THE UNIVERSITY OF SHEFFIELD

DOCTORAL THESIS

The Acceleration of Polynomial Methods for Blind Image Deconvolution Using Graphical Processing Units (GPUs)

Adam Petterson

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

October 2019

Abstract

Image processing has become an integral part of many areas of study. Unfortunately, the process of capturing images can often result in undesirable blurring and noise, and thus can make processing the resulting images problematic. Methods are therefore required that attempt to remove blurring. The main body of work in this field is in Bayesian methods for image deblurring, with many algorithms aimed at solving this problem relying on the Fourier transform. The Fourier transform results in the amplification of noise in the image, which can lead to many of the same problems as blurring.

Winkler presented a method of blind image deconvolution (BID) without the Fourier transform, which treated the rows and columns of the blurred image as the coefficients of univariate polynomials [1]. By treating the rows and columns of the image in this way, the problem of computing the blurring function becomes a problem of computing the greatest common divisor (GCD) of these polynomials. The computation of the GCD of two polynomials is ill posed, as any noise in the polynomials causes them to be coprime. Thus an approximate GCD (AGCD) must be computed instead.

The computation of an AGCD is a computationally expensive process, resulting in the BID algorithm being expensive. The research presented in this thesis investigates the fundamental mathematical processes underpinning such an algorithm, and presents multiple methods through which this algorithm can be accelerated using a GPU. This acceleration results in an implementation that is 30 times faster than a CPU parallel approach. The process of accelerating the BID algorithm in this way required a first of its kind GPU accelerated algorithm for the computation of an AGCD, with multiple novel techniques utilised to achieve this acceleration.

For Lianne. Without your inspiration, friendship, and support I would never have accomplished this. I'll always remember you.

Acknowledgements

I wish to thank all my supervisors and advisers for their support over the past four years. I would first like to thank Dr. Joab Winkler, for his guidance, support and encouragement in his role as my primary supervisor. I would also like to thank Dr. Paul Richmond for his help with understanding GPGPU concepts and advice on developing the algorithms central to this research. Thanks also go to thank Dr. Su Yi and Dr. Calvin Lim Chi Wan for their support when my research moved to Singapore. I would also like to thank Dr. Dirk Sudholt for his advice.

I would like to thank Peter Heywood for providing the code for the warp shuffle algorithm, and both Peter Heywood and Robert Chisholm for helping to debug my code when I came across challenging bugs.

I would like to thank my partner, Jenny, for the love and support she gave me throughout my PhD, and for staying by my side, even when I was on the other side of the world. I would also like to thank my family, for the support and encouragement they have given to me throughout my entire academic career. I would additionally like to thank all my friends in the UK for giving me help and support in dealing with the more stressful moments of my research, and to all the new friends I made in Singapore, who helped me to adapt and to settle in to life in a new country.

Contents

1 Introduction **1**

2 Image Blurring and Deblurring **7**

2.1 Convolution and the Point Spread Function 7

2.1.1 The Mathematics of Convolution 8

2.1.2 Effects of Image Convolution 11

2.1.3 The Boundary Area 12

2.1.4 Separable and Non-Separable point spread function (PSF)s 14

2.1.5 Spatial variance 14

2.1.6 Common Artifacts in Deconvolved Images 15

2.2 Existing Techniques and Literature Survey 16

2.2.1 Matlab functions 16

2.2.2 Blind Image Deconvolution Algorithms 19

2.2.3 Solutions to the Spatial Variance Problem 21

2.2.4 Parallel Image Deconvolution 23

2.3 Conclusion 24

3 Greatest Common Divisors and Approximate Greatest Common Divisors **25**

3.1 The Mathematics of GCDs and AGCDs 25

3.1.1 The Sylvester Resultant Matrix 26

3.1.2 The Bézout Resultant Matrix 28

3.1.3 QR and Singular Value Decomposition 29

3.1.4 Sylvester Subresultant Matrices 31

3.2 Literature Survey of AGCD Computation Methods 33

3.2.1 Euclid's algorithm 33

3.2.2 Matrix Methods 33

3.3 The QR Decomposition and QR Updates 35

3.3.1 Full Decomposition 36

3.3.2 QR Column Deletions 46

3.4 Literature Survey of Parallel QR Factorisation Methods 48

3.4.1 Parallelisation of QR Decomposition 48

3.5 Conclusion 52

4 Polynomial Blind Image Deconvolution 53

4.1 Algorithm Overview 53
4.2 Degree Computation 55
4.2.1 Algorithm 55
4.2.2 Improving Reliability of Degree Results 61
4.3 Computation of the Coefficients 64
4.3.1 Computation of an optimal column 66
4.3.2 Optimisations and Improvements 68
4.4 Image Deconvolution Results 68
4.4.1 Test 1 69
4.4.2 Test 2 69
4.4.3 Test 3 70
4.5 Simple CPU Parallelisation 71
4.6 Profiling and Parallelism Potential 72
4.7 Conclusion 74

5 Introduction to Parallel and GPU Computing 75

5.1 Parallel Computing and Accelerators 75
5.1.1 Data Parallel vs Task Parallel 75
5.1.2 Types of Parallel Coprocessors 76
5.2 Introduction to GPU Hardware and GPGPU Concepts 77
5.2.1 Grids, Blocks, Threads and Warps 77
5.2.2 Race Conditions and Synchronisation 77
5.2.3 Memory Usage 78
5.2.4 Warp Shuffling 81
5.3 Conclusion 81

6 GPU Acceleration of the Computation of the Degree of an AGCD 83

6.1 Algorithm design 84
6.1.1 Computation of the Upper Triangular Factor of the Subresultant Matrices 84
6.1.2 Computation of Rank and Degree Estimation 95
6.2 Implementation 96
6.2.1 Efficient Storage of Upper Triangular Matrices 96
6.2.2 Computation of the Upper Triangular Factors 98
6.2.3 Estimating the Degree 103
6.2.4 Profiling the Final GPU Implementation and Establishing Launch Parameters 114
6.3 Results 117
6.3.1 Reliability testing 117

6.3.2 Runtime testing 118
6.4 Issues and Potential Improvements 122
6.5 The Reduced Algorithm 122
6.5.1 Implementation 123
6.5.2 Results 123
6.6 Conclusion 124

7 A Low GPU Memory Approach to Degree Computation 127
7.1 Algorithm Changes 127
7.1.1 Example 128
7.2 Implementation 132
7.2.1 New Memory Structure 132
7.2.2 Kernel Changes 133
7.2.3 graphics processing unit (GPU) Profiling and and Establishing Launch Parameters 137
7.3 Memory Usage 139
7.4 Results 141
7.4.1 Reliability Testing 142
7.4.2 Runtime Testing 142
7.4.3 Testing on Limited Hardware 146
7.5 Profiling of the Accelerated Blind Image Deconvolution Algorithm 148
7.6 Conclusion 149

8 GPU Acceleration of the Computation of the Coefficients of an AGCD151
8.1 Algorithm 151
8.1.1 Algorithm Overview 152
8.1.2 Optimisations 153
8.1.3 Example 156
8.2 Implementation 162
8.2.1 Computing the Givens Matrices 162
8.2.2 Computing Column w of the Orthogonal Matrices 164
8.2.3 Computing the Residuals 166
8.2.4 Finding the Minimum Residual 166
8.2.5 GPU profiling 166
8.3 Results 169
8.3.1 Reliability testing 169
8.3.2 Runtime testing 169
8.3.3 Testing on Limited Hardware 171
8.4 Conclusion 172

9 Full Image Deconvolution and Results **175**
9.1 Standard Test Images 175
9.1.1 Error Computation 176
9.1.2 Error and Runtime Results 176
9.1.3 Reliability Improvements 184
9.2 Large Images 186
9.3 Conclusion 193

10 Conclusion **195**
10.1 Key Findings 195
10.1.1 Changes to the Original Algorithm 195
10.1.2 Parallelisation of the Degree Computation 197
10.1.3 Parallelisation of the Computation of the Coefficients 199
10.1.4 Overall Improvements to the Blind Image Deconvolution Algorithm 199
10.2 Future Research Opportunities 200
10.2.1 Further Improvements to the GPU Algorithms 200
10.2.2 Applications of the Algorithm 202
10.3 Conclusion 202

List of Figures

1.1 Example of image blurring . . . 1

2.1 Example of signal convolution where $a \otimes b = c$ . . . 8
2.2 Example of 2 dimensional convolution where $A \otimes B = C$ . . . 9
2.3 Examples of blurring functions . . . 12
2.4 Original exact image . . . 12
2.5 Exact image from Figure 2.4 blurred with Gaussian and motion PSFs shown in Figure 2.3 . . . 13
2.6 Example of two functions that could be used to taper the edges of images . 13
2.7 Photo of parrots demonstrating the spatially variant bokeh effect . . . 14
2.8 Image showing ringing artifacts in an image deconvolved by Lucy Richardson deconvolution . . . 15
2.9 Blurred and noisy image . . . 17
2.10 Figure 2.9 restored with the Lucy Richardson deconvolution function available in MATLAB . . . 17
2.11 Figure 2.9 restored using the two filtering methods available in MATLAB . . 18
2.12 Figure 2.9 restored with the blind deconvolution algorithm available in MATLAB . . . 19

4.1 Flowchart showing the serial process of the degree estimation algorithm . . 56
4.2 Blurred image with red lines to show the areas excluded from consideration when selecting rows and columns . . . 57
4.3 Example results from a single trial resulting in correct degree estimation . . 61
4.4 Example results from a single trial resulting in incorrect degree estimation . 62
4.5 Flowchart showing the serial process of the optimal column algorithm . . . 67
4.6 Exact, blurred, and restored images for Test 1 . . . 69
4.7 Exact, blurred, and restored images for Test 2 . . . 70
4.8 Exact, blurred, and restored images for Test 3 . . . 71
4.9 Comparison of central processing unit (CPU) serial and parallel implementations . . . 72
4.10 Results of profiling the serial implementation of the image deconvolution algorithm . . . 73

5.1 The hierachy of grids, blocks and threads 78
5.2 The difference between strided and coalesced global memory access 80
5.3 Efficient shared memory usage 80
5.4 Poor shared memory usage 81

6.1 The structure of the matrix computations in this algorithm 90
6.2 Diagram showing the progression of triangular numbers 97
6.3 Diagram demonstrating how work is balanced before and after load balancing is applied 101
6.4 Graph showing theoretical GPU memory usage of computing a single trial of an approximate greatest common divisor (AGCD) computation on the GPU 102
6.5 Time spent in each kernel of the GPU implementation 116
6.6 Runtimes of the GPU method for increasing polynomial degrees and varying numbers of trials 118
6.7 Runtimes of the GPU implementation compared CPU methods for increasing polynomial trials 119
6.8 Runtime improvement of the GPU method compared to the CPU methods for increasing numbers of trials 120
6.9 Runtimes of the GPU implementation compared CPU methods for increasing polynomial degrees 121
6.10 Runtime improvement of the GPU method compared to the CPU methods for increasing numbers of polynomial degrees 122
6.11 The number of trials that can be executed in a single kernel call by GPUs with different amounts of memory 123
6.12 Runtimes of the GPU original and reduced implementations for an increasing number of trials 124
6.13 Runtimes of the CPU original and reduced implementations for an increasing number of trials 125
6.14 Runtimes of the GPU original and reduced implementations for increasing polynomial degrees 126
6.15 Runtimes of the GPU original and reduced implementations for increasing polynomial degrees 126

7.1 GPU memory usage of the original method from Chapter 6 139
7.2 GPU memory usage of the low memory method 140
7.3 Comparison of the memory usage between the two methods 140
7.4 Runtimes of the low memory implementation for increasing polynomial degrees and various numbers of trials, with runtimes from the original implementation for comparison 143
7.5 Runtimes of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of trials 144

7.6 Runtime improvement of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of trials 144
7.7 Runtimes of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing polynomial degrees 145
7.8 Runtime improvement of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of polynomial degrees 146
7.9 Runtimes of the low memory GPU method on an NVIDIA GTX 780 compared to the NVIDIA Titan V and CPU serial implementation for increasing numbers of trials 147
7.10 Runtimes of the low memory GPU method on an NVIDIA GTX 780 compared to the NVIDIA Titan V and CPU parallel implementation for increasing polynomial degrees 148
7.11 Results of profiling the MATLAB implementation of the image deconvolution algorithm using the low memory GPU degree computation 149

8.1 Pie chart showing the percentage of time spent in each kernel 168
8.2 Runtimes of the GPU method and the CPU serial and parallel methods . . 170
8.3 Speedup of the GPU method over those of the CPU serial and parallel methods 171
8.4 Runtime of the GTX 780 against the Titan V, as well as the CPU parallel implementation as a comparison 172

9.1 The eight exact images used in this experiment 177
9.2 Blurred images from Figure 9.1 178
9.3 256×256 images restored on the CPU 179
9.4 256×256 images restored with the GPU 180
9.5 512×512 images restored on the CPU 182
9.6 512×512 images restored with the GPU 183
9.7 Failed deconvolution from incorrect trial selection 185
9.8 2000×2000 image that will be used for the tests in this section 186
9.9 The blurred and deconvolved images for a PSF of dimensions 25×25 with low levels of noise 188
9.10 The blurred and deconvolved images for a PSF of dimensions 51×51 with low levels of noise 188
9.11 The blurred and deconvolved images for a PSF of dimensions 75×75 with low levels of noise 189
9.12 The blurred and deconvolved images for a PSF of dimensions 101×101 with low levels of noise 189
9.13 The blurred and deconvolved images for a PSF of dimensions 25×25 with medium levels of noise 190

9.14 The blurred and deconvolved images for a PSF of dimensions 51×51 with medium levels of noise 190
9.15 The blurred and deconvolved images for a PSF of dimensions 75×75 with medium levels of noise 192
9.16 The blurred and deconvolved images for a PSF of dimensions 25×25 with high levels of noise 192
9.17 The blurred and deconvolved images for a PSF of dimensions 51×51 with high levels of noise 194

List of Tables

4.1 The percentage of trials identifying the correct degree for lower levels of noise 63

4.2 The percentage of trials identifying the correct degree for higher levels of noise 64

6.1 Rate of successful trials for varying polynomial degrees and noise levels . . 118

9.1 Time taken and error before and after the deconvolution on the CPU and GPU for the 256×256 images 181

9.2 Time taken and error before and after the deconvolution on the CPU and GPU for the 512×512 images 184

9.3 Time taken and errors for the deconvolution of large images at low levels of noise 187

9.4 Time taken and errors for the deconvolution of large images at low levels of noise 191

9.5 Time taken and errors for the deconvolution of large images at low levels of noise 191

Chapter 1

Introduction

Blurring in images is an important problem across many different fields. When an image is blurred, the blurring function used acts as a low pass filter, obscuring high frequency areas of an image. This means that detail in the blurred image is lost, and edges of features with in the image become less defined. Due to this degradation, blurring presents a significant barrier for many image processing techniques, such as feature selection and recognition. The removal of this blur is therefore an important area of research.

Blurring can be considered to be the convolution of an exact image $\mathcal{F}$, with a blurring function $\mathcal{H}$, also known as a point spread function (PSF), to result in a degraded image $\mathcal{G}$,

$$\mathcal{G} = \mathcal{F} \otimes \mathcal{H}.$$

Removing blur from these images can therefore be seen as a deconvolution problem, where the exact image must be separated from the PSF. An example of this convolution can be seen in Figure 1.1.

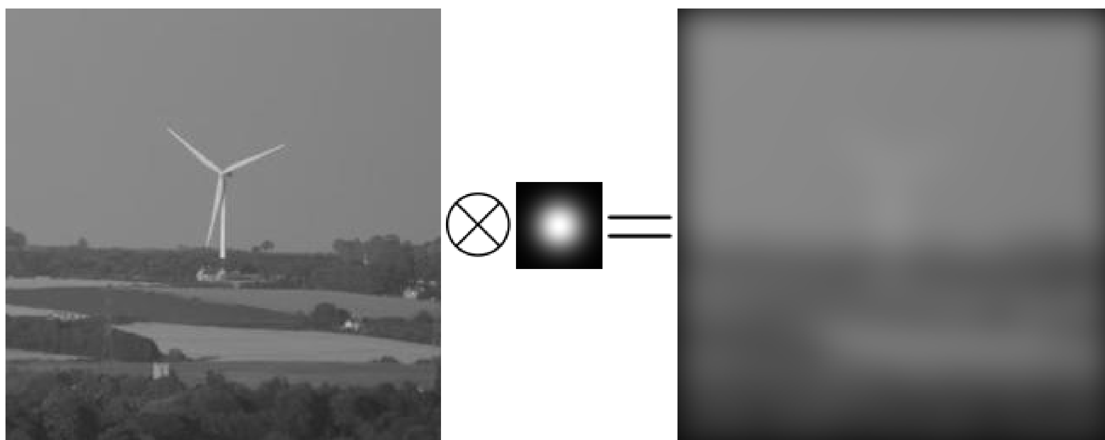


Figure 1.1: Example of image blurring

Many existing techniques for image deconvolution make assumptions about the PSF, in order to deconvolve it from the exact image. These assumptions can include prior

knowledge of the size of the PSF, or even of the exact form of the PSF. Unfortunately, in many real world scenarios, it is not possible to know exact details about the PSF, and thus these solutions are not effective in most realistic scenarios. Image deconvolution where these assumptions are not made is known as blind image deconvolution (BID), which is one of the most challenging problems in image processing.

Deconvolution can be particularly problematic when blurring is accompanied by noise, which can be amplified by many deblurring algorithms, or in some cases even prevent deblurring algorithm from working at all. Unfortunately, in most real world scenarios, noise is inevitable. Noise can be introduced to an image in the PSF iteslf, or after the convolution. A more accurate representation of convolution could therefore be described with

$$\mathcal{G} = \mathcal{F} \otimes (\mathcal{H} + \mathcal{N}_1) + \mathcal{N}_2.$$

where $\mathcal{N}_1$ is additive noise present in the PSF, and $\mathcal{N}_2$ is equal to the additive noise in the resulting blurred image.

In the presence of this noise, BID becomes an ill posed problem. The convolution and deconvolution of images will be discussed further in Chapter 2.

There are many existing methods of performing image deconvolution, including Bayesian methods [2, 3, 4, 5, 6], zero sheet separation [7], machine learning approaches [8, 9, 10, 11], and greatest common divisor (GCD) methods [12, 13, 14, 15]. These approaches are discussed further in Chapter 2. A common technique utilised in many of these methods, both blind and non-blind, is that of the Fourier transform. A major side effect of the use of a Fourier transform is the amplification of noise in the image, which leads to images being less clear, and can also interfere with image processing techniques. It is due to this that a method to perform blind image deconvolution without using the Fourier transform is desirable.

One method of achieving blind image deconvolution without the Fourier transform was presented by Winkler in his 2016 paper [1], where polynomial methods were used. This technique provided impressive results, with low errors compared to many other contemporary techniques. However, it is also computationally expensive. This algorithm is considered in detail in Chapter 4. Modifications to the original algorithm are also described in Chapter 4, that will improve the runtime and reliability of these computations.

Parallel programming can offer solutions to accelerate computationally expensive problems. In this thesis the BID algorithm presented by Winkler is investigated, and general-purpose computing on graphics processing units (GPGPU) is considered for the investigation into how the algorithm can be accelerated by utilising a graphics processing unit (GPU). By utilising GPGPU methods the runtime of the algorithm can be significantly reduced. To utilise GPUs effectively, parallelism must be found at a low level within the algorithm, and a significant amount of care must be taken to ensure the work is adequately balanced across the GPU. The use of GPUs will be discussed in Chapter 5, including

factors that must be considered to ensure the efficiency of the implemented algorithms.

Key to the approach proposed by Winkler is to consider the horizontal and vertical elements of a PSF to be the GCD of two polynomials, the coefficients of which are taken from the pixel values of rows and columns of the blurred image. Unfortunately, due to noise introduced in many imaging systems, finding an exact GCD of the polynomials is ill posed, as the polynomials are coprime and therefore have no common divisors. When the polynomials are coprime an approximate greatest common divisor (AGCD) must be computed instead. An AGCD is a form of GCD in which the coefficients can be perturbed within a given threshold, such that an exact divisor can be found for each of the polynomials.

Computing the AGCD is the most computationally expensive part of the BID algorithm by a large margin. This is due to the need for many large matrix operations, which can involve thousands of operations each. It is therefore a good candidate for acceleration by utilising the massive parallelism available on a GPU. While the focus of this thesis is on accelerating the BID algorithm, the AGCD algorithm that is accelerated to this end has potential applications in other fields. Two examples of such fields are control theory [16], and the computation of multiple roots of a polynomial [17, 18, 19]. The fast computation of an AGCD is thus motivated not just by image deconvolution, but also has more general applications.

The two most expensive computations, as part of the AGCD algorithm, are considered for acceleration in this thesis. The first section to be accelerated is the computation of a degree of an AGCD. In the algorithm proposed by Winkler the degree computation represents the majority of the overall compute time of the BID algorithm. Chapters 6 and 7 will consider the acceleration of this algorithm, and a GPU accelerated implementation will be proposed that demonstrates computation of the degree at speeds of up to 132 times faster than the serial implementation.

The second section to be considered is that of structured non-linear total least norm (SNTLN) [20], which is the method through which the coefficients of the AGCD are computed. The BID algorithm presented by Winkler utilised a modified form of the SNTLN method [21], in which an optimal column of a subresultant matrix must be computed to perform the remainder of this operation. This modification will be described Chapter 4. Chapter 8 proposes a GPU accelerated form of this computation, using the GPU to compute this optimal column. This accelerated algorithm demonstrated runtime improvements of up to 543 times faster than the serial implementation for the computation of the optimal column.

The result of the advancements described in this thesis is a GPU accelerated algorithm which performs BID faster, and more reliably, than the original BID algorithm. Chapter 9 presents the improvements made to the overall algorithm, with runtimes demonstrated to be up to 30 times faster than the central processing unit (CPU) parallel implementation when tested on high end hardware, despite many features of the algorithm not yet having been accelerated.

As this thesis covers a number of different areas the literature survey has been split into sections within chapters that cover these areas. Chapter 2 will discuss the literature for image deconvolution methods, including parallel implementations, while Chapter 3 will discuss both the literature of AGCD computation and of parallel QR factorisation.

This thesis will be organised into the following chapters:

Chapter 1 This chapter gives an introduction to the subject of the research presented in this thesis, provides the motivations behind this work, and provides a overview of the chapters of the thesis.

Chapter 2 This chapter describes the process through which images are blurred, and deblurred, from a high level, as well as some of the key challenges faced by deblurring algorithms. Several existing methods for image deconvolution are discussed, with a section describing how some of these algorithms could be parallelised.

Chapter 3 This chapter discusses the mathematics crucial to the computation of polynomial GCDs and AGCDs. Two resultant matrices that are of particular interest to this computation are defined, these being Sylvester and Bézout matrices. Two matrix decompositions that are useful for this computation are also discussed, with particular attention paid to the QR decomposition, which is central to the algorithm that is explored in this thesis. Methods of computing the AGCD from the literature are explored, as well as parallel methods for the computation of the QR decomposition.

Chapter 4 This chapter gives an overview of the BID algorithm proposed by Winkler, which uses polynomial AGCDs. The algorithm will be described in detail, with focus on the sections of the algorithm that will be parallelised in this research. This chapter will also detail some of the modifications made to this algorithm to improve its performance and reliability. A simple CPU parallel algorithm will be introduced that can be compared to the GPU accelerated algorithm in Chapters 6, 7, 8 and 9.

Chapter 5 This chapter introduces the concept of parallel computing, with a focus GP-GPU techniques. Important hardware and software aspects of utilising GPUs to accelerate algorithms will be discussed.

Chapter 6 This chapter discusses how degree computation was achieved on the GPU. The algorithm will be detailed, with focus on the computations within the algorithm where parallelism was found. The implementation of this algorithm on the GPU, and the challenges that had to be overcome to achieve this, will be discussed. Finally the results of this section will be detailed, with comparisons made against CPU serial and parallel implementations of the same computation.

Chapter 7 This chapter details an alternate approach to the degree computation, using a significantly lower amount of memory than the original implementation. This implementation offers a significantly greater amount of scalability, and shorter runtimes,

than the algorithm described in chapter 6. This will again be compared against the CPU implementations, and also against the implementation proposed in Chapter 6.

Chapter 8 This chapter details the GPU implementation of sections within the coefficient computation of the AGCD, with focus on a section of the modified SNTLN technique proposed by Winkler and Hasan in [12]. The runtime and reliability of this algorithm will again be tested against CPU serial and parallel implementations.

Chapter 9 This chapter applies the work from the previous chapters to the full image deconvolution algorithm. The accelerated algorithm is compared against the best performing CPU implementation. All the improvements described throughout this thesis will be tested, including tests for reliability and runtime improvements. Large images are also tested, to investigate how well the algorithm scales.

Chapter 10 The final chapter gives an overview of the key findings of this thesis, and some of the most important results detailed in the other chapters. This chapter will also suggest areas which could be of interest for further studies.

