This is exactly what must be avoided as much as possible. Alternatively there are the consensus [19] and diffusion [1] strategies. Both strategies only use direct local neighbours to improve their state. The localised nature of both strategies fits the framework derived earlier much better. The strategies have in common that they use the state estimate of their neighbours to improve their local state estimation.

The way they incorporate the state estimate is slightly different. The consensus strategy sums the difference between the state estimation under consideration and the state estimation of its neighbours. This is then multiplied by a factor ϵ and used as correction factor for its own state estimation. In contrast the diffusion strategy takes the weighted average of its own state estimation and that of its neighbours. Due to the weights of the diffusion algorithm it offers more degrees of freedom in incorporating information. Making it likely that a higher accuracy could be achieved from the diffusion algorithm. Mainly due to the degrees of freedom the focus will lie on diffusion strategies in the upcoming sections.

This chapter is organised as follows: Section 3-1 discusses a diffusion algorithm proposed in literature, which is used as a basis for the localised wavefront reconstructor. Section 3-2 recognises the intended application field of the proposed diffusion algorithm and reveals some differences with respect to the localised wavefront reconstructor. To account for these differences several modifications are suggested. Section 3-3 determines an expression for the

state prediction error covariance matrix including the diffusion update step. Section 3-4 shows the advantages of using the diffusion algorithm and also points out some issues that need further research.

3-1 Basis diffusion algorithm

In [1] a diffusion algorithm was stated that is used as a basis for an extension of the wavefront reconstructor. The diffusion algorithm uses local state estimates in the neighbourhood of the segment under consideration $\mathcal{S}_{(i,j)}$. For this a set of neighbours including the segment itself is defined by $\mathcal{N}_{(i,j)}$. Furthermore, for the ease of notation in this chapter the dual indices $\mathcal{S}_{(i,j)}$ will be denoted by a single index $\mathcal{S}_i$. Implying that the segment numbering will be ongoing, formatted column wise from top to bottom.

State exchange is motivated by the knowledge that together more information is available in contrast to estimation solely based on the measurements. In the diffusion algorithm incorporation of the neighbouring information is done by the diffusion update and is given by equation (3-1). The diffusion update is performed each iteration. Equation 3-1 updates the state estimation $\hat{x}_i(k|k)$ by taking the weighted average of the set of neighbouring states and is given by

$$\hat{x}_i(k|k) = \sum_{j \in N_i} c_{i,j} \psi_j(k) \tag{3-1}$$

where $\psi_j(k)$ is the state estimate of segment j at time instance k and the weights $c_{i,j}$ are selected such that:

$$\sum_{j \in N_i} c_{i,j} = 1 \tag{3-2}$$

Taking the average usually has a smoothing effect, implying a reduction of extremes. Since the diffusion step is nothing more then a weighted average this also applies. Based on the fact that neighbours have different measurement information, segments differ in state estimation accuracy also per state element. Meaning that a neighbour can be better in estimating a certain state compared to the segment itself. By the diffusion step the segment can benefit from this and improve that state. This has a side effect as also badly estimate state information is injected due to the averaging effect. Crucial over here are the weight parameters, these weights influence how strong a state estimation is taken into account. Due to the interconnection structure and the recursion effect state information from all segments will propagate through the whole network to a certain degree.

As a starting point Algorithm 1 as stated in [1] is taken, its notation is adjusted such that it matches the notation of previous chapters. Consider the state-space model as in equation (3-3) till (3-4).

$$x(k+1) = A(k)x(k) + K(k)e(k) \tag{3-3}$$

$$y_i(k) = C_i(k)x(k) + v_i(k) \qquad i = 1, \ldots, N \tag{3-4}$$

Then the diffusion algorithm is given by Algorithm 2.

Algorithm 2 Diffusion Kalman filter

Compute for all segments i and at every time instance k:
Step 1: Incremental Update

1: $\psi_i(k) \leftarrow \hat{x}_i(k|k-1)$
2: $P_i(k) \leftarrow P_i(k|k-1)$

3: **for all** neighbour segments $j \in \mathcal{N}_i$ **do**
4: $\quad R_e \leftarrow R_e + C_j(k)P_i(k)C_j(k)^T$
5: $\quad \psi_i(k) \leftarrow \psi_i(k) + P_i(k)C_j(k)^T R_e^{-1}[y_j(k) - C_j(k)\psi_i(k)]$
6: $\quad P_i(k) \leftarrow P_i(k)C_j(k)^T R_e^{-1} C_j(k)P_i(k)$
7: **end for**

Step 2: Diffusion Update

8: $\hat{x}_i(k|k) \leftarrow \sum_{j \in \mathcal{N}_i} c_{i,j}\psi_j(k)$

Step 3: Time Update

9: $P_i(k|k) \leftarrow P_i(k)$
10: $\hat{x}_i(k+1|k) = A(k)\hat{x}_i(k|k)$
11: $P_i(k+1|k) = A(k)P_i(k|k)A(k)^T + K(k)Q(k)K(k)^T$

Let us briefly review what happens in the algorithm. In the incremental update step on line 1 and 2, the first procedure is to assign the local state estimates $\hat{x}_i(k|k-1)$ and the covariance matrix $P_i(k|k-1)$ to intermediate variables $\psi_i(k)$ and $P_i(k)$. Then the loop on line 3 till 7 iterates through a set of neighbours $\mathcal{N}_i$ and incorporates all the measurements of the neighbours into the estimation. According to the measurement update also the covariance matrix is updated. The next step is the actual diffusion on line 8, where state information is exchanged by taking a weighted average of the segments own estimation and that of its neighbours. Line 9 till 11 completes the algorithmic by performing a time update, it updates the state estimation and covariance matrix for the next time instance. Revealing the recursive nature of the algorithm.

3-2 Modification based on structure wavefront reconstructor

The proposed diffusion algorithm in [1] is intended for a network of nodes (figure 3-1) without a specified hierarchy where each node has one or more sensors, e.g wireless sensor networks (WSN). In opposite the localised wavefront reconstructor is a structured grid as depicted in figure 2-2(a), divided into segments with a sensor array. In its presented form the diffusion algorithm does not match the framework for the wavefront reconstructor so in this section some modifications will be made.

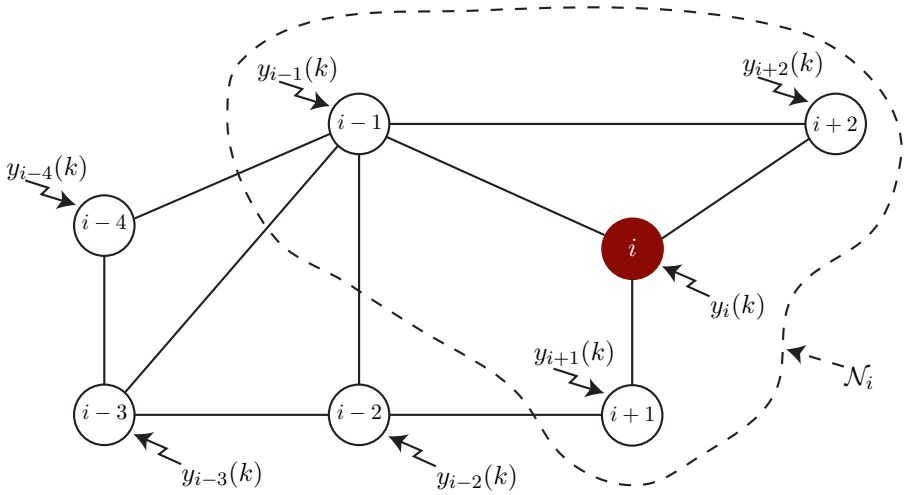


Figure 3-1: Example network structure as assumed in [1].

In [1] it is assumed that the system is time varying while in our case it involves an linear time invariant (LTI) system. This allows for a simplification of the algorithm. Not only on paper this simplification has effect it also will have a positive effect on the computational complexity in the final implementation since the number of online matrix multiplication reduces, because a stationary covariance matrix can be determined such that the Kalman gain can be pre-computed. In addition the original algorithm also incorporates all the measurements of it's neighbours into the state estimation (line 3 - 7 in algorithmic 2). Whereas in the case of the wavefront reconstructor each segment $\mathcal{S}_i$ does not have its own measurements it rather gets measurements assigned. Figure 3-2 depicts the structure of the local wavefront reconstructor

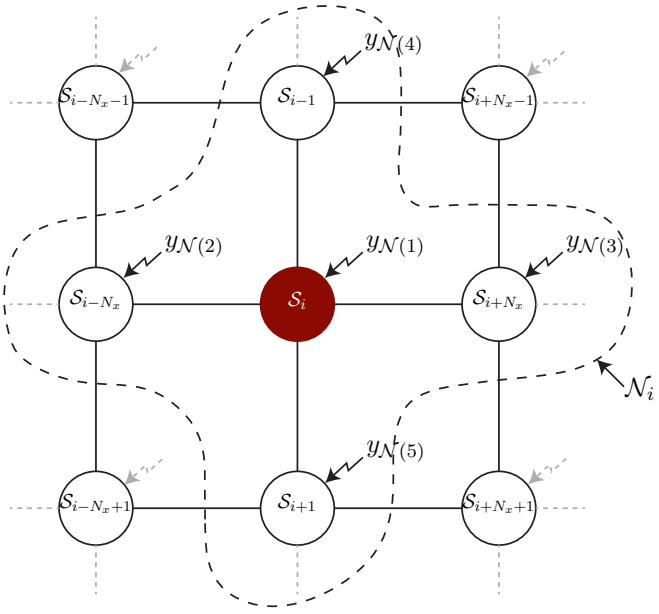


Figure 3-2: Structure of the local wavefront reconstructor with respect to the diffusion algorithm.

Depending on the assignment there is a certain overlap between segments. For example if we consider the level one version each segment has one measurement with his neighbour in common. Define for level 1 the set $\mathcal{N}_i = \{\mathcal{S}_i, \mathcal{S}_{i-1}, \mathcal{S}_{i+1}, \mathcal{S}_{i-N_x}, \mathcal{S}_{i+N_x}\}$ then the measurements are given by

$$\begin{aligned}
y_{\mathcal{N}(1)} &= \left[\underline{s_{x,i}(k)}, \underline{s_{x,i+1}(k)}, \underline{s_{y,i}(k)}, \underline{s_{y,i+N_x}(k)}\right]^T \\
y_{\mathcal{N}(2)} &= \left[s_{x,i-1}(k), \underline{s_{x,i}(k)}, s_{y,i-1}(k), s_{y,i+N_x-1}(k)\right]^T \\
y_{\mathcal{N}(3)} &= \left[\underline{s_{x,i+1}(k)}, s_{x,i+2}(k), s_{y,i+1}(k), s_{y,i+N_x+1}(k)\right]^T \\
y_{\mathcal{N}(4)} &= \left[s_{x,i-N_x}(k), s_{x,i-N_x+1}(k), s_{y,i-N_x}(k), \underline{s_{y,i}(k)}\right]^T \\
y_{\mathcal{N}(5)} &= \left[s_{x,i+N_x-1}(k), s_{x,i+N_x}(k), \underline{s_{y,i+N_x}(k)}, s_{y,i+2*N_x}(k)\right]^T
\end{aligned} \tag{3-5}$$

Where the underlined indices reveal the overlap, as they point out equal indices. The overlap in measurements between segments is different from the assumption in [1]. In [1] the same value is also measured by its neighbour by a different sensor, such that the noise on the measurements can be assumed to be uncorrelated, where for the wavefront reconstructor the overlapping measurements are exactly the same measured by the same sensor such that the measurements are correlated. Consequently we could stack the measurements in a single vector $\tilde{y}_i$ and remove all double entries such that

$$\begin{aligned}
\tilde{y}_i &= [s_{x,i-1}(k), s_{x,i}(k), s_{x,i+1}(k), s_{x,i+2}(k), s_{x,i-N_x}(k), s_{x,i-N_x+1}(k) \\
&\quad , \quad s_{x,i+N_x-1}(k), s_{x,i+N_x}(k), s_{y,i-1}(k), s_{y,i}(k), s_{y,i+N_x}(k) \\
&\quad , \quad s_{y,i+N_x-1}(k), s_{y,i+N_x+1}(k), s_{y,i-N_x}(k), s_{y,i+2*N_x}(k), s_{y,i-N_x}(k)]^T
\end{aligned} \tag{3-6}$$

Furthermore all the measurements in $\tilde{y}_i$ could be made directly available to segment $\mathcal{S}_i$. So for the wavefront reconstructor incorporating neighbouring measurements is the same as assigning more measurements to each segment at the beginning. From which we can conclude that it is possible to perform another simplification, namely replacing the "for loop" (line 3 till 7 in Algorithm 2) by a single update equation based on the measurements assigned to the segment (line 2 in Algorithm 3).

Since for the localised reconstruction problem, assigning which measurements belong to a segment is a design parameter, we are able to select the assigned measurements per segment freely. To achieve a fair comparison the measurement set $y_i(k)$ is chosen such that the set for diffusion algorithm corresponds to either a level 1 or a level 2 measurement set. This also reduces the measurements per segment compared to $\tilde{y}_i$, which is advantageous for the computational complexity. The reformulated version is stated in Algorithm 3

Algorithm 3 Diffusion Kalman filter LTI form

Consider the state-space model in (2-21) till (2-23). Compute for all segments i and at every time instance k:

Step 1: Incremental Update

1: $\psi_i(k|k-1) \leftarrow \hat{x}_i(k|k-1)$

2: $\psi_i(k|k) \leftarrow \psi_i(k|k-1) + K_i^f \left[y_i(k) - C_i\psi_i(k|k-1)\right]$

Step 2: Diffusion Update

3: $\hat{x}_i(k|k) \leftarrow \sum_{j\in\mathcal{N}_i} c_{i,j}\psi_j(k|k)$

Step 3: Time Update

4: $\hat{x}_i(k+1|k) = A_i\hat{x}_i(k|k)$

In both Algorithm 2 and 3 it is assumed that the process and measurement noise are uncorrelated, resulting in the following Kalman gain for Algorithm 3

$$K_i^f = P_i(C_i^y)^T(C_i^y P_i(C_i^y)^T + R_i)^{-1} \tag{3-7}$$

and P_i is the stabilising positive definite solution of the DARE

$$P_i \quad = \quad P_i - P_i(C_i^y)^T(C_i^y P_i(C_i^y)^T + R_i)^{-1}C_i^y P_i \tag{3-8}$$

Note that the covariance matrix P_i does not take into account the diffusion update. It is the standard formulation obtained by applying standard Kalman methodology, which is rather peculiar since the diffusion update influences the state estimates and with that also the covariance matrix.

The wavefront reconstructor is obtained to provide a prediction. In the current format the filter step is done for the current state and not for the predicted state, this introduces unnecessary complexity. Since $\hat{x}_i(k|k)$ is not required, a favourable modification would be to replace the filter step (line 2 of Algorithm 3) to a combined filter and predictor step (line 1 of Algorithm 4). This would involve performing the diffusion step on the predicted state (line 4 till 5 of Algorithm 4). The two previous algorithms did not include the output equation $\hat{\phi}(k+1|k,i) = C_{\phi,i}\hat{x}_i(k+1|k)$, where for the wavefront reconstructor the output should also be determined. Based on experiments we know that the output benefits in accuracy if it is performed directly after the filter step (line 2 of Algorithm 4). This is possible due to the averaging effect of the diffusion step, which will diminish the accuracy of the best estimated states. The best estimated states are the ones most related to the available measurements as these are best observable. In addition these measurements are also strongly related to the local wavefront phase. The weights play a crucial role in this, one can imagine that if the weights are chosen optimal and state dependent there will be no degradation for the best estimated states. In this case it will probable be that the highest output accuracy could be achieved by placing the output equation after the diffusion step. Additional research could verify this. In the current situation non optimal weights are considered, which will leave us with Algorithm 4.

Algorithm 4 Diffusion Kalman one-step ahead predictor

Consider the one-step ahead predictor in equations (2-26) till (2-28). Compute for all segments i and at every time instance k:

Step 1: Time Update

1: $\hat{x}_i(k+1|k) = A_i\hat{x}_i(k|k-1) + K_i^y\left[y_i(k) - C_i\hat{x}_i(k|k-1)\right]$
2: $\hat{\phi}(k+1|k,i) = C_{\phi,i}\hat{x}_i(k+1|k)$

Step 2: Diffusion Update

3: $\psi_i(k+1|k) \leftarrow \hat{x}_i(k+1|k)$
4: $\hat{x}_i(k+1|k) = \sum_{j\in\mathcal{N}_i} c_{i,j}\psi_j(k+1|k)$

3-3 Prediction error covariance matrix

The Kalman gain determination is based on the covariance matrix P_i. Since the diffusion step also influences the covariance matrix we should establish a new expression for the covariance matrix that incorporates the diffusion update.

Suppose we introduce a lifted state space system including the segment and all the nearest neighbours $\mathcal{N}_i$. For this purpose let the number of nearest neighbours including the segment itself be given by nn_i, also denoted as the degree of a segment. Such that we have

$$\underbrace{\begin{bmatrix} x_{\mathcal{N}_i(1)}(k+1) \\ \vdots \\ x_{\mathcal{N}_i(nn_i)}(k+1) \end{bmatrix}}_{X_i(k+1)} = \underbrace{\begin{bmatrix} A_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & A_{\mathcal{N}_i(nn)} \end{bmatrix}}_{\bar{A}_i} \underbrace{\begin{bmatrix} x_{\mathcal{N}_i(1)}(k) \\ \vdots \\ x_{\mathcal{N}_i(nn_i)}(k) \end{bmatrix}}_{X_i(k)} + \underbrace{\begin{bmatrix} K_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & K_{\mathcal{N}_i(nn_i)} \end{bmatrix}}_{\bar{K}_i} \underbrace{\begin{bmatrix} e_{\mathcal{N}_i(1)}(k) \\ \vdots \\ e_{\mathcal{N}_i(nn_i)}(k) \end{bmatrix}}_{\bar{e}_i(k)}, \tag{3-9}$$

$$\underbrace{\begin{bmatrix} y_{\mathcal{N}_i(1)}(k+1) \\ \vdots \\ y_{\mathcal{N}_i(nn_i)}(k+1) \end{bmatrix}}_{\bar{y}_i(k)} = \underbrace{\begin{bmatrix} C_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & C_{\mathcal{N}_i(nn_i)} \end{bmatrix}}_{\bar{C}_i} \underbrace{\begin{bmatrix} x_{\mathcal{N}_i(1)}(k) \\ \vdots \\ x_{\mathcal{N}_i(nn_i)}(k) \end{bmatrix}}_{\bar{x}_i(k)} + \underbrace{\begin{bmatrix} v_{\mathcal{N}_i(1)}(k) \\ \vdots \\ v_{\mathcal{N}_i(nn_i)}(k) \end{bmatrix}}_{\bar{v}_i(k)} \tag{3-10}$$

Thus the one step ahead predictor of the lifted system given by equation (3-9) till (3-10) is defined as

$$\begin{aligned}
\hat{\bar{e}}_i(k) &= \bar{y}(k) - \bar{C}_i\hat{X}_i(k|k-1) && \text{(3-11)}\\
\hat{X}_i(k+1) &= \bar{A}_i\hat{X}_i(k|k-1) + \bar{K}_i^y\hat{\bar{e}}_i(k) && \text{(3-12)}\\
\hat{\bar{\phi}}_i(k+1|k) &= \bar{C}_i\hat{X}_i(k+1|k) && \text{(3-13)}
\end{aligned}$$

Theorem 1. *Prediction error covariance matrix When considering a one step ahead predictor based given by equation* (3-11) *till* (3-13)*, then the covariance matrix of the prediction error*

$$\bar{P}_i(k+1|k) = E\left[\Big(X(k+1) - \hat{X}(k+1|k)\Big)\Big(X(k+1) - \hat{X}(k+1|k)\Big)^T\right] \quad \text{(3-14)}$$

is given by

$$\begin{aligned}
\bar{P}_i(k+1|k) &= \bar{A}_i\bar{P}_i(k|k-1)\bar{A}_i^T - (\bar{A}_i\bar{P}_i(k|k-1)\bar{C}_i^T + \bar{S}_i)(\bar{K}_i^y)^T + \bar{Q}_i\\
&- \bar{K}_i^y(\bar{S}_i^T + \bar{C}_i\bar{P}_i(k|k-1)\bar{A}_i^T) + \bar{K}_i^y(\bar{C}_i\bar{P}_i(k|k-1)\bar{C}_i^T + \bar{R}_i)(\bar{K}_i^y)^T
\end{aligned} \quad \text{(3-15)}$$

where $\bar{S}_i$, $\bar{Q}_i$ and $\bar{R}_i$ are given by

$$\begin{aligned}
\bar{S}_i &= \begin{bmatrix} S_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ & & \ddots & \\ \vdots & & & 0 \\ 0 & \dots & 0 & S_{\mathcal{N}_i(nn_i)} \end{bmatrix}, \quad \bar{Q}_i = \begin{bmatrix} Q_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ & & \ddots & \\ \vdots & & & 0 \\ 0 & \dots & 0 & Q_{\mathcal{N}_i(nn_i)} \end{bmatrix},\\
\bar{R}_i &= \begin{bmatrix} R_{\mathcal{N}_i(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ & & \ddots & \\ \vdots & & & 0 \\ 0 & \dots & 0 & R_{\mathcal{N}_i(nn_i)} \end{bmatrix}
\end{aligned} \quad \text{(3-16)}$$

Furthermore for the diffusion algorithm it is possible to define a weight matrix W_i such that we have

$$P_i(k+1|k) = W_i\bar{P}_i(k+1|k)W_i^T \quad \text{(3-17)}$$

Proof. To verify the theorem we start with line 4 of algorithm 4.

$$\hat{x}_i(k+1|k) = \sum_{j\in\mathcal{N}_i} c_{i,j}\psi_j(k+1|k) \quad \text{(3-18)}$$