Chapter 2

Image Blurring and Deblurring

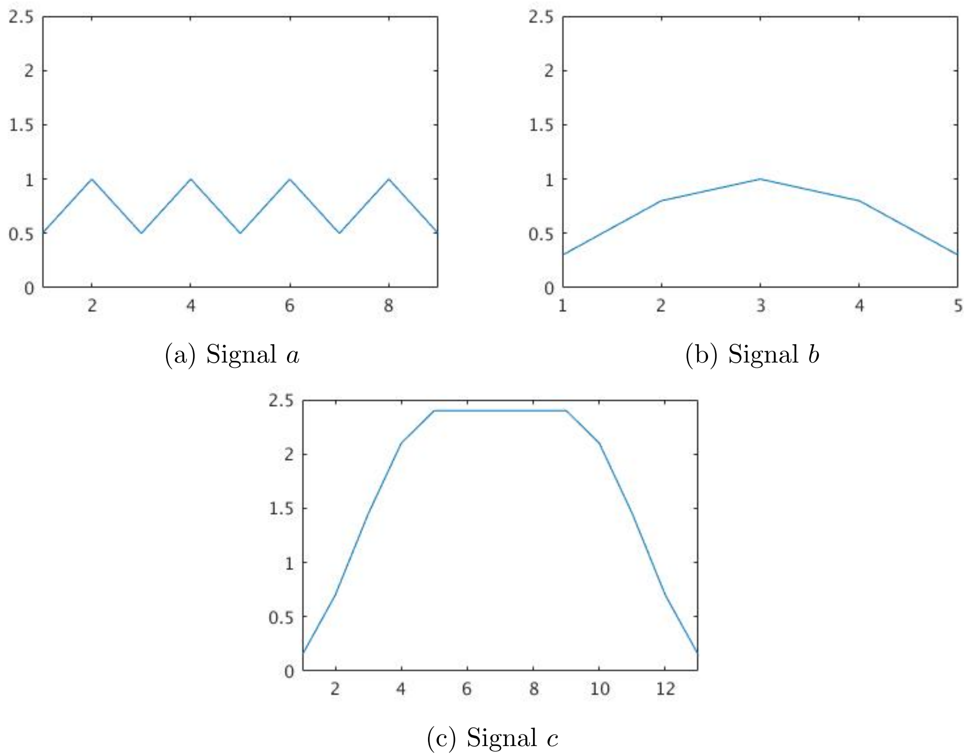
Images and image processing have become important parts of everyday life, with applications in many fields. Unfortunately, in the process of capturing an image, the exact image may be degraded in two significant ways: blurring, where exact information from points in the image are spread over neighbouring pixels, and noise, where pixel values may be perturbed to be greater or less than the exact value that is desired. This thesis concentrates on the removal of blur, but it aims to do so in the presence of noise. The presence of noise is almost inevitable in most real world scenarios, and causes a number of problems for image deconvolution algorithms.

Section 2.1 will give an overview of image blurring, including the mathematics of convolution, and how that relates to blurring in images, as well as the effects blurring can have on images. Section 2.2 will provide a survey of existing deconvolution methods, both blind and non blind. These will be described at a high level, with common issues that are encountered by these algorithms discussed. This section will also investigate how these image deconvolution algorithms can be parallelised.

2.1 Convolution and the Point Spread Function

Convolution is a mathematical function that involves the combination of two signals to result in a single signal. This is performed by measuring how much overlap occurs when one signal passes over the other signal [22]. When two signals are convolved the dimensions of the resulting signal become larger, due to the overlap of the edges of the two signals. Additionally, when one of the signals being convolved acts as a low pass filter, such as a Gaussian distribution, any high frequency features that may have been present in the original signal may be smoothed out and lost in the resulting signal.

For example, consider the signals shown in Figure 2.1. Figure 2.1a shows a high frequency signal a, when convolved with signal b, shown in Figure 2.1b, results in signal c, shown in Figure 2.1c. As can be seen in this example signal a has a width of 9, signal b has a width of 5, and the resulting signal c has a width of 13. Additionally, it is clear that the high frequency elements of signal a have been lost in signal c, as they have been



(a) Signal a

(b) Signal b

(c) Signal c

Figure 2.1: Example of signal convolution where $a \otimes b = c$

smoothed out when convolved with the low frequency signal b which is acting as a low pass filter.

2.1.1 The Mathematics of Convolution

When applied to two polynomials, as opposed to two signals, convolution is the equivalent of the product of the two polynomials. Consider the polynomials defined below, $f(x)$, of degree m and $g(x)$, of degree n,

$$f(x) = \sum_{i=0}^{m} a_i x^i, \quad g(x) = \sum_{j=0}^{n} b_j x^j.$$

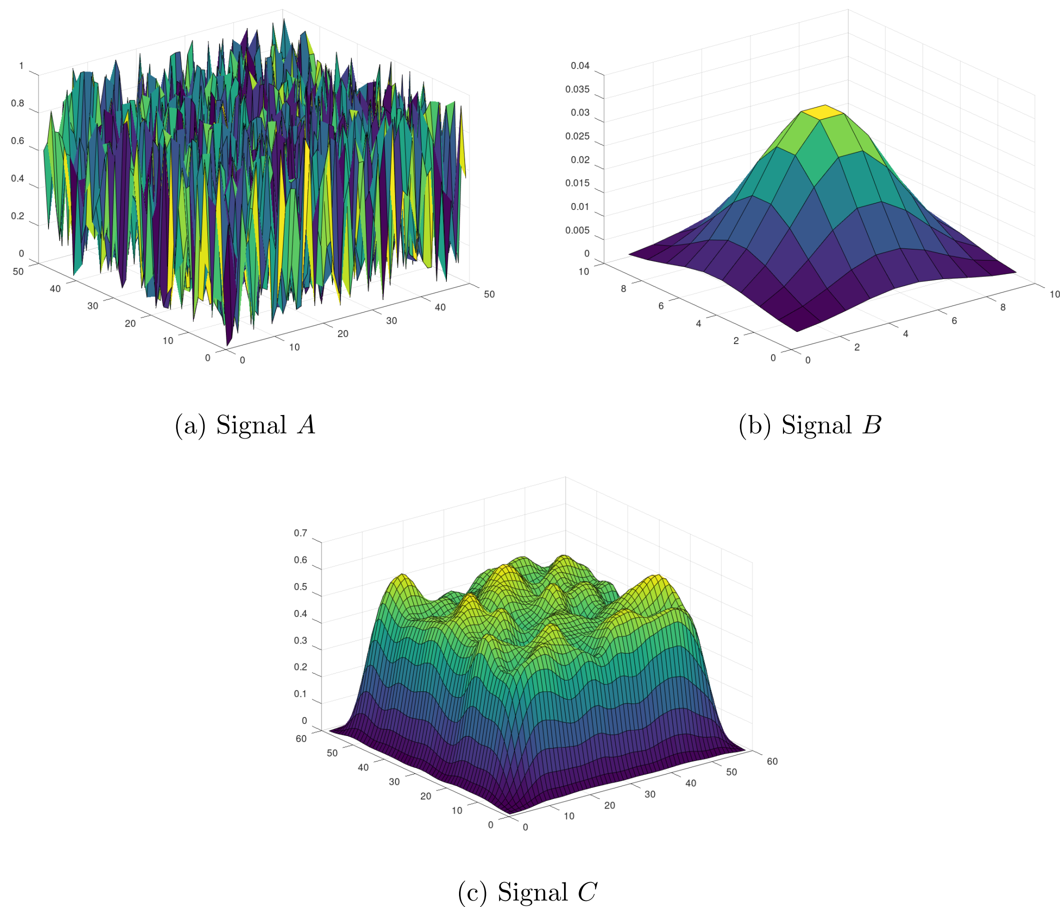
The convolution, or product, of these polynomials results in the polynomial $h(x)$,

$$h(x) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_{k-j} b_j x^k.$$

The computation to calculate the kth coefficient of the convolution of $f(x)$ and $g(x)$, or the kth element of signal c above, can be defined with

$$c_k = \sum_{j=0}^{k} a_{k-j} b_j. \tag{2.1}$$

In Equation 2.1, when considering signals with limited bounds, if the index required



(a) Signal A

(b) Signal B

(c) Signal C

Figure 2.2: Example of 2 dimensional convolution where $A \otimes B = C$

from signal a or b is outside of these bounds the value returned will be zero.

The concept of convolution can be expanded into two dimensions. Take for example the 2-dimensional signals A and B shown in Figure 2.2. Signal A is random noise of dimension 50×50, with values ranging between 0 and 1. Signal B is a Gaussian distribution of dimension 10×10, with a peak of 0.033. This Gaussian signal acts as a low pass filter. Similarly to the convolution example in Figure 2.1 it can be seen that the high frequency elements of signal A have been lost when convolved with B to result in C. Note that the dimensions of the resulting signal C have been expanded in the same way as occurred in Figure 2.1, with the dimensions of the new signal being 60×60.

Blurring in images is often modelled as a discrete 2-dimensional convolution of a 2-dimensional signal. The exact image $\mathcal{F}$ is convolved with a signal $\mathcal{H}$ to give the convolved image $\mathcal{G}$.

$$\mathcal{G} = \mathcal{F} \otimes \mathcal{H}. \tag{2.2}$$

When the signal $\mathcal{H}$ acts as a low pass filter the image becomes blurred, and the signal $\mathcal{H}$ is known as a PSF or blurring function.

While Equation 2.2 is sufficient for modelling blur in a artificial scenario, the majority

of imaging situations in the real world would not be as clean, and noise can be introduced to the image. This noise is introduced at two points. Firstly, in the process of deconvolution, where the noise is introduced to the PSF ($\mathcal{N}_1$), or after the convolution, where noise is added to the resulting image ($\mathcal{N}_2$). Therefore a more accurate equation to represent the blurring caused in a realistic scenario would be

$$\mathcal{G} = \mathcal{F} \otimes (\mathcal{H} + \mathcal{N}_1) + \mathcal{N}_2.$$

Treating blurring as convolution of polynomials, the signals $\mathcal{F}$ and $\mathcal{H}$ can be modelled as bivariate polynomials $F(x, y)$, of degree m in x and n in y, and $H(x, y)$ of degree p in x and q in y [12].

$$F(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} f(i, j) x^{m-i} y^{n-j},$$

$$H(x, y) = \sum_{k=0}^{p} \sum_{l=0}^{q} h(k, l) x^{p-k} y^{q-l}.$$

The product, or convolution, of these polynomials results in the bivariate polynomial $G(x, y)$,

$$G(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \sum_{l=0}^{q} f(i, j) h(k, l) x^{m+p-(i+k)} y^{n+q-(j+l)}.$$

Substituting $s = i + k$ and $t = j + l$ into this equation gives

$$G(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{s=i}^{p+i} \sum_{t=j}^{q+j} f(i, j) h(s - i, t - j) x^{m+p-s} y^{n+q-t}.$$

Therefore the coefficients of the resulting bivariate polynomial, or the elements of a discrete 2-dimensional signal, such as an image, can be defined as

$$g_1(s, t) = \sum_{i=0}^{m} \sum_{j=0}^{n} f(i, j) h(s - i, t - j). \tag{2.3}$$

Equation 2.3 represents the computation of convolution using a spatially invariant PSF. This means that the blurring function, in this case $H(x, y)$, does not change as a function of the position in the image. It therefore follows that when modelling blurring as convolution a spatially invariant PSF is implied.

Equation 2.3 is able to describe every form of 2-dimensional, spatially invariant PSF. However, deconvolving a PSF modelled as a bivariate polynomial is a significant challenge. A subset of these PSFs can be separated into vertical and horizontal elements, which can be treated independently [1]. This means that the vertical element and the horizontal element of a separable PSF can be computed separately, and deconvolved from the image independently.

A separable PSF can be defined mathematically as

$$\begin{aligned} H(x,y) &= \sum_{k=0}^{p}\sum_{l=0}^{q} h_r x^{p-k} h_c y^{q-l}, \\ &= h_r(x)h_c(y). \end{aligned}$$

Therefore, as the horizontal and vertical elements of $H(x, y)$ can be split into $h_r(x)$ and $h_c(y)$, this PSF can be applied to an image by the convolution of $h_r(x)$ with all polynomials representing rows in the image, and then $h_c(y)$ can be convolved with all of the columns in the result of the previous convolution. The resulting image would be the same as if the bivariate polynomial had been convolved with a bivariate polynomial representing the image directly. It then follows that the horizontal and vertical elements of this convolution can be deconvolved individually, which is a simpler problem.

It was previously noted that when a signal with high frequency elements is convolved with another signal, such as a Gaussian, then the high frequency elements are lost. When applied to imaging the PSF will act as a low pass filter, leading to high frequency components in the image being lost, and only low frequency components remaining. This defines the process of blurring, where areas of detail in the image are lost. The process of deblurring is the restoration of high frequency components within the image, and when treating the blur as a convolution, this is known as image deconvolution.

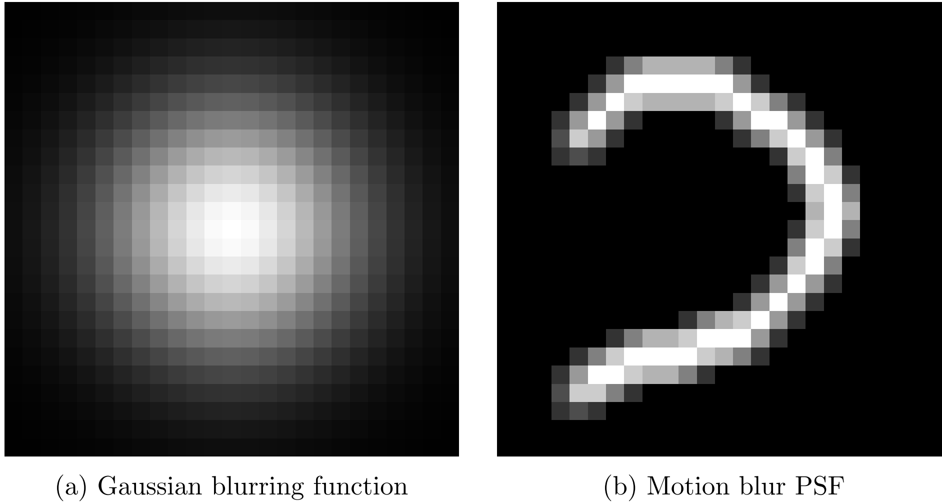
2.1.2 Effects of Image Convolution

In section 2.1.1, the mathematics of PSFs were defined. In this section, the effects of different PSFs on images that they are applied to will be investigated, including non-separable and spatially variant PSFs, and where these forms of PSF can be observed in real world scenarios.

Some examples of artificially generated PSFs can be seen in Figure 2.3. Figure 2.3a shows a Gaussian PSF. Gaussian PSFs are often used in image processing to simulate defocus blur, which can be caused by the incorrect aperture or focal length settings on a camera. Figure 2.3b represents an artificially generated example of what a PSF may look like from camera shake, causing motion blur.

Figure 2.4 shows an exact image in greyscale that is to be blurred. In Figure 2.5 we see the results of blurring this image with the two blurring functions in Figure 2.3. Both of the blurred images exhibit a clear loss of detail, which can particularly be seen in the shadows on the windmill, and the trees at the bottom of the image.

Additionally, a black area is seen around the blurred images. This boundary area occurs for the same reason that in the simple 1-dimensional convolution example seen previously results in a larger signal than the two input signals. However, this feature does not occur in most real world situations, though some image processing techniques assume its presence.



(a) Gaussian blurring function (b) Motion blur PSF

Figure 2.3: Examples of blurring functions

Figure 2.4: Original exact image

2.1.3 The Boundary Area

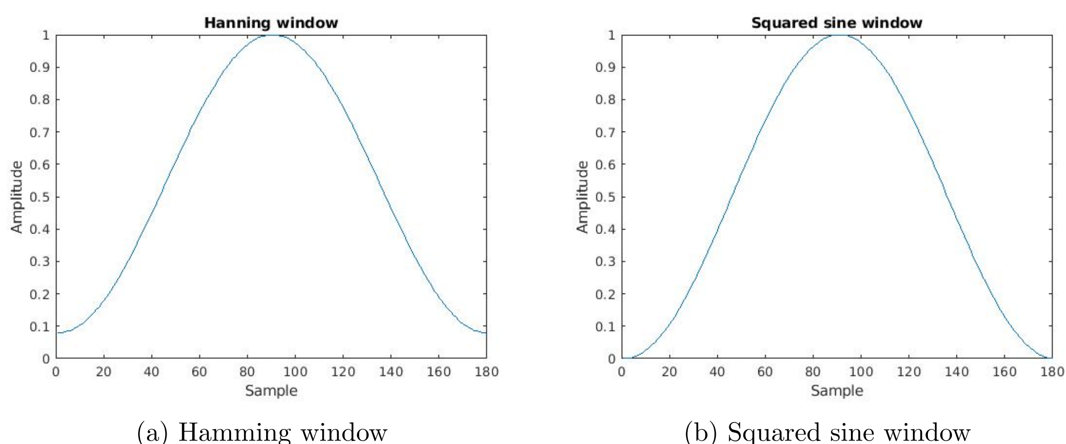
As was discussed previously, artificially blurred images such as those in Figure 2.5 show a black border around the images, where the image fades to black. The presence of this boundary area is due to the pixels outside of the dimensions of the exact image being regarded to be equal to be zero. This area is, for obvious reasons, not present in a naturally blurred image.

The fact that this boundary area is present is a problem for GCD methods for image deconvolution, which rely on having complete polynomials on which the GCD must be computed. Thus the boundary area must be extrapolated instead. This can be performed with window functions, which can taper the edges of the images in order to simulate this border area.

Some examples of windowing functions that can be used are the Hamming window, a sine curve where only positive entries are considered, and a squared sine curve. The

(a) Gaussian blurred image (b) Motion blurred image

Figure 2.5: Exact image from Figure 2.4 blurred with Gaussian and motion PSFs shown in Figure 2.3



(a) Hamming window (b) Squared sine window

Figure 2.6: Example of two functions that could be used to taper the edges of images

limit on the potential windowing functions is that they must be monotonically decreasing, such that the edge appears to be a smooth fade to black, as the weighted sum in the boundary area will be considering an increasing number of zero entries. Figure 2.6 shows two examples of window functions that can be used to taper the edge of an image.

As the GCD and its degree are not known in BID this presents a significant problem for GCD BID methods. Tapering the edges of the image would require knowledge at the very least of the size of the PSF. Simply repeating attempts at computing the GCD with multiple widths of taper is not desirable, due to the computational expense. Thus a method of estimating this width would be required. This is a problem that is as yet unsolved in the literature.

2.1.4 Separable and Non-Separable PSFs

The terms seperable and non-seperable refer to whether a PSF can or cannot be split into vertical and horizontal components, as described previously in this section. The Gaussian PSF in Figure 2.3a is an example of a separable PSF. A 2-dimensional Gaussian can be separated into two, 1-dimensional, Gaussian PSFs. When the horizontal element is applied to the rows of the exact image, then the vertical component is applied to the columns of the horizontally blurred image, the resulting image is identical to that of an image convolved with the 2-dimensional Gaussian.

Non-separable PSFs, however, provide a greater challenge, as the row and column components cannot be treated as independent. The motion blur PSF in Figure 2.3b is an example of an non-separable PSF. It is only possible to fully describe this PSF in the same way as with the Gaussian when analysis and computations are performed in the Fourier domain.

2.1.5 Spatial variance

A spatially variant PSF is a PSF that varies across the image. One key example of this is defocus blurring. This is often done purposefully, for artistic effect, which is known as the bokeh effect, though it can also be a caused by an incorrectly set aperture on the camera for the subject matter. The effect this has is a greater amount of blurring the further away an object is from the focal range that the camera is set up for.

Figure 2.7: Photo of parrots demonstrating the spatially variant bokeh effect

Figure 2.7 demonstrates the bokeh effect. In this photo the heads of both parrots are completely in focus. However, the feet of the parrot have a low level of blurring, the wooden railing has a moderate amount of blurring, and finally the foliage in the background is heavily blurred.

Other examples of spatially variant blurs involve motion blurs, where a subject in the image moves, and heat distortion, where movement of warm air causes distortion when an image is captured through it. The spatial variance problem represents one of the greatest challenges in image restoration.

2.1.6 Common Artifacts in Deconvolved Images

Many commonly used deconvolution techniques cause various forms of artifacts in the deconvolved images that are not present in the exact image. Most significantly noise can be introduced or amplified, and ringing can occur around edges in the image. This section will discuss some of these artifacts, and describe the causes.

2.1.6.1 Ringing

Ringing is a form of artifact present in images deconvolved by the Fourier transform. The Fourier transform is used to attempt to restore high frequency components of the image, and thus can cause repetition in some elements of the image. Ringing artifacts show repetition and ghosting around edges of elements of the image, where there is a large difference in the pixel values of the elements in an image. It is often also very apparent around the edges of the image, where there is a sharp gradient between the pixel values at the edge of the image, and the values outside of the image bounds. Figure 2.8 shows both of these forms of ringing, particularly at the edges of the photo, and around the subject of the photo.

Figure 2.8: Image showing ringing artifacts in an image deconvolved by Lucy Richardson deconvolution

The ringing at the edges of the images can be mitigated to some extent by tapering

the edges of the image, for example with the use of the `edgetaper` function in MATLAB. Tapering the edges in this way simulates the boundary conditions that would exist if the image had been convolved artificially, as was described in Section 2.1.3. Unfortunately tapering requires prior knowledge of the PSF, and can not be used in blind image deconvolution.

2.1.6.2 Noise

Deconvolution techniques that use the Fourier transform also create problems with noise. As previously discussed, the purpose of the Fourier transform in image deconvolution is to reintroduce high frequency elements that have been smoothed out by convolution, by transforming the image into the frequency domain. Unfortunately, through this process, low amplitude noise that may have been present in the original image may be significantly amplified by the Fourier transform.

2.2 Existing Techniques and Literature Survey

This section will discuss a selection of the various techniques used in image deconvolution, from the methods available in MATLAB to some more recent techniques from the literature. The majority of these techniques use the Fourier transform, which, as discussed in Section 2.1.6, results in an image with artifacts, and struggles to cope with even minor levels of noise in the input image. This section will also discuss methods in which the spatial variance problem have been tackled, and parallel image deconvolution algorithms.

As was discussed previously, a major distinction between forms of deconvolution algorithm is whether the algorithm is blind. Non-blind algorithms assume prior knowledge of the PSF, and simply attempt to deconvolve the provided PSF from the input image. Blind algorithms on the other hand prove to be a much greater challenge, as before deconvolution can occur the PSF must be estimated. The methods provided in MATLAB, as will be discussed, are non-blind. However, the methods discussed from the literature are primarily blind.

2.2.1 Matlab functions

MATLAB provides four functions for performing image deconvolution in the Image Processing Toolbox. These functions all provide spatially invariant deconvolution, though they do allow for non-separable PSFs. None of these techniques provide truly blind image deconvolution, though some make fewer assumptions about the PSF than others.

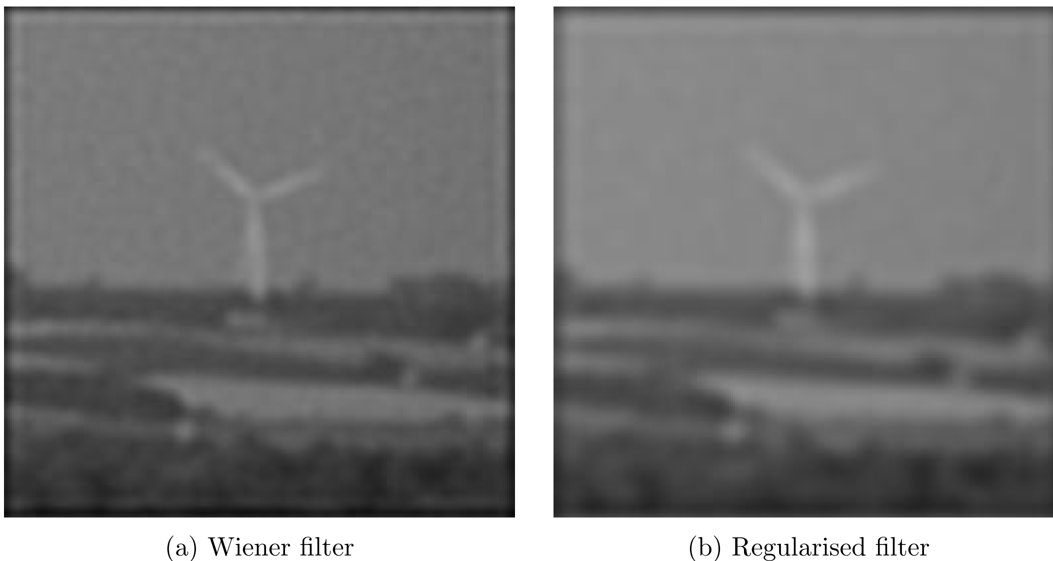
These methods in MATLAB have all been tested using the exact image in Figure 2.4, blurred with a Gaussian PSF of width 15, and with added noise. This is shown in Figure 2.9.

The first of the MATLAB functions tested is `deconvlucy` [23]. This function uses the Lucy Richardson method for deconvolution, which is a Bayesian approach that was

Figure 2.9: Blurred and noisy image

Figure 2.10: Figure 2.9 restored with the Lucy Richardson deconvolution function available in Matlab

independently described by Richardson in 1972 [2], and Lucy in 1974 [3]. The result of this algorithm on the blurred image in Figure 2.9 is shown in Figure 2.10. This algorithm attempts to maximise the likelihood that the resulting image, when convolved with the PSF, results in the blurred image. This method is non-blind, and requires full knowledge of the PSF beforehand. The result of this restoration, while it does restore some clarity to the edges of the fields and the windmill, exhibits both forms of artifacts discussed in Section 2.1.6. Though the ringing around elements in the image is minor, it is noticeable around the edge of the image, and the noise has become amplified and less defined than in the blurred image.



(a) Wiener filter (b) Regularised filter

Figure 2.11: Figure 2.9 restored using the two filtering methods available in MATLAB

The next two functions, `deconvwnr` [24] and `deconvreg` [25], use filters to attempt to deconvolve the image, respectively Weiner and regularised filters. Both functions require the exact PSF to be known, and also a value representing the amount of noise in the image, and thus neither option is blind. Figure 2.11 shows the results of these functions. Both implementations show a significant improvement in the amount of noise present when compared to the degraded image, with very little ringing present. The image restored with the Wiener filter is shown in Figure 2.11a. This image shows that the Wiener filter restored the details of the image reasonably well. However, the regularised filter approach in Figure 2.11b still shows a significant amount of blurring around the edges of features, and many details are still unrecognisable.

The final function, `deconvblind` [26] provides a semi-blind deconvolution method, in which aspects of the PSF, such as the degree, are assumed to be known. The algorithm uses maximum likelihood to estimate a PSF, and deconvolve this PSF from the blurred image. While the PSF can be estimated by the algorithm the function requires an initial matrix, to act as a starting point for the algorithm. This matrix can either be an estimate of the PSF, or a matrix of ones, in which case the algorithm will attempt to estimate the PSF. Even if no estimate is given, providing the algorithm with an initial matrix in which

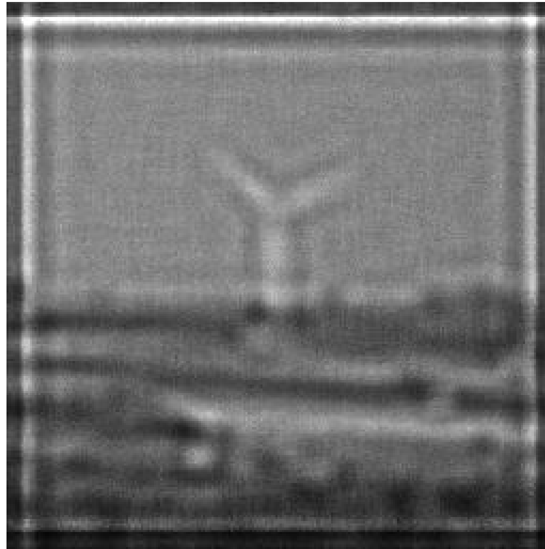


Figure 2.12: Figure 2.9 restored with the blind deconvolution algorithm available in MATLAB

to store the PSF is enough to make this method non-blind, as this provides the function with prior knowledge of the width of the PSF, which in most cases will not be known. Figure 2.12 provides the results of this function. The result shows very high levels of noise and relatively extreme levels of ringing. The details of the image are even less distinct than in the blurred image.

2.2.2 Blind Image Deconvolution Algorithms

This section will give an overview of a variety of BID algorithms from the literature, with a focus on four of main approaches to image deconvolution. Those being zero sheet separation, Bayesian approaches, machine learning approaches, and polynomial approaches.

2.2.2.1 Zero Sheet Separation

Zero sheet separation is a method of blind image deconvolution first proposed by Lane and Bates in 1987 [7]. In this method a single image, or any signal with more than 1 dimension, can be deconvolved from multiple PSFs without any complementary images, or prior knowledge of the PSF. This paper was significant at the time it was written, as it provided evidence that blind image deconvolution was possible based on a single image. This method relies on analytic properties of the Z-transform, which transforms signals into the frequency domain. Zeros in the Z-transform of a component of the convolution, that being a PSF when related to images, are usually continuous and lie on a $2K - 2$ dimensional hyper surface, where K is the dimension of the aforementioned component. These hyper-surfaces can be separated to calculate the components of the convolution. Zero sheet separation makes several assumptions about the image, most prominently this algorithm relies on there being no noise in the imaging system [27]. This is unlikely to be

the case in most real world situations.

2.2.2.2 Bayesian Approaches

Bayesian approaches to image deconvolution take a probabilistic approach to computing and deconvolving the PSF. While a Bayesian method was discussed previously, that being the Lucy Richardson algorithm [2, 3], this technique was for non-blind deconvolution.

Blind image deconvolution with Bayesian methods can be performed as either a maximum likelihood (ML) or a maximum a posteriori (MAP) problem, in which the probability of the exact image given the PSF, or the PSF given the exact image are maximised respectively [4].

Katsaggelos and Lay in their 1991 paper proposed a maximum likelihood approach using an iterative expectation-maximisation algorithm to estimate the unknown PSF [5], with no prior knowledge of the noise. Likas and Galatsanos in their 2004 paper proposed a method using variations approximation, a generalisation of the aforementioned expectation-maximisation algorithm [6]. This approach was found to be computationally efficient, even for large images, while retaining all of the advantages of previous Bayesian methods.

More recently the focus of research in the area of Bayesian BID has been on multi-frame deconvolution, requiring multiple blurred images of the same subject, unlike the single frame deconvolution considered in this thesis.

2.2.2.3 Machine Learning Approaches

Recently, much attention in the field of BID has focused on machine learning methods. Machine learning involves the creation of models from sets of training data by using algorithms and statistics. These models can then be used to infer properties about a previously unseen situation, which can be used to solve problems [28]. In the case of deconvolution, the machine learning methods can infer properties about the PSF of an unseen image, based on sets of training data of previously seen images with known PSFs [29].

Panchapakesan, Sheppard, Marcellin and Hunt presented a vector quantiser encoder distortion method for estimating the PSF [8]. The method involved using codebooks trained from example images to estimate the most likely PSF, that being the one that provided the least encoder distortion, from a finite set of candidate PSFs. Nakagaki and Katsaggelos take a similar approach in their 2003 paper [9].

More recently focus has been on neural networks, particularly convolutional neural networks and deep neural networks, though these have primarily been non-blind methods [10, 11]. Sun et al presented a blind method for deconvolving a spatially variant motion blur [30]. This will be discussed further in Section 2.2.3.

2.2.2.4 Polynomial Approaches

The primary focus of this thesis is on the method proposed by Winkler in his 2016 paper [1], though this is not the only research into polynomial approaches. This section will detail some of these methods, highlighting their strengths, weaknesses, and how they differ from the method investigated in this thesis.

The method proposed by Winkler aims to provide blind image deconvolution for a separable PSF without the Fourier transform. This method treats rows and columns of the blurred image as polynomials, and the PSF as two GCDs, and thus the vertical and horizontal components of a separable PSF can be found by treating its computation as the computation of the GCD. Due to noise in the image the GCD cannot be computed, and thus an AGCD must be computed instead. This method can be split into several steps, firstly the degree of an AGCD in each dimension must be computed, representing the height and width of the PSF. The computation of the degree essentially becomes a rank estimation problem, which is computed using QR decomposition, and QR updates computed in sequence. Once the degree has been computed the coefficients are calculated using structured matrix methods. The resulting 1-dimensional PSFs are then deconvolved from the blurred image using least squares problems. This method will be discussed in detail in Chapter 4. An extension to this algorithm is discussed in [12], that considers the computation of a non-separable PSF when two images convolved with the same blurring function are available. This method is computationally expensive, with longer runtimes than many other contemporary methods. It is this problem that this thesis seeks to address.

Another method for polynomial deconvolution was presented by Li, Yang, and Zhi in their 2010 paper [13], which concentrates on execution speed and generalisation. When multiple blurred images with a shared common image are available, the GCD of the z-transform of these images should be the exact image. The method proposed uses GCD algorithms to compute the exact image and the blurring function. The authors claim to have successfully recovered true images when the SNR is greater than or equal to 50dB.

Liang and Pillai presented a method of polynomial deconvolution for pairs of blurred images with a common exact image and different PSFs [14, 15]. The method presented used a Sylvester based AGCD algorithm, similar to that proposed by Winkler's method. Instead of using an AGCD algorithm to find the PSF however, the exact image is treated as an AGCD of the two blurred images. This method uses the Fourier transformation to deconvolve the image, and therefore will likely have issues with noise amplification and other artifacts.

2.2.3 Solutions to the Spatial Variance Problem

As discussed previously in this chapter, many existing techniques for image deconvolution consider only spatially invariant PSFs. Solving the spatial variance problem has been the subject of a considerable amount of research. In this section some of these methods will

be discussed. Spatial variance, as discussed in Section 2.1.5, refers to the PSF varying as a function of the location of the pixel in the image. This is present in many forms of real-world forms of blur, such as out of focus blurring.

The most common solution to the spatial variance problem covered in recent research involves splitting the image into sections with a common PSF, then deconvolving these sections of the image independently. One method, presented by Deshpande and Patnaik [31] in 2014 involved splitting the image into square subimages in order to deconvolve a spatially variant motion blur. Due to the use of traditional frequency domain techniques, using the Fourier transform, the results of this deconvolution result in a high number of the artifacts discussed in Section 2.1.6. This is particularly apparent around the edges of the subimages, which feature significant ringing. Windowing was used in the computation of individual images, aiming to prevent the ringing, then a low pass filter was applied to attempt to remove the remaining artefacts. This algorithm relies on the assumption that a PSF, while spatially variant, is spatially invariant in the in individual subimages. This is not the case in a natural spatially variant motion blur, as the blur will vary across individual pixels. While the results were claimed to be successful on real world images, only results for artificially blurred images that meet this assumption were shown.

Zhang, Wang, Jiang, Wang, and Gao presented a method in their 2016 paper [32] of deconvolving spatially variant defocus blur. In defocus blurring, as discussed in Section 2.1.5, the PSF varies as a function of distance from the camera. The method presented in this paper uses edge data, and Kth nearest neighbour techniques to estimate areas of the image likely to be a similar distance away from the camera. This results in irregular shapes in which the pixels are likely to be a roughly uniform distance from the camera. These shapes can be deconvolved under the assumption that they have the same PSF. This implementation is likely to fall into difficulty with images with extreme perspective changes, as this may have continuous areas that are the same colour with different levels of blur, which could result in the edge data creating an area of the image with drastically different levels of blurring.

Temerinac-Ott, Ronneberger, Nitschke, Driever and Burkhardt in their 2011 conference paper [33] presented a different method for computing a spatially variant PSF for single plane illumination microscopy. The method used was to capture images from multiple angles of an object, then, by breaking each image into segments, and inspecting overlapping segments of the image, to create a composite PSF. This was then then deconvolved these spatially variant PSFs using the Lucy Richardson algorithm. While this implementation is not suitable for all forms of imagery, it does provide insight into alternative methods attempting to solve the spatial variance problem.

Sun, Cao, Xu and Ponce proposed a deep learning approach to solving the spatial variance problem for smooth motion blurs [30]. Unlike other methods described in this section the implementation here assumed motion smoothness, as opposed to the segmented images where the segments have uniform PSFs discussed in the rest of this section. This algorithm provided impressive results, avoiding some of the noise present in comparable

algorithms.

2.2.4 Parallel Image Deconvolution

Image processing is a natural fit for large scale parallelisation such as that provided by GPUs, as it often requires performing the same operation on a number of pixels simultaneously, or large scale matrix operations. While Chapter 5 will give an introduction to GPU hardware and programming techniques, this section will give an overview of some of the current implementations of deconvolution on GPUs, and other parallel systems.

As discussed throughout this chapter, the Fourier transform is prevalent in most methods for deconvolution, both blind and non-blind. The implementation of the Fourier transform on a GPU is a well researched problem [34, 35, 36], with speedups provided by these algorithms between 2 and 80 times faster than contemporary Fourier transform libraries. Due to this, many of the methods discussed throughout this chapter can be accelerated. Additionally, many of the machine learning implementations presented use GPUs and other large scale parallelism in the training of their models [11]. In this section however, focus will be on specific deconvolution implementations on GPUs.

Domanski, Valloton, and Wang in their 2009 conference paper [37], implemented a Lucy Richardson deconvolution method on a GPU, using a GPU accelerated Fourier transform. As was noted in Section 2.2.1 the Lucy Richardson method is non-blind. This implementation achieved a speedup of 4.2-8.5 times faster than that of a CPU method using a well optimised fast Fourier transform method.

Matson et al. in their 2009 paper [38] presented a parallel method for blind deconvolution of images of space objects through atmospheric distortion. While not implemented on a GPU, this algorithm was designed for cluster computing, which relies on a large number of computers linked together, where processes can be distributed to different CPUs within the cluster. This was an multi-frame approach, meaning the algorithm used a series of images to estimate an exact image, as opposed to attempting to deconvolve an individual image as is investigated in this thesis. Execution time of this algorithm ranged from around 1 second to several hundred seconds, dependent on the number of processors used, and the size of the images, which was the main limiting factor in terms of execution time.

In 2011 Klosowski and Krishnan presented a method of non-blind image deconvolution on GPUs [39]. This was an accelerated implementation of the method presented by Krishnan and Fergus in 2009 [40]. This method takes a regularisation approach using hyper-Laplacian image priors, with an assumption that all of the properties of the blurring function are known. Polynomial root finding in the Fourier domain is used to solve optimisation problems that arise through this approach. This method gained much attention due to its relative efficiency for the results generated. Klowski and Krishnan tested the original CPU implementation against their GPU implementation. The GPU used was an NVIDIA GTX 260, which was tested against an Intel Xeon E5506. The results detailed in their paper suggest deconvolution was accomplished over 27 times faster on the GPU than the CPU, though it was noted that there were some performance limitations caused by

the CUDA fast Fourier transform library that was used. While the size of the image that could be processed was limited at the time of publication, due to the small amount of GPU memory available in the GTX 260, this limitation would not be as severe on more modern hardware, due to significantly increased memory availability. While this is a non-blind method of deconvolution it is worth noting the acceleration gained on what is now dated hardware. It is possible that this implementation could be used to compliment the PSF estimation methods presented in this thesis.

Goto, Otake and Hurano in their 2017 paper [41] presented a method of GPU accelerated BID algorithm for the deconvolution of motion blurs. Firstly a patch was selected, that enabled the algorithm to process a smaller section of the image to attempt to find the PSF. This patch was selected by finding areas with optimal edge data, found with Laplacian and Sobel filters. The patch was used to estimate the PSF, which is then deconvolved from the image using the Krishnan and Fergus implementation that was described previously. This method was tested on an NVIDIA GTX Titan GPU, against an Intel Xeon E5-2360 CPU. The implementation showed speedups of 2.58 times compared to a GPU, with an image of resolution 1920×1080 taking 3.2 seconds to deconvolve. It was noted that the deconvolution of the estimated PSF was 10 times faster than the CPU implementation.

While the work discussed in this section provided parallel approaches to various different algorithms it is noteworthy that at the time of writing no existing research could be found on the parallelisation of polynomial methods of blind image deconvolution, on which this thesis focuses.

2.3 Conclusion

In this chapter, a brief overview was given to understand the problem of image deconvolution, how it can be modelled computationally, and some of the approaches that have been harnessed to attempt to solve it. The approach of Winkler was briefly discussed in Section 2.2.2.4. It is this approach that the advancements presented in this thesis will be based on, and it will be discussed throughout the rest of this thesis. Before this can happen, however, it is necessary to get a much more in depth understanding of this algorithm, and profile it to see where improvements can be made. This shall be the scope of the next chapter.

Chapter 3

Greatest Common Divisors and Approximate Greatest Common Divisors

Chapter 2 provided an overview of the literature surrounding BID algorithms. One of the methods of performing BID discussed in this chapter was to treat the pixel values of the image as coefficients of a polynomial, and the blurring function as the GCD of these polynomials. It was noted that the problem of finding the GCD of these polynomials is ill-posed, due to noise in the image causing the polynomials to be coprime. In this situation an AGCD must be computed instead. The method of BID investigated in this thesis, proposed by Winkler, makes use of AGCDs, and thus this Chapter will discuss GCDs and AGCDs in detail.

Section 3.1 will discuss the mathematics involved in the computation GCDs and AGCDs, including two forms of resultant matrix that can be used for this purpose. Section 3.2 will provide a survey of the literature surrounding methods used to compute AGCDs. Section 3.3 will give a detailed overview of QR decomposition and QR updates, computations which are central to the AGCD method used in Winkler's BID algorithm. Finally, Section 3.4 will provide a survey of parallel QR implementations from the literature.

3.1 The Mathematics of GCDs and AGCDs

The GCD of two or more polynomials is a polynomial of the greatest degree possible that is a factor, and can be deconvolved from, all of the source polynomials. The computation of polynomial GCDs has been the subject of a significant amount of research. However, as was discussed previously, when noise is present in images, or other signals in which a GCD must be found, the exact GCD cannot be computed. Instead, an AGCD must be computed.

Consider the coprime univariate polynomials $u(x)$ and $v(x)$. When both of these polynomials are convolved with a third polynomial $d(x)$, this will result in the polynomials

$f(x)$ and $g(x)$, this is the same as polynomial multiplication.

$$f(x) = u(x)d(x),$$
$$g(x) = v(x)d(x).$$

The GCD of $f(x)$ and $g(x)$ in this case would be $d(x)$.

The computation of the GCD of two or more polynomials has applications in multiple fields. While in this thesis the main focus is on their use in image deconvolution [1, 42, 14, 15, 13], some further examples include control theory [43, 16] and polynomial root finding [17, 18, 19].

In the exact case discussed above, the coefficients of $f(x)$ and $g(x)$ must be free of any errors. Unfortunately, many practical problems cannot be solved by finding the GCD, as noise present in the convolved signals, or in the convolution process itself, would result in a high likelihood of the polynomials $f(x)$ and $g(x)$ being coprime. Thus the problem of computing the GCD in these cases is ill posed.

While it is not possible to compute the GCD in the inexact case, it is still possible to deconvolve these polynomials by finding an AGCD instead. An AGCD takes the form of a polynomial that, while the degree remains constant, the coefficients can be perturbed within a specified threshold and can be deconvolved from the inexact polynomials $f(x)$ and $g(x)$. This deconvolution will result in polynomials roughly equal to the exact polynomials $u(x)$ and $v(x)$.

When applied to the convolution of an image with a separable PSF, such as in the BID algorithm proposed by Winkler, two rows, or two columns, are selected at random from the original image. As was discussed in Chapter 2, convolution of an image is performed in the same way as convolution of polynomials, with the pixel values of the selected rows and columns considered to be the coefficients of polynomials.

For the purposes of image deconvolution, it is always the case that the degrees of the two polynomials, m and n, are equal. However, when applying this AGCD algorithm in situations other than image deconvolution, this will not always be true. For the purpose of generality, this section will consider the possibility that they are not equal, though it will be assumed that $m \geq n$.

3.1.1 The Sylvester Resultant Matrix

The Sylvester matrix is frequently used for the computation of an AGCD of two univariate polynomials. In this section the properties of the Sylvester matrix that make it useful for this computation will be explained.

The Sylvester matrix $S(f, g)$ is formed from two Toeplitz matrices $T(f)$ and $T(g)$, where f and g are the coefficients of the matrices $f(x)$ and $g(x)$ respectively.

$$f = [f_0, f_1, \ldots, f_{m-1}, f_m] \quad , \quad g = [g_0, g_1, \ldots, g_{n-1}, g_n] \,.$$

Toeplitz matrices $T(f)$ and $T(g)$ are constructed for these vectors.

$$T(f) = \begin{bmatrix} f_0 & & \\ f_1 & \ddots & \\ \vdots & \ddots & f_0 \\ f_{m-1} & \vdots & f_1 \\ f_m & \ddots & \vdots \\ & \ddots & f_{m-1} \\ & & f_m \end{bmatrix}, T(g) = \begin{bmatrix} g_0 & & \\ g_1 & \ddots & \\ \vdots & \ddots & g_0 \\ g_{n-1} & \vdots & g_1 \\ g_n & \ddots & \vdots \\ & \ddots & g_{n-1} \\ & & g_n \end{bmatrix}.$$

These Toeplitz matrices are then concatenated to compute the Sylvester matrix,

$$\begin{aligned} S(f,g) &= [T(f), T(g)], \\ &= \begin{bmatrix} f_0 & & & g_0 & & \\ f_1 & \ddots & & g_1 & \ddots & \\ \vdots & \ddots & f_0 & \vdots & \ddots & g_0 \\ f_{m-1} & \vdots & f_1 & g_{n-1} & \vdots & g_1 \\ f_m & \ddots & \vdots & g_n & \ddots & \vdots \\ & \ddots & f_{m-1} & & \ddots & g_{n-1} \\ & & f_m & & & g_n. \end{bmatrix}. \end{aligned}$$

The main property of $S(f, g)$ that is of interest for GCD degree computation is the rank. When the polynomials $f(x)$ and $g(x)$ have a GCD of degree p the rank of this matrix will be equal to $m+n-p$, and thus p can be easily computed from the rank loss. However, in the presence of noise, $f(x)$ and $g(x)$ become coprime, and $S(f, g)$ will be of full rank. In this case further analysis will be required in order to approximate the rank of $S(f, g)$. The singular value decomposition (SVD) and the QR decomposition are frequently used for this purpose. The QR decomposition of the Sylvester matrix constructed from polynomials with an exact GCD results in a matrix in which the last non-zero row will contain the coefficients of the GCD.

Due to these properties the computation of an AGCD can be treated as a problem of computing a low rank estimate of the Sylvester matrix, in which the coefficients of the AGCD appear on the last non-zero row of its QR decomposition.

3.1.2 The Bézout Resultant Matrix

An alternative to the Sylvester matrix for GCD computations is the Bézout matrix. The main advantage of the Bézout matrix over the Sylvester matrix is that it is significantly smaller, and thus less expensive to process. While the Sylvester matrix is of dimension $(m+n) \times (m+n)$, the Bézout matrix is of dimension $\max(m,n) \times \max(m,n)$, where m and n are the degrees of the polynomials used to construct each matrix. This smaller matrix means that processing the Bézout matrix involves significantly fewer operations than the Sylvester matrix. Additionally the Bézout matrix is symmetric. This means that complexity is reduced even further by processing only a half of the entries of the matrix.

The Bézout matrix $B(f,g)$ is constructed from the polynomials $f(x)$ and $g(x)$, where the vectors f and g are the coefficients of these polynomials respectively.

$$f = [f_0, f_1, \ldots, f_{m-1}, f_m] \quad , \quad g = [g_0, g_1, \ldots, g_{n-1}, g_n] \, .$$

The entries b_{ij} of the Bézout matrix can be computed with

$$b_{ij} = \sum_{k=0}^{m_{ij}} f_{i+k+1} g_{j-k} - f_{j-k} g_{i+k+1}, \tag{3.1}$$

where $m_{ij} = \min(i, \max(m,n) - 1 - j)$.

As discussed the Bézout matrix is both smaller than the Sylvester matrix and symmetric. This can lead to significantly faster computation of the GCD. Adding to this the Sylvester matrix often requires preprocessing of the vectors, which will be described in Section 4.2.1.2. The Bézout matrix does not require this step due to how the coefficients of the matrices are combined.

While the processing of a Bézout matrix can be fast, this comes at the cost of robustness of algorithms using it, particularly when considering inexact polynomials. When an entry b_{ij} is computed using $f_i g_j - f_j g_i$, where $f_i g_j \approx f_j g_i$, then issues may arise due to round off and precision errors, leading to b_{ij} appearing to equal zero when this is not necessarily the case.

Similarly to the Sylvester matrix, the rank of the Bézout matrix can be used to compute the degree of an exact GCD, and the last non-zero row of the QR decomposition of this matrix will give the coefficients of the GCD.

Example

Consider for example the polynomials $f(x)$ and $g(x)$,

$$f(x) = 4 + 2x + 7x^2,$$
$$g(x) = 3 + 6x + 4x^2.$$

The vectors f and g represent the coefficients of these polynomials,

$$f = \begin{bmatrix} 4 & 2 & 7 \end{bmatrix},$$
$$g = \begin{bmatrix} 3 & 6 & 4 \end{bmatrix}.$$

The structure of a 3×3 Bézout matrix, constructed using Equation 3.1, is shown below.

$$B(f,g) = \begin{bmatrix} f_1g_0 - f_0g_1 & f_2g_0 - f_0g_2 & f_3g_0 - f_0g_3 \\ \begin{pmatrix} f_1g_1 - f_1g_1 + \\ f_2g_0 - f_0g_2 \end{pmatrix} & \begin{pmatrix} f_2g_1 - f_1g_2 + \\ f_3g_0 - f_0g_3 \end{pmatrix} & \begin{pmatrix} f_3g_1 - f_1g_3 + \\ f_4g_0 - f_0g_4 \end{pmatrix} \\ \begin{pmatrix} f_1g_2 - f_2g_1 + \\ f_2g_1 - f_1g_2 + \\ f_3g_0 - g_0f_3 \end{pmatrix} & \begin{pmatrix} f_2g_2 - f_2g_2 + \\ f_3g_1 - f_1g_3 + \\ f_4g_0 - g_0f_4 \end{pmatrix} & \begin{pmatrix} f_3g_2 - f_2g_3 + \\ f_4g_1 - f_1g_4 + \\ f_5g_0 - g_0f_5 \end{pmatrix} \end{bmatrix}.$$

Several of the expressions in this matrix can be cancelled out to simplify computation. Every expression $f_jg_i - f_ig_j$ where $i = j$ will equal zero so can be removed, for example in $b_{0,1}$. Every expression where i or j is greater than m or n will result in zero, so can also be removed, for example in $b_{1,2}$. Finally, as is seen only in $b_{0,2}$, the expression $f_1g_2 - f_2g_1 + f_2g_1 - f_1g_2$ will result in zero, so this can also be removed. This results in the simpler structure shown below.

$$B(f,g) = \begin{bmatrix} f_1g_0 - f_0g_1 & f_2g_0 - f_0g_2 & 0 \\ f_2g_0 - f_0g_2 & f_2g_1 - f_1g_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Constructing this matrix using f and g results in the matrix

$$B(f,g) = \begin{bmatrix} 2 \times 3 - 4 \times 6 & 7 \times 3 - 4 \times 4 & 0 \\ 7 \times 3 - 4 \times 4 & 7 \times 6 - 2 \times 4 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$
$$= \begin{bmatrix} -18 & 5 & 0 \\ 5 & 34 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

3.1.3 QR and Singular Value Decomposition

The SVD involves the decomposition of an $m \times n$ matrix A into three factors[1]. U, an $m \times m$ orthogonal matrix, Σ, an $m \times n$ rectangular matrix in which the only non-zero

[1]For the purposes of this section m and n are not defined by the degrees of polynomials. As should be obvious from the context, m and n here refer to the dimensions of the matrix being decomposed.

entries are non-negative and lie on the principal diagonal, and V, an $n \times n$ orthogonal matrix matrix.

$$A = U\Sigma V.$$

For the purposes of finding the degree of a GCD the matrix Σ is of interest. The diagonal values of this matrix are known as the singular values of A. A notable feature of this matrix is that the rank of Σ is equal to the rank of the original matrix A.

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & \\ & \sigma_2 & & & \\ & & \ddots & & \\ & & & \sigma_{m-1} & \\ & & & & \sigma_m \end{bmatrix}.$$

As noted the only non-zero values of this matrix are along the diagonal. Consider a Sylvester matrix, constructed using the method above using polynomials with an exact GCD of degree p. When this matrix is split into its factors using SVD, the last p entries on the diagonal of Σ will equal zero. Thus, the degree of the GCD is equal to m minus the rank of Σ.

When an AGCD is required, rather than a GCD, these entries will no longer be equal to zero, due to the polynomials being coprime. Thus Σ will be of full rank. While the last p singular values will now be non-zero, they will, with high likelihood, be smaller than the other singular values. Thus the rank estimation of Σ, and therefore A, can be computed by the use of thresholding to detect the smaller singular values.

While the SVD is frequently used for GCD computations, it encounters problems in some situations. Particularly where rounding errors may indicate that matrices are singular, when theoretically they should not be [44]. It also requires that a threshold is set with which to detect the small values of Σ.