By introducing a weight matrix $W_i \in \mathbb{R}^{N_m \times nnN_m}$ related to the diffusion weights $c_{i,j}$ and stacking $\hat{\psi}_i(k+1|k) \in \mathbb{R}^{1 \times N_m}$, such that

$$\hat{\Psi}_i(k+1|k) = \left[\begin{array}{c|c|c} \hat{\psi}^T_{\mathcal{N}_i(1)} & \dots & \hat{\psi}^T_{\mathcal{N}_i(nn_i)} \end{array} \right]^T, \qquad W_i = \left[\begin{array}{c|c|c} c_{i,\mathcal{N}_i(1)} I_{N_m} & \dots & c_{i,\mathcal{N}_i(nn_i)} I_{N_m} \end{array} \right] \tag{3-19}$$

and thus it is possible to write the summation as

$$\hat{x}_i(k+1|k) = W_i \hat{\Psi}_i(k+1|k) \tag{3-20}$$

On line 3 of algorithm 4 a reassignment takes place, such that by backward substitution this results in

$$\hat{x}_i(k+1|k) = W_i \hat{X}_i(k+1|k) \tag{3-21}$$

Since W_i now defines the relation between the lifted state $X_i(k)$ and a local state $x_i(k)$ it is possible to continue with determining the covariance matrix of the prediction error of the lifted system defined by equation (3-14). Note that based on equation (3-2) it is also possible to define W_i as a relation between the true state of the local an the lifted system, which is given by $x_i(k) = W_i X_i(k)$. With the introduction $\hat{X}_i(k|k-1)$ we are able to substitute $\hat{X}_i(k+1|k)$ as an update formulation based on the one step ahead predictor on line 2 of algorithm 4.

$$\begin{aligned} \bar{P}_i(k+1|k) &= cov\left(X_i(k+1) - \hat{X}_i(k+1|k)\right) \\ \bar{P}_i(k+1|k) &= cov\left(X_i(k+1) - \left[\bar{A}_i \hat{X}_i(k|k-1) + \bar{K}_i^y \left(\bar{y}_i(k) - \bar{C}_i \hat{X}_i(k|k-1)\right)\right]\right) \end{aligned} \tag{3-22}$$

where $cov(e(k)) = E[e(k)e(k)^T]$ is the covariance operator. Further, substitute the measurement vector $\bar{y}_i(k)$ by the output equation (3-10) based on the true state $\bar{x}_i(k)$.

$$\bar{P}_i(k+1|k) = cov\left(X_i(k+1) - \left[\bar{A}_i \hat{X}_i(k|k-1) + \bar{K}_i^y \left(\bar{C}_i X_i(k) + \bar{v}_i(k) - \bar{C}_i \hat{X}_i(k|k-1)\right)\right]\right) \tag{3-23}$$

Substitute equation (3-9) for $X_i(k+1)$ to obtain equation (3-24) in terms of $X_i(k)$.

$$\begin{aligned} \bar{P}_i(k+1|k) &= cov\left(\left[\bar{A}_i X_i(k) + \bar{K}_i \bar{e}_i(k)\right]\right. \\ &- \left.\left[\bar{A}_i \hat{X}_i(k|k-1) + \bar{K}_i^y \left(\bar{C}_i X_i(k) + \bar{v}_i(k) - \bar{C}_i \hat{X}_i(k|k-1)\right)\right]\right) \end{aligned} \tag{3-24}$$

Collecting the vectors results in

$$\bar{P}_i(k+1|k) = cov\left((\bar{A}_i - \bar{K}_i^y \bar{C}_i)\left(X_i(k) - \hat{X}_i(k|k-1)\right) + \bar{K}_i \bar{e}_i(k) - \bar{K}_i^y \bar{v}_i(k))\right) \quad (3\text{-}25)$$

Since the measurement and process noise are uncorrelated with relation to the state

$$\bar{P}_i(k+1|k) = cov\left((\bar{A}_i - \bar{K}_i^y \bar{C}_i)\left(X_i(k) - \hat{X}_i(k|k-1)\right)\right) + cov\left(\bar{K}_i \bar{e}_i(k) - \bar{K}_i^y \bar{v}_i(k))\right) \quad (3\text{-}26)$$

By the properties of vector covariance we can rewrite it to

$$\begin{aligned} \bar{P}_i(k+1|k) &= (\bar{A}_i - \bar{K}_i^y \bar{C}_i) cov\left(X_i(k) - \hat{X}_i(k|k-1)\right)(\bar{A}_i - \bar{K}_i^y \bar{C}_i)^T \\ &+ \bar{K}_i cov(\bar{e}_i(k))\bar{K}_i^T - \bar{K}_i^y E[\bar{v}_i(k)^T \bar{e}_i(k)]\bar{K}_i^T \\ &- \bar{K}_i E[\bar{e}_i(k)\bar{v}_i(k)^T](\bar{K}_i^y)^T + \bar{K}_i^y cov(\bar{v}_i(k))(\bar{K}_i^y)^T \end{aligned} \quad (3\text{-}27)$$

From here we can substitute $cov(X_i(k) - \hat{X}_i(k|k-1)$ by $\bar{P}_i(k|k-1)$ defined in equation (3-14), which leaves us with

$$\bar{P}_i(k+1|k) = (\bar{A}_i - \bar{K}_i^y \bar{C}_i)\bar{P}_i(k|k-1)(\bar{A}_i - \bar{K}_i^y \bar{C}_i)^T + \bar{Q}_i - \bar{K}_i^y \bar{S}_i^T - \bar{S}_i(\bar{K}_i^y)^T + \bar{K}_i^y \bar{R}_i(\bar{K}_i^y)^T \quad (3\text{-}28)$$

where $\bar{Q}_i = \bar{K}_i cov(\bar{e}_i(k))\bar{K}_i^T$, $\bar{S}_i = \bar{K}_i E[\bar{e}_i(k)\bar{v}_i(k)^T]$ and $\bar{R}_i = cov(\bar{v}_i(k))$. By rewriting equation (3-28) we arrive at

$$\begin{aligned} \bar{P}_i(k+1|k) &= \bar{A}_i \bar{P}_i(k|k-1)\bar{A}_i^T - (\bar{A}_i \bar{P}_i(k|k-1)\bar{C}_i^T + \bar{S}_i)(\bar{K}_i^y)^T + \bar{Q}_i \\ &- \bar{K}_i^y(\bar{S}_i^T + \bar{C}_i \bar{P}_i(k|k-1)\bar{A}_i^T) + \bar{K}_i^y(\bar{C}_i \bar{P}_i(k|k-1)\bar{C}_i^T + \bar{R}_i)(\bar{K}_i^y)^T \end{aligned} \quad (3\text{-}29)$$

This proofs part of the theorem as it is equal to equation (3-15) and leaves us to proof equation (3-17). Based on equation (3-21) we have for the diffusion algorithm a covariance for the prediction error of

$$\begin{aligned} P_i(k+1|k) &= E\left[(x_i(k+1) - \hat{x}_i(k+1|k))(x_i(k+1) - \hat{x}_i(k+1|k))^T\right] \\ P_i(k+1|k) &= E\left[(x_i(k+1) - W_i\hat{X}_i(k+1|k))(x_i(k+1) - W_i\hat{X}_i(k+1|k))^T\right] \end{aligned} \quad (3\text{-}30)$$

We can also write $x_i(k+1)$ in a lifted form y using the observation that $x_i(k+1) = W_i X_i(k+1)$, since that the true state is equal for all segments and equation (3-2) holds. Such that

$$\begin{aligned} P_i(k+1|k) &= E\left[(W_i X_i(k+1) - W_i\hat{X}_i(k+1|k))(W_i X_i(k+1) - W_i\hat{X}_i(k+1|k))^T\right] \\ P_i(k+1|k) &= E\left[W_i(X_i(k+1) - \hat{X}_i(k+1|k))(W_i(X_i(k+1) - \hat{X}_i(k+1|k)))^T\right] \\ P_i(k+1|k) &= W_i E\left[(X_i(k+1) - \hat{X}_i(k+1|k))(X_i(k+1) - \hat{X}_i(k+1|k))^T\right] W_i^T \end{aligned} \quad (3\text{-}31)$$

Observe that we can substitute $\bar{P}_i(k+1|k)$ into equation (3-31), which results in

$$P_i(k+1|k) \quad = \quad W_i \bar{P}_i(k+1|k) W_i^T \tag{3-32}$$

completing the proof of theorem 1. From here it is possible to determine the optimal Kalman gain $\bar{K}_i^y$, observe however that by the diffusion algorithm for $\bar{K}_i^y$ a structure is induced. Consisting out of a block diagonal matrix based on the Kalman gains K_i^y from the set of neighbours $j \in \mathcal{N}_i$, defined by equation (3-33) below.

$$\bar{K}_i^y = \begin{bmatrix} K^y_{\mathcal{N}(1)} & 0 & \dots & 0 \\ 0 & & & \vdots \\ & & \ddots & \\ \vdots & & & 0 \\ 0 & \dots & 0 & K^y_{\mathcal{N}(nn)} \end{bmatrix} \tag{3-33}$$

where K_i^y is defined as in equation (2-34) with the covariance matrix $P^l_{(i,j)}$ defined by equation (3-31). Note that equation (3-33) is not the optimal Kalman gain due to the induced block diagonal structure. Furthermore in the case of time invariant matrices it is also feasible to compute a time invariant covariance matrix and a fixed Kalman gain. At least if we assume the covariance matrix will converge to a solution. Algorithm 8 in appendix B is an example of pseudo code that will do the job for a set of segments. □

3-4 Performance of the diffusion algorithm

The term performance is ambiguous and can refer to different measures which is also the case over here. First we will consider the performance in terms of accuracy. Where we will relate the diffusion algorithm to both the global reconstructor and the localised reconstructor. Till this moment the weights $c_{i,j}$ are still not defined. So lets assume we choose them according to the number of neighbours denoted by the degree nn_i of the segment such that $c_{i,j} = \alpha_i N n n_i$, where $\alpha_{(i,j)}$ is a scaling factor such that condition 3-2 holds. The simulation results shown in figures 3-3 and 3-4 are based on the same parameters with respect to turbulence and grid size as in chapter 2. Again we have simulated both the level 1 and level 2 case. Three different methods are compared to each other, the global reconstructor, the diffusion based reconstructor and the isolated localised reconstructor. For both the global as the local reconstructor the strehl ratios where derived analytically, however for the diffusion algorithm this was done based on data generated by the simulation. This is immediately visible by the slightly less smooth results of the diffusion algorithm. Since the simulations are based on a significant amount (10.000) of sample points it should give an accurate representation of the strehl ratio.

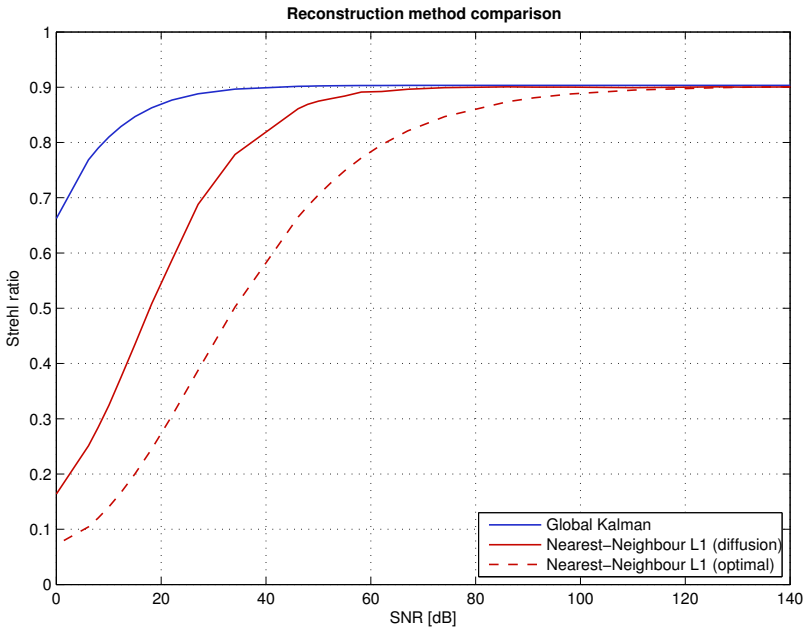


Figure 3-3: Prediction error in terms of strehl versus SNR for various wavefront reconstructors. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$. The plot shows the global reconstructor, the isolated localised reconstructor and the localised reconstructor including diffusion algorithm. The local models in this plot are based on level 1 measurements.

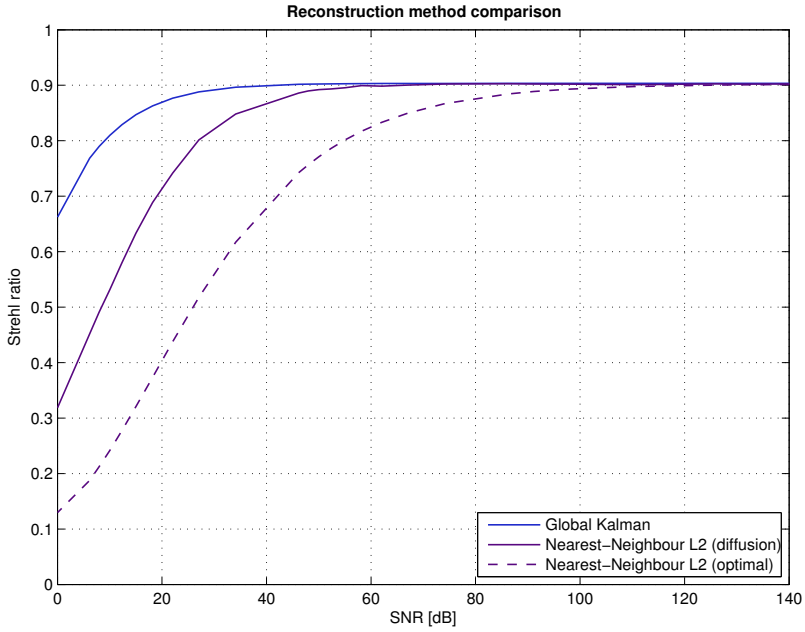


Figure 3-4: Prediction error in terms of strehl versus SNR for various wavefront reconstructors. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$. The plot shows the global reconstructor, the isolated localised reconstructor and the localised reconstructor including diffusion algorithm. The local models in this plot are based on level 2 measurements.

The diffusion algorithm is expected to perform better than the isolated version. Which is confirmed for both cases by the simulation results. The most significant difference is in the higher noise ratios at the border of where the diffusion algorithm still achieves to perform well while the isolated version already starts to degrade. In this border region a gain of approximately 20 dB in noise sensitivity is achieved. Still there is a gap between the global reconstructor and the localised reconstructor, but it has been reduced significantly. Besides there is an open issue that could improve the results even further. Right now the weights of the diffusion update are determined rather heuristically. So they are probably far from the optimal weights. Would we be able to choose optimal weights it may further improve the final reconstruction performance.

The second performance measurement we are interested in is the complexity. In addition to the computational complexity the diffusion algorithm also introduces communication. Communication is often a slow process and should be kept to the minimum. In the case of the wavefront reconstructor it helps a lot that it is in LTI format (Algorithm 4) instead of time varying (Algorithm 2). This ensures for a reduction in computations and communication as now only the state estimate of the neighbour has to be communicated and diffused. Note however this solely applies for the real-time reconstruction. The offline identification process does not have this advantage as there, a segment requires knowledge about the output matrices and the joint covariance matrix of its neighbours. Also in the identification process the lifted covariance matrix has to be determined given by the procedures in Algorithm 8, which is rather computational complex. Still for the reconstruction it is more important that it is as computationally efficient as possible then for the identification process. So, what does the diffusion process add to the computational complexity? Considering Algorithm 4 the only modification is the diffusion step when compared to the localised reconstructor. The summation is the basis of the diffusion step and consist out of a sum over a set of scalars multiplied by a set of vectors. Where the vectors are state estimations with size N_m. The summation is done over the number of neighbours giving a set size of nn_i. As with the localised reconstructor this is done for each segment each iteration resulting in the following additional computational complexity.

$$O(2Nnn_iN_m) \tag{3-34}$$

As already denoted also the state estimates have to be communicated from the neighbours to each other. Also this happens each iteration for each segment resulting in the following complexity with respect to the communication.

$$O(Nnn_iN_m) \tag{3-35}$$

When the diffusion algorithm is related to the isolated localised reconstructor based on full order models the algorithm is rather successful. It gives only a slight increase in computational complexity which is almost negligible. In addition the accuracy as discussed earlier shows a major improvement. From which we could conclude that the results are very promising though we should be critical. All these results were achieved on local models which still have their full states. As discussed before local full order models will give a tremendous increase in computational complexity compared to the global reconstructor. The localised reconstructor was made interesting by reducing the order of the local models. Similar to make

the diffusion algorithm interesting for the wavefront reconstruction the step to reduced order models has to be made. For the original local reconstructor the suggestion was made to use identification procedures for determining low order models. Problem there is that the models are no longer in the same basis, but are related to each other by a similarity transformation. In addition this similarity transformation is not known and certainly not straightforward to derive. Alternatively we could perform order reduction technologies on the full order models and take the increase in computational complexity during the identification process for granted. This makes it possible to determine the similarity transformations between the systems. Suggesting that when the similarity transformation is defined by T_i the diffusion step is given by

$$\hat{x}_i(k|k) = \sum_{j \in N_i} c_{i,j} T_i T_j^{\dagger} \psi_j(k) \tag{3-36}$$

Order reduction does not only transform the state but also removes state information. By taking the pseudo inverse of the similarity transformation a conversion back to the original full state is given. This does not completely cover up the removed state information since this conversion cannot reconstruct the missing information perfectly. When a matrix is not full rank, the solution to the inverse is not unique. The pseudo inverse works around this by returning a specific possible solution. However for the diffusion algorithm this results that incorrect information is injected into state estimates. Take for example that for a certain segment a state does not contribute at all to the output and is removed by order reduction. While for another segment this specific state is crucial. The pseudo inverse reconstructs this state but does not necessarily have to be correct or even close to the real state. Followed up by the diffusion step which averages both state estimations and introduces unwanted errors, degrading the accuracy. So for diffusion on reduced order models the weights become even more crucial. When it would be feasible to define state element dependent weights and also take into account the relation between segments it will probably possible to benefit from diffusion strategies.

Chapter 4

Reconstructor Implementation

The term implementation refers to the mapping of a designed algorithm to a practical usable setup. During the implementation of the wavefront reconstructor the focus lies on performance. Achieving a high performance is strongly dependent on the processing device. Furthermore to use processing devices to their full extend the algorithm must fit the hardware structure tightly. Already during the algorithm design in chapter 2 parallel computing was taken into consideration. Pointing towards devices that are capable of executing algorithms in parallel. There are two important properties induced by the wavefront reconstructor that affects the choice of hardware.

The first property is the expected high number of parallel sub problems originating from the E-ELTs design parameter that the WFS will have between 10.000 and 40.000 segments. The load balancing principle and Ahmdal's law prescribe that the closer the number of physical processors are to the number of sub problems the higher the potential performance. When reviewing the possibilities of parallel devices it is unavoidable to pass by the multi-core CPU. However the multi-core CPU fails to be a strong contender since they have just a few processing units, which is hardly a massively parallel device. The GPU on the other hand can contain up to several hundreds of cores also it requires more sub problems than physical present, which favours the GPU with respect to wavefront reconstruction. In Appendix A a summary of the architecture of the GPU can be found.

The second property is the high number of floating point operations in combination with a reasonable amount of memory access. In [7] also the FPGA was proposed as a possible contender for parallel wavefront reconstruction. It also stated that for the FPGA the performance is in the 300 gigaflop range while the GPU is capable of achieving teraflop peak performance. Such that with respect to floating point operations the FPGA is not the best option and clearly giving way to the GPU on this matter.

Due to the previously mentioned qualities the GPU was selected for the hardware. Consequently this chapter will discuss the implementation with respect to the GPU.

This chapter is organised as follows: Section 4-1 discusses the requirements on the memory of the GPU and evaluates if this would be sufficient. Section 4-2 determines the theoretically required FLOPs and compares it to a modern GPU. Section 4-3 describes the experimental setups that were used during the tests. Section 4-4 suggests two different implementations,

one based on a fine grained approach and the other based on a coarse grained approach. Section 4-5 compares the timing results achieved for both methods and points out the best one.

4-1 Memory induced requirements

Memory is a crucial element in every processing device. The memory stores and delivers the data to the processing unit, where the processing unit performs the operations on this data. Two essential parameters for memory are its capacity and its bandwidth. There are several reasons why the global memory is most interesting to analyse.

First, memory often has a certain hierarchy, where the smallest memory is close to the processing unit and can be accessed the fastest. In opposite, the biggest memory is usually the slowest memory. The global memory is the biggest memory inside the GPU. Consequently it is also the slowest memory, placed at the bottom of the memory hierarchy. While the global memory is several factors slower then the processing unit, other types of memory available in the GPU like the shared memory and registers are so fast that their latency with respect to the cores is minimal. The global memory is greatly influencing the latency times and therefore play a significant role in the performance.

Second, specific for the parallel computer domain is that either each processing unit has its own memory or the processing units share a memory. For the GPU the term shared memory is ambiguous, since the GPU consist of several multi-core processors. For the GPU the term shared memory refers to memory that is shared among cores however not between processors. The global memory on the other hand is accessible by every core. In addition the fact that the GPU is controlled by the CPU and the CPU only has access to the global memory, will make sure that the global memory is extensively used.

Third, the global memory is the only memory that is persistent between kernel calls. Persistent means that the variables are guaranteed to exists and will not be modified in between kernel calls. Since the wavefront reconstruction process is a real-time process it is obligatory to output the results every time instance. The architecture of the GPU dictates that communication with the CPU automatically results in a kernel call. Returning the wavefront phases and retrieving new measurements require a communication with the CPU, such that all data that is required at the next time instance should be stored into the global memory.

4-1-1 Memory capacity

Memory capacity refers to the amount of data that can be stored. Since memory is always confined it is important to check weather sufficient memory is available. When more data has to be processed also more data has to be stored, thus requiring a larger memory capacity. In relation to the E-ELT, the increased grid size results in more data.

The isolated localised wavefront reconstructor will be implemented, this reconstructor is defined by

$$\hat{x}^l_{(i,j)}(k+1|k) = \tilde{A}^l_{(i,j)}\hat{x}^l_{(i,j)}(k|k-1) + K^{yl}_{(i,j)}y^l_{(i,j)}(k)) \quad (4\text{-}1)$$

$$\hat{\phi}_{(i,j)}(k+1|k) = C^l_{\phi(i,j)}\hat{x}^l_{(i,j)}(k+1|k) \quad (4\text{-}2)$$

By observing the equations (4-1) till (4-2) it is possible to derive that the following variables needs to be stored: $\tilde{A}^l_{(i,j)} \in \mathbb{R}^{N_m \times N_m}$, $K^{yl}_{(i,j)} \in \mathbb{R}^{N_m \times N_u}$, $C^l_{\phi,(i,j)} \in \mathbb{R}^{1 \times N_m}$, $\hat{x}^l_{(i,j)}(k|k-1) \in \mathbb{R}^{N_m \times 1}$, $\hat{x}^l_{(i,j)}(k+1|k) \in \mathbb{R}^{N_m \times 1}$ and $\hat{\phi}^l_{(i,j)}(k+1|k) \in \mathbb{R}^{1 \times 1}$. The state space models existing out of standard matrices are accessed once each time instance, due to memory persistence and the real-time character of the wavefront they can be best stored in global memory. In addition this also applies to the measurements and the wavefront phases as they will be communicated back to the host each time instance. Less obvious is the fact that also the state estimates belongs should also be sored in global memory. One could say this is an intermediate result. Again, due to memory persistence the state estimate should be stored in global memory to be available the next time instance. Let us assume that we use single precision such that a single precision floating point decimal takes 4 bytes. Leaving us with the following expression for the required memory capacity

$$m = 4N(N_m^2 + N_m N_u + 3N_m + N_u + 1). \quad (4\text{-}3)$$

Where m defines the required memory size in bytes. Figure 4-1 shows how theoretical memory usage evolves in relation to the grid size, where the calculations have been based on level 1 and level 2 measurements or respectively involving $N_u = 4$ and $N_u = 8$ number of measurements. In addition the order of the local models are assumed to be maximal $N_m = 12$, due to the SIMD architecture of the GPU all cores perform the same operations such that the maximal order is decisive.

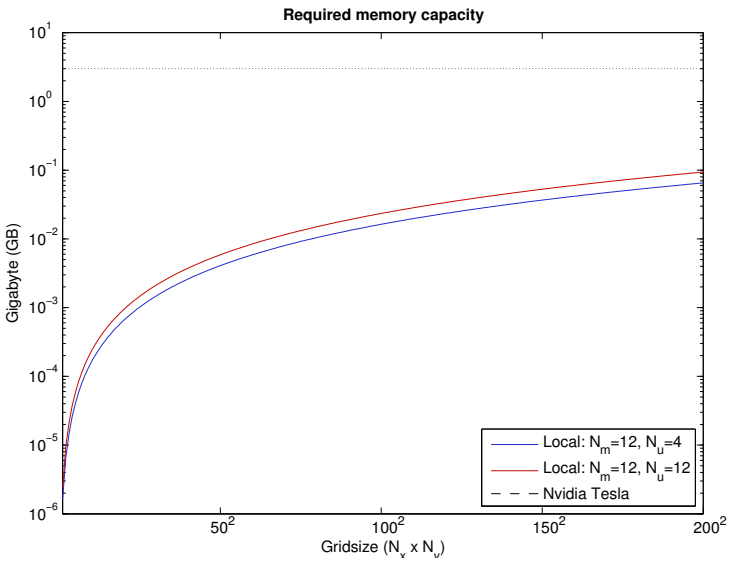


Figure 4-1: The by the localised reconstructor required memory resources in megabytes (MB). Both the level 1 and level 2 measurement set are evaluated against the available memory in a Nvidia Tesla C2070.

The black dotted line shows the memory capacity available on a Tesla C2070. Observe that the capacity is given in a logarithmic scale, which makes it obvious that memory capacity will not be an issue as there is still plenty of headroom for unexpected aspects.

4-1-2 Memory bandwidth

The memory bandwidth is a far more important parameter, especially in parallel processing. In the case of shared memory each processing unit is accessing the memory separately, meaning that they can access the memory all at the same time instance, stressing the access time of the memory. This effect holds especially for the GPU due to its SIMD structure where each processor is doing the same.

Besides data access also data storage has to be performed. Values that where changed in between have to be stored back to global memory. Of course this only applies if the data is required at the next time step and not for intermediate values. Now let us assume that a load operation is expensive as a store operation for the global memory. Furthermore there is also a distinction between the memory bandwidth inside the GPU and the communication bandwidth between the GPU and CPU. To be explicit, each time instance new measurements have to be send to the GPU and the resulting wavefront phases have to be communicated back to the host. This communication time does not depend on the speed of the global memory alone. For example the PCI Express bus also influences the duration. To make the theoretical analysis comprehensible we will only consider the data access inside the GPU. Based on equations 4-1 till 4-2 it can be noticed that $\hat{x}^l_{(i,j)}(k|k-1)$ is required twice. Since the state is relatively small it is a good candidate to put in the shared memory to avoid the multiple access off global memory. Similar the overlap between measurements can introduce multiple global memory access. However loading the measurements into shared memory introduces a lot of branch statements, i.e "for", "if" and "else" statements. Branch statements should be kept to the minimum due to the SIMD architecture of the GPU. Certainly for the level 1 case, the overlap introduces limited additional memory access (each measurement is accessed twice). Such that there is chosen to keep it in global memory. To calculate the bandwidth equation (4-3) should be multiplied with the framerate, in the case of the wavefront reconstructor 3000 Hz.

Figure 4-2 shows the required bandwidth with respect to the grid size. The limit for the level 2 case is reached for a grid size of approximately 190×190 at a frequency of 3000 Hz. One should say this is already pretty good. However the current analysis is still theoretical at a reasonable abstract level, often with respect to the final implementation this is an underestimate. Almost always there is overhead, for instance control instructions and variables. Not to mention simplifications that were made during theoretical analysis, e.g. neglecting the host to GPU communication. Due to this we should consider memory bandwidth as a serious limitation.

4-2 Computational complexity imposed requirements

Determining the computational load is strongly related to the computational complexity. Where the computational complexity is given for a single time instance the computational load is defined in the number of calculations per second. Given the desired framerate of

3000 Hz and the expression of the computational complexity (Equation 2-52) we are able to determine the computational load.

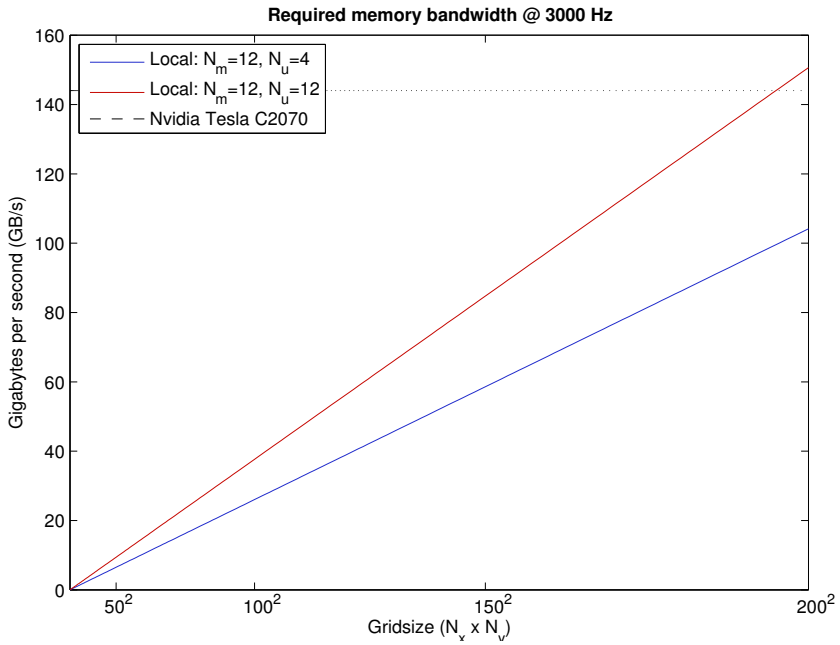


Figure 4-2: Required memory bandwidth with respect to the gridsize for a sample frequency of 3000 Hz.

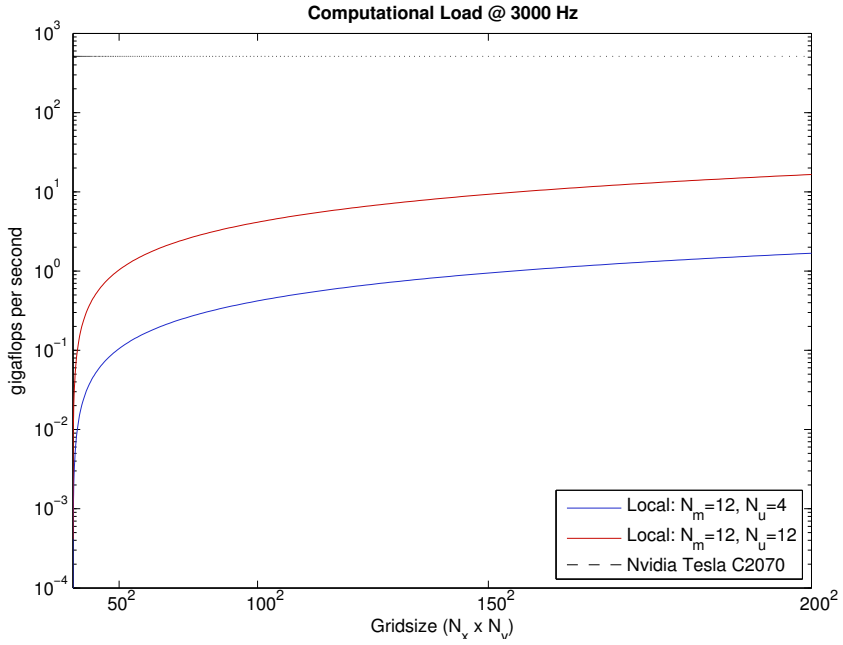


Figure 4-3: Required performance with respect to the gridsize for a sample frequency of 3000 Hz.

Figure 4-3 shows that the theoretical required load is significantly below the peak performance of a Nvidia Tesla C2070. Of course this does not show the total picture. As achieving the peak performance given by the manufacturer can be hard to achieve. Also since it is theoretically determined we have again an underestimation of the required performance. However with our prior knowledge about the memory bandwidth we can be sure that the computational load will not be the first restrictive factor.

4-3 Experimental setups

To compare the theoretical determined performance with a real setup we used several test setups. The experimental simulations were purely directed on a feasibility study of the implementation of the algorithm on a GPU. They do not include real life turbulence data and thus cannot be used for validating the applicability aspect.

The use of various types of GPUs with different capabilities will help in demonstrating the scalability. This is especially interesting when we are not yet capable of achieving sufficient performance. In this case these results can then help in extrapolating the timing values to see whether what will be feasible with future devices. The three different configurations that where used, which are depicted in Tables 4-1 till 4-3.

System 1: HP Compaq 8000 elite CMT	
CPU	Intel Quad Core CPU Q9650@3.000 GHz
Cache	12 MB
Cores	4
GPU	Nvidia Quadro NVS 290
Cores	2
Peak Performance (SP)	-
Memory Capacity	256 MB
Memory Bandwidth	6.4 GB/sec

Table 4-1: Nvidia Quadro NVS 290 based hardware configuration which is used as experimental setup to perform simulations.

System 2: HP XW4600	
CPU	Intel Quad Core CPU Q9550@2.836 GHz
Cache	6 MB
Cores	4
GPU	Nvidia Tesla C2070
Cores	448
Peak Performance (SP)	1.03 TFlop
Memory Capacity	6 GB
Memory Bandwidth	144 GB/sec

Table 4-2: Nvidia Tesla C2070 based hardware configuration which is used as experimental setup to perform simulations.

System 3:	
CPU	Intel Duo Core CPU E8500@3.16 GHz
Cache	6 MB
Cores	1
GPU	Nvidia Tesla C1060
Cores	240
Peak Performance (SP)	622 GFlop
Memory Capacity	4 GB
Memory Bandwidth	102.4 GB/sec

Table 4-3: Nvidia Tesla C1060 based hardware configuration which is used as experimental setup to perform simulations.

As one can observe also the CPU details are stated, as the GPU can not operate without a host. Obviously the CPU will also influence the performance. Even not the CPU alone does have an influence but the complete system configuration, for instance the PCI bus should transport all the data to the GPU.

4-4 Software design

The algorithm which was developed has to be converted into software. Preferable this should be done in an optimal way such that the execution time is minimal and the numerical accuracy is still within bounds. To validate the working principle of the algorithm turbulence data is required to supply the software with relevant data. Furthermore in the real application the reconstruction process will be the time critical part as this has to be performed in real time and the identification process can be done offline. Figure 4-4 gives an impression of how the software is implemented. For the simulation of the turbulence a model is developed on basis of the Kolmogorov theory and the frozen flow assumption. The model derivation is done in Matlab together with the turbulence data generation. The turbulence data is then preprocessed conform the WFS format including noise as defined by equation 4-4.

$$s_k = G\phi_k + n_k. \tag{4-4}$$

This data is send to the GPU by the use of the C-MEX interface and the CUDA API. There it is processed and the results are send back to Matlab by the same interface, in between the timing of the individual steps are being recoded. In Matlab the post processing and validation takes place.

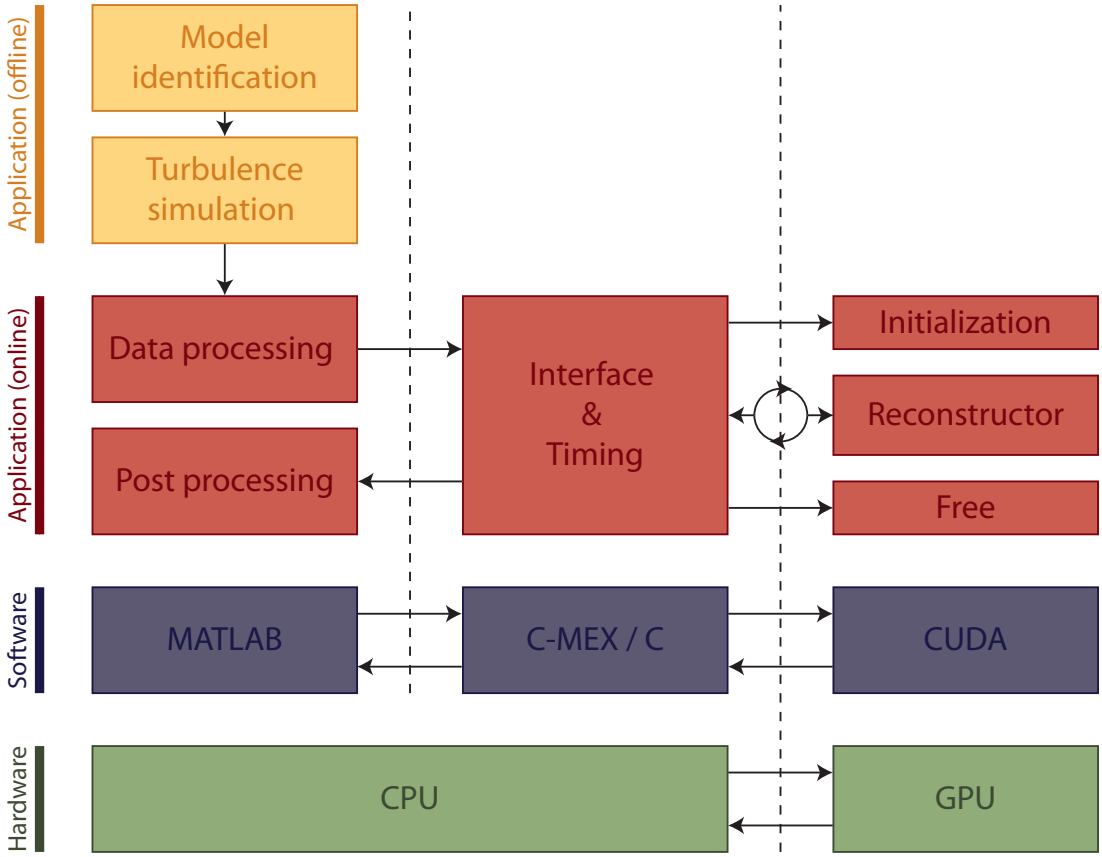


Figure 4-4: Schematic overview of the experimental setup used to simulate the wavefront reconstruction process on a GPU.

As efficiency is especially important for the wavefront reconstruction, let us zoom in how the real wavefront reconstruction takes place. Actually for the implementation on the GPU there is chosen to use two completely different strategies:

The first method is based on the coarse grained approach. Because the algorithm was developed with parallelism in mind a partitioning is possible that exactly fits the structure of the algorithm and threats every grid point as a sub problem. Such as denoted in equation 4-1 till 4-2. As this is a very specific case it requires custom software. This of course has the disadvantage that it can cost a lot of effort to reach highly optimised code. Especially as it can already be interesting to optimise for a specific type of GPU. Since this would lead too far for a proof of concept, the code is only optimised based on general GPU properties.

The second method on the other hand uses a highly optimised library called CUSPARSE. The library CUSPARSE offers an interface for sparse matrix vector operations specific targeting the GPU. To be able to use this library the algorithm should be rewritten in a single matrix vector multiplication where the matrix should be as sparse as possible. Denote that each states pace model can be written in the following format:

$$\left[\begin{array}{c} x(k+1) \\ \hline y(k) \end{array}\right] = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \left[\begin{array}{c} x(k) \\ \hline u(k) \end{array}\right] \tag{4-5}$$

Which makes it able to argue that a set of decoupled state space models can also be written

as a matrix vector multiplication. The wavefront reconstruction algorithm consisting out of decoupled models is allowed to be modified to equation 4-6.

$$\left[\begin{array}{c} \hat{x}^l_{(1,1)}(k+1) \\ \vdots \\ \hat{x}^l_{(N,N)}(k+1) \\ \hline \hat{\phi}^l(k+1,1,1) \\ \vdots \\ \hat{\phi}^l(k+1,N,N) \end{array}\right] = \left[\begin{array}{cccc|cccc} \tilde{A}^l_{(1,1)} & 0 & \dots & 0 & K^{yl}_{(1,1)} & 0 & \dots & 0 \\ 0 & & & \vdots & 0 & & & \vdots \\ & & \ddots & & & & \ddots & \\ \vdots & & & 0 & \vdots & & & 0 \\ 0 & \dots & 0 & \tilde{A}^l_{(N,N)} & 0 & \dots & 0 & K^{yl}_{(N,N)} \\ \hline C^l_{(1,1)} & 0 & \dots & 0 & 0 & \dots & & 0 \\ 0 & & & \vdots & & & & \\ & & \ddots & & \vdots & & \ddots & \vdots \\ \vdots & & & 0 & & & & \\ 0 & \dots & 0 & C^l_{(N,N)} & 0 & \dots & & 0 \end{array}\right] \left[\begin{array}{c} \hat{x}^l_{(1,1)}(k) \\ \vdots \\ \hat{x}^l_{(N,N)}(k) \\ \hline y^l_{(1,1)}(k) \\ \vdots \\ y^l_{(N,N)}(k) \end{array}\right]^* \tag{4-6}$$

by the observation that the measurement $y^l_{(i,j)}$ is a subset of the total set of measurements $s(k)$ and that there is overlap between the different subsets we can even reduce the complexity further. Since in the current form most of the measurements are multiple times present in the vector (equation 4-6*) introducing unnecessary additional complexity. By changing the structure of the upper right block matrix from diagonal to coincide with the measurements the repeating entries can be removed.

The matrix vector multiplication has to be executed every time step. Where the matrix is time invariant and can be determined during the identification procedure. The vector is based on the states which are taken from the previous time step present in the upper lock of resulting vector. Concatenated with the current measurements.

The usage of two different methods is done for comparison purposes between own coded software and using already optimised libraries. Beside, this allows us to validate the hypothesis that course grained is preferred above fined grained methods. Due to the fact that most libraries are build for evaluating more fundamental operations, which will automatically introduce a more fine grained approach.

4-5 Suitability of two different implementations

Two different methods were discussed to solve the wavefront reconstruction problem. Let us now determine which method should be used to proceed. As both methods solve exactly the same problem, we are interested in the one that solves the problem in the most efficient way with relation to the selected hardware. To determine the fastest method both were implemented and executed on a GPU for various grid sizes. To compare the effect of an appropriately designed algorithm with respect to the selected hardware, the reconstruction methods were also implemented on a CPU, both tests were executed on system 3. Figure 4-5 and 4-6 shows the results respectively the GPU and CPU.

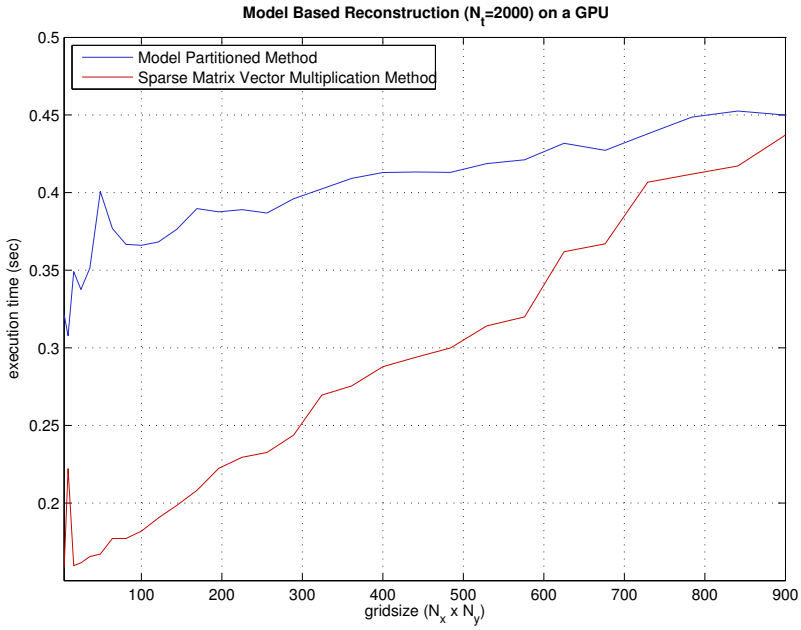


Figure 4-5: Timing results of the model sparse matrix vector multiplication method versus partitioned method implemented by using a GPU with grid sizes from 2×2 till 20×20. The results are based on 2000 reconstructions excluding the initialisation time.