The QR decomposition can be used for this function instead of the SVD [45, 21]. This form of decomposition was shown to be more resistant to the round off problems of the SVD, and the method proposed by Winkler in [21] does not require a threshold to be specified. The complexity for both of these decompositions performed on square matrices is $O(n^3)$.

The QR decomposition splits the original $m \times n$ matrix A into two factors. Those factors being an $m \times m$ orthogonal factor Q and an $m \times n$ upper triangular factor R.

$$A = QR.$$

Similarly to the SVD, only the upper triangular factor R is of interest when computing a degree. As was the case with Σ in the SVD, the rank of R is equal to the rank of A, as the entries of the last p rows will equal zero.

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n-1} & r_{1,n} \\ & r_{2,2} & \cdots & r_{2,n-1} & r_{2,n} \\ & & \ddots & \vdots & \vdots \\ & & & r_{m-1,n-1} & r_{m-1,n} \\ & & & & r_{m,n} \end{bmatrix}.$$

Consider the matrix R that was computed from the Sylvester matrix constructed using polynomials of degrees m and n, with a GCD of degree p. In this case the matrix is square, as it is assumed that $m = n$. The entries on the final p rows of the upper triangular section of this matrix will be equal to zero, just like the SVD. The degree p of the GCD will therefore be equal to n minus the rank of R. However, just like with the SVD, when the polynomials are coprime these entries will no longer equal zero, and the matrix will be of full rank. The algorithm proposed by Winkler demonstrates a method to compute the rank of the Sylvester matrix without a threshold needing to be specified.

The coefficients of the GCD of the polynomials $f(x)$ and $g(x)$ can be found on the last non-zero row of the upper triangular factor of the Sylvester matrix $S(f, g)$ [43, 12]. Though when the polynomials are coprime, and an AGCD is required to be computed, there is more complexity to the coefficient computation, as a low rank estimate of the Sylvester matrix must be computed.

The QR decomposition, as the decomposition that is of primary interest in this thesis, will be explored further in Section 3.3.

3.1.4 Sylvester Subresultant Matrices

The method proposed by Winkler uses subresultant matrices constructed from the original Sylvester matrix S_1. While the exact use for these subresultant matrices will be discussed in Section 4.1, this section will define the subresultant matrices, and discuss a method with which to efficiently compute the QR decomposition of all subresultant matrices.

Each subresultant matrix $S_{2...n}$ is computed from S_{k-1}, with S_1 being the original Sylvester matrix $S(f, g)$. Take for example the matrix S_1, constructed using the method shown in Section 3.1.1, using polynomials of degree 4.

$$S_1 = \begin{bmatrix} f_0 & & & & g_0 & & & \\ f_1 & f_0 & & & g_1 & g_0 & & \\ f_2 & f_1 & f_0 & & g_2 & g_1 & g_0 & \\ f_3 & f_2 & f_1 & f_0 & g_3 & g_2 & g_1 & g_0 \\ f_4 & f_3 & f_2 & f_1 & g_4 & g_3 & g_2 & g_1 \\ & f_4 & f_3 & f_2 & & g_4 & g_3 & g_2 \\ & & f_4 & f_3 & & & g_4 & g_3 \\ & & & f_4 & & & & g_4 \end{bmatrix}.$$

Using this matrix, the first subresultant matrix, S_2, can be computed. This is per-

formed by removing two columns and one row from S_1. The last column of the left section, consisting of the Toeplitz matrix $T(f)$, and the last column of the right section, consisting of the Toeplitz matrix $T(g)$, are removed. Additionally, the last row of the matrix will be removed, which after the column deletions will have no non-zero entries.

$$S_2 = \begin{bmatrix} f_0 & & & g_0 & & \\ f_1 & f_0 & & g_1 & g_0 & \\ f_2 & f_1 & f_0 & g_2 & g_1 & g_0 \\ f_3 & f_2 & f_1 & g_3 & g_2 & g_1 \\ f_4 & f_3 & f_2 & g_4 & g_3 & g_2 \\ & f_4 & f_3 & & g_4 & g_3 \\ & & f_4 & & & g_4 \end{bmatrix}.$$

By removing the last columns from each section of S_2, as well as the last row, the second subresultant matrix S_3 is found. Continuing this pattern will compute the rest of the subresultant matrices.

$$S_3 = \begin{bmatrix} f_0 & & g_0 & \\ f_1 & f_0 & g_1 & g_0 \\ f_2 & f_1 & g_2 & g_1 \\ f_3 & f_2 & g_3 & g_2 \\ f_4 & f_3 & g_4 & g_3 \\ & f_4 & & g_4 \end{bmatrix}, \quad S_4 = \begin{bmatrix} f_0 & g_0 \\ f_1 & g_1 \\ f_2 & g_2 \\ f_3 & g_3 \\ f_4 & g_4 \end{bmatrix}.$$

When the polynomials that were used to construct the Sylvester matrix have a GCD, the index k of the first rank deficient subresultant matrix S_k is equal to the degree of the GCD. When an AGCD must be computed instead, the subresultant matrices, much like the Sylvester matrix, will be of full rank. Finding either the SVD or QR decomposition of all of the subresultant matrices allows for tests to be run to compute the matrices that are closer to being rank deficient.

Winkler investigated both the SVD and QR decomposition of the subresultant matrices for the estimation of the rank. It would be possible to compute the QR decomposition, or the SVD, of every subresultant matrix in order to compute the rank. However, this would be computationally expensive, and result in a large quantity of redundant computation.

Another advantage of the QR decomposition is the ability to perform QR updates when columns, or rows, are added or removed from S_1. QR updates can be used to amend the original upper triangular factor R_1, to compute the matrices $R_{2\ldots n}$ without full recomputation of the QR decomposition. By performing QR updates, and computing the factors of S_k from S_{k-1}, the complexity of the algorithm can be reduced from $O(n^3)$ to $O(n^2)$. While SVD update algorithms do exist [46], these are less ubiquitous, and often have problems with numerical stability. Various forms of the QR decomposition, and QR updates, will be discussed further in Chapter 3.

3.2 Literature Survey of AGCD Computation Methods

While this thesis will focus on the AGCD algorithm presented by Winkler, this is not the only method for performing AGCD computation. This section will give an overview of methods from the literature for computing AGCDs, which are sometimes referred to in the literature as ϵ-GCDs.

3.2.1 Euclid's algorithm

In 1985 Schönhage presented a method with which to compute a so called quasi-GCD of polynomials [47]. The idea presented of a quasi-GCD is related, but differs to the concept of an AGCD discussed in this thesis. In a situation where a quasi-GCD can be computed the coefficients of the input polynomials are exact, but only known to a limited precision. However, in an AGCD the exact values of the coefficients of the input polynomials may have been perturbed by noise. This method used an adaptation of Euclid's algorithm for the computation of GCDs. Euclud's algorithm was also used in Noda and Sasiki's method [48], and Hribernig and Stetter's method [49] which found clusters of roots of polynomials.

While these algorithms were significant at the time of publication, they have been largely superseded by structured matrix methods, and will not be described in detail in this thesis.

3.2.2 Matrix Methods

More recently focus has been on matrix methods for the computation of AGCDs. Two matrices are of particular interest when considering this computation: the Sylvester matrix, described in Section 3.1.1, and the Bézout matrix, described in Section 3.1.2.

Corless, Gianni, Trager, and Watt proposed a method of using the SVD of a Sylvester resultant matrix to compute an AGCD [50]. The Sylvester matrix is constructed in the same way as was described in Section 3.1.1, from polynomials $f(x)$ of degree m and $g(x)$ of degree n. The singular values $\sigma_{1 \ldots m+n}$ are computed, and the maximum value of k is found such that σ_k is above a specified error threshold and σ_{k+1} is below this threshold. This value of k can be used to find the degree of an AGCD. Corless et al. suggested four different methods with which the coefficients of an AGCD could be computed after the degree is computed. It is important to note that this algorithm requires the error present in the input polynomials to be known. If this is not the case Corless et al. suggest some methods in which an AGCD can still be computed, or negative results be returned.

Corless, Watt, and Zhi [51] proposed a QR method using structured linear total least norm (STLN) [52]. This method was shown to successfully compute AGCDs from input polynomials with degrees of up to 1020. This method, similar to that presented in [50], requires a tolerance level to be input alongside the polynomials such that an AGCD can be computed within that tolerance. This means that the method requires prior knowledge of the amount of error in the input polynomials. The algorithm starts by preprocessing the polynomials, scaling them by their 2-norms and ensuring the leading coefficient is positive.

The Sylvester matrix S is then formed with the two normalised polynomials $f(x)$ and $g(x)$, and it is split into the orthogonal and upper triangular factors through QR decomposition. Submatrices R_k of dimension $(k+1) \times (k+1)$ are extracted from the bottom right of the matrix for $k = 0, \ldots, m+n-1$. Each of these submatrices are tested by computing their norms, and finding the value of k such that the norm of R_k is greater than or equal to the specified error threshold, and the norm of R_{k-1} is less than the threshold. The top row of R_k will give the coefficients of an AGCD. The polynomials $f(x)$ and $g(x)$ are divided by the computed AGCD, and are tested to ensure they are coprime.

Zarowski, Ma, and Fairman also presented a QR method [53], which proposed manipulation of the Sylvester matrix to form a smaller problem size, to achieve a faster algorithm. The manipulation of the matrix resulted in an $n \times n$ upper triangular matrix when considering polynomials of equal length, while the method of using the QR decomposition of the full Sylvester matrix requires processing a $2n \times 2n$ matrix. While the algorithm proposed had similar complexity to that of the SVD algorithm presented by Corless, it was discussed that the constant factor in the time complexity of QR decomposition is lower than that of the SVD, and by decreasing the problem size the complexity of the algorithm is also reduced. This method would be difficult to adapt to the processing of subresultant matrices, as are discussed in this thesis, as the manipulation of the Sylvester matrix makes it computationally expensive to construct subresultant matrices in the smaller form proposed in this paper.

Bini and Boito proposed another method using Gaussian elimination on Sylvester and Bézout matrices, considering the displacement structure properties of these matrices [54]. The method involved transforming the Sylvester or Bézout matrix into a Cauchy-like matrix, and using Gaussian elimination on this to estimate the coefficients of a divisor of degree k, where k is arbitrarily decided. Optimisation is used to attempt to minimise the error of this divisor. If a divisor is found within specified bounds a degree of $k+1$ is tested, and this step is repeated until a divisor cannot be computed within the specified bounds. The last divisor found will be the actual result of an AGCD. If the first value of k does not return a divisor within the error limits then the process is repeated for $k-1$ until a divisor is found. This paper presented results for polynomial degrees of up to 500, and provided examples in which this algorithm could resolve AGCDs when the QR algorithm proposed by Corless et al. failed due to incorrect estimation of the degree. It should be noted that this implementation still requires a threshold to be set based on the noise in the inputs for an AGCD, and the repeated process of computation of divisors and optimisation is particularly expensive.

Li, Yang, and Zhi proposed a method of computing an AGCD by finding a low rank estimate of a Sylvester matrix by using the Schur algorithm and STLN [13]. While not naming them as such this method also used subresultant matrices, which shall be used in this thesis, and are described in Section 3.1.4. This method uses STLN to attempt to compute a divisor within a specified threshold for each potential degree k, and iterating through possible values of k until an AGCD is reached that satisfies the stated tolerance.

This research focused on the speed of execution, and the proposed algorithm achieved quadratic complexity.

Zeng proposed a method of computing an AGCD using an iterative algorithm using QR updates [45]. This method uses Sylvester subresultant matrices, as was described in Section 3.1.4. The algorithm iterates over subresultant matrices, starting at the original subresultant matrix, and computing the QR decomposition of each in sequence. For each subresultant matrix a divisor is attempted to be computed, moving over the potential degree values until a AGCD is computed that is below the specified threshold. The results shown in this paper demonstrated that this algorithm is able to find AGCDs in polynomials of up to 2000, suggesting a significant increase in the reliability of the computation of an AGCD from the use of subresultant matrices.

Winkler and Allan proposed using the QR decomposition of Sylvester subresultant matrices using STLN to solve the optimisation problem to find an AGCD [55]. This method first computes the degree, through QR decomposition of Sylvester subresultant matrices to compute the rank loss, and then uses STLN to compute a low rank estimate of the Sylvester matrix to compute the coefficients. Winkler and Hasan extended this research with the use of SNTLN [20], as opposed to STLN that had been used previously [21], to compute a low rank estimation of the Sylvester matrix and thus compute an AGCD. This was taken further by making the SNTLN method more robust in [56]. The work presented by Winkler et al. differs to that proposed previously. These methods separate out the degree computation from the computation of the coefficients, similarly to the work presented by Corless et al., but they also consider subresultant matrices, which have been shown to compute AGCDs more effectively than reliance on a single Sylvester or Bézout matrix. This work will be explained in detail in Section 4.1.

While processing subresultant matrices was shown to provide more robust algorithms, the process of analysing multiple large matrices, as is required by such an algorithm, is computationally expensive. While the research presented by Winkler et al. and Zeng attempted to solve this by using QR update algorithms, the computational expense is an issue that still must be addressed.

3.3 The QR Decomposition and QR Updates

QR decomposition, as was discussed in Section 3.1, is the process through which a matrix A is split into its factors, an orthogonal matrix Q and an upper triangular matrix R.

$$A = QR.$$

This decomposition is central to the computation of an AGCD in several of the algorithms discussed in Section 3.2, and is central to the algorithm investigated in this thesis. This section will therefore give an overview of this decomposition.

When a column or row is deleted from, or added to, the original matrix A, the Q

and R matrices can be updated to reflect this change. This is significantly more efficient than recomputation of the full decomposition, and gives the QR decomposition significant advantages when compared to the SVD. It should be noted that the complexity of a QR update is quadratic, and generally considered efficient. However, the scale of the matrices that are required to be processed for polynomials of high degree can make this process computationally expensive.

3.3.1 Full Decomposition

QR decomposition is the process through which a matrix A is split into its factors, Q, an orthogonal matrix, and R, an upper triangular matrix. This section will discuss the three main methods through which QR factorisation can be computed.

3.3.1.1 Givens Rotations

Givens rotations are the simplest method of computing the QR decomposition. In this method Givens rotations are used to introduce a single 0 at a time to the original matrix A to result in the upper triangular component R [57].

Consider a 5x5 matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}.$$

The Givens rotation begins at the bottom left corner of A. Firstly a Givens matrix G is calculated from the values a_{51} and a_{41}. G is constructed from the values γ and σ, which are calculated with the equations below.

$$\tau = \frac{a_{51}}{a_{41}}, \quad \gamma = \frac{1}{\sqrt{1+\tau^2}}, \quad \sigma = \gamma\tau.$$

The Givens matrix G, with elements $g_{i,j}$, where i and j are the row and column indices respectively, can now be constructed. Starting with an identity matrix, the dimensions of which are equal to that of A, the values for γ and σ are substituted into this matrix in positions related to the indices of the rows in A on which the rotation is taking place. In the matrix G, g_{xx} and g_{yy} will equal γ, g_{xy} will equal σ and g_{yx} will equal $-\sigma$, where x is equal to the index of the first row being considered, in this case 4, and y is equal to the index of the second row, in this case 5.

Thus the full rotation matrix for the first step will be

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \gamma & \sigma \\ 0 & 0 & 0 & -\sigma & \gamma \end{bmatrix}.$$

The matrix product of G_1 and A will give the first step of computing R. This in progress matrix will be known here as $\tilde{R}_1$.

$$R_1 = G_1 A.$$

As only the last two rows of the Givens matrix differ from the identity matrix, only the last two rows of $\tilde{R}_1$ will have changed from A in the matrix product of G_1 and A. The entry on the bottom column of the last row of $\tilde{R}_1$ will equal zero.

As the remainder of the Givens matrix has the same properties as the identity matrix, only two rows of A will actually be affected by the product of it and G_1. Due to this, a more compact form of the Givens matrix can be used, and applied just to the necessary rows of the matrix. The result of the matrix product with the compact matrix does not differ to that in the matrix product with the full matrix. It is, however, useful computationally, as the full matrix product is not required to be computed. The compact form of this matrix is shown below.

$$G_1 = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}.$$

By computing a matrix product of this compact form and the submatrix of A consisting only of rows 4 and 5, the result is a submatrix of $\tilde{R}_1$, the first step in computing R. This submatrix can be substituted into rows 4 and 5 of A to give $\tilde{R}_1$.

The further rotations will be performed sequentially, moving up the rows of $\tilde{R}_i$. The second rotation will be performed using the third and fourth rows of $\tilde{R}_1$ to compute $\tilde{R}_2$. This continues with the second and third rows of $\tilde{R}_2$ to compute $\tilde{R}_3$, and finally the first and second rows of $\tilde{R}_3$ to give $\tilde{R}_4$. This will leave the entry on the first row as the only non-zero value of the first column.

Once the first column has only zeros below the principal diagonal, the process is then repeated for the second column. Progressively introducing zeros up this column until all entries below the principal diagonal are zero. This process is repeated for all columns of $\tilde{R}_i$, where i is the index of the iteration. In each case, starting at the last row, and progressively performing rotations to introduce zeros in each column. The order of these operations is shown below, with $\bullet$ representing non-zero values in R, and the values below the diagonal representing the iteration in which a zero was introduced through Givens rotation into that position.

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ 4 & \bullet & \bullet & \bullet & \bullet \\ 3 & 7 & \bullet & \bullet & \bullet \\ 2 & 6 & 9 & \bullet & \bullet \\ 1 & 5 & 8 & 10 & \bullet \end{bmatrix}$$

Once this process has been completed for all columns in $\tilde{R}_i$, the result will be the upper triangular matrix R.

The matrix Q can be computed by successively calculating the matrix product of the transpose of the Givens matrices. This can be done at the time each Givens matrix is computed, but is sometimes simply performed after R has been fully computed. The equation for this is shown below.

$$Q = \prod_{i=0}^{T_{n-1}} G_i^T,$$

where n is the number of rows in the original matrix A, and T_{n-1} is the $(n-1)$th triangular number, which is computed with

$$T_n = \frac{(n-1)(n-2)}{2}.$$

Example

For this example, a random 3×3 matrix was generated in MATLAB. The full version of each Givens matrix will be shown here for clarity and mathematical completeness.

$$A = \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.5975 \end{bmatrix}.$$

The initial step in the first rotation is to calculate σ and γ from the first entries of the second and third rows.

$$\begin{aligned} \tau &= \frac{a_{31}}{a_{21}} &&= \frac{0.1270}{0.9058} &&= 0.1402, \\ \gamma &= \frac{1}{\sqrt{1+\tau^2}} &&= \frac{1}{\sqrt{1+0.1402^2}} &&= 0.9903, \\ \sigma &= \gamma\tau &&= 0.9903 \times 0.1402 &&= 0.1388. \end{aligned}$$

Now the Givens matrix can be constructed.

$$G_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \gamma & \sigma \\ 0 & -\sigma & \gamma \end{bmatrix},$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9903 & 0.1388 \\ 0 & -0.1388 & 0.9903 \end{bmatrix}.$$

This rotation can then be applied to the original matrix to give $\tilde{R}_1$.

$$\tilde{R}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9903 & 0.1388 \\ 0 & -0.1388 & 0.9903 \end{bmatrix} \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.5975 \end{bmatrix},$$
$$= \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9147 & 0.6398 & 0.6745 \\ 0 & 0.0088 & 0.8723 \end{bmatrix}.$$

The second Givens matrix will be constructed from the first and second rows, using entries from the first column.

$$\tau = \frac{\hat{r}_{21}}{\hat{r}_{11}} = \frac{0.9147}{0.8147} = 1.1227,$$
$$\gamma = \frac{1}{\sqrt{1+\tau^2}} = \frac{1}{\sqrt{1+0.1227^2}} = 0.6651,$$
$$\sigma = \gamma\tau = 0.6651 \times 1.1227 = 0.7467.$$

$$G_2 = \begin{bmatrix} 0.6651 & 0.7467 & 0 \\ -0.7467 & 0.6651 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

This rotation is applied to $\tilde{R}_1$ to give $\tilde{R}_2$.

$$\tilde{R}_2 = \begin{bmatrix} 0.6651 & 0.7467 & 0 \\ -0.7467 & 0.6651 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9147 & 0.6398 & 0.6745 \\ 0 & 0.0088 & 0.8723 \end{bmatrix},$$
$$= \begin{bmatrix} 1.2248 & 1.0852 & 0.6889 \\ 0 & -0.2565 & 0.2407 \\ 0 & 0.0088 & 0.8723 \end{bmatrix}.$$

As there are now no non-zero values below the principal diagonal of R in the first column, work must begin on the second column. This starts with the construction of the Givens matrix.

$$\begin{aligned}
\tau &= \frac{\hat{r}_{32}}{\hat{r}_{22}} &&= \frac{0.0088}{-0.2565} &&= -0.0343,\\
\gamma &= \frac{1}{\sqrt{1+\tau^2}} &&= \frac{1}{\sqrt{1+(-0.0343)^2}} &&= 0.9994,\\
\sigma &= \gamma\tau &&= -0.0343 \times 0.9994 &&= -0.0343.
\end{aligned}$$

$$G_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9994 & -0.0343 \\ 0 & 0.0343 & 0.9994 \end{bmatrix}.$$

Applying this rotation to $\tilde{R}_2$ provides the final form of R.

$$\begin{aligned}
R &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9994 & -0.0343 \\ 0 & 0.0343 & 0.9994 \end{bmatrix} \begin{bmatrix} 1.2248 & 1.0852 & 0.6889 \\ 0 & -0.2565 & 0.2407 \\ 0 & 0.0088 & 0.8723 \end{bmatrix},\\
&= \begin{bmatrix} 1.2248 & 1.0852 & 0.6889 \\ 0 & -0.2567 & 0.2106 \\ 0 & 0 & 0.8800 \end{bmatrix}.
\end{aligned}$$

All entries below the principal diagonal of R are now equal to zero, and all of the Givens matrices have been computed. The orthogonal matrix Q can therefore be computed by sequentially calculating the matrix product of the transpose of each Givens matrix.

$$Q = (G_1^T G_2^T) G_3^T = \begin{bmatrix} 0.6651 & -0.7463 & -0.0256 \\ 0.7395 & 0.6631 & -0.1162 \\ 0.1037 & 0.0583 & 0.9929 \end{bmatrix}.$$

3.3.1.2 Householder Reflections

Householder reflections, also called Householder transformations, are a method designed to reduce the number of operations required to compute the QR decomposition when compared to Givens rotations. This is accomplished by introducing zeros to entire columns of the upper triangular matrix R per matrix product computation [58].

Considering the same matrix A from Section 3.3.1.1

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}.$$

Each reflection is calculated from its respective column from A, using the entries from the principal diagonal to the last row in every column. This column vector shall be called c_i, where i is the index of the column being processed. A Householder matrix H_i, is constructed by first taking the Euclidean norm of this column. This is added to the first value of c_i to compute the vector v_i,

$$v_i = \begin{bmatrix} c_{i1} + \|c_i\|_2 \\ c_{i2} \\ \vdots \\ c_{in} \end{bmatrix}.$$

The vector v_i is then used to create a Householder transformation by calculating

$$H_i = I - cv_iv_i^T,$$

where $c = {}^2/_{v_i^T v_i}$ and I is the identity matrix with the same dimensions as A.

The product of the Householder matrix and the original matrix A is computed, which gives the first stage of the computation of R, with all the necessary zeros introduced to the first column. The state of this matrix after the first transformation has been completed will be referred to here as $\tilde{R}_1$. Once this computation has been completed a new vector v_2 can be computed, using entries from the second column, below the principle diagonal, of $\tilde{R}_1$. This is used to construct a Householder matrix H_2, and the product of this with $\tilde{R}_1$ gives the matrix $\tilde{R}_2$. Continuing in this pattern for $n-1$ reflections will compute the final upper triangular matrix R.

The product of the transpose of successive Householder matrices H_i will give the final result for Q, similar to the method used with Givens rotations.

$$Q = \prod_{i=0}^{n-1} H_i^T.$$

Householder reflections provide for a significantly more efficient computation of the full QR decomposition than Givens rotations, while maintaining the numerical stability. Givens rotations are approximately twice as expensive as Householder reflections [59].

Example

This example shall use the same matrix A as used in the Givens rotation example.

$$A = \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.5975 \end{bmatrix}.$$

The first transformation shall be performed on the first column of A which will here be referred to as a_1.

$$a_1 = \begin{bmatrix} 0.8147 \\ 0.9058 \\ 0.1270 \end{bmatrix} \quad , \quad \|a_1\|_2 = 1.2249.$$

$$v_1 = \begin{bmatrix} 0.8147 + 1.2249 \\ 0.9058 \\ 0.1270 \end{bmatrix} = \begin{bmatrix} 2.0396 \\ 0.9058 \\ 0.1270 \end{bmatrix}.$$

The Householder transformation will be calculated from this vector.

$$c = \frac{2}{v_1^T v_1} = 0.4003,$$

$$\begin{aligned} H_1 &= I - c v_1 v_1^T, \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 0.4003 \left(\begin{bmatrix} 2.0396 \\ 0.9058 \\ 0.1270 \end{bmatrix} \begin{bmatrix} 2.0396 & 0.9058 & 0.1270 \end{bmatrix} \right), \\ &= \begin{bmatrix} -0.6651 & -0.7395 & -0.1037 \\ -0.7395 & 0.6716 & -0.0460 \\ -0.1037 & -0.0460 & 0.9935 \end{bmatrix}. \end{aligned}$$

Applying this to the original matrix gives the first matrix in the computation of R, here known as $\tilde{R}_1$

$$\begin{aligned} \tilde{R}_1 &= \begin{bmatrix} -0.6651 & -0.7395 & -0.1037 \\ -0.7395 & 0.6716 & -0.0460 \\ -0.1037 & -0.0460 & 0.9935 \end{bmatrix} \begin{bmatrix} 0.8147 & 0.9143 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.5975 \end{bmatrix}, \\ &= \begin{bmatrix} -1.2249 & -1.0853 & -0.6889 \\ 0 & -0.2552 & 0.1173 \\ 0 & -0.0269 & 0.8973 \end{bmatrix}. \end{aligned}$$

The process is now repeated for the second column from the principal diagonal down to the last row.

$$a_2 = \begin{bmatrix} -0.2552 \\ -0.0269 \end{bmatrix}, \quad \|a_2\|_2 = 0.2566.$$

To preserve the 3×3 structure of the upper triangular matrix, zero can be prepended to the start of v_2.

$$v_2 = \begin{bmatrix} 0 \\ -0.2552 + 0.5566 \\ -0.0269 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.0014 \\ -0.0269 \end{bmatrix},$$

$$c = \frac{2}{v_2^T v_2} = 2757.4,$$

$$\begin{aligned} H_2 &= I - cv_1v_1^T, \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 2757.4 \left(\begin{bmatrix} 0 \\ 0.0014 \\ -0.0269 \end{bmatrix} \begin{bmatrix} 0 & 0.0014 & -0.0269 \end{bmatrix} \right), \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9945 & 0.1048 \\ 0 & 0.1048 & -0.9945 \end{bmatrix}. \end{aligned}$$

The product of H_2 and $\tilde{R}_1$ gives the final value for R.

$$\begin{aligned} R &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9945 & 0.1048 \\ 0 & 0.1048 & -0.9945 \end{bmatrix} \begin{bmatrix} -1.2249 & -1.0853 & -0.6889 \\ 0 & -0.2552 & 0.1173 \\ 0 & -0.0269 & 0.8973 \end{bmatrix}, \\ &= \begin{bmatrix} -1.2249 & -1.0853 & -0.6889 \\ 0 & -0.2566 & 0.2107 \\ 0 & 0 & -0.8801 \end{bmatrix}. \end{aligned}$$

To compute Q, the formula described previously is used. Starting with the transpose of the first Householder matrix H_1, the progressive product of the transpose of each Householder matrix H_i computed. Once the product of the final matrix H_{n-1} is computed the resulting orthogonal matrix Q will have been computed.