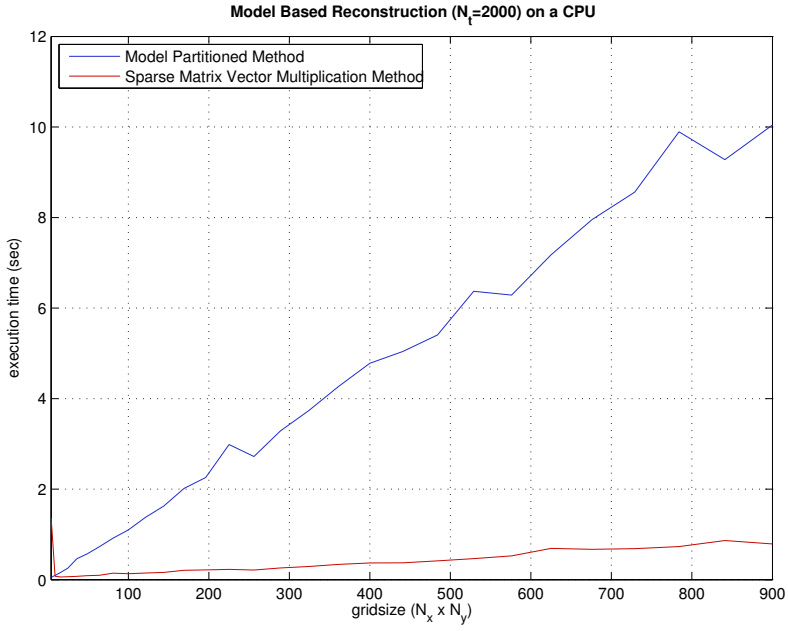


Figure 4-6: Timing results of the model sparse matrix vector multiplication method versus partitioned method implemented by using a CPU with grid sizes from 2×2 till 20×20. The results are based on 2000 reconstructions excluding the initialisation time.

If both figures are compared it is obvious that the GPU is a lot faster in solving the given problem. A speed-up factor of 22.22 and 2.1 is achieved for respectively the MP method and

the SMVM method. Note the steep increase of the MP method on the CPU, which could be explained by the knowledge that the MP method depends most on parallelism since it is coarser grained. To determine the most suitable method on the GPU the results in figure 4-5 should be analysed. The results are relatively close and it would be easy to conclude the SMVM method is better as in this range it has the fastest executions times. However at the end the scalability will be decisive and over here the MP method is in the advantage. The incline of the MP method is smaller what is favourable, certainly when the difference in a confined simulation range is relatively small. Take also notice of the fact that the CUSPARSE libary is already highly optimised and for the custom developed software there is probably still room for some improvement. This allows us to safely conclude that the MP method is the preferred way to go for the GPU implementation,which directly means that from here on we will only consider the MP method with relation to the analysis.

Another crucial aspect in the upcoming tests was to determine whether single precision calculations were sufficient and achieve comparable results compared to dual precision calculations. Some types of GPUs can perform both single as dual precision operations. However the dual precision calculations are always significantly slower making it preferable to do the calculations in single precision format. Resulting in the fact that this will have a major impact on both the timing results and the hardware choice. What would provide an answer is an evaluation of the absolute accuracy on both the GPU and CPU. Figure 4-7 shows the simulation of the absolute accuracy expressed in strehl ratio.

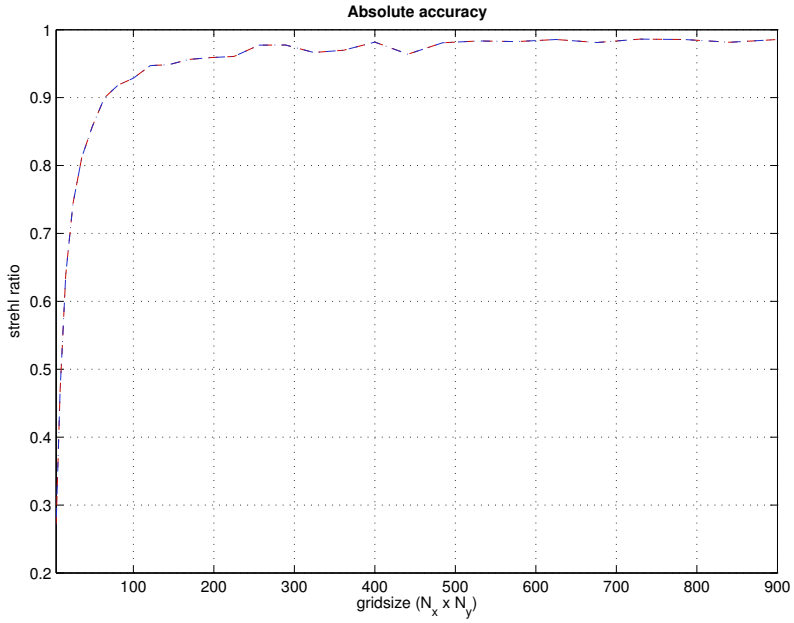


Figure 4-7: Absolute accuracy for verification of the single precision GPU implementation.

The absolute accuracy measure gives already a pretty conclusive result. The plots are exactly on top of each other. From which we can safely assume that the single precision does not affect the accuracy of the reconstruction. Still it might be interesting to know if there is a difference between the result obtained from the GPU and the CPU. Which can be revealed by a relative accuracy measure, to be more precise the normalised root mean square error.

$$RMS(\hat{\phi}_{cpu} - \hat{\phi}_{gpu}) = \sqrt{\frac{\left(\frac{\hat{\phi}_{cpu}(1)-\hat{\phi}_{gpu}(1)}{\hat{\phi}_{cpu}(1)}\right)^2 + \cdots + \left(\frac{\hat{\phi}_{cpu}(N)-\hat{\phi}_{gpu}(N)}{\hat{\phi}_{cpu}(N)}\right)^2}{N}} \tag{4-7}$$

clearly, as we can observe from figure 4-8 the relative error is non-zero. Which raises the question why the difference did not affect the absolute performance. This is because the magnitude of the relative error is so small that is does not propagate in the strehl ratio. To put it in other words on the current plotting scale it is just not visible.

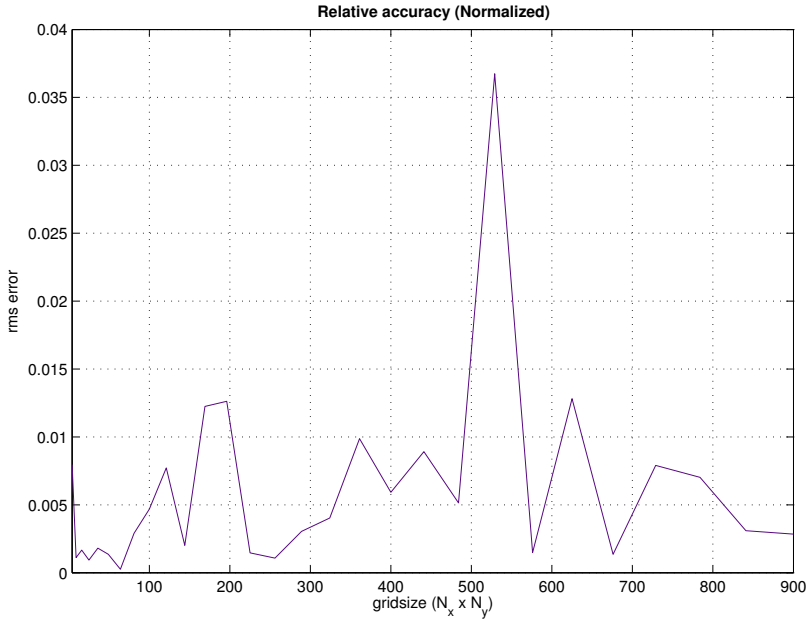


Figure 4-8: Relative normalised accuracy for verification of the single precision GPU implementation.

Chapter 5

Simulation Results

To validate and verify the methods developed, a wide variety of simulations where performed, including a number of tests on experimental setups. These tests were aimed at finding timing details of the algorithm and to show how the algorithm performs under different conditions. In chapter 4 we have already seen some preliminary results, whereas in this chapter you will find a more in depth discussion.

This chapter is organised as follows: Section 5-1 discusses how variations in the localised wavefront reconstructor affects the reconstruction accuracy. Section 5-2 emphasises on the GPU timing and shows the results of reconstruction on some high end GPUs.

5-1 Varying system parameters

Each system has design parameters and external conditions which affect the performance levels of the systems functioning. This also applies for the local wavefront reconstructor derived in chapter 2. An important design parameter is the number of measurements. Whereas external conditions direct more towards turbulence properties like wind speed and direction. In the upcoming sections the effect of variations in some of these variables will be evaluated.

5-1-1 The effect of spatial information

The amount of spatial information is one of the design parameters that influence the wavefront reconstruction accuracy. To see what the effect is of the quantity of measurements assigned to a single local model, several cases shall be evaluated. Each case will reflect on an increasingly larger number of measurements.

Already in chapter 2 we have seen the results for a level 1 and level 2 case with respect to a 8×8 grid. Three additional sets of measurements are defined, a level 3, level 4 and the case where all measurements are included. To be able to make a fair comparison associated to the level 1 and 2 case, all the simulation parameters are kept equal to the ones defined in

chapter 2. The measurement set definitions of the different cases are given. Starting with the level 3 case, which is defined by

$$\begin{aligned} y_{(i,j)}(k) &= [s_{x,i-2,j-1}(k), s_{x,i-1,j-1}(k), s_{x,i,j-1}(k), s_{x,i+1,j-1}(k), s_{x,i-2,j}(k) \\ &, \quad s_{x,i-1,j}(k), s_{x,i,j}(k), s_{x,i+1,j}(k), s_{x,i-2,j+1}(k), s_{x,i-1,j+1}(k) \\ &, \quad s_{x,i,j+1}(k), s_{x,i+1,j+1}(k), s_{y,i-1,j-2}(k), s_{y,i-1,j-1}(k), s_{y,i-1,j}(k) \\ &, \quad s_{y,i-1,j+1}(k), s_{y,i,j-2}(k), s_{y,i,j-1}(k), s_{y,i,j}(k), s_{y,i,j+1}(k) \\ &, \quad s_{y,i+1,j-2}(k), s_{y,i+1,j-1}(k), s_{y,i+1,j}(k), s_{y,i+1,j+1}(k)]^T \end{aligned} \tag{5-1}$$

Followed by the level 4 case where the assigned measurements are

$$\begin{aligned} y_{(i,j)}(k) &= [s_{x,i-2,j-2}(k), s_{x,i-1,j-2}(k), s_{x,i,j-2}(k), s_{x,i+1,j-2}(k), s_{x,i-2,j-1}(k) \\ &, \quad s_{x,i-1,j-1}(k), s_{x,i,j-1}(k), s_{x,i+1,j-1}(k), s_{x,i-2,j}(k), s_{x,i-1,j}(k) \\ &, \quad s_{x,i,j}(k), s_{x,i+1,j}(k), s_{x,i-2,j+1}(k), s_{x,i-1,j+1}(k), s_{x,i,j+1}(k) \\ &, \quad s_{x,i+1,j+1}(k), s_{x,i-2,j+2}(k), s_{x,i-1,j+2}(k), s_{x,i,j+2}(k), s_{x,i+1,j+2}(k) \\ &, \quad s_{y,i-2,j-2}(k), s_{y,i-2,j-1}(k), s_{y,i-2,j}(k), s_{y,i-2,j+1}(k), s_{y,i-1,j-2}(k) \\ &, \quad s_{y,i-1,j-1}(k), s_{y,i-1,j}(k), s_{y,i-1,j+1}(k), s_{y,i,j-2}(k), s_{y,i,j-1}(k) \\ &, \quad s_{y,i,j}(k), s_{y,i,j+1}(k), s_{y,i+1,j-2}(k), s_{y,i+1,j-1}(k), s_{y,i+1,j}(k) \\ &, \quad s_{y,i+1,j+1}(k), s_{y,i+2,j-2}(k), s_{y,i+2,j-1}(k), s_{y,i+2,j}(k), s_{y,i+2,j+1}(k)]^T \end{aligned} \tag{5-2}$$

For the last case all measurements are taken into consideration and $y_{(i,j)}(k)$ is given by

$$y_{(i,j)}(k) = [s_x, s_y]^T \tag{5-3}$$

Figures 5-1 till 5-3 show the results obtained for the proposed cases. The simulations were performed for the global reconstructor, the local optimal reconstructor and the local identified reconstructor. The figures present their results in terms of strehl ratio versus SNR.

The trend visible from the three figures is that the localised reconstructor asymptotically approaches the global reconstruction accuracy with respect to the number of measurements. Incorporating all measurements in the local models leads to a similar performance compared to the global reconstructor, which can be established by figure 5-3. Consequently it is possible to conclude that increasing the number of measurements assigned to a local model can be used to reduce the noise sensitivity of the local reconstructor. However, the number of measurements per level increases fast while the noise sensitivity decreases relatively slow. For instance, while the level 1 case consist out of only 4 measurements, the level 4 case already consists of 40 measurements. Since increasing the measurements will also lead to higher order models, it should be clear that both the model order and number of inputs influences the computational complexity negatively. All in all this confines the applicability of increasing the number of measurements to reduce noise sensitivity.

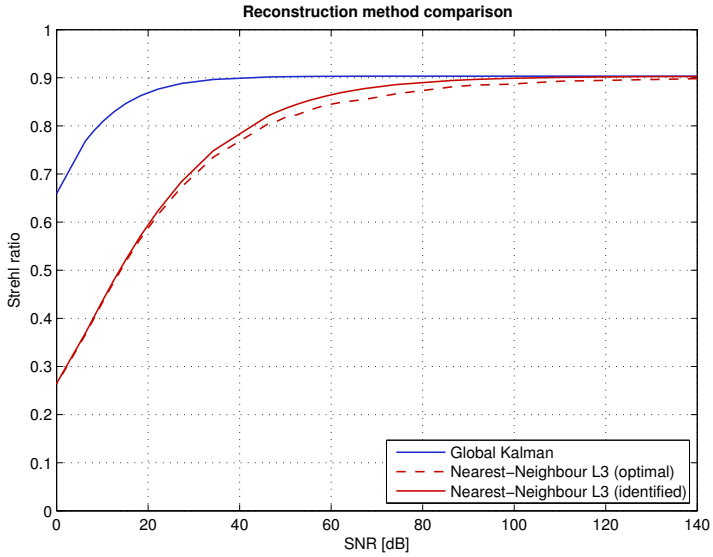


Figure 5-1: Prediction error in terms of strehl versus SNR to evaluate the effect of an increasingly larger set of measurements. This figure reflects on the level 3 case. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

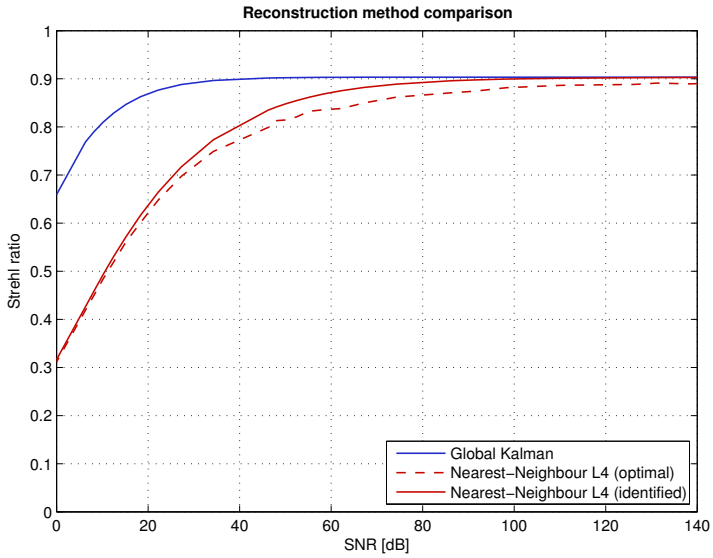


Figure 5-2: Prediction error in terms of strehl versus SNR to evaluate the effect of an increasingly larger set of measurements. This figure reflects on the level 4 case. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

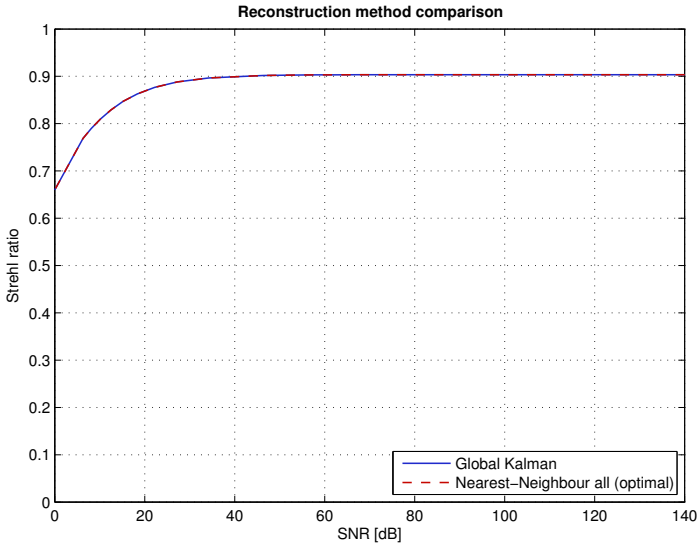


Figure 5-3: Prediction error in terms of strehl versus SNR to evaluate the effect of an increasingly larger set of measurements. This figure reflects on the case where all measurements are taken into account. Applied to a gridsize of $(N_x \times N_y) = (8 \times 8)$.

5-1-2 The influence of different grid sizes

The linear order is a strong property of the local reconstructor. After all this is what makes the algorithm scalable. Till now it was only mentioned that the order of the models should be much smaller than the order of the global model. Moreover for a grid size of 8×8 it was shown that it was possible to reduce the local models below or equal to an order of 12. When it is feasible to identify local models with similar size for bigger grids the system will be truly linear. For this purpose a simulation was performed for a grid size of 16×16. Figure 5-4 shows the results of the simulation. The remaining simulation parameters were kept equal to the ones used in previous simulations and can be found in chapter 2.

We can again recognise the global reconstructor, the local optimal reconstructor and the local identified reconstructor. Also for this situation the models were identified with an order between 5 and 12. Nevertheless comparable results are obtained with respect to the 8×8 grid. Similar as before the local reconstructor starts to drop around 80 dB.

When observed closely it can be seen that it is shifted a few decibels. Actually this points to a more remarkable effect, the global reconstructor result is shifted over a full 20 dB. The shift can be clarified by the increased amount of spatial information over the same area. Since all the other parameters are kept equal including the diameter of the aperture, by increasing the grid size the resolution is increased. Such that there is more spatial information for the same area available.

Another effect visible is the slight increase in strehl ratio. This can be explained by the frozen flow property of the turbulence. In the direction where the wind is coming from, there is no information present for the first segments. While the upwind segments have the measurements down wind till their disposal which allows them to better predict the wavefront. Since for the 16×16 grid the first row covers a relatively small area the influence is smaller, leading to better end results. Figure 5-5 visualises the location dependent properties by

showing the prediction error per segment.

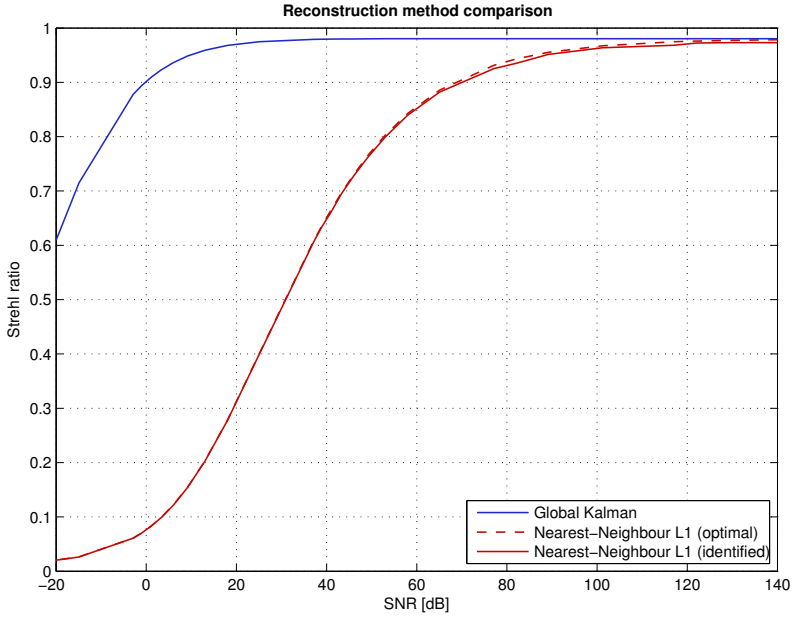


Figure 5-4: Prediction error in terms of strehl versus SNR applied to a gridsize of $(N_x \times N_y) = (16 \times 16)$.

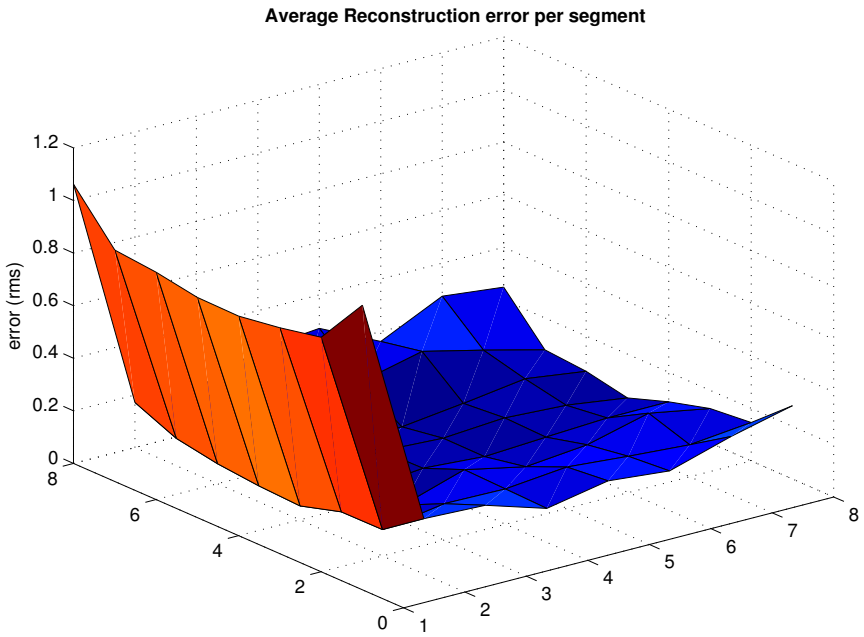


Figure 5-5: Error surface of 8×8 grid viewed segment wise.

To come back on the scalability it was shown that models of similar order could be identified for larger grid sizes from which it is possible to conclude that this would also be feasible for

E-ELT sized grids.