In this simple example only two transformations were required, and so the matrix Q can be computed by

$$
\begin{aligned}
Q &= H_1^T H_2^T, \\
&= \begin{bmatrix} -0.6651 & -0.7395 & -0.1037 \\ -0.7395 & 0.6716 & -0.0460 \\ -0.1037 & -0.0460 & 0.9935 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.9945 & 0.1048 \\ 0 & 0.1048 & -0.9945 \end{bmatrix}, \\
&= \begin{bmatrix} -0.6651 & -0.7463 & 0.0256 \\ -0.7395 & 0.6631 & 0.1161 \\ -0.1037 & 0.0584 & -0.9929 \end{bmatrix}.
\end{aligned}
$$

3.3.1.3 Gram-Schmidt Orthogonalisation

In Gram-Schmidt orthogonalisation either the orthogonal matrix Q, or the upper triangular matrix R, can be computed initially, and whichever is computed can be used to find the other. For this section the matrix Q will be computed first. Firstly, the orthonormal basis is computed using all the columns of A, those being referred to as a_i where $i = 1 \ldots n$ and n is the number of columns in the matrix. These vectors are used to compute the column vectors that make up the orthogonal matrix Q [60].

The main disadvantage of Gram-Schmidt orthogonalisation is a lack of stability when compared to Householder and Givens methods. While modified and stabilised versions of the Gram-Schmidt orthogonalisation have been developed, these are still not as stable as the Householder and Givens methods. The main advantage, however, is that when only a partial orthogonalisation is necessary, where only certain columns of Q, or rows of R are required, the Gram-Schmidt process is able to compute only these columns or rows individually. Both Householder and Givens methods on the other hand need to progressively compute rows of R and columns of Q in sequence. This makes Gram-Schmidt suitable for certain iterative algorithms, where stability is not a major concern.

Example

Consider the same 3×3 matrix A from Section 3.3.1.1.

$$
A = \begin{bmatrix} 0.8147 & 0.9134 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.9575 \end{bmatrix}.
$$

Firstly the orthonormal basis needs to be calculated. The matrix is separated into three column vectors, a_1 being the first column, a_2 being the second and a_3 being the third.

The vectors that make up the orthonormal basis are calculated sequentially. The projection of each vector onto the previously calculated vector is subtracted from the original vector. This gives the vectors $\tilde{q}_1$, $\tilde{q}_2$, and $\tilde{q}_3$. Dividing each of these vectors by its norm gives the vectors q_1, q_2, and q_3 which together form the matrix Q. The matrix

product of the transpose of Q and the matrix A gives the upper triangular matrix R.

The vector $\tilde{q}_1$ is equal to a_1, and thus q_1 is simply a_1 divided by its norm, ensuring the resulting vector has a norm of 1.

$$q_1 = \frac{a_1}{\|a_1\|} = \begin{bmatrix} 0.6651 \\ 0.7395 \\ 0.1037 \end{bmatrix}.$$

The vector $\tilde{q}_2$ is calculated using the following formula.

$$\begin{aligned} \tilde{q}_2 &= a_2 - (a_2^T q_1) q_1, \\ &= \begin{bmatrix} 0.9134 \\ 0.6324 \\ 0.0975 \end{bmatrix} - \left(\begin{bmatrix} 0.9134 \\ 0.6324 \\ 0.0975 \end{bmatrix} \begin{bmatrix} 0.8147 \\ 0.9058 \\ 0.1270 \end{bmatrix} \right) \begin{bmatrix} 0.8147 \\ 0.9058 \\ 0.1270 \end{bmatrix} = \begin{bmatrix} 0.1915 \\ -0.1702 \\ -0.0150 \end{bmatrix}. \end{aligned}$$

And again, to ensure the vector has a norm of 1, $\tilde{q}_2$ is divided by its norm to give

$$q_2 = \begin{bmatrix} 0.7463 \\ -0.6631 \\ -0.0583 \end{bmatrix}.$$

Finally, to calculate $\tilde{q}_3$,

$$\tilde{q}_3 = a_3 - (a_3^T q_1) q_1 - (a_3^T q_2) q_2 = \begin{bmatrix} -0.0225 \\ -0.1022 \\ 0.8738 \end{bmatrix}.$$

And again, dividing $\tilde{q}_3$ by its norm results in

$$q_3 = \begin{bmatrix} -0.0256 \\ -0.1162 \\ 0.9926 \end{bmatrix}.$$

The matrix Q is constructed by combining these column vectors,

$$Q = \begin{bmatrix} 0.6651 & 0.7463 & -0.0256 \\ 0.7395 & -0.6631 & -0.1162 \\ 0.1037 & -0.0583 & 0.9926 \end{bmatrix}.$$

Finally, to compute R, the equation $A = QR$ can be rearranged to give $R = Q^T A$. Thus the factor R is calculated.

$$R = \begin{bmatrix} 0.6651 & 0.7463 & -0.0256 \\ 0.7395 & -0.6631 & -0.1162 \\ 0.1037 & -0.0583 & 0.9926 \end{bmatrix}^T \begin{bmatrix} 0.8147 & 0.9134 & 0.2785 \\ 0.9058 & 0.6324 & 0.5469 \\ 0.1270 & 0.0975 & 0.9575 \end{bmatrix},$$

$$= \begin{bmatrix} 1.2249 & 1.0853 & 0.6889 \\ 0 & 0.2566 & -0.2106 \\ 0 & 0 & 0.8800 \end{bmatrix}.$$

Computationally there are two implementations of Gram-Schmidt. These are commonly known as Classical Gram-Schmidt and Modified Gram-Schmidt. When implementing Classical Gram-Schmidt rounding errors can cause instability in the results, leading to a non-orthogonal Q. The modified Gram-Schmidt algorithm addresses some of these problems by performing computations on computed values, as opposed to stored values, thus minimising the error. Most implemented algorithms are based on the modified Gram-Schmidt. As was discussed at the beginning of this section, Gram Schmidt can also be used to compute R before Q, by performing the Gram Schmidt process on the rows, as opposed to the columns used in this example.

3.3.2 QR Column Deletions

One of the features of QR decomposition is the ability to update the Q and R matrices upon changes to the original matrix A. These changes can be in four forms: column deletion, column insertion, row deletion, and row insertion. QR updates are also known as reorthogonalisation. The fact that the QR decomposition can be updated gives the QR decomposition an advantage over SVD. This is due to the fact that the complexity of a QR update is quadratic, while the full QR decomposition and SVD are both cubic.

This section will discuss various techniques of performing QR column deletion, as this is the only form of update that is necessary in this thesis.

3.3.2.1 Givens Rotations

Givens rotations provide a relatively simple update algorithm for column deletions, following largely the same pattern as the original full decomposition [58]. Consider the matrix A which has the upper triangular factor R. Columns 2 and 3 are removed from A to give the matrix B.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}, \quad B = \begin{bmatrix} a_{11} & a_{14} & a_{15} \\ a_{21} & a_{24} & a_{25} \\ a_{31} & a_{34} & a_{35} \\ a_{41} & a_{44} & a_{45} \\ a_{51} & a_{54} & a_{55} \end{bmatrix}.$$

As these columns have been removed, the upper triangular factor of A, R_A, is not a

factor of B. The aim of the update algorithm is to compute the new upper triangular matrix R_B, which is a factor of B, from R_A. The same columns as were removed from A to construct B are removed from R_A to give R_B. In this case these columns are 2 and 3. This in-progress upper triangular matrix will be known as $\tilde{R}_{B1}$.

$$R_A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} \\ 0 & 0 & r_{33} & r_{34} & r_{35} \\ 0 & 0 & 0 & r_{44} & r_{45} \\ 0 & 0 & 0 & 0 & r_{55} \end{bmatrix}, \quad R_{B1} = \begin{bmatrix} r_{11} & r_{14} & r_{15} \\ 0 & r_{24} & r_{25} \\ 0 & r_{34} & r_{35} \\ 0 & r_{44} & r_{45} \\ 0 & 0 & r_{55} \end{bmatrix}.$$

Note that in the matrix R_{B1} there are non-zero entries below the principal diagonal. Similarly to the full decomposition performed with the Givens rotation as described in Section 3.3.1.1, the Givens rotations will start from the left most column of R_{B1} and move right across the matrix.

The first column of R_{B1} contains no non-zero entries below the principal diagonal, and therefore requires no update. The second column has two values below the principal diagonal, which means two rotations are required. For the first rotation, a Givens matrix can be constructed from the values r_{44} and r_{34} of column 2. Similarly to in the full decomposition Givens rotations are performed starting with the first column with non-zero entries below the principal diagonal, moving up each column in sequence.

Once the four necessary rotations have taken place, R_B will have been computed.

$$R_B = \begin{bmatrix} r_{11} & r_{14} & r_{15} \\ 0 & r_{B22} & r_{B23} \\ 0 & 0 & r_{B33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Note that only the values r_{Bxy} have changed, and the values on the top row remain the same as in the original matrix R_{A1}. Only entries to the right of the deleted column will change, and only rows below the principal diagonal of the column prior to the deleted column will change. In this case, this means that the first row of R_B remains the same from R_A, with the exception of the removed columns. It is also worth noting that deletion of the last column of a matrix will require no Givens rotations in order to update the matrix, and simply removing the column will suffice.

Again, similarly to the full decomposition, the updated orthogonal matrix Q_B for the matrix B can be computed by applying the inverse of each rotation in order to the original orthogonal matrix Q_A. The dimensions of the Q matrices stay constant, therefore no columns or rows need to be removed.

3.3.2.2 Householder Reflections

QR column deletion through Householder reflections are performed in the same way as in the full decomposition when using Householder reflections [58]. Using the same A and B matrices as in Section 3.3.2.1 the first column of R_{B1} needs no update, and thus the first reflection will be applied using entries from the second column.

The vector v_2 can be computed from the second column of R_{B1}, using the entries below the principal diagonal. Moving right along the columns of the matrix, and progressively computing and applying the Householder reflection to the in progress R matrix. This will result in the updated matrix R_B.

The orthogonal matrix Q_B can be computed from Q_A, by applying the computed Householder reflections to Q_A in the same way that they are applied in the full decomposition.

Householder reflections for updates have similar benefits over Givens updates that were discussed in Section 3.3.1.2, but when processing single column deletions this advantage is not present, as the Householder transformation and Givens rotation will essentially reduce to the same computation with deletions of this size.

3.3.2.3 Gram Schmidt

The Gram Schmidt algorithm is not well suited for updating algorithms, and thus is not widely used. Daniel, Gragg, Kaufman, and Stewart, in their 1976 paper [61], investigated the use of both Givens rotations and Gram Schmidt reorthogonalisation to update a matrix after each of the four types of Givens update. The approach used here for column deletions, that are the prime focus of this thesis, is simply reorthogonalisation through Givens rotations, as was described in Section 3.3.2.1. Thus Gram Schmidt will not be considered for the task of QR updating in this thesis.

3.4 Literature Survey of Parallel QR Factorisation Methods

This section will discuss low-level methods of parallelising the methods for the computation of a QR decomposition, and also for computing QR updates discussed previously in this chapter. While some of these techniques were not originally designed with GPU hardware in mind, the techniques in question could still have potential implications for how the computations can be parallelised on a GPU.

3.4.1 Parallelisation of QR Decomposition

QR decomposition presents a difficult problem for parallelisation. While all forms of QR decomposition require large matrix operations, that are relatively trivial to parallelise, all forms of QR decomposition are iterative processes. Therefore the ordering of the operations is important for parallelisation. Despite this difficulty, the parallelisation of QR decompositions has been a widely researched topic, with attempts having been made

to parallelise the three main methods of full decomposition described in Chapter 4, as well as parallelising the update algorithms.

3.4.1.1 Givens Rotations

The earliest example of parallelisation of QR deconvolution is the Sameh and Kuck method, proposed in their 1978 paper [62]. This method uses an iterative pattern, which gradually gains more parallelism over the course of the iterations. In the first iteration only one rotation occurs, on the last two rows of the matrix.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}.$$

Starting with the matrix A, rotations will be performed, introducing a zero to the matrix starting in the bottom left corner. The first rotation will be on the last two rows, as highlighted. This will give the first stage in the computation of R, $\tilde{R}_1$.

$$\tilde{R}_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ \tilde{r}_{41} & \tilde{r}_{42} & \tilde{r}_{43} & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & \tilde{r}_{52} & \tilde{r}_{53} & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix}.$$

The second iteration will again perform a single rotation, moving one row up from the previous iteration, introducing another 0 in the first column.

$$\tilde{R}_2 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \tilde{r}_{31} & \tilde{r}_{32} & \tilde{r}_{33} & \tilde{r}_{34} & \tilde{r}_{35} \\ 0 & \tilde{r}_{42} & \tilde{r}_{43} & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & \tilde{r}_{52} & \tilde{r}_{53} & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix}.$$

In the third iteration multiple rotations can be performed in parallel. Values for the last two rows have been calculated, the entries in the first column of these rows both equal zero. This means that the rotation starting from the second column can be performed. Therefore, the third rotation on the first column, and the first rotation on the second column, can be performed simultaneously.

$$\tilde{R}_3 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ \tilde{r}_{21} & \tilde{r}_{22} & \tilde{r}_{23} & \tilde{r}_{24} & \tilde{r}_{25} \\ 0 & \tilde{r}_{32} & \tilde{r}_{33} & \tilde{r}_{34} & \tilde{r}_{35} \\ 0 & \tilde{r}_{42} & \tilde{r}_{43} & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & 0 & \tilde{r}_{53} & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix}.$$

In the fourth iteration two rotations are run simultaneously again, introducing two more zeros, in this iteration the final entries of the top row of R .

$$\tilde{R}_4 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & \tilde{r}_{22} & \tilde{r}_{23} & \tilde{r}_{24} & \tilde{r}_{25} \\ 0 & \tilde{r}_{32} & \tilde{r}_{33} & \tilde{r}_{34} & \tilde{r}_{35} \\ 0 & 0 & \tilde{r}_{43} & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & 0 & \tilde{r}_{53} & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix}.$$

In the fifth iteration the rotations on the first column are complete. However, rotations can now begin on the third column, as the last two entries of the second column are equal to zero. This pattern continues for the next three iterations, until the matrix is upper triangular in form.

$$\tilde{R}_5 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} \\ 0 & 0 & \tilde{r}_{33} & \tilde{r}_{34} & \tilde{r}_{35} \\ 0 & 0 & \tilde{r}_{43} & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & 0 & 0 & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix},$$

$$\tilde{R}_6 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} \\ 0 & 0 & r_{33} & r_{34} & r_{35} \\ 0 & 0 & 0 & \tilde{r}_{44} & \tilde{r}_{45} \\ 0 & 0 & 0 & \tilde{r}_{54} & \tilde{r}_{55} \end{bmatrix},$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} \\ 0 & 0 & r_{33} & r_{34} & r_{35} \\ 0 & 0 & 0 & r_{44} & r_{45} \\ 0 & 0 & 0 & 0 & r_{55} \end{bmatrix}.$$

After the seventh iteration, the final upper triangular matrix R has been computed, with all entries below the principal diagonal being equal to zero.

While this small 5×5 matrix only resulted in only two Givens rotations at maximum able to be computed simultaneously, this increases with larger matrices. Take for example the 8×8 matrix shown below. In this matrix the numbers show the iteration in which a

Givens rotation can introduce a zero in its position, with the rest of the entries represented by $\bullet$ showing spaces in the final matrix that will be populated by non-zero values.

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 7 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 6 & 8 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 5 & 7 & 9 & \bullet & \bullet & \bullet & \bullet & \bullet \\ 4 & 6 & 8 & 10 & \bullet & \bullet & \bullet & \bullet \\ 3 & 5 & 7 & 9 & 11 & \bullet & \bullet & \bullet \\ 2 & 4 & 6 & 8 & 10 & 12 & \bullet & \bullet \\ 1 & 3 & 5 & 7 & 9 & 11 & 13 & \bullet \end{bmatrix}$$

The final R matrix takes $2n - 1$ iterations to complete, with the maximum number of simultaneous rotations being $\lfloor n/2 \rfloor$. The orthogonal matrix Q can be computed in the same number of iterations, with the same rotation on both Q and R occurring in the same iteration.

McGraw-Herdeg implemented the Sameh and Kuck method on a GPU in 2007. He used an NVIDIA GTX 8800 [63]. While this GPU is not very powerful compared to modern GPUs, the implementation achieved a speedup of 2.6 times compared to leading CPU implementations at the time. More recently Marcellino and Nevarra presented a method of computing the SVD on the GPU using QR decomposition through the Sameh and Kuck method [64]. They reported a speedup of up to 50 times when compared to a CPU implementation.

Further optimisations

While the Sameh and Kuck method maximises the number of Givens rotations that can be computed simultaneously, there are other methods of providing further parallelism. The simplest of these methods is to parallelise the matrix products involved in the rotations.

A more involved solution was proposed by Hofmann and Kontoghiorghes in their 2006 paper [65]. In this implementation partially computed rows were used in the computation of rows above and below, in a pattern they called pipelining. The pipelined algorithm achieved greater levels of parallelism compared to the original Sameh and Kuck algorithm, as well as presenting an alternative parallel implementation for situations where fewer processors are available. The resulting algorithm is capable of solving QR decompositions on tall and skinny matrices in half the time of the original Sameh and Kuck algorithm.

3.4.1.2 Householder Reflections

The most widely used form of parallel Householder reflections are blocked Householder reflections, originally proposed by Rotella and Zabettakis in their 1999 paper [66].

The method proposed an extension of the original Householder transformation, in which the original matrix A is split into submatrices, which can be processed in paral-

lel. Kerr, Campbell and Richards presented an implementation of blocked Householder matrices on GPUs [67], and achieved speedups of up to 5 times faster than high performance CPU libraries at the time.

3.4.1.3 Gram Schmidt

The Gram Schmidt method, in addition to the numerical instability reported in Section 3.3.2.3, is not well suited to parallelisation, particularly on a GPU, as it would require a significant amount of thread synchronisation [67].

3.4.1.4 Parallel QR updates

Andrew and Dingle investigated methods of implementing QR updates on GPUs [68]. This paper was the first to implement QR updates on a GPU. While this paper investigated all forms of parallel deletions, the only form of update of interest in this thesis is the QR column deletion. Andrew and Dingle tested both Householder and Givens methods of QR updates. Their findings suggested that for column deletions where large numbers of columns are removed at a time, the blocked Householder methods were significantly faster. However, when fewer columns are removed the advantage of the blocked Householder implementation has over the Givens implementation is reduced. This is expected, as when single columns are removed the matrix computations required of the Householder and Givens methods reduce to the same computation.

3.5 Conclusion

This chapter gave a background into the mathematics necessary to compute AGCDs, as well as an overview of techniques used for this computation in the literature. As was noted in Section 3.2, the algorithm presented by Winkler et al., which was used in the BID algorithm presented in [1], provides state of the art results for this computation. Unfortunately these results also come at significant computational expense.

Reducing the runtime of this algorithm would be of benefit to this algorithm, but to do so the algorithm must be explored in greater detail. While Section 3.4 gave an overview of techniques in the literature for the parallelisation of the QR methods discussed in Section 3.3. To understand how these methods could be applied to the BID algorithm proposed by Winkler it is necessary to investigate this algorithm in more detail. Therefore, in the next chapter, a more detailed analysis of this algorithm shall be provided.

Chapter 4

Polynomial Blind Image Deconvolution

This chapter will discuss in detail the polynomial method of blind image deconvolution proposed by Winkler [1]. The main premise of this algorithm is to consider the rows and columns of the blurred image as polynomials, with the pixel values being the coefficients. The vertical and horizontal elements of a separable PSF convolved with this blurred image can then be considered to be a GCD of these polynomials. Unfortunately, when the blurred image contains noise, an exact GCD cannot be computed. Thus an AGCD must be computed instead, as was described in Chapter 3. In an AGCD the coefficients must be perturbed slightly to account for noise in the coefficients of the input polynomials.

Section 4.1 will provide an overview of the BID algorithm from a high level, discussing the sections of the AGCD computation, and how these fit into the overall BID algorithm. Section 4.2 will discuss how the degree is computed using the Sylvester matrix and QR decomposition that were discussed in Chapter 3. This section will also discuss a change made to the degree computation to aid with reliability. Section 4.3 will provide a brief overview of how the coefficients are computed with the AGCD algorithm. Section 4.4 will provide three sample results from the deconvolution algorithm. Section 4.5 will discuss how the BID algorithm can be parallelised on a CPU. Finally, Section 4.6 will discuss the profiling results of the serial implementation, and identify the section of this algorithm that acceleration would benefit most.

4.1 Algorithm Overview

The most significant section of the BID algorithm proposed by Winkler is the computation of an AGCD. This section will give an overview of the AGCD computation. The computation of an AGCD in the algorithm proposed by Winkler can be broken down into two parts, the computation of a degree of an AGCD, and the computation of the coefficients.

The computation of the degree is the main focus of this thesis, as degree computation is the most computationally expensive part of the algorithm, as will be shown in Section 4.6.

Therefore this chapter will focus more on understanding this section, which is explained in detail in Winkler and Hasan's 2012 paper [21]. The degree computation can be further broken down into several sections. Firstly, the Sylvester matrix S_1 is constructed from the polynomials in question, $f(y)$ and $g(y)$, then the upper triangular factor of this Sylvester matrix R_1 is computed through QR decomposition. Then the upper triangular factors of all subresultant matrices of the Sylvester matrix $R_{2...n}$ must be computed. This is a computationally expensive process of progressive QR column deletions from the original upper triangular matrix R_1.

After each upper triangular factor of a subresultant matrix has been computed, it must be tested to estimate its rank. Two tests are performed, each of these tests will estimate the rank of the Sylvester matrix S_1. The first is to find ratio of the minimum and maximum values for the diagonals of each matrix $R_{1...n}$, and the second is to find the ratio of the minimum and maximum row norm values of each matrix $R_{1...n}$. The gradient is computed across all ratios in both tests, and the index of the minimum gradient provides an estimate of the rank of the Sylvester matrix, and thus the degree.

The process of computing an estimate of the degree, from the selection of the rows (or columns) to the computation of the minimum gradient, is repeated an unspecified number of times for different pairs of polynomials. These pairs of polynomials that are used to compute a single estimate of the degree will henceforth be referred to as trials. Computing multiple trials allows the computation of the degree to be more robust, with the modal value from all trials representing the final degree.

The coefficient computation, while discussed in less detail than the degree in this thesis, will still be optimised and accelerated in Chapter 8. The first stage of computing the coefficients is to select a pair of rows (or columns) with which to compute the coefficients, the original method proposed by Winkler suggests that this trial should be selected from the set of pairs of polynomials that returned the correct degree during the degree computation, and suggested that the first of these trials would suffice. Experimentally an improved method was found for the selection of the trial, that provides more reliable results. This improved method shall be discussed in Section 4.3.

Winkler's work has considered two different coefficient computation methods, firstly using the Sylvester matrix, and secondly using approximate polynomial factorisation. In this thesis the focus will be on the Sylvester method. After a pair of polynomials are selected, they are scaled and normalised in the same way as during the degree computation, and the pth Sylvester subresultant matrix $S_p(f, g)$ is formed, where p is equal to the degree computed previously.

The problem of computing the optimal coefficients of the AGCD is a non-linear optimisation problem. The method employed by Winkler is that of SNTLN [20], an extension of STLN [52]. This is an iterative method, through which a low rank estimation of the Sylvester matrix, and thus the coefficients of the AGCD, can be computed.

The algorithm presented by Winkler [69] uses a modified form of the SNTLN [20]. The modifications made by Winkler and Hasan involve the computation of an optimal column

of the subresultant matrix $S_p(f, g)$. This optimal column is defined as the column c_o, for which the residual of an approximate linear algebraic equation is minimised when column c_o is moved to the left side of the matrix [69]. This modification was made to ensure the computation of an AGCD of the polynomials $f(x)$ and $g(x)$ provided the correct result, even if the Toeplitz matrices that were used to construct the Sylvester matrix were swapped. This is not the case in the original SNTLN method. This modification, while improving the robustness of SNTLN, is computationally expensive, and therefore will be the focus of the acceleration and optimisation investigated in Chapter 8.

After AGCDs have been computed for both the rows and the columns, these need to be deconvolved from the original image. This is described in [12], and will not be considered for optimisation in this thesis, as it does not take a significant amount of time to compute, so will not be covered in detail here.

4.2 Degree Computation

The most computationally expensive section of the algorithm, as will be shown in section 4.6, is the computation of the degree of an AGCD. In the algorithm proposed by Winkler, this is performed by computing a numerical rank of the Sylvester matrix constructed from two polynomials. This section will discuss this algorithm in detail.

4.2.1 Algorithm

This section will describe the original implementation of the degree computation in Winkler's 2012 paper [21]. While the overall algorithm remains very similar to the original proposed version, several modifications were made for the purpose of efficiency. These changes will be highlighted in this section. The algorithm can be broken down into multiple stages, thus this section is divided into subsections discussing each of these stages. Figure 4.1 shows an overview of the algorithm, from a high level.

4.2.1.1 Selection of Rows or Columns

As the PSF is assumed to be separable, rows and columns are to be considered separately. The first step in computing the degree is to select row (or column) vectors from the pixel values of the original image. While it would be possible to get an estimate for the degree from a single pair of vectors, the degree of the AGCD cannot be computed with certainty, particularly when high levels of noise are present. Therefore multiple pairs of row or column vectors are used, and the modal degree estimation from these vectors is found, to give a final estimate for the degree. While the number of trials can be scaled arbitrarily, experimentally it was shown that 20 to 25 trials generally produced good results, even at high levels of noise.

Selecting values from the edge area of the convolved image, as described in Chapter 2, could potentially lead to problems. These pixels are influenced by their proximity to

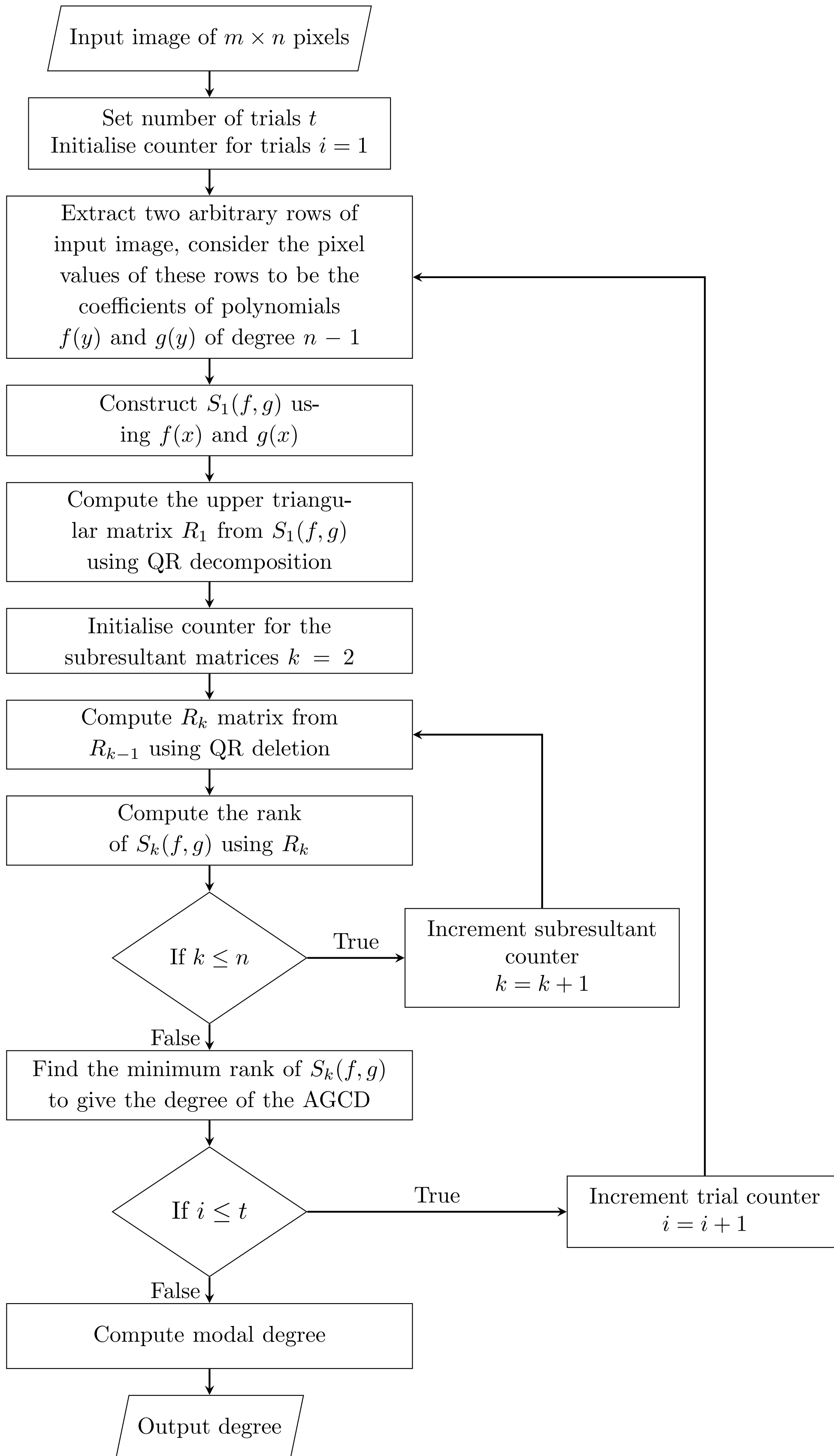


Figure 4.1: Flowchart showing the serial process of the degree estimation algorithm

the border, and thus appear darker than the remainder of the image. Due to this they do not contain enough information to compute the degree, therefore the vectors should be selected from a central portion of the image.

It is unfortunately impossible to know the width of the border area, without first knowing the degree of the AGCD. Therefore bounds must be placed on the rows and columns eligible for selection such that the border area is likely to be avoided. However, enough of the rows and columns must remain to be able to extract the required number of columns and rows for the number of trials. In this implementation the central third of both the rows and columns are assumed to be outside of the edge area, as when the border area is inside these regions the image is likely to be too blurred to be deconvolved anyway. Therefore, only this section will be used to select columns.

Figure 4.2 shows a large image split into thirds in both dimensions. Rows outside of the central sections of these the two lines in each dimension will not be considered.

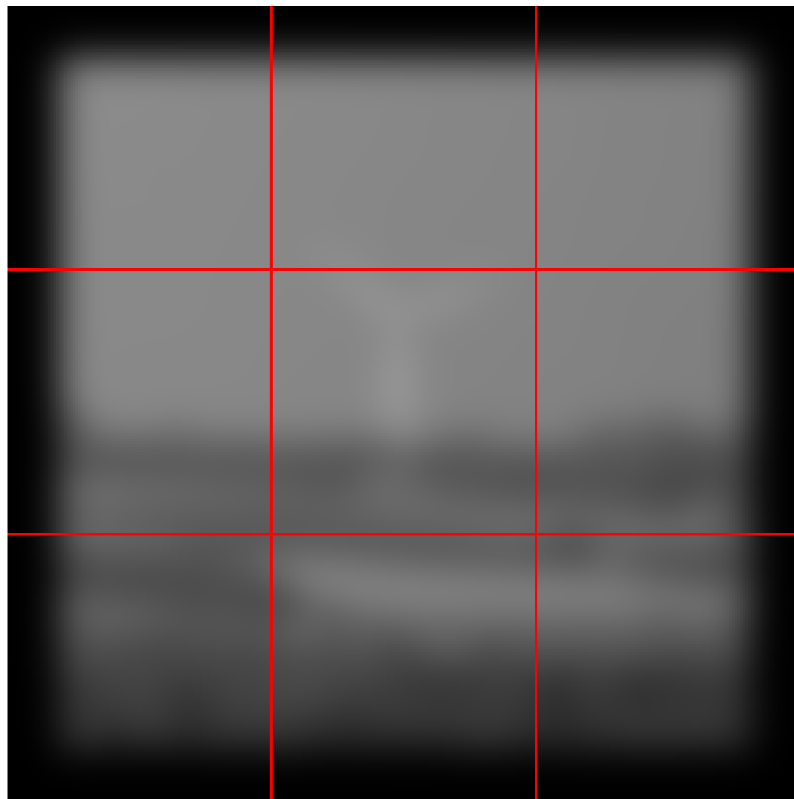


Figure 4.2: Blurred image with red lines to show the areas excluded from consideration when selecting rows and columns

An extra precaution against problematic rows or columns was included in this implementation. It was found that when a trial of two vectors resulted in the incorrect degree being estimated that usually only one of these rows was the source of the problem, and attempting the trial with that row and any other row often also resulted in an incorrect estimation. In order to avoid this the algorithm was modified, such that any row or column may only be selected once.

4.2.1.2 Preprocessing of the Vectors

The last section described the extraction of two vectors, f and g, from an image. From these vectors an estimate of the degree must be computed. These vectors are likely to be unbalanced, and thus require normalisation and scaling in order to be balanced, avoiding issues in the computation.

In this section the preprocessing that must be performed on the vectors will be described. Firstly, the vectors are normalised to minimise the difference between values in the two vectors, then one of the vectors will be scaled by a constant to minimise the difference in additive error between the vectors [21]. In [21] several examples are given of situations in which the algorithm fails without these prepossessing techniques. While Winkler's paper goes into detail on the mathematical justification of these preprocessing algorithms, the work presented in this thesis does not modify these preprocessing algorithms, and so only the final form of these operations are discussed.

Normalisation

Winkler considered several methods to perform normalisation [21]. Normalisation by the 1 and 2 norms, normalisation by the leading coefficient, and normalisation by the geometric mean of the polynomial. Ultimately it was decided that using the geometric mean was the optimal approach. This approach was chosen for three reasons. Firstly, normalisation by the geometric mean considers all coefficients $\hat{s}_i$ of the polynomial $\hat{s}(x)$ when computing the normalised polynomial $\bar{s}(x)$, while the other methods only consider a single coefficient, this means that this form of normalisation is better suited when there are large differences between the values of the coefficients in the provided polynomials. Secondly, unlike the other methods, the geometric mean retains the uniform probability distribution of the relative error of $\hat{s}(x)$. Finally the paper showed that when the coefficients of $\hat{s}(x)$ vary wildly the relative errors of $\bar{s}(x)$ could only be computed reliably when $\hat{s}(x)$ has been normalised by the geometric mean.

The normalisation method used is therefore the geometric mean, which is defined as

$$\bar{s}_i = \frac{\hat{s}_i}{\left(\prod_{j=0}^{m} \hat{s}_j\right)^{\frac{1}{m+1}}}, \quad i = 0 \ldots m.$$

Thus the polynomials $f(x)$ and $g(x)$ are redefined by their geometric means. The non-normalised coefficients of these polynomials are $\hat{f}_i$ and $\hat{g}_i$ respectively, with their normalised coefficients being $\bar{f}_i$ and $\bar{g}_i$

$$f(x)=\sum_{i=0}^{m}\bar{f}_i x^{m-i} \quad , \quad \bar{f}_i=\frac{\hat{f}_i}{\left(\prod_{j=0}^{m}\hat{f}_j\right)^{\frac{1}{m+1}}},$$

$$g(x)=\sum_{i=0}^{n}\bar{g}_i x^{n-i} \quad , \quad \bar{g}_i=\frac{\hat{g}_i}{\left(\prod_{j=0}^{n}\hat{g}_j\right)^{\frac{1}{n+1}}}.$$

Scaling $g(x)$ by an Arbitrary Constant

After normalisation, due to the two polynomials being scaled differently, the componentwise error between the two polynomials is no longer equal, and thus one of the polynomials should be adjusted to account for this.

After the normalisation presented above, the updated componentwise error of the polynomial $f(x)$ will be

$$\left(\frac{m}{m+1}\right)\varepsilon,$$

and the error of $g(x)$ will be

$$\left(\frac{n}{n+1}\right)\varepsilon.$$

Therefore, to correct this difference in the error, $g(x)$ should be scaled by a variable α.

$$\alpha=\frac{\beta m(n+1)}{n(m+1)},$$

where β is an independent variable, the scale of which is determined through a linear programming problem, in which the difference between the maximum and minimum absolute entries of $f(x)$ and $\alpha g(x)$ is minimised.

4.2.1.3 Computation of the Upper Triangular Factors

The normalised vectors described in Section 4.2.1.2 can be used to construct the Sylvester matrix S_1, as is shown in Chapter 3. With the Sylvester matrix S_1 constructed, the upper triangular factors of the Sylvester matrix, and the subresultant matrices, can be computed. R_1 is first computed as a full QR decomposition of S_1. The upper triangular factors $R_{2\ldots n}$ of the subresultant matrices $S_{2\ldots n}$ must then be computed from R_1. The computation of these factors is the most computationally expensive part of the degree computation.

In the case of the exact polynomials the degree of the GCD can be found by investigating each of the subresultant matrices, and finding the first rank deficient matrix [19]. Unfortunately, in the case where the polynomials are inexact, all of the subresultant matrices will be of full rank, and thus the rank must be approximated instead.

As was stated previously, the upper triangular factor R_k, of S_k, shares the rank of S_k. However, the properties of R_k make it easier to compute an approximate rank than those

from the subresultant matrix, and thus in this section the upper triangular matrices $R_{2...n}$ are computed, to be used in the rank estimation computations that will be described in Section 4.2.1.4.

Chapter 3 showed how the subresultant matrices $S_{2...n}$ are constructed. While the rank of these matrices is of interest, it is not necessary to actually construct these matrices in the algorithm. As was discussed is the previous chapter it would be possible to compute a decomposition of every subresultant matrix, though computational complexity can be reduced by performing QR updates to calculate the upper triangular factor of these matrices instead.

As was discussed in Chapter 3, when the last columns of each Toeplitz submatrix of S_k are removed, the entries in the last row of the remaining columns of the matrix will always be equal to zero, and thus will have no impact on the rank calculation. This means that, for the purposes of computing the rank of the upper triangular R matrix, the row deletion described previously is not required. This is of great benefit when optimising the code, as while QR row deletions require both Q and R matrices to compute the updated upper triangular matrix R, this is not the case with column deletions. In a QR column deletion only the original upper triangular matrix R is required to compute the updated upper triangular matrix. For the purposes of this algorithm this means that the upper triangular matrix of R_k can be computed from R_{k-1}, without needing to consider the Q matrices. This optimisation halves the number of matrix operations required to perform the updates, as well as bypassing the row deletions entirely.

4.2.1.4 Testing the Upper Triangular Factors

After all the upper triangular factors $R_{1...n}$ have been computed, an estimation of the rank for the Sylvester matrix $S_1(f, g)$ must be computed. Due to the presence of noise in the polynomials, the rank computation cannot be exact, as the upper triangular matrices $R_{1...p}$ will be of full rank. However, if an AGCD can be computed, there will be near-zero values in the lower rows of $R_{1...p}$, and these matrices will be closer to rank deficient than $R_{p+1...n}$.

Two tests are used to estimate the rank of $S_k(f, g)$ using the computed upper triangular factor R_k. In his 2012 paper Winkler found that these tests, while heuristic, were demonstrated to obtain reliable results over many examples.

The tests are computed in a very similar way, with one considering the row norms of each subresultant matrix, and the other considering the diagonals of each subresultant matrix.

In this section $a_{k,i}$ will represent either the diagonal, or the row norm, for the ith row of kth subresultant matrix. In the first of these tests, which considers the diagonals of R_k, $a_{k,i} = |r_{k,i,i}|$ where $r_{k,i,i}$ is the entry on the ith row of the principal diagonal of R_k. In the second test, which considers the 2-norms of these rows $a_{k,i} = \|r_{k,i}\|_2$ where $r_{k,i}$ is the entire ith row of R_k.

With $a_{k,i}$ defined for both tests the computation to perform these tests can be described. First a ratio is computed for the maximum and minimum values for $a_{k,i}$, for each upper triangular factor R_k.

$$t_k = \frac{\max_i(a_{k,i})}{\min_i(a_{k,i})}.$$

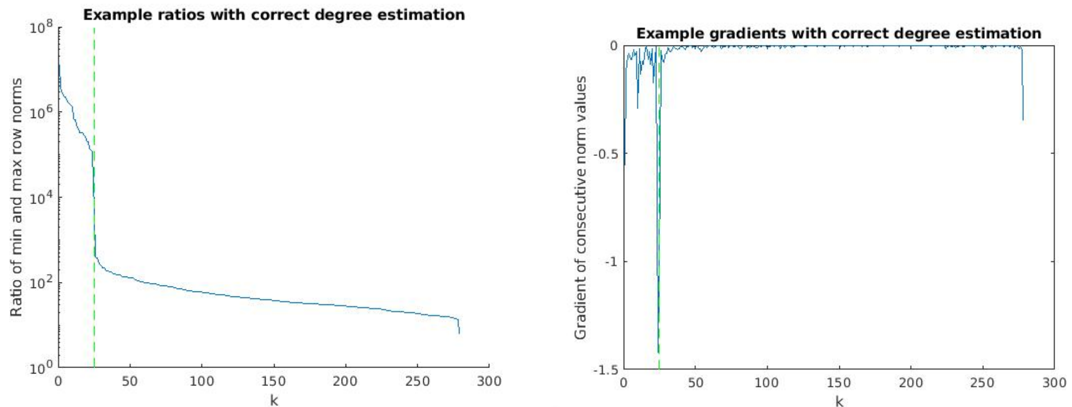
The gradient between the values of t_k is computed for adjacent values of k. The index at which the gradient is at its minimum will provide the estimate of the rank, and thus the degree, for this test. The degree will be known as p.

$$p = \arg \max_{k=1 \ldots n-1} \left(\frac{t_k}{t_{k+1}} \right).$$

These two tests provide two estimates for the degree of the AGCD for each trial. The modal value across all trials provides the final estimate for the degree.

4.2.2 Improving Reliability of Degree Results

It is possible, by making a reasonable assumption about the size of the PSF, to improve both reliability of the degree estimation results, and the speed of the computations. By assuming that the PSF is smaller than an arbitrary size, a correct estimate of the degree can be computed more reliably. This is a reasonable assumption when considering image deconvolution, as when the PSF is too large it would be difficult to discern any details in the image, and the deconvolution would be likely to be unsuccessful. An additional benefit of this is a decrease in the number of operations required, as many of the subresultant matrices would no longer be required, and thus their upper triangular factors need not be computed.



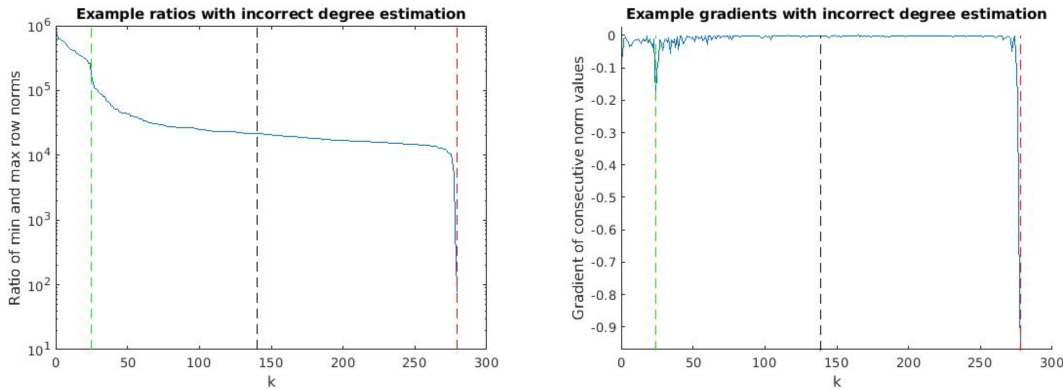
(a) Example of ratios that give an correct degree estimation

(b) Example of gradients that give an correct degree estimation

Figure 4.3: Example results from a single trial resulting in correct degree estimation

Figure 4.3 shows an example of a trial that results in the correct degree being computed. Figure 4.3a shows the ratios computed between the minimum and maximum row norm

values for each value of k on a log scale. Figure 4.3b shows the gradients between row norm ratios for successive values of k. In this case the exact image was 256 wide, with PSF of degree 24. Note that the degree computed, signalled by the green line, shows a clear drop compared to other ratios, and the gradient shows a very prominent trough for $k = 24$. This is the desired outcome. However, note that the high values of k also exhibit a sharp drop at the end of the graph. This drop for higher values of k can often increase as the noise increases, and can result in an incorrect estimation.



(a) Example of ratios that give an incorrect degree estimation

(b) Example of gradients that give an incorrect degree estimation

Figure 4.4: Example results from a single trial resulting in incorrect degree estimation

Figure 4.4 shows two graphs from an example of a trial which produced an incorrect degree. Figure 4.4a shows the ratios of the row norms of the image on a log scale, while Figure 4.4b shows the gradients as was explained for Figure 4.3b. While a clear drop can be seen when $k = 24$ (marked with a green dashed line) in the ratios in Figure 4.4a, and confirmed by the low gradient shown in Figure 4.4b, a more significant drop can be seen when $k > 275$, and the gradient graph shows a significantly lower gradient for these higher values of k than that shown at $k = 24$. This results in an incorrect degree estimation. In this case that result is at $k = 278$ (marked with a red dashed line) with the gradients only being lower than the expected result when $k = 278$ and $k = 277$. This is not always the case, as some trials return results with significantly lower gradients for lower values of k.

For the purposes of this image deconvolution algorithm it is sufficient to place a limit on the expected AGCD degree of 50% of the convolved image size, as when the degree is greater than this limit it is unlikely to result in successful deconvolution of the image, even if an AGCD degree is correctly identified. This limit can be seen in the graphs in Figure 4.4 as black dashed lines.

The code provided by Winkler approached this problem by ignoring any computed degrees that produced results above a specified limit. While this approach works in most cases, it can lead to degree estimations being made with less confidence, as the modal value is computed from a set of values that is smaller than the set of computed trials, as values above the specified limit are ignored. In extreme cases this was found to cause the

deblurring to fail, due to all the values produced by the trials being outside the expected range.

An improved approach, as discussed earlier in this section, is to simply not compute the upper triangular factors of the subresultant matrices outside of this range. This results in a more robust approach, as when excluding the upper values of k more trials will return the desired result.

To demonstrate this, two real world images, one of size 256×256 and one of size 512×512, were convolved with PSFs of sizes 25×25 and 49×49 respectively. Noise uniformly distributed between a lower bound of 1×10^{-5} and an upper bound of 1×10^{-4} was added to the PSF. Noise uniformly distributed between a lower bound of 1×10^{-7} and upper bound 1×10^{-6} was added to the convolved image. The equation detailing this is

$$\mathcal{G} = \mathcal{F} \otimes (\mathcal{H} + \mathcal{N}_1) + \mathcal{N}_2,$$

where $\mathcal{G}$ is the blurred image, $\mathcal{F}$ is the exact image, $\mathcal{H}$ is the exact PSF to be convolved with the image, $\mathcal{N}_1$ is the additive noise introduced to $\mathcal{H}$, and $\mathcal{N}_2$ is additive noise added to the convolved image.

For the purposes of these results, the method of simply ignoring any value outside of the range will be known as the original method, and the new method of avoiding computation of results beyond the limit will be known as the reduced method. Each image was subject to 24 trials on both the columns and the rows. The successful results from both rows and columns were combined to provide an overall percentage of the trials that were successful.

Tables 4.1 and 4.2 show the rate of successful degree computation of the original and reduced algorithms, under the two levels of noise described above. In these tables the success rates of the degree computed from the row norms and the diagonals are separated, to show the improvement this method has on both tests.

	256×256		512×512	
	Row norms	Diagonals	Row norms	Diagonals
Original	85.4%	91.7%	0%	8.3%
Reduced	97.9%	97.9%	97.9%	97.9%

Table 4.1: The percentage of trials identifying the correct degree for lower levels of noise

The results in Table 4.1 show that both the original and the reduced versions of the algorithm provided reasonable results for the smaller image. However, the larger image yielded very poor results in the original implementation, while the reduced implementation still performed well.

When the additive noise introduced to both the PSF and convolved image is increased by a factor of 10, the difference between the two algorithms becomes significantly more apparent. The results in Table 4.2 show that the original method did not provide any successful trials. The reduced method provided reasonable results for both image sizes, though at a reduced success rate for the larger image.

	256 × 256		512 × 512	
	Row norms	Diagonals	Row norms	Diagonals
Original	0%	0%	0%	0%
Reduced	91.7%	89.6%	66.7%	66.7%

Table 4.2: The percentage of trials identifying the correct degree for higher levels of noise

4.3 Computation of the Coefficients

Section 4.1 gave a brief overview of the coefficient computation method. This section will give a more detailed explanation of this algorithm, with focus on the most expensive section, that being the computation of an optimal column as part of the modified SNTLN method.

In a situation with exact polynomials the matrix S_1 will be rank deficient, and the last non-zero row of the upper triangular factor R_1, found through QR decomposition, will provide the coefficients of the GCD. When coprime polynomials are considered instead S_1, and thus R_1, will be of full rank, and thus the coefficients of the AGCD cannot be retrieved in the same way.

The method of SNTLN [20] is used by Winkler and Hasan to solve this problem [69, 21]. The aim of this work was to attempt to find a structured, low rank estimation of S_1, and thus find an approximation of the coefficients.

Before the Sylvester matrix can be constructed, the polynomials must be preprocessed. This is performed in the same way as in the computation of the degree, described in Section 4.2. To simplify the notation it will be assumed that the polynomials $f(x)$ and $g(x)$ have already been processed in this way.

Consider the exact polynomials $\hat{f}(x)$ and $\hat{g}(x)$ with a GCD $d(x)$ of degree p, such that when $\hat{f}(x)$ and $\hat{g}(x)$ are divided by $d(x)$ they produced $u(x)$ and $v(x)$ respectively.

$$d(x) = \frac{\hat{f}(x)}{u(x)} = \frac{\hat{g}(x)}{v(x)}.$$

It follows from this matrix that there exists a polynomial $w(x)$ such that

$$w(x) = v(x)\hat{f}(x) = u(x)\hat{g}(x).$$

Therefore, when considering the vector form of these polynomials, where u is the vector of the coefficients of $u(x)$,

$$w = C_p v = D_p u,$$

where C_p and D_p are the Toeplitz matrices of $\hat{f}$ and $\hat{g}$ respectively, that make up the pth subresultant matrix described in Section 3.1.4.

This equation can be rearranged to give

$$\begin{bmatrix} C_p & D_p \end{bmatrix} \begin{bmatrix} v_p \\ -u_p \end{bmatrix} = 0. \tag{4.1}$$

As described in Section 3.1.4, the matrix $[C_p D_p] = S_p(\hat{f}, \hat{g})$. This Sylvester matrix will be denoted in this chapter in the shorter form, S_p.