5-1-3 The effect of changing model conditions

The proposed reconstructor is model based. Every model is a representation of the reality and it represents the reality to a certain degree. Depending on resemblance between model and reality the model based method can be well or ill conditioned. Up till this moment the emphasis was on a close resemblance between model and reality. In practise it will however be likely that there will be deviations present. After all models need to be identified and during that time wind conditions like velocity and direction can change. In addition models are always a simplification of the real world as modelling the real word would be far to complex. To evaluate the consequence of model inaccuracy, two simulations where performed. Figure 5-6 depicts the situation where between model usage and identification the wind has decreased in velocity with $5\,\mathrm{m/s}$. With the settings of Chapter 2 this results in a wind velocity for the x direction of $v_x = 15\,\mathrm{m/s}$.

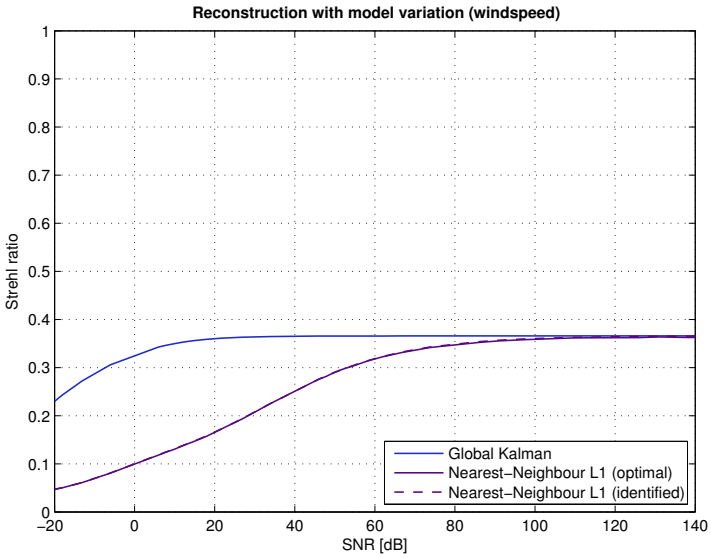


Figure 5-6: Prediction error in terms of strehl versus SNR applied to a grid size of (8×8). Where model inaccuracy is present due to variations in wind speed.

The second situation reflects on a change in wind direction. Again consider the turbulence settings of Chapter 2, for this case the wind parameters are changed to $v_x = 0\,\mathrm{m/s}$ and $v_y = 20\,\mathrm{m/s}$. Figure 5-7 depicts the simulation result of the given situation.

In the figures the same reconstructors as before are compared. The variations between model and reality affects the accuracy severely for all reconstructors. The situation where variation in wind speed is shown looses approximately 0.5 points in strehl ratio. For the variation in direction it is even worse. The reconstruction almost completely fails and is just able to achieve a strehl ratio of 0.01. Both the variation in direction with 90 degrees and the variation of wind speed with a deviation of 25% is rather significant. In practise such huge changes take longer periods of time to occur, despite smaller changes could certainly happen on a minute time scale. Although the influence of smaller variations are limited, it

still affects the accuracy of the reconstruction. It is not possible to completely overcome this issue, however it can be reduced by performing intermediate model updates during longer observations.

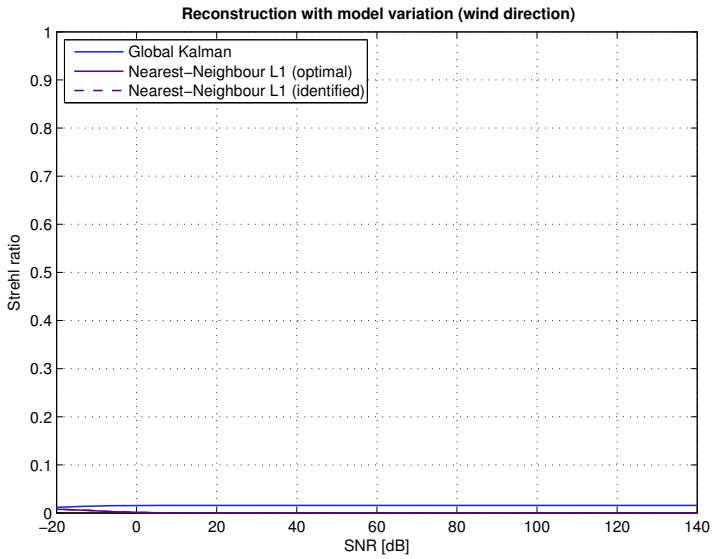


Figure 5-7: Prediction error in terms of strehl versus SNR applied to a grid size of (8×8). Where model inaccuracy is present due to variations in wind direction.

5-2 GPU timing results

In this section an evaluation will be carried out, determining the possibilities with current hardware and the developed algorithm in terms of execution time. Especially when this is compared to requirements set by the E-ELT. In addition a comparison of how the experimental setup relate to the theoretical findings in chapter 4 will be presented.

For the timing of the code seven different timers were introduced, all measure a section or overlapping sections of the reconstruction code. The first timer measures the complete process from loading the models into the GPU till destroying the variables used during execution. The second timer measures the initialisation of the models, which includes reformatting them to single precision format and loading them to the GPU. The third timer determines the sum of all repeating reconstruction instructions for a set of time instances. This involves loading the slopes to the GPU, performing a single reconstruction and storing the reconstructed wavefront on the host. The next consecutive timers 4, 5 and 6 separately measure loading, reconstruction and storing. The seventh timer measures the time required to cleanup all the used data both at the GPU and CPU.

By having defined timers it is possible to determine what to time. With respect to the E-ELT it will be interesting to see how problem size affects the performance. To evaluate the effect of increasingly lager grid sizes, for each grid a turbulence data generation model is required. These are rather computational complex to determine. It is even computational infeasible to simulate up to 200×200 grid. In particular memory limitations restricted us to a maximum grid of 52×52. All experiments on the GPUs were performed with level 1

local models. Again, simulations are based on the turbulence statistics defined in chapter 2. Figure 5-8 depicts the results for gridsizes of 2×2 up to 52×52 .

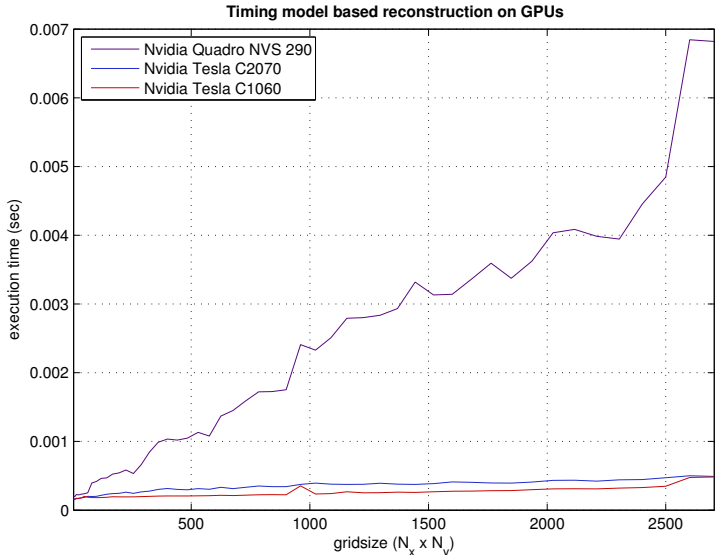


Figure 5-8: Average timing results for a single reconstruction taken over 2000 reconstructions for a gridsizes of 2×2 up to 52×52.

To arrive at the results in figure 5-8, 2000 reconstructions were performed and averaged. Furthermore three different configurations were used, these configurations are denoted in chapter 4. During the execution, timing was performed in the format discussed before. Timer 3 was used for this comparison. Since timer 3 includes repeating code dependent on the number of reconstructions, it can be utilised to determine the time required for a single reconstruction.

Configuration 1 with the NVS 290 is obviously the slowest of the three, based on the specifications this could be expected. On the other hand configuration 2 and 3 are almost comparable. Observe even that the Tesla C1060 is slightly faster, which is quite remarkable since the Tesla C2070 should theoretically be able to achieve a higher performance. A plausible explanation could be that due to the slower CPU kernel start ups configuration 2 is slightly slower. Furthermore it is very likely that the Tesla C2070 will be better in keeping the present trend for bigger grid sizes as it has more cores and a higher memory bandwidth. Extended experiments with bigger grid sizes could verify these statements.

Let us further reflect on the results of configuration 3. Configuration 3 was able to reconstruct a grid of 2500 segment in 0.348 milliseconds. For the desirable framerate of 3000 Khz this would involve a total reconstruction time of just over one second. Despite that the results are almost within bounds, bear in mind this is for a grid size of 50×50, where for the E-ELT a 200×200 grid has to be reconstructed. In addition take into mind that reconstruction is not the only process in AO control. After all the DM has also to be controlled. To be able to draw some conclusions for bigger grid sizes the results are extrapolated under the assumption the GPUs could keep up the linear scale. Figure 5-9 shows the extrapolated timing result of a single reconstruction.

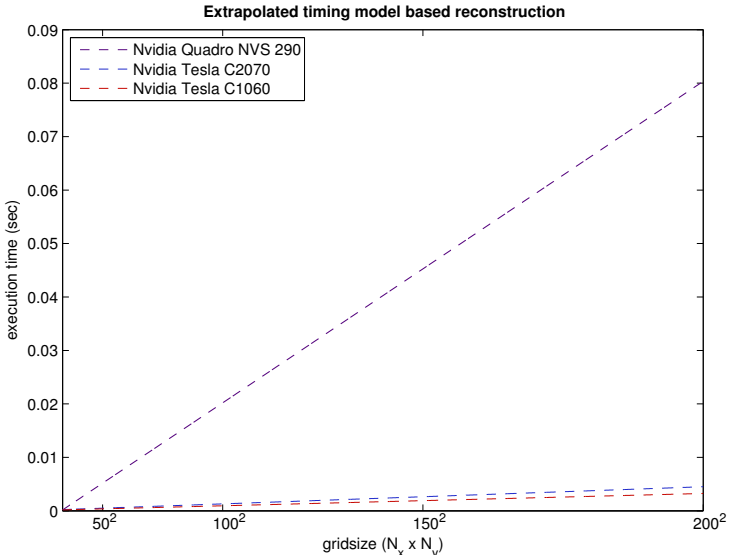


Figure 5-9: Initialisation times for gridsizes of 2×2 up to 52×52.

All the obtained results in figure 5-9 are estimations and so do not give any guarantee how the real execution time would evolve. One could imagine that at a given moment the memory bandwidth gets saturated introducing a sudden increase in runtime. However these results give us a clue what could be achievable for bigger grid sizes.

When we assume a 200×200 grid and derive the execution time, the outcome will be 4.5 milliseconds for a single reconstruction. When converting this to a framerate it results in a frequency of 222 Hz. This is much lower than the goal set by the E-ELT. There are a few approaches that can improve the performance to the desired level.

The first and cheapest approach is to optimise the software. At this moment a straight forward implementation of the algorithm incorporating prototyping and profiling statements. Suggesting there is still room for improvement. Optimising software could give a reasonable gain, but will probably not yet be enough to meet the requirements.

The second approach will be altering the hardware. Spreading the algorithm over multiple GPUs would also spread the computational load. Due to the structure of the algorithm this will be easy to achieve and will give a better processing unit versus sub problem balance. However keep in mind that GPUs require more sub problems then processor units to hide the memory latency. Figure 5-8 shows that a single GPU can reach kilohertz performance for a grid size of 50×50, consequently 16 GPUs are required for a grid size of 200×200.

The third approach points towards another hardware solution already suggested earlier which is scaling up the individual GPU performance. Since the Tesla C2070 is in the top segment, only future devices could provide this enhancements in performance. From figure 5-9 it is possible to give already an indication of the potential gain. When comparing the performance of NVS 290 and Tesla C1060 when reconstruction a 200×200 grid, there is approximately a gain of factor 18. By relating this to the memory bandwidth of both GPUs we see that we have a similar factor of around 16. A comparable enhancement in a future device would imply that the framerate requirement for the E-ELT could be met by making the reconstruction for 3000 frames possible to be carried out well under one second.

Initialisation was not incorporated in figure 5-8 and 5-9. As initialisation is only performed

once it does not influence the reconstruction time and thus the maximum achievable framerate. However due to the turbulence variations, regular model updates might be required. Such that it is crucial that the initialisation time is within bounds. Would the initialisation take up a huge amount of time the models could already be outdated before they are put to use. In addition intermediate model updates require fast model update to avoid disturbing the reconstruction process. For evaluation figure 5-10 depicts the timing results of the initialisation performed in advance of the reconstruction used for the results in figure 5-8.

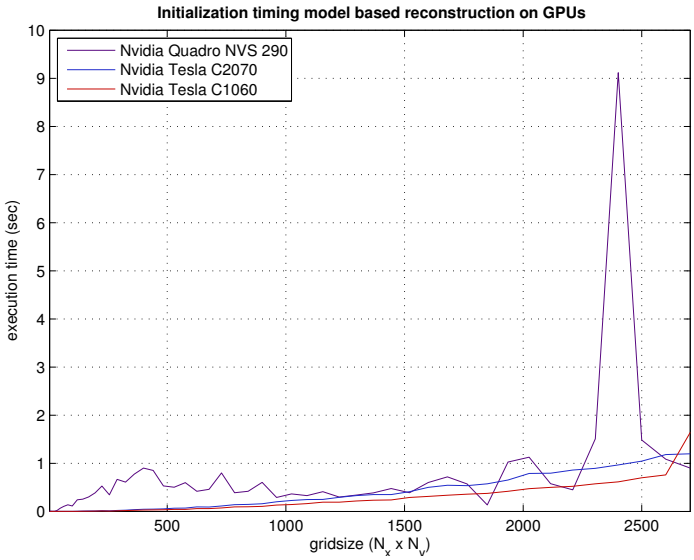


Figure 5-10: Initialisation times for gridsizes of 2×2 up to 52×52.

The initialisation is expressed in execution time versus grid size. Except for some rather huge variation the results between configurations are pretty close to each other. An explanation for this is that significant parts of the initialisation time is determined by the host. Since the hosts do not differ significantly the results are comparable. The variation for configuration 1 is significant and makes drawing conclusions difficult. However the knowledge that none of the systems contained a real-time operating system improves the understanding. Another process could have interrupted, causing a spike in execution time. Taking this into account a stable initialisation time of around one second is certainly achievable for a grid size of 2500 segments. The question whether this is within bounds is not answered by this. The bounds depend strongly on the variation of the turbulence. Would it be required to update the models during a observation, a one second interruption of reconstruction would definitively degrade the image quality. However would a model be valid for longer period of time an initialisation time of a few seconds would be no problem.

Based on the timers a more detailed timing model can be determined. This will give us more insight into the process that contributes most to the execution time. During the analysis every separate process is also assumed affine. Such that each process can be defined by $f(N) = |t_1|N + |t_2|$ where t_i defines the duration in microseconds. Denote that absolute values are taken, due to the fact that negative execution times are not possible. Based on the data retrieved from configuration 3 we arrive at the timing model depicted in table 5-1.

Process	Number of executions	Timing $f(N)$
Initialisation	1	$252.24N + 0.62$
Load Measurements	N_t	$0.024N + 14.43$
Reconstruct	N_t	$0.042N + 142.64$
Store Wavefront	N_t	$0.01N + 13.41$
Cleanup memory	1	$0N + 124.41$
Total	1	$N_t(0.076N + 170.48) + 252.24N + 125.03$

Table 5-1: Timing model for each separate process based on the experimental results achieved on configuration 3.

When only taking into account the repeating processes a remarkable observation can be made. Denote that a significant part of the execution time is made up by the constant factor. The constant factor represents for example the kernel start up in the reconstructing process. To be even more concrete, till a grid size of approximately 2200, the constant factor dominates the execution time. Reducing this constant time by optimisation could thus be very interesting.

The timing model also allows to relate the results to the theoretical results defined in chapter 4. In the theoretical analysis only the reconstruction process was considered. Furthermore the theoretical analysis did not include the host or the communication between the host and the GPU. As the constant time is mainly due to kernel start up it should be neglected. Which leaves us with $f(N, N_t) = 0.042NN_t$ to compare to the theoretical analysis. In chapter 4 we concluded that the memory bandwidth will be the limiting factor for the wavefront reconstructor. Theoretical it should be possible to perform 3000 reconstructions of a grid with size 150×150 well within one second. The experimental result does show a different story, when we fill in the numbers we are left with $f(22500, 3000) = 2.8$ seconds. Still making us wonder where the difference is coming from. Theoretical timing analysis almost always underestimate the final implementation execution time. Certainly when it is done at a relatively abstract level like for the wavefront reconstructor. In this case a lot of final implementation issues are ignored e.g intermediate variables, branches, control instructions, etc.

Chapter 6

Conclusions & Future Work

6-1 Conclusions

The document at hand addresses the problem of efficiently computing the wavefront reconstruction of an extremely large telescope (ELT). The insights that played a crucial role and led towards the solution were; using the vast amount of computational power that parallel computing offers and designing an algorithm that tightly fits the architecture of the hardware.

Imposed by the parallel hardware offering vast amounts of computational power, an algorithm was required that could easily be divided into sub problems. This was achieved by exploiting the grid structure of the wavefront sensor and treating the reconstruction of every sensor segment as a separated problem, which enabled the design and implementation of a set of localised reconstructors. The major advantages of the localised reconstructor are its coarse grained parallelism and its linear order $O(N)$ with respect to the number of segments.

In Chapter 5 it was shown that the localised reconstructor could complete 2000 reconstructions for a grid size of 52×52 segments within 1 second on a single Nvidia Tesla C2070. By spreading the load over multiple GPUs, 16 modern GPUs will be capable of meeting the E-ELT requirements, which would involve completing the reconstruction of a 200×200 grid at kilohertz rate. In addition, future GPUs are expected to be more powerful, since development of processing power is dictated by Moore's law. The law states that the number of transistors at an equal area doubles every one and a half year. Extrapolating results in the fact that in six years, a single GPU should also be capable of performing the wavefront reconstruction for an E-ELT sized grid.

There is however a tradeoff compared to global solutions and this lies in accuracy. The localised reconstructor has a higher noise sensitivity and performance will drop for SNRs below 80 dB. Incorporating more measurements would compensate for this but also increases the computational load. A more efficient solution can be found by introducing communication between the local reconstructors.

Since each local reconstructor in its current form is completely independent, the algorithm could benefit from collaboration between the reconstructor in the form of state exchange. Chapter 3 proposed to use diffusion algorithms as form of state exchange. Diffusion algorithms prove to be able to reduce the noise sensitivity with 20 dB. It should denoted that this

was shown on the full order models, which are extremely computational complex. To apply diffusion efficiently it should be applied on the reduced order models.

Let us conclude by recapitulating on the original thesis subject; *"The development of an adaptive optics control algorithm to compensate for atmospheric turbulence, such that it is suitable to be implemented on an array of processing units. The goal is to set the requirements for the E-ELT application in terms of processing bandwidth and architecture modular scalability"* Throughout the research the emphasis was more on wavefront reconstruction then on AO control. Although the thesis question describes a broader scope the most fundamental problem in AO control with respect to the E-ELT is addressed. Namely the computational complexity of the wavefront reconstructor. Overall it is possible to conclude that the proposed methods are successful in answering the thesis question, since it is one of the few or even the only algorithm at this moment proven to be capable in solving wavefront reconstruction at real-time for E-ELT sized grids. Solving a fundamental issue in controlling the AO system the E-ELT.

6-2 Future Work

Answering a question often raises new questions which was also the case during this research. An overview will be presented of possible future research topics.

- Each segment of the localised reconstructor gets a set of measurements assigned. At this moment an selection of the nearest measurements is made and several sets are defined to show the impact of the number of measurements. However determining which and the amount of measurements taken into account for a certain segment is an optimization problem. Observe for example that measurements upwind could provide more information compared to measurements downwind. Also the amount of noise present on the measurements plays a crucial role. More noise requires more measurements to be able to average the noise and thereby achieve good noise sensitivity. So finding the optimal set of measurements per segment is not straight forward and is still left as an open issue.

- Similar for the diffusion algorithm, there has to be determined which segments are connected to each other and thus exchange state information. Determining this optimum is also not addressed and could significantly improve performance.

- The state exchange in the diffusion algorithm is now a weighted average. The weights of the diffusion algorithm are chosen as the degree of a segment as suggested in [1], which is a rather heuristic method of determining weights. Furthermore the weights are vector wise and it should be clear that the accuracy of state estimation is state element dependent. So modifying it by making the weights state element dependent and choosing optimal weights would benefit the algorithms accuracy.

- At this moment the diffusion algorithm was applied on the full order models since the methods proposed in literature only work on full order models. To make the diffusion algorithm for wavefront reconstruction computational wise attractive, the algorithm has to be modified to work on reduced order models. In Chapter 3 a basis towards reduced

order models was already presented, but still it requires additional research to make it applicable.

- Besides the applicability of diffusion algorithms on reduced order models, also the identification of local models requires research. In its current form the local models are identified separately. With the diffusion algorithm it is required to know the relation between the local models to allow for state exchange. One solution is to derive local models from a global model and apply order reduction, from which it is possible to deduce the similarity transformations. However, this is computational very complex and even although identification can be performed offline it is very hard to solve. The second solution is distributed identification methods. Those are methods that threat every local model separately but also takes notice of the coherence between them.

Appendix A

Parallel processors

Parallel processors refer to a wide variety of hardware that supports parallel computing. To apply the parallel computing concept, not only the algorithms have to be converted. The underlying hardware has to have the ability to cope with this as well. With the current development in CPU technology the MIMD architecture is becoming more and more common. Nowadays desktop computers come standard with a multi-core CPU. When applying parallel technologies, optimal performance is a crucial factor. Choosing the right hardware architecture that matches the problem is thus rather important.

Multi-core CPUs devote a lot of die area to control logic. For common scientific calculations this makes them less attractive. Usually it is necessary to execute the same operations over and over again. Exactly the phenomenon were a SIMD based architecture is specialised in. GPUs, which belong to the class of processors that is also referred to as massively parallel processors, are an example of SIMD architecture based processors. With the advances in GPU technology they have become popular for general purpose computing. In addition they are relatively inexpensive in comparison to e.g. a supercomputer. These factors make GPUs a promising technology for solving the WFS reconstruction algorithm in time.

This chapter is organised as follows: Section A-1 discusses the history and development of the GPU. Section A-2 will dive into the details of the hardware architecture and the structure of the software framework. Section A-3 reasons about the applicability of GPUs by clarifying some related aspects. Section A-3-1 wraps up the chapter with an overview of the most essential GPU aspects.

A-1 Development of the GPU

Since several years GPUs have become of interest for general purpose computing. The processors graphics heritage reveals some of its strength and weaknesses. The GPU originates from the early 1980s, when three-dimensional graphics pipeline hardware was developed for specialised platforms. These expensive hardware platforms evolved to graphics accelerators in the mid-to late 1990s allowing it to be used in personal computers. In this era the graphics

hardware consisted of fixed-function pipelines which were configurable but still lacked the ability to be programmable. Another development in this decade was the increase of popularity of graphics application programming interfaces (APIs). APIs allow a programmer to use software or hardware functionality at a higher level, such that the programmer does not have to know the details of the system he or she is using. Several APIs popped up during the 1990s, e.g. OpenGL® as an open standard and DirectX^{TM} which was developed by Microsoft. Combined with the introduction of the NVIDIA GeForce 3 [20] in 2001, the GPU development took the turn towards programmability. Slowly evolving to a more general purpose platform. The first scientists noted the potential to solve their computational intensive problems. At this phase the GPU was still intended for graphics processing which made it challenging to use them for other purposes. To overcome this, a new field of research arises with the focus on how to map scientific problems to fit in the graphics processing structure. This field is referred to as general-purpose computing on graphics processing units (GPGPU) [21]. Manufactures jumped into the trend by developing tools to acquire easy access to all the resources and further improved the hardware. In addition programming language extensions helped to enhance the accessibility. Compute Unified Device Architecture (CUDA) is an example of such a programming language developed by NVIDIA. In addition under initiative of Apple, Open Computing Language (OpenCL) is under development which should be the first step towards standardisation in GPU tools. At this moment GPUs are almost true unified processors and their development is still continuing. Table A-1 shows the properties of two modern high performance GPUs: the NVIDIA Tesla C2070 [22] and the AMD FireStream 9270 [23].

Property	**NVIDIA Tesla C2070**	**AMD FireStream 9270**
Cores	448	800
Peak Performance (Single precision)	1.03 TFLOP	1.2 TFLOP
Peak Performance (Double precision)	515 GFLOP	240 GFLOP
Memory	6GB GDDR5	2GB GDDR5
Memory Interface	384-bit @ 1.5 GHz	256-bit @ 850 MHz
Memory Bandwidth	144 GB/sec	108.8 GB/s
System Interface	PCIe x16 Gen2	PCIe x16 Gen 2

Table A-1: Performance overview of the modern high performance GPUs: NVIDIA Tesla C2070 and the AMD FireStream 9270.

A-2 Architecture of the GPU

Memory is several factors slower than the processor itself, causing it to stall. Latency arises primarily by accessing the global memory. Reducing stalls by memory access results in an improvement of performance. To overcome this hardware designers use different strategies.

For CPUs it is nowadays common that they consist out of a few cores. Each CPU has it control logic and cache such that a significant part of the die area is devoted to the control logic and cache. This allows the processor to handle a wide scale of problems. In addition

the advanced control hardware minimises latency. Different control functionality makes this work, e.g pipelining, cache, branch predictors, instruction and data prefetchers. This strategy allows the multi-core CPU to support a few heavy weight threads that are each separately optimised for performance.

Where the CPU is a generalist, the GPU is a specialist. For the GPU architecture another approach was taken. There is chosen for lightweight threads with poor single-thread performance. Latency gaps are in a GPU filled up by using the vast number of available threads to fill up the gaps. As noted earlier the GPU is based on the SIMD architecture. Slightly refining this, it can be reformulated as a single instruction multiple thread (SIMT) architecture. The treads are divided by the GPU in groups, such a group of threads is called a warp (Figure A-2). Exactly the same instruction is performed for each thread in a warp. When a stall in a single thread occurs, the complete warp is temporally replaced by another warp available at the pipeline. To maximise performance, it is thus essential that the number of threads exceeds the number of available cores with several factors.

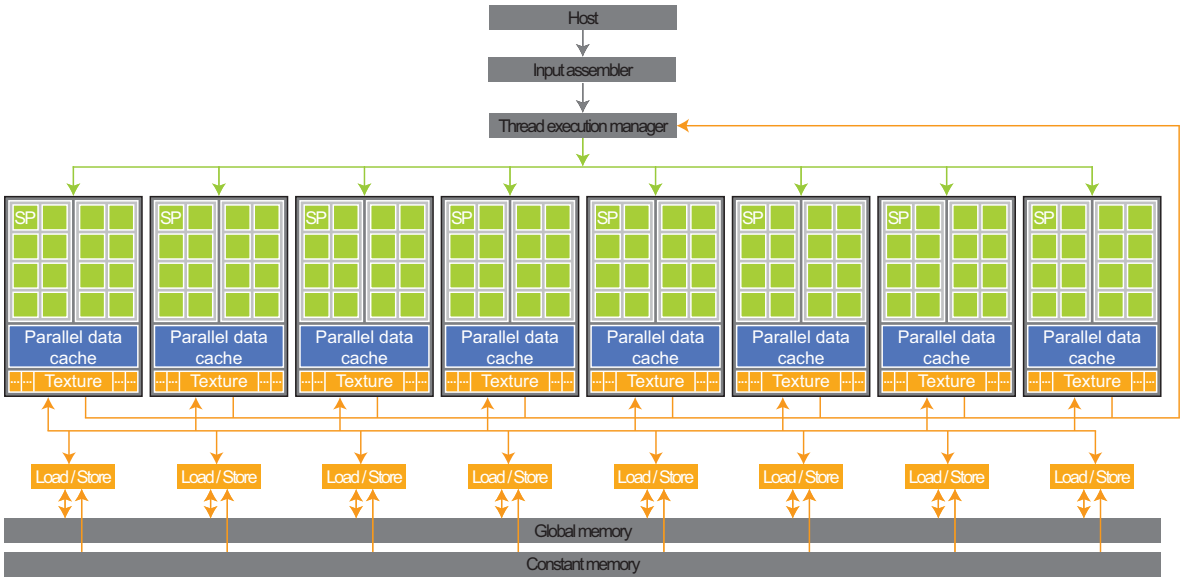


Figure A-1: Architecture overview of GPU.

As already made clear, the performance of the memory affects the overall performance. Usually the principle applies that, when the dimension of the memory is increased also the latency increases. Various aspects play a role here. The smaller the memory, the closer it can be places to the processor and the faster it can be addressed. Second is the technology used to produce the memory, that plays a role. For small memories like registers it is still cost effective to use expensive but fast memory types. While for the global memory each component per memory bit counts. Reason enough to design a memory architecture with a few levels. Combining various types of memory to be able to supply programmers both with fast and sufficient memory.

The GPU uses such a memory hierarchy as well (Figure A-1) from [24]. Close to the processing units we find the fastest memory in the form of registers. Slightly slower is the shared memory which can be seen as a small cache that is shared by a group of cores. Even at a higher level is the global and constant memory. These memories are shared by all the cores of the GPU and is by far the slowest memory. In designing parallel programs it is crucial to

keep the data as local as possible. Accessing the global memory will give a severe load on the bandwidth of the memory. Observe that when a run to the global memory is performed, all threads in a warp do this at exactly the same moment [25]. Consequently resulting in a higher latency compared to when a single processor access the memory alone.

Besides the physical architecture also some knowledge of a programming model will help understanding the behaviour of the GPU. Since there are multiple programming models we will focus on a single one, namely CUDA. Figure A-2 presents the CUDA programming model. At the host a kernel defines some parallel functionality. On the GPU side a kernel resembles a grid. Kernels are always executed sequential, making it impossible that on a GPU two separate grids exists at the same moment. The gird is subdivided into blocks. Again each block is partitioned into threads. Single blocks in a grid operates completely independently from each other, in contrast to the threads in a block. The shared memory allows the threads in a block to communicate and can cooperatively work on a subproblem. All threads and blocks are uniquely identified by an index. The index can be used to select the different data, creating uniqueness. As denoted earlier, it is for the hardware mapping required to create groups existing out of 32 threads. The threads are selected from a single block forming a warp. The warps are scheduled to be executed on the GPU.

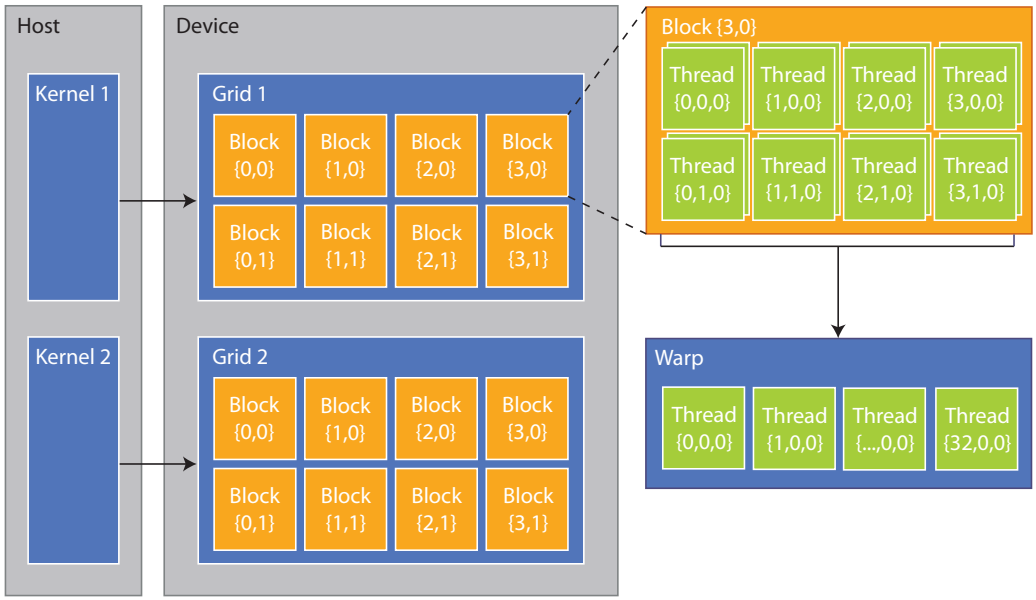


Figure A-2: Programming model of a the GPU programming language CUDA.

A-3 Applicability in practice

The development of the GPU, with respect to general purpose applications, started fairly recently. Using new technologies always brings additional risks. However they have huge potential as well. Both have the same origin. The technology is still in its infancy. Researchers and manufactures are currently doing research and development to develop the GPU itself, tools supporting the development and documentation. All based on advancing insights and user feedback [24]. In the beginning, this often results in impressive improvements in reason-

able short periods of time. When a technology is already well established these improvements are followed up in a much slower rate. All this applies also to the GPU technology: profilers and debuggers are still primitive. Programming languages and compilers are in the first, or second stage of release. The hardware itself is adapted with each new generation, to better fit the needs of general purpose computing, with for example improved double-precision support [22, 23]. Stepping in at this moment will probably require some additional effort. Yet it can mean that you will be a step ahead in the future.

The GPU can not run on its own, it needs a CPU to be controlled by. In practise this means when a CPU is executing a (sequential) program and encounters a parallel section the GPU is requested to solve it. All the data required to solve the parallel section first needs to be transferred to the GPU memory. Afterwards the GPU is able to execute the parallel statements and will, when finished, copy the result back from the GPU memory to the memory addressable by the CPU. An example of this process is shown in listing A.1 which is based on a matrix multiplication example defined in [26, p. 50], the kernel code is sited in the appendix. The impact hereof is that each so called kernel start-up takes a non ignorable time and can give a severe load on the host. The solution lies in parallizing entire sections of the algorithm reducing the number of kernel start-ups. As discussed in the parallel computing chapter this can be problematic. In addition the SIMT architecture structure involves that each thread of a kernel performs exactly the same, implicating that it must be possible to divide the algorithm in equal sub problems. Consequently the GPU approach will greatly benefit of an appropriately chosen algorithm.

```
1  void MatrixMultiplication(float* M, float* N, float* P, int Width){
2    int size = Width * Width * sizeof(float);
3    float* Md, Nd, Pd;
4
5    //Transfer M and N to device memory
6    cudaMalloc((void**) &Md, size);
7    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
8    cudaMalloc((void**) &Nd, size);
9    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
10
11   //Allocate P on the device
12   cudaMalloc((void**) &Pd, size);
13
14   //kernel invocation code
15   dim3 dimBlock(Width, Width);
16   dim3 dimGrid(1,1);
17
18   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
19
20   //Transfer P from device host and free resources
21   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
22   cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
23 }
```

Listing A.1: Example of a kernel startup process

The memory model determines that it is advantageous to keep data as local as possible. However, the size of these local memories is limited and not the only restricting factor for the amount of memory available to each thread. A kernel can exist out of more threads than

the GPU cores contains. In combination with the memory this means that it has to share the resources over the threads. Resulting in the fact that amplifying the number of threads diminishes the number of registers available to each thread. Such that balancing between the number of threads in such a way that latency can be completely hidden, but sufficient local memory is available. This is part of the design parameters that should be determined when converting the algorithm.

CUDA and OpenCL are both extensions to the programming language C. The difference between them is that CUDA is specific designed for the NVIDIA GPUs. Where OpenCL is vendor independent. A major advantage of CUDA is that it can use the full extend of the GPU. For OpenCL compromises were made, to support all the different vendor specific implementations, which resulted in a slight negative effect on the performance. For both applies that the integration with an existing programming language introduces flexibility and low-learning curves. Libraries are already widely available and there is plenty of documentation for general C. Creating a relatively easy platform to program in.

A-3-1 Overview

Consequences to the algorithm design which are imposed when using GPU technology:

- Every process is initialised and controlled by the host. Copying data to and from the GPU is a required step at each process. Resulting in significant latencies. Using the coarse grained approach in parallelizing will reduce the number of start-up procedures.
- The GPU performs best with straight-forward arithmetic, due to its small control units. Meaning that every thread executes exactly the same code. So preferably avoiding branches, especially those that diverge. Recursion is even not feasible.
- Global memory access is expensive and when possible it should be avoided. This can be achieved by either using the available local memory (shared memory and registers) or it might even be justified to do some additional arithmetic. On a higher-level it is important to recognise this in the design stage to try to lower the data dependencies and always take a coarse grained approach.
- The GPU performance benefits from several factors of more threads then available processing units to be able to hide latency. So when feasible it is preferable to split the algorithm in a higher number of threads. Note that is should not be at the expense of memory access.
- There is a fixed amount of local memory and to each thread subsection of memory is assigned. Making the size of the local memory dependent on the number of threads.

Appendix B

Parallel Computing

Engineering often consist of making tradeoffs. Parallel computing is such a tradeoff, this technology trades space for time. Traditionally algorithms are performed sequentially. For huge scientific problems this approach is often impractical. The time it would take before results are available can be enormous. Parallel computing addresses this issue and instead of executing the steps sequential the idea is to perform them simultaneously or in parallel. Especially in the case of real-time problems where time is sparse, like the AO system in telescopes, problems can benefit from parallel computing. Section B-1 points out some aspects that should be taken into consideration when using parallel computing. Section B-2 provides a case study to clarify the differences between a coarse and fine grained approach with respect to parallel computing.

B-1 Basic Concepts

When optimising algorithms for performance the gain can be expressed by the speed-up factor. The speed-up factor is defined by the relation between the algorithms original execution time and the execution time of the redesigned version.

$$S_p = \frac{E_{old}}{E_{new}} = \frac{1}{(1-F) + \frac{F}{S_{p(enhanced)}}} \tag{B-1}$$

Where S_p is the overall speed-up factor, E_{old} and E_{new} are the execution times of respectively the task without and with enhancement, F is the fraction of code that could be enhanced and $S_{p(enhanced)}$ is the speedup factor of the enhanced code. Parallel computing is a possible enhancement which speeds-up a routine by spreading the load. Ideally the speed-up factor for parallel computing would be linear, meaning that when the number of processing units increases the runtime decreases with the same factor. However, in practise this is almost never achievable. This is the consequence of the section of the algorithm that allows to be parallelized. Gene Amdalh observed this and formulated the potential speed-up factor on a

parallel computing platform in Amdalh's Law [27, p. 66]. The law states that the speed-up of a program is dictated by the section of the algorithm which cannot be parallelized. Such that the sequential part will finally determine the runtime. In case of a realistic scientific problem, which will often exist of both parallelizable and sequential parts, this has a significant influence on the runtime. The relationship defined by Amdalh's Law is formulated as

$$S_p = \frac{1}{\gamma + \frac{(1-\gamma)}{p}} \tag{B-2}$$

Where γ is the fraction of the code that is non-parallelizable and p is the number of processing units. Putting the law in perspective: if for example 50 percent of the code cannot be parallelized, the maximal achievable speed-up will be 2x regardless of the number of processors added. The effect is shown in Figure B-2, the number of processing units chosen here is 10 and the sequential portion is 50 %.

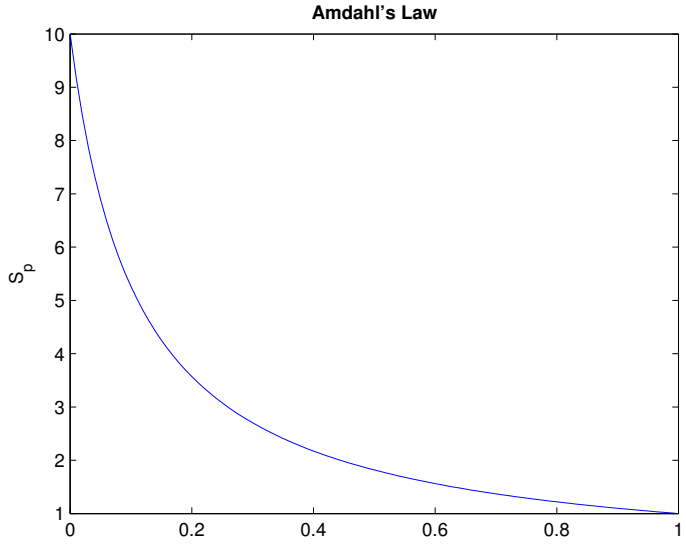


Figure B-1: Degradation of speedup by Amdahl's law. S_p is a function of (γ) the fraction of non-parallelizable code.

If and in how many sub problems a algorithm can be divided depends on data dependencies. Data dependencies refer to instructions where prior calculations are required before the instruction at hand can be evaluated. The largest set of uninterrupted dependent instructions defines the critical path. The optimisation of an algorithm by parallelizing is limited to the runtime of the critical path. An example of a human analog to the critical path was named by Fred Brooks in [28]. It states "Nine women can't make a baby in one month". Since the development of the baby is a sequential process with strong dependencies it is not feasible to divide it into sub problems. Similar for algorithms, certain sections cannot be broken up into sub problems. When there are no data dependencies we can split the algorithm. However often we cannot split it in an arbitrary number of sub problems. Referring back to our analogue, assume that we want four babies. Obviously we can split this up over at most four woman. Again the dependencies prevent us at splitting it in more sub problems.

Splitting algorithms can be performed at different levels. On an elementary operation level this is referred to as fine grained. Or on a level where a complete set of elementary operations form a sub problem, this is defined as coarse grained. Often it is more straightforward to parallelize on a fine grained level like matrix multiplications. On a fine grained level there are often less data dependencies and certainly the insight in the functioning is better. However, when feasible, it will often be better to follow the coarse grained approach. Since this reduces the amount of work performed sequential and more import it also reduces communication with the central unit. A processing unit that can progress without intervention for a longer period of time requires less communication.

Communication in the field of computing is expensive. When communicating the processing units have to use the memory to share their results. Memory access is slow compared to the processing unit, causing the processing unit to stall. Communication is also seen as overhead since the primary objective is the calculation. As a consequence the less communication, the better.

Amdahls law insinuates that the part that can be made parallel can be infinitely speed-up, just by increasing the number of processing units. In practise, at some point augmenting processing units will decrease the speed-up. In the book "The mythical man month" [28] this effect is also revealed with relation to late software development projects. Keep adding man power will not help to get the project finished in time and eventually will have a negative impact. caused by the overhead of communication, when then number of people working on a project is increasing also communication increases. At a particular point meetings will dominate the time spend on a project. Similar for parallel computing, adding processing units implies that the algorithm has to be divided in smaller sub problems. Taking a more and more fine grained approach, drastically increasing the communication. If the problem is not divided into smaller sub problems processing units will stay idle.

Load balancing is the phenomenon when the number of hardware units does not coincide with the number of sub-problems. Hardware comes frequently with a fixed number of processing units. When the algorithm is subdivided, the resulting number of sub problems does not always match the number of processing units defined by the hardware. In case of insufficient sub problems, it means that several processing units will be idle. In the opposite case when there are to much sub problems, the sub problems have to split up in two groups. First executing one group followed by the next, introducing sequential properties. In either case there is a inconsistency in comparison to the theoretical feasible upper limit defined by Amdahls law.

The hardware architecture also has impact on how an algorithm can be converted to a parallel version. Michael J. Flynn [29] was one of the first to define a classification for computation platforms. The two classes which are interesting in relation to parallel computing are the single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD). SIMD has the same control logic for different processing units, this means that it performs the same instruction on distinct data. An advantage is that less die area is required for the logic unit, making space available for more or advanced processing units. Opposite to SIMD stands MIMD, these types of hardware platforms have its own control unit for each processing unit. Allowing to perform different instructions on different data sets creating a set of completely independent processing units. Providing much more flexibility in contrast to SIMD. Both technologies have their advantages and specific application area's. For example the MIMD architecture is applied in current multi-core CPU technology. Where the SIMD architecture can be recognised in GPU hardware designs.

Section B-2 of appendix B contains a case study of a sequential algorithm that is converted to use parallel computing. Several of the discussed aspects will come back in this case study.

B-2 Parallel computing case study

To show the difference between a fine grained and a coarse grained approach with respect to parallel computing an example case is defined. The case is selected for its ability to fit for both approaches and has no direct resemblance with reality. Assume we have a square grid of 16 x 16 sensors that measures the wavefront phase shifts. These phase shifts are monitored over a period of time. A measurement from a single time instant can be represented as a matrix which is from now on referred to as ϕ, such that $\Phi \in \{\phi(1), \phi(2), ..., \phi(N)\}$. We are interested in the average phase shift over a 1 second time period, were the sample frequency is 500 Hz. The hardware used to solve the problems has 200 processing units and a SIMD architecture. A sequential algorithm that would solve the illustrated case is defined by AveragePhaseShift (version 1).