The equation above can be transformed by moving a column of S_p to the right side of the equation.

$$Ay = c, \tag{4.2}$$

where c is the column removed from S_p, A is the remaining columns of the Sylvester matrix S_p after the column is removed, and y is the combined column vector of v and $-u$ as shown in Equation 4.1.

When considering the inexact case, with polynomials $f(x)$ and $g(x)$, the coefficients must be perturbed slightly to form a AGCD. The subresultant matrix S_p is perturbed by the matrix $B(\alpha, \theta, z)$, also denoted as B.

$$B = \begin{bmatrix} z_0\theta^m & & & \alpha z_m\theta^n & & \\ z_1\theta^{m-1} & \ddots & & \alpha z_1\theta^{n-1} & \ddots & \\ \vdots & \ddots & z_0\theta^m & \vdots & \ddots & \alpha z_m\theta^n \\ z_{m-1}\theta & \vdots & z_1\theta^{m-1} & \alpha z_{m+n-1}\theta & \vdots & \alpha z_{m+1}\theta^{n-1} \\ z_m & \ddots & \vdots & \alpha z_{m+n} & \ddots & \vdots \\ & \ddots & z_{m-1}\theta & & \ddots & \alpha z_{m+n-1}\theta \\ & & z_m & & & \alpha z_{m+n}. \end{bmatrix},$$

where α and θ are variables initialised in the preprocessing, and z is a vector of length $m + n$.

Similarly to how S_p was split into the column c and A, B must be split into the vectors h and E. S_p, and therefore A and c, can also be considered by the variables α and θ, as these were used in the preprocessing of the polynomials.

Thus equation 4.2 for the inexact polynomials becomes

$$(A_p(\alpha, \theta) + E_k(\alpha, \theta, z))y = c_p(\alpha, \theta) + h_p(\alpha, \theta, z).$$

The approximate solution to this equation is found with a minimisation problem, with regards to the variables α, θ, y and z. The solution to this yields a low rank estimation of the Sylvester matrix, which can be used to compute the coefficients of the AGCD. This

minimisation problem was not considered in detail for acceleration in this thesis, so shall not be discussed in detail here.

Equation 4.2 showed a column from the subresultant matrix S_p being moved to the right side of the equation. While typically in SNTLN it was thought that the first column of the matrix should be moved, as examples showed that moving the last column led to errors, Winkler and Hasan noted that this approach was not robust [21]. This was because it often led to the solution found for the Sylvester matrix $S(f, g)$ being different to that of the Sylvester matrix $S(g, f)$. This is not desirable when computing the AGCD of two polynomials, as the AGCD should be the same in both situations.

Winkler and Hasan demonstrated that when computing the residual of the equation $A_p x \approx c_p$ for each column, the last element has a residual of several orders of magnitude higher than the other columns.

Winkler and Hasan instead proposed the selection of a different column of the Sylvester matrix to move to the right of the equation, specifying that all columns must be considered as candidates to be moved. This column is called the optimal column, and the computation of the optimal column is second only to the degree computation in terms of computational expense in the BID algorithm.

4.3.1 Computation of an optimal column

The optimal column is defined as the column with which the residual of the approximate linear algebraic equation is minimised when moving the selected column $c_{p,o}$ to the right of the remaining columns in the matrix $S_p(f, g)$ [12]. The computation of this optimal column will be broken down in this section, and can be seen in full in the flow chart in Figure 4.5.

The first step of the computation of the optimal column is to compute the QR factorisation of the matrix $S_p(f, g)$ to create the matrices Q_p and R_p.

The algorithm then takes on an iterative approach, for each column in $S_p(f, g)$ QR column deletion is used to compute the updated factors after the removal of this column, resulting in an updated matrix $Q_{p,t}$ where t is the current column being tested.

Note that while the computation of the updated upper triangular matrix is required in order to compute $Q_{p,t}$, as was discussed in Chapter 3, it is not necessary beyond this, and thus $R_{p,t}$ can be discarded.

Column t is now extracted from $S_p(f, g)$. This column will be known as c_t. The product of $Q_{k,t}^T$ and c_t is computed, giving the column vector a_t.

$$a_t = Q_{p,t}^T c_t.$$

From the resulting vector the norm of the last p entries provides the residual r,

$$r_t = \left\| a_{t,(n-d)\dots n} \right\|_2 .$$

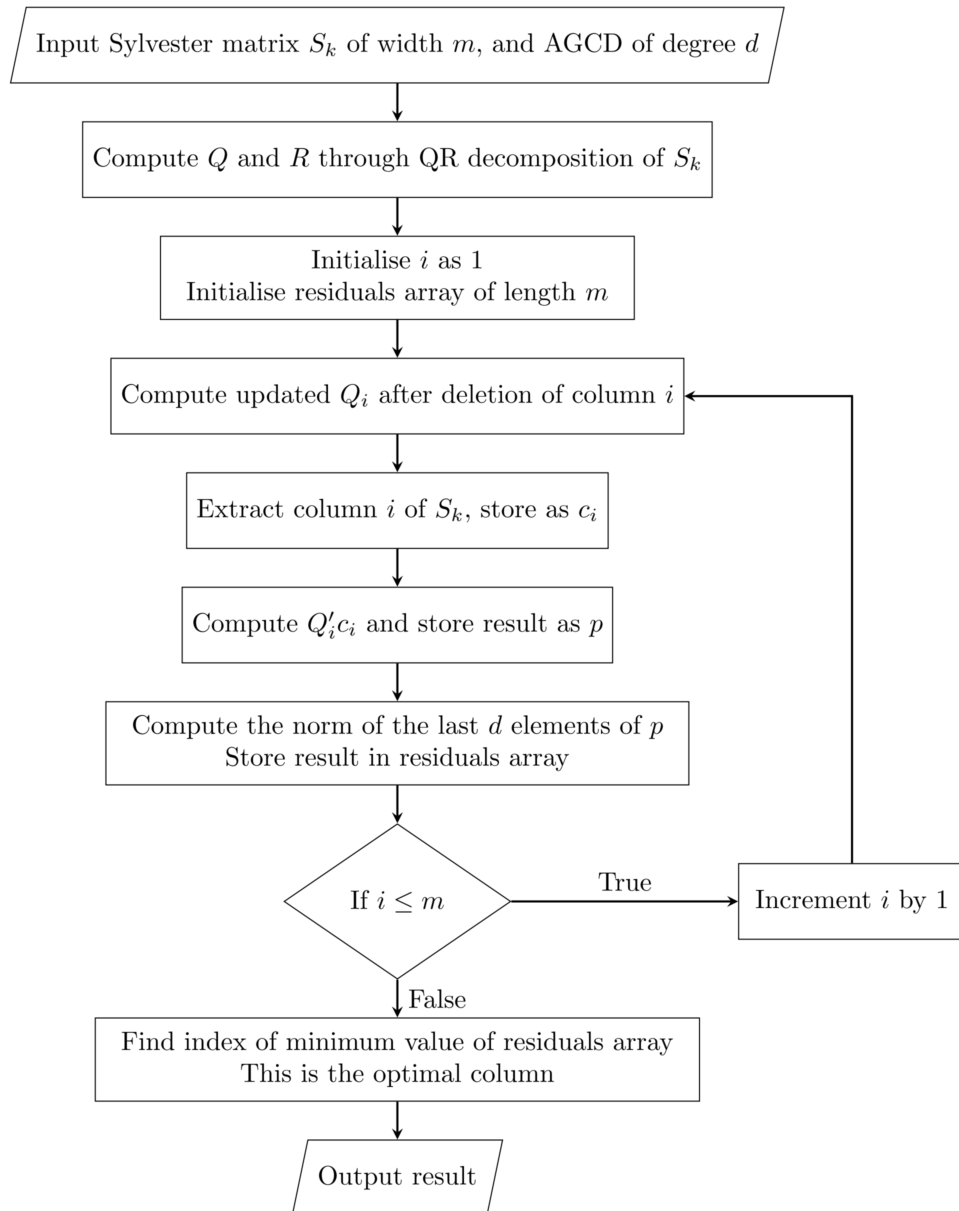


Figure 4.5: Flowchart showing the serial process of the optimal column algorithm

This process is repeated for all $m+n-2d$ columns, and the value of t in which the residual is minimised represents the column index of optimal column o.

$$o = \min_t(r_t) \quad , \quad t = 1 \ldots (m+n).$$

4.3.2 Optimisations and Improvements

The method of computing the residual described above can be optimised by removing unnecessary computations. It should be noted that the matrix $S_k(f,g)$, and its QR decomposition, were previously computed in Section 4.2. While it would be possible to retain these matrices, and thus remove the need for their computation, the degree of the AGCD is not known at this time, and thus the matrix $S_k(f,g)$, and its factors computed from QR decomposition, Q_k and R_k, would need to be retained until the degree is computed. This would place a large burden on the memory, for minimal gain, as this would be trivial to recompute. In systems where a large amount of memory is available however, this would be an easy optimisation to apply.

At a lower level there exist unnecessary computations within the QR update algorithm. While the method discussed thus far computes the entire QR update, it is in fact possible to reduce the number of operations required in this update, as only the last p columns of Q_p will effect the value of the residual. While this section only provided a brief overview of this optimisation, it will be explored in further detail in Chapter 8.

4.4 Image Deconvolution Results

This section will discuss and present a sample of the results produced by the BID algorithm, from the perspective of both relative error and qualitative analysis of the resulting images. Winkler presented a sample of results of his algorithm in his 2016 paper [1] in which he presented the BID algorithm.

The images were compared to the exact image to compute the error measure. The images are normalised against each other, and the norm of the difference between all the pixel values with their equivalents in the exact image was computed. A relative error of 5.55×10^{-3} was observed for the proposed algorithm, compared to the relative errors of the MATLAB methods, in which the PSF was specified in all cases, which ranged from between 0.340 to 0.124.

Each exact image $\mathcal{F}$ was convolved with a Gaussian PSF, $\mathcal{H}$, of the stated degree in each case. Uniformly distributed noise, $\mathcal{N}_1$, was added to the PSF, with stated upper and lower bounds. Uniformly distributed noise $\mathcal{N}_2$ was also added to the convolved image, again with stated upper and lower bounds, resulting in the degraded image $\mathcal{G}$.

$$\mathcal{G} = \mathcal{F} \otimes (\mathcal{H} + \mathcal{N}_1) + \mathcal{N}_2.$$

4.4.1 Test 1

In this first test an image of a wind turbine, of dimensions 256×256, will be used. This image has been degraded with the following properties:

- **Exact image size:** 256×256
- **Gaussian PSF dimensions**: 25×25
- **PSF noise:**
  - **Upper bound:** 1×10^{-4}
  - **Lower bound:** 1×10^{-5}
- **Additive noise:**
  - **Upper bound:** 1×10^{-6}
  - **Lower bound:** 1×10^{-7}

(a) Exact image (b) Blurred noisy image (c) Restored image

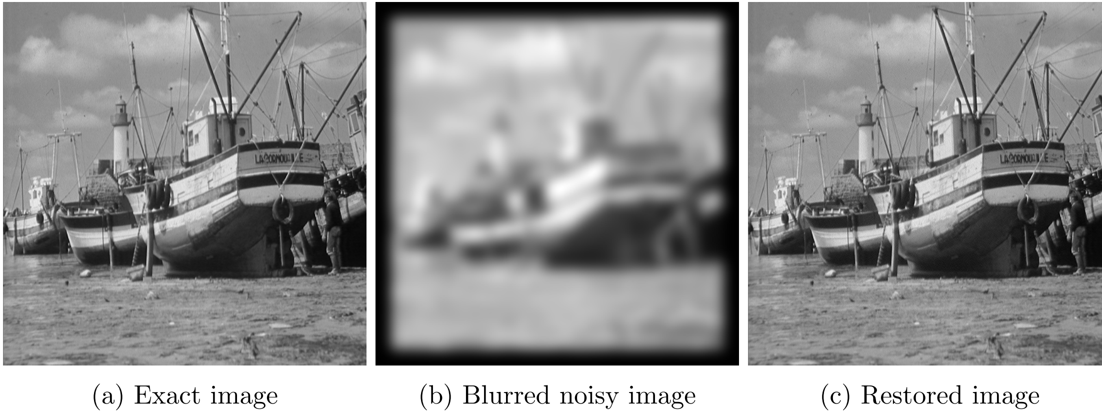
Figure 4.6: Exact, blurred, and restored images for Test 1

It is clear, from the images shown in Figure 4.6, that the algorithm was successful in deconvolving the image. The main differentiating feature between the exact image in Figure 4.6a and the restored image in Figure 4.6c is the presence of some level of noise in the restored image. This is largely due to the additive noise that was introduced to the image. As the algorithm only attempts to deconvolve the image, and not denoise it, this noise will remain. It is worth noting that this is at a low level when compared to many results from algorithms that use the Fourier transform. The actual relative error that was produced by this deconvolution when compared to the exact image is 0.059, an improvement from the relative error in the blurred image of 0.109.

4.4.2 Test 2

The second test used an image of a fishing boat. This is a typical test image for image processing algorithms. This image has been degraded with the following properties:

- **Exact image size:** 512×512
- **Gaussian PSF dimensions**: 49×49
- **PSF noise:**
  - **Upper bound:** 1×10^{-6}
  - **Lower bound:** 1×10^{-7}
- **Additive noise:**
  - **Upper bound:** 1×10^{-7}
  - **Lower bound:** 1×10^{-8}



(a) Exact image (b) Blurred noisy image (c) Restored image

Figure 4.7: Exact, blurred, and restored images for Test 2

In this test, the results of which are shown in Figure 4.7, there is a well restored image, with the only differentiating factor being noise. The larger image and PSF in this test meant that a lower amount of noise was required for a successful deconvolution. However, the relative errors do not differ significantly from those in Test 1, with a relative error between the exact and restored images of 0.023, compared to a relative error between the exact and blurred images of 0.200.

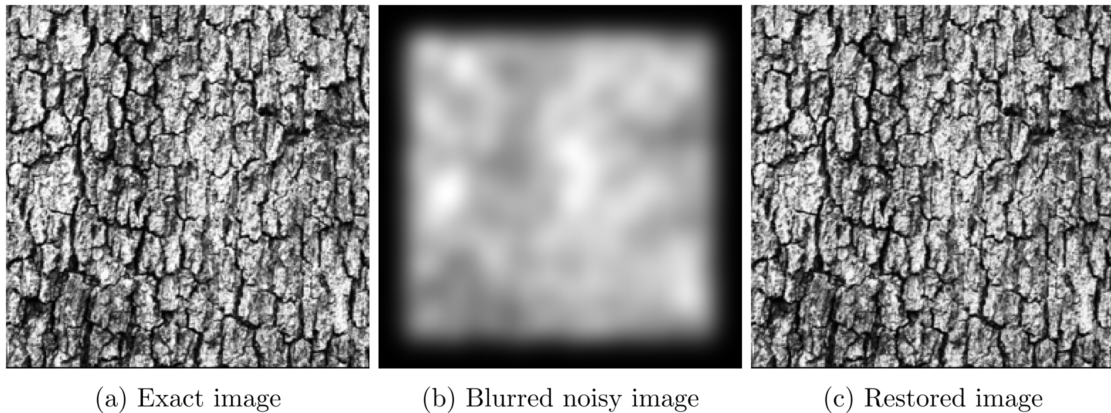
4.4.3 Test 3

The final test used an image of tree bark which is another typical test image. This image has been degraded with the following properties:

- **Exact image size:** 256×256
- **Gaussian PSF dimensions**: 49×49
- **PSF noise:**
  - **Upper bound:** 1×10^{-6}

  - **Lower bound:** 1×10^{-7}

- **Additive noise:**
  - **Upper bound:** 1×10^{-7}
  - **Lower bound:** 1×10^{-8}



(a) Exact image (b) Blurred noisy image (c) Restored image

Figure 4.8: Exact, blurred, and restored images for Test 3

The results of this test, shown in Figure 4.8, show a heavily blurred image, caused by a significantly larger PSF relative to the size of the exact image than in Test 1, though with lower noise. The results again show a successfully deconvolved image. The relative errors in this case are 0.010 when comparing the restored image to the exact image, and 0.462 when comparing the blurred image to the exact image.

4.5 Simple CPU Parallelisation

While the focus of this thesis is on GPU parallelisation, a simple parallelisation of the original MATLAB version on a CPU is also possible. This version can give a baseline for any GPU implementations to be tested against.

As described in Section 4.1, the degree computation algorithm is run as a series of trials. These trials can easily be run in parallel across the cores of a CPU, providing a simple method of parallelisation on a CPU. In this parallel implementation the tasks to parallelise are well balanced, with each trial being exactly the same in terms of overall number of computations.

Similarly, during the computation of the coefficients described in Section 4.3, the computation of an optimal column can be parallelised across the cores of the CPU. Each core processes a single QR column deletion and the product of the column and the matrix required for this section. While this section will not lead to as balanced an implementation as that for the degree computation, as some QR deletions will require less work than others, modern CPUs have efficient task schedulers, which will ensure cores are utilised effectively.

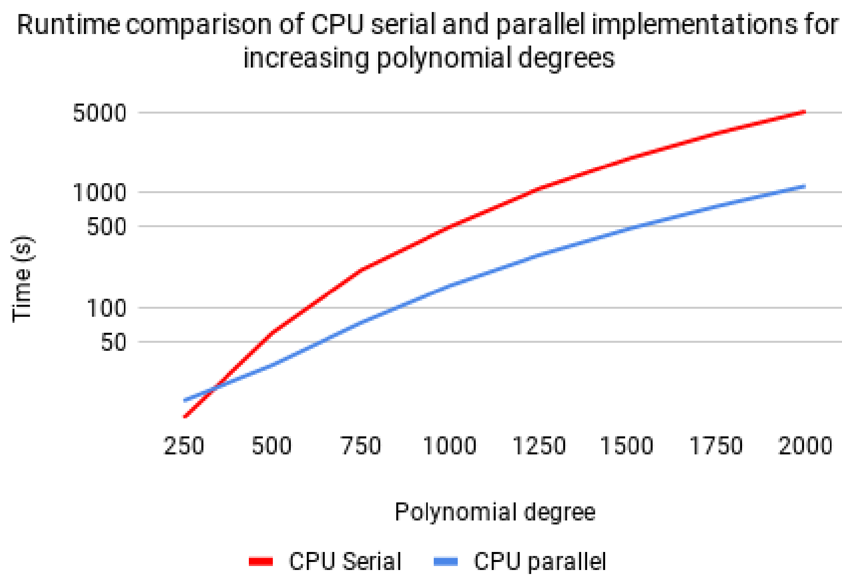


Figure 4.9: Comparison of CPU serial and parallel implementations

One issue with CPU parallelisation in Matlab is the need to initialise a parallel pool. This initialisation step can take a significant amount of time, and when dealing with images of smaller sizes, leading to polynomials of smaller degrees, can cause a parallel implementation to run slower than the serial implementation. This can be seen on the graph in Figure 4.9, which shows a comparison of the runtimes of the serial and parallel implementations of the degree computation on a log scale. These results were gathered on an Intel i7 6850k CPU, with the work in the parallel implementation spread across all 6 cores. In this graph the line showing the parallel implementation starts above that of the serial implementation. The parallel implementation quickly speeds up relative to the serial implementation, however, as the amount of time spent initialising the parallel pool relative to the overall runtime is less significant with larger degrees.

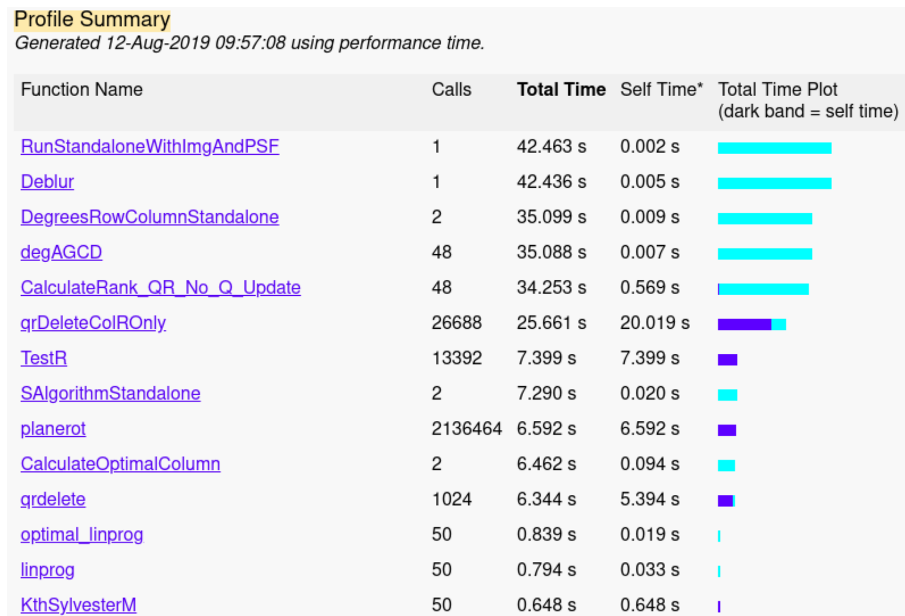
Further results with the CPU parallel implementations can be found in Chapters 6, 8 and 9.

4.6 Profiling and Parallelism Potential

To optimise an algorithm it is necessary to understand the current bottlenecks of the algorithm. In order to investigate this the code must be profiled. This can be accomplished by using the Matlab profiler.

Figure 4.10 shows the timed results output from the Matlab profiler for one execution of the algorithm, processing a 256×256 image convolved with a 25×25 Gaussian PSF, with 24 trials for the rows and 24 trials for the columns. The test was run on a single core of an Intel i7 6850K with a clock speed of up to 4GHz.

In Figure 4.10 the bars representing time taken by each function are split into two


Profile Summary
Generated 12-Aug-2019 09:57:08 using performance time.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
RunStandaloneWithImgAndPSF	1	42.463 s	0.002 s	
Deblur	1	42.436 s	0.005 s	
DegreesRowColumnStandalone	2	35.099 s	0.009 s	
degAGCD	48	35.088 s	0.007 s	
CalculateRank_QR_No_Q_Update	48	34.253 s	0.569 s	
qrDeleteColROnly	26688	25.661 s	20.019 s	
TestR	13392	7.399 s	7.399 s	
SAlgorithmStandalone	2	7.290 s	0.020 s	
planerot	2136464	6.592 s	6.592 s	
CalculateOptimalColumn	2	6.462 s	0.094 s	
qrdelete	1024	6.344 s	5.394 s	
optimal_linprog	50	0.839 s	0.019 s	
linprog	50	0.794 s	0.033 s	
KthSylvesterM	50	0.648 s	0.648 s	

Figure 4.10: Results of profiling the serial implementation of the image deconvolution algorithm

sections. The dark blue bar represents the self time. That being the time taken doing fundamental processing tasks, such as basic arithmetic and control flow within that function, without considering the time spent in child functions. The cyan section of this bar represents the time taken executing the child functions. The combined bars show the total time spent in each function. The times in seconds of the total time and self time of these functions are shown to the right of the bars.

Starting from the top of this figure, `RunStandaloneWithImgAndPSF` is the main test function, which takes input of an image and a PSF, blurs the image, and attempts deconvolution using the `Deblur` function, which can be seen on the next line. The `Deblur` function is where the full deconvolution happens, including computing the degree of the AGCD, calculating its coefficients, and deconvolving the computed AGCDs from the blurred image. The next three functions, `DegreesRowColumnStandalone`, `degAGCD` and `CalculateRank_QR_No_Q_Update` all appear to have similar runtimes, and all have very little self time, due to the fact that they call each other, in sequence, with little processing to do aside from this.

The next function on the list, called from `CalculateRank_QR_No_Q_Update`, is where the first significant portion of self time is found. The function `qrDeleteColROnly` performs the QR column deletions discussed in Section 4.2.1.3. While this function is primarily self time, there is still a section of the function that is not accounted for in this self time. This remaining time is made up of the function `planerot`, which, as can be seen on its row of the graph is purely self time. This function performs Givens plane rotation to compute the Givens matrix to apply to the upper triangular matrix, which is pure arithmetic that can easily be combined with the rest of the QR deletion.

Note that while qrDeleteColROnly is the most significant part of its parent function (CalculateRank_QR_No_Q_Update), there is a significant section of this function not accounted for in the self time or the time of CalculateRank_QR_No_Q_Update. This time is taken executing TestR, which can be seen further down the chart. This function involves performing the two tests described in Section 4.2.1.4 on the computed triangular factors to estimate the rank of each matrix, and thus compute the degree. As will be discussed in Chapter 5, copying data to and from a GPU adds overhead to the computation and must be avoided. If the computation of the upper triangular factors is moved to the GPU, and not the tests, this would require all of the subresultant matrices for every test to be copied back to the main system memory. Thus this section is best computed on the GPU as well.

Together, the computation and testing of the upper triangular factors of the subresultant matrices represent over half of the overall runtime of the serial algorithm, and thus these are the first algorithms that should be accelerated.

Unfortunately the MATLAB profiler does not provide clear results when processing CPU parallel implementations, grouping all parallel for loops as a single function. This makes the results difficult to interpret, and due to this the profiling of the CPU implementation described in Section 4.5 is unable to be assessed in this way.

4.7 Conclusion

This chapter described a reliable method for computation of an AGCD of multiple pairs of univariate, coprime polynomials. This was performed as part of a BID algorithm which, as was seen by the results shown in Section 4.4, provides very promising results. However, as has been discussed, this method is computationally expensive, and thus requires optimisation for certain practical applications.

The profiling in Section 4.6 showed that the main bottleneck in performance for this algorithm is in the degree computation, particularly in the QR column deletions. Therefore this is where the focus should be when it comes to optimising this algorithm. However, before the algorithm can be accelerated on a GPU, an understanding must be gained on how to properly utilise such devices. The next chapter will therefore discuss parallel programming, and how GPUs can be effectively utilised.

Chapter 5

Introduction to Parallel and GPU Computing

The primary aim of this research is to find methods of optimising and accelerating the blind image deconvolution algorithm presented by Winkler [1]. As was described in Chapter 4, the part of the algorithm with the highest computational expense is the QR column deletions, as part of the degree computation of an AGCD of a set of pairs of polynomials.