Algorithm 5 $AveragePhaseShift(\Phi)$ (version 1)

1: $A \leftarrow zeros(16, 16)$
2: **for** $k = 0$ **to** N **do**
3: $\quad A \leftarrow A + \phi(k)$
4: **end for**
5: $A \leftarrow A./N$
6: **return** A

Denote that both line 3 and line 5 consist out of matrix operations. Matrix operations result in a series of computations depending on the size of the matrix.

B-2-1 Fine grained approach

The suggested algorithm is build from two elementary operations; a matrix summation and a elementary divide. Both operations are completely independent and can easily be converted to a parallel version.

Algorithm 6 $AveragePhaseShift(\Phi)$ (version 2)

1: $A \leftarrow zeros(16, 16)$
2: **for** $k = 0$ **to** N **do**
3: $\quad << initparallel(i, j) >>$
4: $\quad A[i, j] \leftarrow A[i, j] + \phi(k)[i, j]$
5: $\quad << synchronise(A) >>$
6: **end for**
7: $<< initparallel(i, j) >>$
8: $A[i, j] \leftarrow A[i, j]/N$
9: $<< synchronise(A) >>$
10: **return** A

First clarify some elements, on line 3 and 7 we find an $<< initparallel(i, j) >>$ command. The command resembles the procedure of distributing the proceeding code over parallel processors. Similar the $<< synchronise(A) >>$ command on line 5 and 9 retrieves the results from the parallel processors and progresses sequentially. Note that each matrix element of a single elementary operation can now be calculated by a different processing unit. Obvious will be the fact that the main path of the algorithm is still sequential. Also each iteration, a parallel start-up procedure and a synchronising procedure has to be performed causing a reasonable amount of overhead for a relatively simple operation. The grid size was 16 x 16, meaning that this would result in 256 parallel processes, however the number of available processing units is hard defined by the hardware at 200. So each parallel calculation process has to be split in two batches and scheduled to be executed.

B-2-2 Coarse grained approach

In opposite to the fine grained approach, the algorithm is converted to a parallel version at top-level. The algorithm is an example of the type of algorithms that allows to be completely parallelized and promises to have a linear speed-up factor. There are two options, parallelize temporally or spatially. Since there are data dependency constraints in the time domain this could be difficult. In contrary in the space domain each of the operations on the elements of the matrix are completely independent and can be computed separately.

Algorithm 7 $AveragePhaseShift(\Phi)$ (version 3)

1: $<< initparallel(i, j) >>$
2: $A[i, j] \leftarrow 0$
3: **for** $k = 0$ **to** N **do**
4: $\quad A[i, j] \leftarrow A[i, j] + \phi(k)[i, j]$
5: **end for**
6: $A[i, j] \leftarrow A[i, j]/N$
7: $<< synchronise(A) >>$
8: **return** A

Clearly the communication has been drastically reduced in comparison to the fine grained approach. While the load balancing issue is still there and cannot be resolved by adapting the software alone. In the sketched case study, the coarse grained approach would favour the fine grained approach. This is mainly caused due to the fact that all the operations performed are independent over the different matrix elements.

Appendix C

Pseudocode for covariance matrix

In this appendix we will provide pseudocode for the prediction error covariance matrix from which a mathematical definition is given by Chapter 3. To be able to compute a static covariance matrix it is assumed that the covariance matrix will become invariant after several iterations.

Algorithmic 8 shows an iterative covariance matrix solver for a set of local models. Line 1 defines the stopping criteria, which is satisfied when the difference between $\bar{P}(k)$ and $\bar{P}(k-1)$ is smaller the a small value ϵ. The stopping criteria could be extended to contain a maximum number of iterations to overcome non convergence. Line 2 and 3 in combination with line 17 and 18 loops through the local models in a 2D way. Line 5 till 7 initialises at the first time instance the lifted covariance matrix. Where *diag* stands for the block diagonal operator such that the lifted covariance matrix is defined as a block diagonal matrix containing a set of local covariance matrices. Line 8 till 11 defines all lifted matrices required to compute the lifted covariance matrix for the next iteration. Line 12 and 13 compute the updated lifted covariance matrix. Line 14 and 15 compute the local covariance matrix and Kalman gain on the just derived lifted covariance matrix.

Algorithm 8 Covariance Matrix Solver

while $\max(\bar{P}(k) - \bar{P}(k-1)) \geq \epsilon$ **do**
  for $i = 1$ to N_x **do**
    for $j = 1$ to N_y **do**

      if k==0 **then**
        $\bar{P}^l_{(i,j)}(0) = diag\{\mathcal{N}_{i-1,j-1}.P, \ldots, \mathcal{N}_{i+1,j+1}.P\}$
      end if

      $\bar{A} = diag\{\mathcal{N}_{i-1,j-1}.A, \ldots, \mathcal{N}_{i+1,j+1}.A\}$, $\bar{K_y} = diag\{\mathcal{N}_{i-1,j-1}.K_y, \ldots, \mathcal{N}_{i+1,j+1}.K_y\}$
      $\bar{C} = diag\{\mathcal{N}_{i-1,j-1}.C, \ldots, \mathcal{N}_{i+1,j+1}.C\}$, $\bar{S} = diag\{\mathcal{N}_{i-1,j-1}.S, \ldots, \mathcal{N}_{i+1,j+1}.S\}$
      $\bar{Q} = diag\{\mathcal{N}_{i-1,j-1}.Q, \ldots, \mathcal{N}_{i+1,j+1}.Q\}$, $\bar{R} = diag\{\mathcal{N}_{i-1,j-1}.R, \ldots, \mathcal{N}_{i+1,j+1}.R\}$
      $W = [c_{(i,j),(i-1,j-1)}I, \ldots, c_{(i,j),(i+1,j+1)}I]$

      $\bar{P}(k+1) = \bar{A}\bar{P}(k)\bar{A}^T - (\bar{A}\bar{P}(k)\bar{C}^T + \bar{S})\bar{K_y}^T - \bar{K_y}(\bar{S}^T + \bar{C}\bar{P}(k)\bar{A}^T)$
        $+ \bar{K_y}(\bar{C}\bar{P}(k)\bar{C}^T + \bar{R})\bar{K_y}^T + \bar{Q}$
      $P(k+1) = W\bar{P}(k+1)W^T$
      $Ky = (S + AP(k+1)C^T)(R + CP(k+1)C^T)^{-1}$
      $k \leftarrow k+1$
    end for
  end for
end while

Appendix D

Sequentially Semi-separable Matrices

The sequentially semi separable (SSS) matrix approach described by Justin K. Rice in [30] to design distribute controllers is used by Rufus Fraanje in [13] for wavefront reconstruction. As parallel computing is required for solving the wavefront reconstruction problem in relation to a European extremely large telescope (E-ELT), the question is raised if this approach is intrinsically parallel. Section D-1 demonstrated this is not the case for the general SSS structure. Section D-2 introduces delays to overcome the problems encountered by the general SSS structure.

D-1 General SSS structure

Distributed is different from parallel by definition. In both cases sub problems are executed on different devices, however parallel imposes the additional property of being simultaneously executed, which is different from distributed where this is not required. Let us consider an general SSS structure. Figure D-1 shows a standard distributed configuration of a string of subsystems resulting from a general SSS structure.

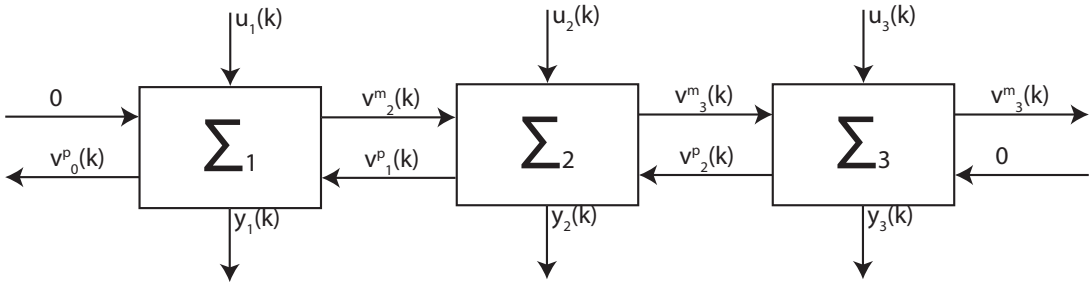


Figure D-1: Subsystem string interconnection.

The discrete time subsystems can be modelled as Equation (D-1).

$$\begin{bmatrix} x_s(k+1) \\ v^p_{s-1}(k) \\ v^m_{s+1}(k) \\ y_s(k) \end{bmatrix} = \begin{bmatrix} A_s & B^p_s & B^m_s & B_s \\ C^p_s & W^p_s & 0 & L^p_s \\ C^m_s & 0 & W^m_s & L^m_s \\ C_s & H^p_s & H^m_s & D_s \end{bmatrix} \begin{bmatrix} x_s(k) \\ v^p_s(k) \\ v^m_s(k) \\ u_s(k) \end{bmatrix} \tag{D-1}$$

To be able to compute the subsystems in parallel there should be no dependency between the output $y_i(k)$ and the interconnection variables $v^p_i(k)$ $v^m_i(k)$ that themselves depends on prior interconnection variables $v^p_{i+1}(k)$ or $v^m_{i-1}(k)$. To present that such a dependency exists equations (D-2) till (D-4) are used. Equation (D-2) starts with calculating the first output $y_1(k)$.

$$y_1(k) = C_1 x_1(k) + H^p_1 v^p_1(k) + H^m_1 v^m_1(k) + D_1 u_1(k) \tag{D-2}$$

We can see $y_1(k)$ depends on four variables $x_1(k)$, $v^p_1(k)$, $v^m_1(k)$ and $u_1(k)$. For showing the dependency we will use $v^p_1(k)$ leading us to Equation (D-3).

$$v^p_1(k) = C^p_2 x_2(k) + W^p_2 v^p_2(k) + L^p_2 u_2(k) \tag{D-3}$$

Here the dependency starts to show up, because $v^p_1(k)$ depends on $v^p_2(k)$. Progressing one step further in Equation (D-4) shows that again there is the dependency. Allowing us to conclude that the standard SSS structure is not parallel on a subsystem level.

$$v^p_2(k) = C^p_3 x_3(k) + W^p_3 v^p_3(k) + L^p_3 u_3(k) \tag{D-4}$$

D-2 SSS with introduced delay

In some cases it is possible to introduce delay to the interconnection lines as depicted by Figure D-2. This enables a parallel implementation of the SSS structure parallel.

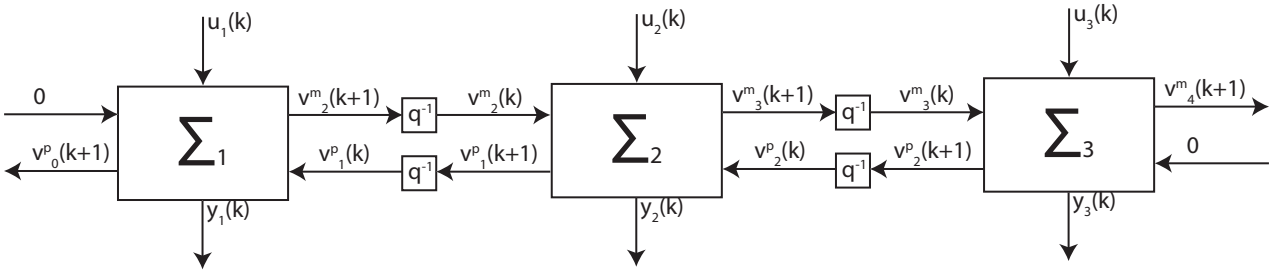


Figure D-2: Subsystem string interconnection with delay in interconnection lines.

Equations (D-5) and (D-6) now describe the discrete subsystem models with introduced delay.

$$\begin{bmatrix} x_s(k+1) \\ v^p_{s-1}(k+1) \\ v^m_{s+1}(k+1) \\ y_s(k) \end{bmatrix} = \begin{bmatrix} A_s & B^p_s & B^m_s & B_s \\ C^p_s & W^p_s & 0 & L^p_s \\ C^m_s & 0 & W^m_s & L^m_s \\ C_s & H^p_s & H^m_s & D_s \end{bmatrix} \begin{bmatrix} x_s(k) \\ v^p_s(k) \\ v^m_s(k) \\ u_s(k) \end{bmatrix} \tag{D-5}$$

$$\begin{bmatrix} v_s^p(k) \\ v_s^m(k) \end{bmatrix} = \begin{bmatrix} q^{-1}I & 0 \\ 0 & q^{-1}I \end{bmatrix} \begin{bmatrix} v_s^p(k+1) \\ v_s^m(k+1) \end{bmatrix} \tag{D-6}$$

These can be rewritten to the following Equation.

$$\left[\begin{array}{c} x_s(k+1) \\ v_{s-1}^p(k+1) \\ v_{s+1}^m(k+1) \\ \hline v_{s-1}^p(k) \\ v_{s+1}^m(k) \\ \hline y_s(k) \end{array}\right] = \left[\begin{array}{ccc|cc|c} A_s & 0 & 0 & B_s^p & B_s^m & B_s \\ C_s^p & 0 & 0 & W_s^p & 0 & L_s^p \\ C_s^m & 0 & 0 & 0 & W_s^m & L_s^m \\ \hline 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 \\ \hline C_s & 0 & 0 & H_s^p & H_s^m & D_s \end{array}\right] \left[\begin{array}{c} x_s(k+1) \\ v_{s-1}^p(k) \\ v_{s+1}^m(k) \\ \hline v_s^p(k) \\ v_s^m(k) \\ \hline u_s(k) \end{array}\right] \tag{D-7}$$

Similar to the SSS structure without delay we can now demonstrated that it is possible to implement the system in parallel. As now the interconnection variables depend on the results off the previous iteration. See Equation (D-8) till (D-10).

$$y_1(k) = C_1 x_1(k) + H_1^p v_1^p(k) + H_1^m v_1^m(k) + D_1 u_1(k) \tag{D-8}$$

$$v_1^p(k) = v_1^p(k) \tag{D-9}$$

$$v_1^p(k) = C_2^p x_2(k-1) + W_2^p v_2^p(k-1) + L_2^p u_2(k-1) \tag{D-10}$$

Appendix E

Coding

For the implementation of the wavefront reconstructor the coding was performed partially in CUDA. As already mentioned before CUDA is a extension on C. With CUDA it is possible to use the GPUs of Nvidia for general purpose computing. Furthermore all code is executed through Matlab such that an interface is required between the high level Matlab code and the low level C code. C-MEX offers such an interface and is extensively used throughout the code implementation. To give some insight in the implementation peaces of the code shall be depicted in this chapter.

E-1 GPU kernel

In Chapter A a reference is made to a matrix multiplication defined in [26, p. 50]. To be comprehensive also the kernel code is given by listing E-1. Besides this rather straight forward implementation the book also gives some examples of more optimised versions.

```
1  __global__ void MatrixMuKernel(float* Md, float* Nd, float* Pd, int Width){
2    // 2D Thread ID
3    int tx = threadIdx.x;
4    int ty = threadIdx.y;
5
6    // Pvalue stores the Pd element that is coomputed by the thread.
7    float Pvalue = 0;
8
9    for(int k=0; k< Width; k++){
10     float Mdelement = Md[ty * Width + k];
11     float Ndelement = Nd[k * Width + tx];
12
13     Pvalue += Mdelement * Ndelement;
14   }
15
16   // Write the matrix to device memory, each thread writes one element
17   Pd[ty * Width + tx] = Pvalue;
18 }
```

Listing E.1: Kernel from a matrix multiplication implemented on GPU.

E-2 Reconstructor implementation

E-2-1 Sequential reconstructor

Listing E-2-1 shows the code off the sequential reconstructor designed in chapter 2, which was executed on a CPU based platform. Let us point out the most crucial elements. The mex function on line 90 has a specific structure that is induced by the C-MEX interface between Matlab and C. In addition in this reconstructor all calculations are performed in dual precision since the data is delivered in dual precision by Matlab. Further we see on Line 107 till 123 the actual reconstruction process iterating through time. Where the reconstruction function on line 121 solves a reconstruction for a single time step.

```
1  /* incudes and defines*/
2  ...
3
4  void reconstruction(int Nx, int Ny, const mxArray * sys,  mxArray *Sx, mxArray
       * Sy, mxArray * phi, mxArray * x){
5    /* declarations */
6    ...
7
8      /* retrieve dimensions*/
9      ...
10
11     /* allocate memory*/
12     ...
13
14     result = mxGetPr(phi);
15   for(n=0; n<Ny; n++){
16     for(m=0; m<Nx; m++){
17              /*retrieve system pointer*/
18              subsSys[0] = m;
19              subsSys[1] = n;
20              id = mxCalcSingleSubscript(sys, nodSys, subsSys);
21              sys_mn = mxGetCell(sys, id);
22
23              /*retrieve system matrices*/
24              ...
25
26              /*retrieve state pointer and initialize if nessesary*/
27              subsX[0] = m;
28              subsX[1] = n;
29              id = mxCalcSingleSubscript(x, nodX, subsX);
30              x_mn = mxGetCell(x, id);
31
32              /*Retrieve model order */
33              nodAnn = mxGetNumberOfDimensions(Ann);
34              dAnn = mxCalloc(nodAnn,sizeof(mwSize));
35              dAnn = mxGetDimensions(Ann);
36
```

```
37              if(x_mn == NULL){
38                  x_mn = mxCreateDoubleMatrix(dAnn[1], 1, mxREAL);
39                  mxSetCell(x, id, x_mn);
40              }
41
42              /*Determine required gradients */
43              off[0] = -1*(m > 0);
44              off[1] = (m < (Nx-1))-1;
45              off[2] =  -1*(n > 0);
46              off[3] = (n < (Ny-1))-1;
47
48              /*Allocate measurments memory */
49              size_sx = abs(off[0]) + off[1] + 1;
50              size_sy = abs(off[2]) + off[3] + 1;
51              y=mxCreateDoubleMatrix(size_sx + size_sy, 1, mxREAL);
52
53              /*Data collection from Sx and Sy */
54              Sx_mn = mxGetPr(y);
55              Sy_mn = mxGetPr(y) + size_sx;
56
57              subsSx[0] = m+off[0];
58              subsSx[1] = n;
59              index = mxCalcSingleSubscript(Sx, nodSx, subsSx);
60              memcpy(Sx_mn, mxGetPr(Sx)+index, size_sx*mxGetElementSize(Sx));
61
62              subsSy[0] = m;
63              subsSy[1] = n+off[2];
64              for(i=0;i<size_sy;i++){
65                  subsSy[1] = n+off[2]+i;
66                  index = mxCalcSingleSubscript(Sy, nodSy, subsSy);
67                  memcpy(Sy_mn+i, mxGetPr(Sy)+index, mxGetElementSize(Sy));
68              }
69
70              /* Perform a local reconstruction */
71              t1 = matrix_mult(Knn, y);
72              t2 = matrix_mult(Ann, x_mn);
73              t3 = matrix_add(t1, t2);
74              memcpy(mxGetPr(x_mn), mxGetPr(t3), dAnn[1]*mxGetElementSize(x_mn));
75              mxDestroyArray(t1);
76              mxDestroyArray(t2);
77              mxDestroyArray(t3);
78
79              t1 = matrix_mult(Cphi, x_mn);
80              memcpy(result+(n*Nx)+m, mxGetPr(t1), mxGetElementSize(t1));
81              mxDestroyArray(t1);
82              mxDestroyArray(y);
83          }
84      }
85
86      /* Free some memory and return*/
87      ...
88  }
89
90  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
91      /* Declarations */
92      ...
93
```

```
94      /* Check for proper and number of arguments */
95    ...
96
97    /* rename pointers for readabillity */
98    data = prhs[DATA_ID];
99    sys = prhs[SYS_ID];
100
101   /* Initialize output data and states */
102   phi_est = mxCreateCellMatrix(1, Nt);
103   x = mxCreateCellMatrix(Nx, Ny);
104   dim = mxGetNumberOfDimensions(data);
105   subs=mxCalloc(dim,sizeof(mwIndex));
106
107   for(k=0; k<Nt; k++){
108     /* Allocate some memory for result */
109     phi = mxCreateDoubleMatrix(Nx, Ny,mxREAL);
110
111     /* retrieve the data for the current timestep */
112     subs[0] = 0;
113     subs[1] = k;
114     id = mxCalcSingleSubscript(data, dim, subs);
115     Sx = mxGetCell(data, id);
116     subs[0] = 1;
117     id = mxCalcSingleSubscript(data, dim, subs);
118     Sy = mxGetCell(data, id);
119
120     /* Reconstruct the wavefront based on the data; */
121     reconstruction(Nx, Ny, sys, Sx, Sy, phi, x);
122     mxSetCell(phi_est, k, phi);
123   }
124   /* set the output to contain the phase estimates; */
125   plhs[0] = phi_est;
126
127   /* clear the memory and return*/
128   mxFree(subs);
129     return;
130 }
```

Listing E.2: C-MEX file that performs a sequential reconstruction based on local models.

E-2-2 Parallel reconstructor using CUSPARSE

Listing E-2-2 represents a parallel implementation of the wavefront reconstructor designed in chapter 2. The parallel partition is performed on a fine grained level based on a sparse matrix vector multiplication. For this the reconstruction problem has been mapped to fit this structure in chapter 4. Te mapping allowed the use of an of the shelf library called CUSPARSE. Line 27 till 35 initialises the library to use the correct sparse format. On line 109 till 112 the CUSPARSE library is really applied to solve a single matrix vector multiplication.

```
1  /* incudes and defines*/
2  ...
3
4  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
5    /*sparse library declarations */
6    cusparseHandle_t handle;
```

```
7   cusparseMatDescr_t desc;
8
9   /* Declarations */
10  ...
11
12    /* Check for proper and number of arguments */
13  ...
14
15  /*timer declarartions*/
16  ...
17
18  /// rename pointers for readabillity
19  const mxArray *data = prhs[DATA_ID];
20  const mxArray *sys = prhs[SYS_ID];
21
22  //Initialize output data and states
23  mxArray *phi_est = mxCreateCellMatrix(1, Nt);
24  mwSize dim = mxGetNumberOfDimensions(data);
25  mwIndex *subs=(mwIndex*) mxCalloc(dim,sizeof(mwIndex));
26
27  //initialize Sparse library handle
28  cusparseStatus_t stat_1 = cusparseCreate(&handle);
29  cusparseStatus_t stat_2 = cusparseCreateMatDescr(&desc);
30  cusparseStatus_t stat_3 = cusparseSetMatType(desc,
        CUSPARSE_MATRIX_TYPE_GENERAL);
31  cusparseStatus_t stat_4 = cusparseSetMatIndexBase(desc,
        CUSPARSE_INDEX_BASE_ZERO);
32
33  if(stat_1 != CUSPARSE_STATUS_SUCCESS || stat_2 != CUSPARSE_STATUS_SUCCESS ||
        stat_2 != CUSPARSE_STATUS_SUCCESS || stat_2 != CUSPARSE_STATUS_SUCCESS){
34    mexErrMsgTxt("Initialization error cusparse library");
35  }
36
37  // model matix and state vector
38  int size_sx = (Nx-1)*Ny;
39  int size_sy = Nx*(Ny-1);
40  int size_y = Nx*Ny;
41  int size_x = 0;;
42  coo hSysMatrix;
43  coo cuSysMatrixCoo;
44  csr cuSysMatrixCsr;
45  cuArray hStateVector;
46  cuArray cuStateVector;
47  cuArray hOutputVector;
48  cuArray cuOutputVector;
49
50  if(mxGetNumberOfDimensions(sys) == 2){
51    int m = mxGetM(sys);
52    int n = mxGetN(sys);
53    size_x = n - (size_sx+size_sy);
54
55    int sys_size = mxGetNzmax(sys);
56    hCreateSparseCooMatrix(&hSysMatrix, sys_size);
57    cuCreateSparseCooMatrix(&cuSysMatrixCoo, sys_size);
58    cuCreateSparseCsrMatrix(&cuSysMatrixCsr, sys_size, m);
59
60    hCreateFloatMatrix(&hStateVector, n, 1, false, 0);
```

```
61      cuCreateFloatMatrix(&cuStateVector, n, 1);
62
63      hCreateFloatMatrix(&hOutputVector, m, 1, false, 0);
64      cuCreateFloatMatrix(&cuOutputVector, m, 1);
65
66      /* Convert sparse matrix format and store to GPU*/
67      matlab2coo(sys, &hSysMatrix);
68      cudaMemcpy(cuSysMatrixCoo.Ir, hSysMatrix.Ir, sys_size*sizeof(int),
            cudaMemcpyHostToDevice);
69      cudaMemcpy(cuSysMatrixCoo.Ic, hSysMatrix.Ic, sys_size*sizeof(int),
            cudaMemcpyHostToDevice);
70      cudaMemcpy(cuSysMatrixCoo.data, hSysMatrix.data, sys_size*sizeof(float),
            cudaMemcpyHostToDevice);
71
72      if(cusparseXcoo2csr(handle, cuSysMatrixCoo.Ir, cuSysMatrixCoo.n, m,
            cuSysMatrixCsr.Ir, CUSPARSE_INDEX_BASE_ZERO) != CUSPARSE_STATUS_SUCCESS)
            {
73        mexErrMsgTxt("Conversion error in sparse formats");
74      }
75
76      cudaMemcpy(cuStateVector.data, hStateVector.data, (size_x+size_sx+size_sy)*
            sizeof(float), cudaMemcpyHostToDevice); //Clear the state vector
77      cudaMemcpy(cuOutputVector.data, hOutputVector.data, (size_x+size_y)*sizeof(
            float), cudaMemcpyHostToDevice); //Clear the output vector
78    }else{
79      mexErrMsgTxt("The system matrix is of a higer dimension than 2");
80    }
81
82    stop_timing(ti_init);
83    start_timing(ti_reconstruct);
84    for(int k=0; k<Nt; k++){
85      resume_timing(ti_load);
86
87      //Allocate some memory for result
88      mxArray *phi = mxCreateDoubleMatrix(Nx, Ny,mxREAL);
89
90      //retrieve the data for the current timestep
91      subs[0] = 0;
92      subs[1] = k;
93      int id = mxCalcSingleSubscript(data, dim, subs);
94      mxArray *Sx = mxGetCell(data, id);
95      subs[0] = 1;
96      id = mxCalcSingleSubscript(data, dim, subs);
97      mxArray *Sy = mxGetCell(data, id);
98
99      /* Load measurments and copy to the GPU*/
100     loadMeasurements(Sx, &hStateVector, size_x);
101     loadMeasurements(Sy, &hStateVector, size_x + size_sx);
102     cudaMemcpy(cuStateVector.data, hOutputVector.data, size_x*sizeof(float),
            cudaMemcpyHostToDevice);
103     cudaMemcpy(cuStateVector.data + size_x, hStateVector.data + size_x, (
            size_sx+size_sy)*sizeof(float), cudaMemcpyHostToDevice);
104
105     stop_timing(ti_load);
106     resume_timing(ti_calc);
107
108     //sparse calculation using sparse library
```

```
109     stat_1 = cusparseScsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, mxGetM(
            sys), mxGetN(sys), 1.0, desc, cuSysMatrixCoo.data, cuSysMatrixCsr.Ir,
            cuSysMatrixCoo.Ic, &cuStateVector.data[0], 0.0, &cuOutputVector.data[0])
            ;
110     if(stat_1 != CUSPARSE_STATUS_SUCCESS) {
111       mexErrMsgTxt("Sparse matrix vector calculation error");
112     }
113     cudaThreadSynchronize();
114
115     stop_timing(ti_calc);
116     resume_timing(ti_store);
117
118     /* Copy result ack to host */
119     cudaMemcpy(hOutputVector.data, cuOutputVector.data, (size_x+size_y)*sizeof(
            float), cudaMemcpyDeviceToHost);
120     storeOutput(&hOutputVector, phi, size_x, size_y);
121
122     //Convert the data and store it
123     mxSetCell(phi_est, k, phi);
124     stop_timing(ti_store);
125   }
126   stop_timing(ti_reconstruct);
127   start_timing(ti_destroy);
128
129   //destroy sparse library handle
130   cusparseDestroy(handle);
131   cusparseDestroyMatDescr(desc);
132
133   //clear the memory
134   ...
135
136   //timer display
137   ...
138
139   //set the output to contain the phase estimates;
140   ...
141 }
```

Listing E.3: C-MEX file that performs a parallel reconstruction based on a sparse matrix vector multiplication.

E-2-3 Parallel reconstructor based on model partitioned method

Listing E-2-3 and E-2-3 represent a parallel implementation which is based on a model based partitioning. The model based partitioning offers a far more coarse grained approach compared to the sparse matrix multiplication. In this case a custom kernel was build, Listing E-2-3. Also some different format conversions are required compared to Listing E-2-2, beyond that the structure is rather similar.

```
1 /* incudes and defines*/
2 ...
3
4 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
5   /* Declarations */
6   ...
```

```
7
8     /* Check for proper and number of arguments */
9   ...
10
11  /*timer declarartions*/
12  ...
13
14  /* rename pointers for readabillity */
15  const mxArray *data = prhs[DATA_ID];
16  const mxArray *sys = prhs[SYS_ID];
17
18  /* Initialize output data and states */
19  mxArray *phi_est = mxCreateCellMatrix(1, Nt);
20  mwSize dim = mxGetNumberOfDimensions(data);
21  mwIndex *subs=(mwIndex*) mxCalloc(dim,sizeof(mwIndex));
22
23  /* initialize GPU (copy models and create state vectors)*/
24  dim3 dimBlock(Nx, 1);
25  dim3 dimGrid(Ny, 1);
26  size_t sx = 2*MAX_MODEL_ORDER*Nx*sizeof(float);
27
28  cuCell x;
29  cuModel models;
30  cuCreateCellMatrix(&x, Nx, Ny);
31  cuCreateModelMatrix(&models, Nx, Ny);
32  cuLoadModels(sys, &models, &x);
33
34  /*initialize imput and output matrices */
35  cuArray cuSx, cuSy, cuPhi, hSx, hSy, hPhi;
36  cuCreateFloatMatrix(&cuSx, (Nx-1), Ny);
37  cuCreateFloatMatrix(&cuSy, Nx, (Ny-1));
38  cuCreateFloatMatrix(&cuPhi, Nx, Ny);
39
40  hCreateFloatMatrix(&hSx, (Nx-1), Ny, false,0);
41  hCreateFloatMatrix(&hSy, Nx, (Ny-1), false,0);
42  hCreateFloatMatrix(&hPhi, Nx, Ny, false,0);
43
44  stop_timing(ti_init);
45  start_timing(ti_reconstruct);
46  for(int k=0; k<Nt; k++){
47    resume_timing(ti_load);
48
49    /* Allocate some memory for result */
50    mxArray *phi = mxCreateDoubleMatrix(Nx, Ny,mxREAL);
51
52    /* retrieve the data for the current timestep */
53    subs[0] = 0;
54    subs[1] = k;
55    int id = mxCalcSingleSubscript(data, dim, subs);
56    mxArray *Sx = mxGetCell(data, id);
57    subs[0] = 1;
58    id = mxCalcSingleSubscript(data, dim, subs);
59    mxArray *Sy = mxGetCell(data, id);
60
61    /* Convert the data to single precision */
62    DP2SP(Sx, &hSx);
63    DP2SP(Sy, &hSy);
```

```
64
65      /* Copy data to GPU */
66      cudaMemcpy(cuSx.data, hSx.data, (Nx-1)*Ny*sizeof(float),
            cudaMemcpyHostToDevice);
67      cudaMemcpy(cuSy.data, hSy.data, Nx*(Ny-1)*sizeof(float),
            cudaMemcpyHostToDevice);
68
69      stop_timing(ti_load);
70      resume_timing(ti_calc);
71
72      /* GPU kernel call */
73      process <<<dimGrid, dimBlock, sx>>> (cuSx, cuSy, cuPhi, models, x);
74      cudaThreadSynchronize();
75
76      stop_timing(ti_calc);
77      resume_timing(ti_store);
78
79      /* Copy result back to host*/
80      cudaMemcpy(hPhi.data, cuPhi.data, Nx*Ny*sizeof(float),
            cudaMemcpyDeviceToHost);
81
82      /* Convert the data and store it */
83      SP2DP(&hPhi, phi);
84      mxSetCell(phi_est, k, phi);
85      stop_timing(ti_store);
86
87    }
88    stop_timing(ti_reconstruct);
89    start_timing(ti_destroy);
90
91    /* destroy GPU data */
92    ...
93
94    /* clear the host memory */
95    ...
96
97    /*timer stop*/
98    ...
99
100   /* set the output to contain the phase estimates; */
101   ...
102 }
```

Listing E.4: C-MEX file that performs a parallel reconstruction based on a model based partitioning.

```
1  /* incudes and defines*/
2  ...
3
4  __global__ void process(cuArray Sx, cuArray Sy, cuArray phi, cuModel models,
       cuCell x){
5    int m = threadIdx.x;
6    int n = blockIdx.x;
7    int Nx = blockDim.x;
8    int id = Nx*n + m;
9    int order = models.order[id];
10   int off_x = x.offset[id];
11
```

```
12    /*init shared state vectors and load states into shared memory*/
13    extern __shared__ float sm_x[];
14    for(int i=0; i<order;i++){
15      sm_x[MAX_MODEL_ORDER*m + i] = x.data[off_x + i ];
16    }
17
18    /* compute the state estimate and the output update*/
19    cuStateUpdate(&models, &Sx, &Sy, id, sm_x, order);
20    cuOutputUpdate(&models, &x, &phi, id, sm_x, order);
21  }
22
23  __device__ void cuStateUpdate(cuModel *models, cuArray *Sx, cuArray * Sy, int
      id, float * sm_x, int order){
24    int index = 0;
25    int m = threadIdx.x;
26    int n = blockIdx.x;
27    int Ny = gridDim.x;
28    int Nx = blockDim.x;
29
30    int off_a = models->Ann_offset[id];
31    int off_k = models->Knn_offset[id];
32
33    /*Determine required gradients */
34          int off_0 = -1*(m > 0);
35          int off_1 = (m < (Nx-1))-1;
36          int off_2 =  -1*(n > 0);
37          int off_3 = (n < (Ny-1))-1;
38
39    /*Size measurments memory */
40    int size_sx = abs(off_0) + off_1 + 1;
41    int size_sy = abs(off_2) + off_3 + 1;
42    int nin = models->nin[id];
43
44    /* Perfom matix vector multiplication and vecor addition in the form x(k+1|k)
          = (A-KC)x(k) + Ky(k) */
45    for(int j = 0; j < order;j++){
46      float result = 0;
47
48      for(int i = 0; i < order;i++){
49        index = order*i + j;
50        result += models->Ann[off_a + index]*sm_x[MAX_MODEL_ORDER*m + i];
51      }
52
53      for(int i = 0; i < size_sx;i++){
54        index = order*i + j;
55        result += models->Knn[off_k + index]*Sx->data[n*(Nx-1) + m+off_0 + i];
56      }
57
58      for(int i = size_sx; i < nin;i++){
59        index = order*i + j;
60        result += models->Knn[off_k + index]*Sy->data[(n+off_2 + i - size_sx)*Nx
              + m];
61      }
62      sm_x[MAX_MODEL_ORDER*Nx + MAX_MODEL_ORDER*m + j] = result;
63    }
64
65  }
```

```
66
67 __device__ void cuOutputUpdate(cuModel *models, cuCell *x, cuArray * phi, int
       id, float * sm_x, int order){
68   int off_x = x->offset[id];
69   int off_c = models->Cphi_offset[id];
70   int n = blockIdx.x;
71   int m = threadIdx.x;
72   int Nx = blockDim.x;
73
74   /* Perform matrix vector mutiplication y(k)= Cx(k+1|k) */
75   float result = 0;
76   for(int i = 0; i < order;i++){
77     result += models->Cphi[off_c + i]*sm_x[MAX_MODEL_ORDER*Nx + MAX_MODEL_ORDER
         *m + i];
78     x->data[off_x + i] = sm_x[MAX_MODEL_ORDER*Nx + MAX_MODEL_ORDER*m + i];
79   }
80
81   phi->data[id] = result;
82 }
83
84 /* Load model method */
85 ...
```

Listing E.5: Kernel methods for parallel reconstruction on a GPU.

Bibliography

[1] F. S. Cattivelli and A. H. Sayed, "Diffusion strategies for distributed kalman filtering and smoothing," *IEEE Transactions on automatic control*, September 2010.

[2] R. N. Wilson, *Reflecting Telescope Optics I.* Germany: Springer-Verlag Berlin Heidelberg, 1996.

[3] K. J. Hinnen, *Data-Driven Optimal Control for Adaptive Optics.* Dutch Institute of Systems and Control, 1st ed., 2006.

[4] E. Fedrigo and R. Donaldson, "Sparta roadmap and future challenges," *Adaptive Optics Systems II*, vol. 7736, no. 1, p. 77364O, 2010.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann, 4th ed., 2007.

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, 1965.

[7] W. Klop, *E-ELT under control: Adaptive optics low-level control solution for E-ELT.* Delft: TU Delft, 2010.

[8] T. N. Truong, A. H. Bouchez, R. G. Dekany, S. R. Guiwits, J. E. Roberts, and M. Troy, "Real-time wavefront control for the palm-3000 high order adaptive optics system," *Proceedings of the SPIE*, vol. 7015, 2008.

[9] G. Hovey, R. Conan, F. Gamache, G. Herriot, Z. Ljusic, D. Quinn, M. Smith, J. Veran, and H. Zhang, "An fpga based computing platform for adaptive optics control," *1st AO4ELT conference*, 2010.

[10] L. A. Poyneer, D. T. Gavel, and J. M. Brase, "Fast wave-front reconstruction in large adaptive optics systems with use of the fourier transform," *Optical Society of America*, vol. 19, October 2002.

[11] B. L. Ellerbroek, "Efficient computation of minimum-variance wave-front reconstructors with sparse matrix techniques," *Optical Society of America*, vol. 19, September 2002.

[12] R. Fraanje and N. Doelman, "Modelling and prediction of turbulence-induced wavefront distortions," *Adaptive Optics Systems II*, July 2010.

[13] R. Fraanje, J. Rice, M. Verhaegen, and N. Doelman, "Fast reconstruction and prediction of frozen flow turbulence based on structured kalman filtering," *Optical Society of America*, 2010.

[14] R. Conan, *Modélisation des effects de l'échelle externe de cohérence spatiale du front d'onde pour l'observation á Haure Résolution Angulaire en Astronomie.* Université de Nice-Sophia Antipolis, 1st ed., 2000.

[15] A. Beghi, A. Cenedese, and A. Masiero, "Stochastic realization approach to the efficient simulation of phase screens," *Optical Society of America*, February 2008.

[16] P. van Overschee and B. de Moor, "Subspace algorithms for the stochastic identification problem.," *Automatica*, vol. 29, no. 3, pp. 649–660, 1993.

[17] M. Verhaegen and V. Verdult, *Filtering and System Identification.* Cambridge University, 1st ed., 2007.

[18] H. R. Hashemipour, S. Roy, and A. J. Laub, "Decentralized structures for parallel kalman filtering," *IEEE Transactions on automatic control*, January 1988.

[19] R. Olfati-Saber, ed., *Distributed Kalman Filter with Embedded Consensus Filters*, (Sevilla, Spain), the European Control Conference, 44th IEEE Conference on Decision and Control, December 2005.

[20] K. Fatahalian and M. Houston, "NVIDIA tesla: a unified graphics and computing architecture," *Hot Chips*, April 2008.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics forum*, vol. 26, no. 1, 2007.

[22] NVIDIA Corporation, "NVIDIA Tesla C2050/C2070 Datasheet." www.nvidia.com/tesla, July 2010.

[23] Advanced Micro Devices, Inc., "AMD FireStream^{TM} 9270 GPU Compute Accelerator ." www.amd.com/stream, May 2010.

[24] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, May 2008.

[25] K. Fatahalian and M. Houston, "A closer look at GPUs," *communications of the acm*, vol. 51, oktober 2008.

[26] D. Kirk and W. Hwu, *Programming Massively Parallel Processors.* Elsevier Science and Technology, 2010.

[27] G. Golub and J. M. Ortega, *Scientific Computing An Introduction with Parallel Computing.* London, United Kingdom: Academic Press inc, 1993.

[28] F. P. Brooks, *The mythical man-month*. Addison Wesley Publishing, 1995.

[29] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on computers*, 1972.

[30] J. K. Rice and M. Verhaegen, "Distributed control: A sequentially semi-separable approach for spatially heterogeneous linear systems.," *IEEE Transactions on automatic control*, 2009.

Glossary

List of Acronyms

AO	adaptive optics
API	application programming interface
AR	auto regressive
CPU	central processing unit
CUDA	Compute Unified Device Architecture
DCSC	Delft Center for Systems and Control
DM	deformable mirror
E-ELT	European extremely large telescope
ELT	extremely large telescope
ESO	European Southern Observatory
FFT	fast Fourier transform
FLOP	floating point operation
FPGA	field programmable gate array
GPGPU	general-purpose computing on graphics processing units
GPU	graphics processing unit
IC	integrated circuit
LLS	linear least squares
LTI	linear time invariant
MIMD	multiple instruction multiple data

MP	Model Partitioned
OpenCL	Open Computing Language
SIMD	single instruction multiple data
SIMT	single instruction multiple thread
SMVM	Sparse Matrix Vector Multiplication
SNR	signal-to-noise
SSS	sequentially semi separable
TMT	Thirty Meter Telescope
VMM	vector matrix multiply
WSN	wireless sensor networks
WFS	wavefront sensor

List of Symbols

α	Phase subscript mapping index
β	Measurement subscript mapping index
ΔT	Temporal resolution
ΔX	Spatial resolution in the x-direction
ΔY	Spatial resolution in the y-direction
γ	The fraction of code that is not parallelizable
$\Gamma(.)$	Gamma function
$\hat{y}^{id}_{(i,j)}$	Estimated output vector for segment i, j
λ	Wavelength of the light emitted by the object under observation
$\mathcal{N}_{(i,j)}$	Set of neighbours connected to segment i, j
$\mathcal{S}_{(i,j)}$	Grid segment on spatial coordinate i, j
ϕ	Phase difference vector
$\phi(k)$	Wavefront phase vector on time instance k
$\psi_j(k)$	Intermediate state estimate of segment j at time instance k
θ	Angular resolution
A	State transition matrix
A_d	The transition matrix
C	Output matrix
C	The capacitance switched per clock cycle
C_d	The output matrix
$c_{i,j}$	Weight of the state estimation of segment j with respect to segment i

e	Gaussian distributed white noise process
$E(.)$	Expectation operator
$e(k)$	Stochastic process wit zero-mean white noise
E_{new}	Executing time for entire task using the enhancement when possible
E_{old}	Execution time for entire task without using the enhancement
F	The frequency at which the IC is running
G	The phase-to-WFS influence matrix
H	The DM influence matrix
H_1	AR-1 predictor model
$H_Q^\dagger$	Is the regularised pseudo inverse of the DM influence matrix H
K	Kalman gain
k	Discrete time index
$K_{(i,j)}^{yl}$	Kalman gain for segment i, j based on a local full order model
K_d	The kalman gain
$K_{5/6}(.)$	Third order modified Bessel function
L_0	The outer turbulence scale
n	Measurement noise
N_m	Model order
N_u	Number of inputs
N_x	Grid size in the x-direction
N_y	Grid size in the y-direction
nn_i	The degree of segment i
p	The number of processing units
$P_{(i,j)}^{l}$	Covariance matrix for segment i, j based on a local full order model
r_0	The Fried parameter
S_p	Speedup factor related to the run time of the original code
$s_x[i,j]$	Measured phase gradients in the x direction
$s_y[i,j]$	Measured phase gradients in the y direction
T_i	Similarity transformation matrix for grid segment i
u	Control inputs
V	The supply voltage
$v(k)$	Measurement noise
v_x	Wind velocity in the x direction
v_y	Wind velocity in the y direction
W_i	Weight matrix representing the weights $c_{i,j}$
$x(k)$	State estimate at time instance x
$X_i(k)$	Stacked vector based on state estimates of $\mathcal{N}_i$
$y_{(i,j)}^{id}$	Output vector for segment i, j
z^{-1}	Discrete shift operator
$\varnothing$	The diameter of the aperture
N	The number of grid segments based on the WFS