QR methods of AGCD computation can involve large scale matrix operations, which have the potential to take advantage of the massive parallelism available on GPU hardware. This chapter will provide an introduction to GPU hardware, and the software that utilises it, and explain the best practices for ensuring efficient use of the hardware.

Section 5.1 will give an overview of types of parallelism, and how these relate to specific forms of hardware. Section 5.2 will give a more detailed description of the hardware and software aspects of GPUs, and how they can be utilised effectively.

5.1 Parallel Computing and Accelerators

Parallel computing involves splitting the work that needs to be computed into jobs, which can be run independently from each other. This means the computer can take advantage of the availability of multiple processors, or multiple cores, to process work more quickly.

5.1.1 Data Parallel vs Task Parallel

Two key paradigms in parallel computing are task parallelism and data parallelism. Task parallelism is the most common form of parallelism, and is found in the task management of every modern operating system. Task parallelism relies on the work being split into groups known as tasks, which can be completely unrelated to each other. The tasks are then assigned to a processor, or a core within a processor, to be computed. This means that unrelated tasks can be running at the same time. Multi-core CPUs are task parallel processors.

Data parallelism involves parallelism across data, running the same instructions on many pieces of data simultaneously. This type of parallelism is well suited to GPUs, where many cores can be utilised to process many data points simultaneously.

Parallel processors can be split into several categories. Three of these categories, that describe the flexibility of the parallelism available, are simultaneous multi-threading (SMT), single instruction multiple data (SIMD), and single instruction multiple threads (SIMT). SMT is used to describe task parallelism, where operations run independently across multiple processors, providing for the greatest amount of flexibility in operation. SIMD is used to describe data parallelism, where the same operation is run across all cores simultaneously, and different branches of the code can be run in sequence. SIMT is a hybrid of these approaches, where each processor has multiple threads, which can be swapped in and out, with the operations in each thread running in lockstep. While SMT offers the greatest flexibility in programming, handling branching code well, SIMD provides the greatest performance when the task is suited to it. Even though SIMT does differ from SIMD, the best performance on a GPU can often be achieved by treating the GPU as a SIMD device, and avoiding divergence [70].

5.1.2 Types of Parallel Coprocessors

In recent years there has been interest in using coprocessors in order to accelerate certain computations, moving these computations off the main CPU of a computer and onto specialist hardware. The most dominant forms of parallel coprocessors are GPUs, and massively parallel CPU based accelerators, known as many integrated core (MIC), such as the Intel Xeon Phi line of products.

The MIC products have many similarities to GPU technology, including dedicated memory, and a combination of SMT and SIMD approaches. However, they have the advantage of using the same x86 instruction set as the majority of desktop and server CPUs. This makes their usage more flexible than GPUs, allowing for a greater level of branching within the code, and often requiring fewer alterations to the code. GPU technology, on the other hand, is more ubiquitous, as for many years they have been used for computer graphics intensive processes, such as video gaming. This also means there is a wide range of devices available at different budgets. The computational advantage a particular type of accelerator provides is dependent on the situation in which it is placed, though most situations where the MIC has advantage over the GPU this difference is slight [71, 72].

Due to the ubiquity and maturity of GPUs, and their significant advantages when it comes to data parallel programming problems, the GPU was selected as the means with which to accelerate this algorithm. GPU hardware and technologies will be discussed in detail in the next section.

5.2 Introduction to GPU Hardware and GPGPU Concepts

This section will explain the basics of the structure of GPUs, and how the architecture necessitates a different approach to algorithmic design and implementation.

While a CPU has a small number of powerful cores, a GPU has a larger number of weaker, specialised cores. A typical modern CPU is likely to have between 4 and 16 cores, while modern GPUs, even at a consumer level, can have over one thousand cores. These cores are significantly less powerful than those in a CPU, but by utilising all the cores to simultaneously perform cohesive operations, such as large matrix operations, significant speedups can be achieved.

While finding enough parallelism to sufficiently occupy the device is a significant part of GPU programming, this is not the only factor that should be considered. The architectural structure of GPUs necessitates careful memory management and work distribution.

The GPU implementations presented in this thesis were all developed in NVIDIA's proprietary CUDA (Compute Unified Device Architecture) platform, written in the C programming language. Therefore discussion about the techniques used in this section, and the thesis as a whole, will be from this perspective. While there are other platforms and libraries for GPGPU, such as the open source OpenCL, which works on a wider range of GPUs, these generally lack many of the cutting edge features of CUDA, and do not perform as well.

5.2.1 Grids, Blocks, Threads and Warps

Work is distributed across the cores of the GPUs at several levels [73, 74]. At the lowest level are threads. Threads are organised into blocks, which in turn are organised into a grid. When calling a GPU kernel the program must specify the number and size of blocks. In modern GPUs blocks can have up to 1024 threads, with up to 2.815×10^{14} blocks in a grid. This hierarchy of grids, blocks, and threads is seen in Figure 5.1.

When distributing threads to cores, the GPU assigns groups of threads to a warp. A warp is a group of 32 threads that will execute the same instruction simultaneously. When the code branches, via an if statement, the branches will be executed sequentially, leaving threads that are not entering that branch of code idling. It is due to this that branching tends to be computationally expensive on GPUs, and it is best for threads that follow the same branch to be processed in contiguous threads. Due to the warps being groups of 32 threads it is best for the block size to be a multiple of this, as otherwise cores will be left idling.

5.2.2 Race Conditions and Synchronisation

Race conditions occur when one thread will modify a value that another thread needs to read from, and order of operations is not enforced. Due to the uncertainty in order of operations present in parallel computing this can cause issues. Threads inside a block can be synchronised, meaning that all threads, or a subset of the threads, wait until they reach

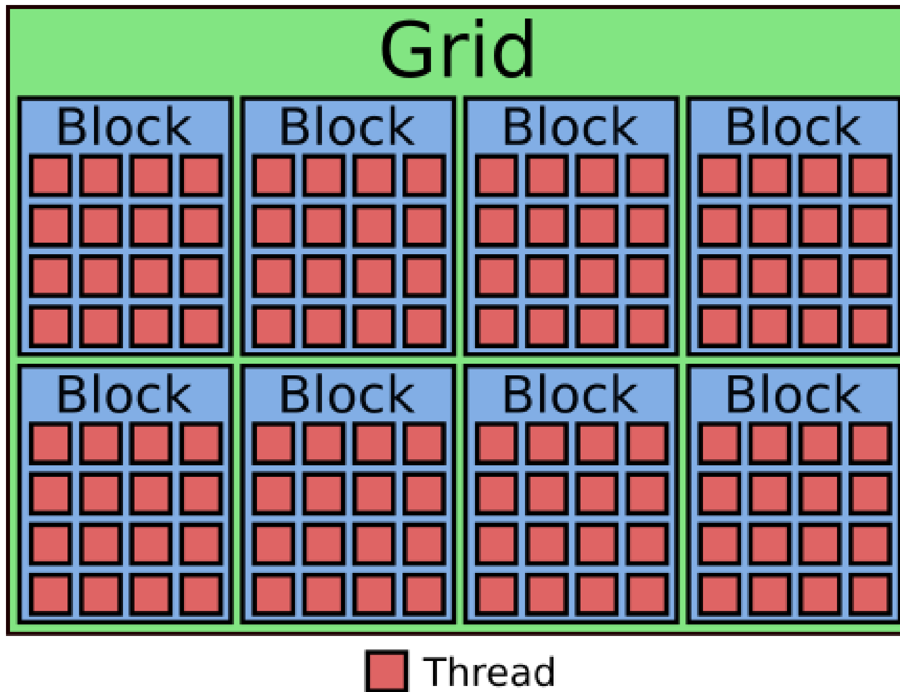


Figure 5.1: The hierachy of grids, blocks and threads

the same point. Additionally in newer GPUs using the Pascal and Volta architectures, the entire grid can be synchronised [75]. Synchronisation, especially above block level, can significantly impact performance, and should be kept to a minimum.

5.2.3 Memory Usage

While work performed on a CPU can use the main system memory (also known as host memory) directly, a GPU requires any data to be processed to be stored in its own memory. This section will discuss the various forms of GPU memory that are relevant to this thesis [73, 76].

The main part of a GPUs memory is known as global memory. Transferring data between host and global memory takes time, so the amount of data transferred should be minimised. The amount of global memory varies between GPUs. Modern, high-end GPUs typically have between 4GB and 24GB of global memory. While this memory makes up the largest part of the GPUs dedicated memory it is also the slowest to access.

In addition to global memory, the GPU also has access to a small amount of shared memory. Shared memory is significantly faster to access than global memory, but is very limited. Modern GPUs have 48KB available. When shared memory is requested, every block gets a portion of the shared memory, though requesting too much shared memory can limit the number of blocks that can be processed simultaneously, due to the shared memory limits. For example if a block required 25KB of shared memory only a single block would be able to run at a time, as there is not enough shared memory to execute two blocks simultaneously. Each block can only access memory locations for the portion

of shared memory allocated to it.

Finally, while not explicitly managed in the code, care must be taken with registers. In larger, more complex kernels, it is possible for the GPU to run out of registers, which are required for caching data while it is processed. If too many registers are requested by the kernel the occupancy of the device can be limited, until eventually the kernel simply cannot be executed. While the compiler manages what gets stored in registers, in complex kernels unnecessary variables should be kept to a minimum. The user can specify the maximum number of threads in a block, and the minimum number of blocks to be run on each multi-processor, using the `__launch_bounds__` arguments for the kernel [77]. These parameters allow the number of registers required by the kernel to be limited. This can affect the performance of the kernel, as more variables will be moved to memory rather than stored in registers. The increased latency of storing these variables in memory can often be offset by the increased occupancy. Due to this uncertainty, the optimal values for these arguments may need to be found experimentally.

As well as being aware of the memory limitations, and the different types of memory, it is also important to consider memory access patterns when developing algorithms for GPUs. Due to the latency of reading from and writing to global memory, it is important to make good use of the memory bandwidth. Modern GPUs, when retrieving data from global memory, will access continuous sections of data of 128 bytes, and it is not possible to extract multiple smaller segments instead of these continuous sections. This means that, if the data needed by a warp is spread out between many data segments, the GPU will have to read a significant amount more data than if all the data were grouped in a single segment. It is therefore important for threads to access contiguous memory locations where possible. This memory access is known as coalesced memory access. Failure to access memory this way, with data spread across multiple segments of memory, is known as strided memory access, and can result in significant slowdown of the program.

Figure 5.2 shows the difference between strided and coalesced memory access patterns. In the strided example, the required memory locations are spread across two segments of memory. In the coalesced example, this is all concentrated into a single segment. While both examples require the same amount of data from memory, the strided access results in twice the amount of data being retrieved. Note that there is only a stride of one between required memory locations in this example. As the stride increases more data will be required to be read.

When accessing shared memory, one needs to be aware of memory bank conflicts. Shared memory is split into banks, with successive words within shared memory belonging to different banks. When reading from shared memory complete words are retrieved. Bank conflicts occur when threads from the same warp request different values from the same bank in the same request. If this occurs then the data will be retrieved from that bank sequentially, slowing down the entire warp. It is important to note that if the same word is requested by multiple threads in the same warp, then the value will only be read once. This value will then be broadcast to all threads. Additionally, given that the access latency

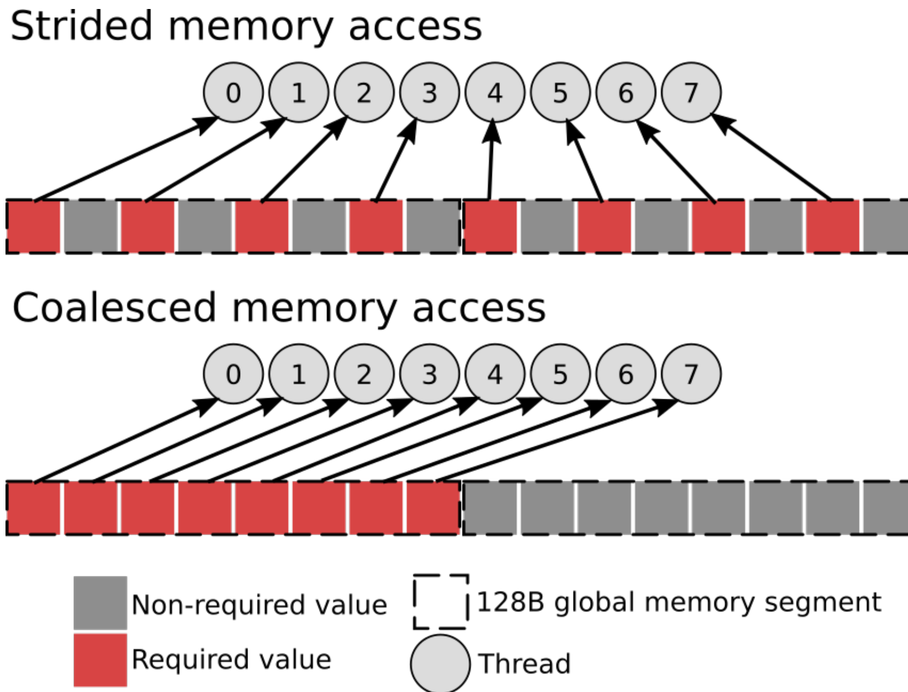


Figure 5.2: The difference between strided and coalesced global memory access

for shared memory is significantly lower than that of global memory, it is often still better to use shared memory, if bank conflicts cannot be avoided, than to use global memory.

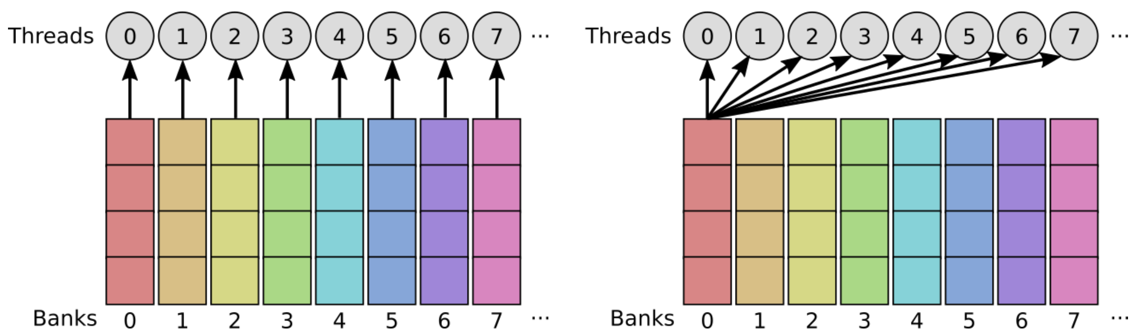


Figure 5.3: Efficient shared memory usage

Figure 5.3 shows two efficient uses of shared memory. For the sake of clarity only 8 of the 32 threads, and 8 of the 32 memory banks, have been shown. In the first diagram every thread reads from a different memory bank, meaning all values can be read simultaneously. The second diagram shows all threads reading from the same value from the same memory bank. In this case the value will be read, and then broadcast to all threads, meaning that the read is still performed efficiently.

Figure 5.4 shows two inefficient uses of shared memory. In both cases the threads are reading from the same memory bank as other threads. This will lead to sequential reads from memory. As data is not being retrieved from many of the memory banks this results in a significant amount of unused bandwidth, and thus an increased runtime from serialised memory accesses.

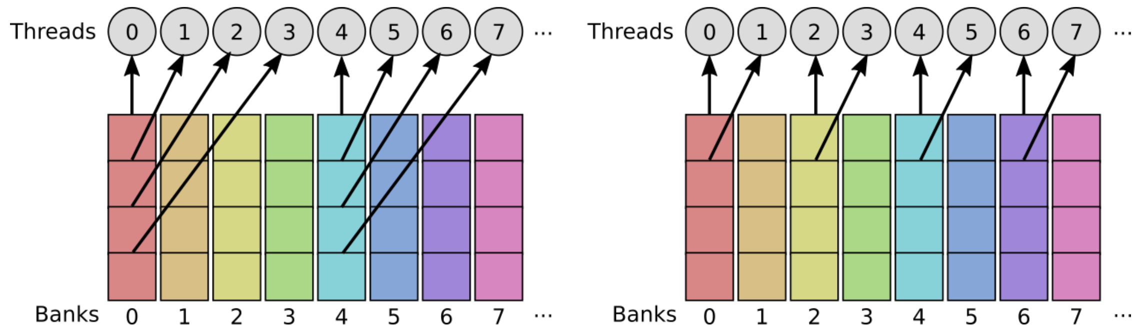


Figure 5.4: Poor shared memory usage

5.2.4 Warp Shuffling

Warp shuffling is a technique introduced in the NVIDIA Kepler architecture [74, 78]. This technique makes use of register swapping to allow threads to access registers of threads in the same block. This helps to reduce memory latency, as threads can read directly from these registers, rather than reading the values from memory.

One common usage of shuffle commands is in reduction algorithms, such as the prefix sum. In these algorithms an iterative approach allows the registers of threads to be read by the next thread in sequence, and allows for rapid amalgamation of data.

The main disadvantages of this technique are that many algorithms that make use of it require a significant amount of shared memory, which could limit the number of blocks that can be processed simultaneously. Additionally, registers can only be exchanged within the same block, which means that reductions of data above the maximum block size, currently 1024, must be batched, or split across multiple blocks.

5.3 Conclusion

This chapter gave an overview of how GPUs work, and how they are best utilised. While there is no existing research into accelerating the implementation of the computation of an AGCD, it is possible to infer from the parallel QR implementations discussed in Chapter 3. The next chapter will discuss how the GPU can be effectively utilised to accelerate the degree computation discussed in Chapter 4, implementing the parallel QR techniques seen in Chapter 3.

Chapter 6

GPU Acceleration of the Computation of the Degree of an AGCD

The most computationally expensive part of the algorithm described in Chapter 4 is the computation of the degree of an AGCD. Due to this computational expense, the computation of the degree of the AGCD was the first part of the BID algorithm considered for acceleration. This chapter will discuss optimisations and accelerations that can be made to this algorithm.

As discussed in Chapter 3, the computation of the degree reduces to a rank estimation problem, with the rank loss of the Sylvester matrix being equal to the degree of an AGCD. The algorithm adapted for this research was that of Winkler and Hasan, in their 2012 paper [21]. In this paper, two methods for computation of the degree of an AGCD were considered, using the QR decomposition and the SVD. This paper found that, while both methods computed the degree accurately, the QR algorithm had significant advantages with regards to computational complexity, due to algorithms providing an efficient method with which to update a QR decomposition after alterations are made to the original matrix. Thus the QR algorithm proposed by Winkler and Hasan is the approach that will be investigated in this research.

Throughout this chapter, as this is primarily an algorithm for computing an AGCD of polynomials, the degree of these polynomials will be discussed, as opposed to the width of the image. The degree of a polynomial in the BID algorithm is equal to the length of the row or column of pixels minus one.

Section 6.1 will describe the algorithm used to compute the degree of the AGCD. The primary focus in this section will be on the computation of the upper triangular factors of the subresultant matrices, as this is where the main algorithmic challenge lies. Section 6.2 presents an implementation of the algorithm described in Section 6.1. This section will describe how all of the kernels of the GPU accelerated implementation were designed, including pseudocode. Section 6.3 shows the results from this section, testing the algorithm

for both reliability and runtime, and comparing it against CPU implementations. Section 6.4 highlights the issues that this implementation has, and suggests ways in which the algorithm could be improved. Section 6.5 presents the tests that were performed on the algorithm with the reduced number of subresultant matrices, as originally described in Chapter 4.

6.1 Algorithm design

This section will discuss the design choices made when developing the algorithm to be implemented on a GPU. While certain hardware constraints must be taken into consideration to ensure the algorithm can perform efficiently on a GPU, the constraints considered in this section will primarily be high level, with more intricate details regarding the implementation of the algorithm on the specific hardware discussed in Section 6.2. This section will be split into two subsections. The first of these will focus on the algorithm for the computation of the upper triangular factors of the subresultant matrices. The second will focus on the two tests to compute the degree from the upper triangular factors, as was discussed in Chapter 4.

6.1.1 Computation of the Upper Triangular Factor of the Subresultant Matrices

The first step in computing the degree is the computation of the upper triangular factors of the subresultant matrices, these being matrices R from the QR decomposition. As was discussed in the profiling section of Chapter 4, computing these upper triangular matrices involves a large number of QR updates. In Chapter 3 several methods of QR update were investigated, and parallel implementations of these methods were discussed. In this section a parallel algorithm will be investigated that can be applied on the GPU to accelerate the computation of the upper triangular factors.

6.1.1.1 Update Method

Chapter 3 discussed various methods for updating the original orthogonal and upper triangular matrices Q and R to reflect a column deletion from the original matrix A. Given the numerical instability of the Gram-Schmidt algorithm, and the difficulty in applying it to QR updates as is required here, the Gram-Schmidt algorithm shall not be further considered for this implementation.

Therefore the choice of update method is between Givens rotations and Householder reflections. As discussed in Chapter 4, in each iteration of the QR update algorithm two non-contiguous columns are deleted from the Sylvester subresultant matrix S_k, to give the subresultant matrix S_{k+1}. These subresultant matrices can be separated into their orthogonal and upper triangular factors through QR decomposition such that $S_k = Q_k R_k$.

As was discussed in Chapter 4 the full decomposition of every subresultant matrix is unnecessary, and instead QR updates can be used to compute from R_k from R_{k-1}.

For each triangular factor $R_{2\ldots n}$, the column removed from index $m+n-2k+2$ (the last column of the matrix) will require no update, as was described in chapter 3. Therefore the only significant computations necessary will be the deletion of the column at $m-k+1$, which will require an update on the last half of the remaining columns of the matrix.

Householder and Givens update algorithms can both be used in this instance, and both result in the exact same computations for a QR update once the respective matrices have been constructed. Due to this, the difference between the update methods is relatively minor. Givens updates were selected due to the relative simplicity of constructing the Givens matrix compared to the Householder matrix.

6.1.1.2 Parallel Pattern

The implementation described in this chapter utilises a modified version of the Sameh and Kuck method, described in Chapter 3. Sameh and Kuck presented the first parallel pattern for QR decomposition using Givens rotations in their 1978 paper [62]. This algorithm was shown to provide speedups when implemented on a GPU in [63, 68, 64]. This section will describe how this algorithm is similar to the iterative approach utilised in the computation of the upper triangular factors of the subresultant matrices.

The Sameh and Kuck method presented in Chapter 3 was designed for the computation of a full QR decomposition, whereas the implementation presented in this research will involve a series of QR deletions. Thus the Sameh and Kuck algorithm needed to be modified to suit the purposes of this algorithm.

To understand how the algorithm was modified, it is important to examine the operations that are required to be computed, and how they can fit together into one coherent algorithm.

In this example, the upper triangular factor of the kth subresultant matrix S_k will be denoted as R_k. Consider the upper triangular matrix R_1 below,

$$R_1 = \begin{bmatrix} r_{1,1,1} & r_{1,1,2} & r_{1,1,3} & r_{1,1,4} & r_{1,1,5} & r_{1,1,6} & r_{1,1,7} & r_{1,1,8} & r_{1,1,9} & r_{1,1,10} \\ & r_{1,2,2} & r_{1,2,3} & r_{1,2,4} & r_{1,2,5} & r_{1,2,6} & r_{1,2,7} & r_{1,2,8} & r_{1,2,9} & r_{1,2,10} \\ & & r_{1,3,3} & r_{1,3,4} & r_{1,3,5} & r_{1,3,6} & r_{1,3,7} & r_{1,3,8} & r_{1,3,9} & r_{1,3,10} \\ & & & r_{1,4,4} & r_{1,4,5} & r_{1,4,6} & r_{1,4,7} & r_{1,4,8} & r_{1,4,9} & r_{1,4,10} \\ & & & & r_{1,5,5} & r_{1,5,6} & r_{1,5,7} & r_{1,5,8} & r_{1,5,9} & r_{1,5,10} \\ & & & & & r_{1,6,6} & r_{1,6,7} & r_{1,6,8} & r_{1,6,9} & r_{1,6,10} \\ & & & & & & r_{1,7,7} & r_{1,7,8} & r_{1,7,9} & r_{1,7,10} \\ & & & & & & & r_{1,8,8} & r_{1,8,9} & r_{1,8,10} \\ & & & & & & & & r_{1,9,9} & r_{1,9,10} \\ & & & & & & & & & r_{1,10,10} \end{bmatrix}.$$

The computation of all matrices $R_{2\ldots n}$ will be computed from this original upper triangular matrix R_1. To compute R_2 from R_1 the 5$^{\text{th}}$ and 10$^{\text{th}}$ columns will be deleted from R_1. The matrix with these columns removed, but not yet updated, will be referred to as $\tilde{R}_2$.

$$\tilde{R}_2 = \begin{bmatrix} r_{1,1,1} & r_{1,1,2} & r_{1,1,3} & r_{1,1,4} & r_{1,1,6} & r_{1,1,7} & r_{1,1,8} & r_{1,1,9} \\ & r_{1,2,2} & r_{1,2,3} & r_{1,2,4} & r_{1,2,6} & r_{1,2,7} & r_{1,2,8} & r_{1,2,9} \\ & & r_{1,3,3} & r_{1,3,4} & r_{1,3,6} & r_{1,3,7} & r_{1,3,8} & r_{1,3,9} \\ & & & r_{1,4,4} & r_{1,4,6} & r_{1,4,7} & r_{1,4,8} & r_{1,4,9} \\ & & & & r_{1,5,6} & r_{1,5,7} & r_{1,5,8} & r_{1,5,9} \\ & & & & r_{1,6,6} & r_{1,6,7} & r_{1,6,8} & r_{1,6,9} \\ & & & & & r_{1,7,7} & r_{1,7,8} & r_{1,7,9} \\ & & & & & & r_{1,8,8} & r_{1,8,9} \\ & & & & & & & r_{1,9,9} \end{bmatrix}.$$

Removing columns 4, 5, 9 and 10 from R_1 gives the matrix $\tilde{R}_3$

$$\tilde{R}_3 = \begin{bmatrix} r_{1,1,1} & r_{1,1,2} & r_{1,1,3} & r_{1,1,6} & r_{1,1,7} & r_{1,1,8} \\ & r_{1,2,2} & r_{1,2,3} & r_{1,2,6} & r_{1,2,7} & r_{1,2,8} \\ & & r_{1,3,3} & r_{1,3,6} & r_{1,3,7} & r_{1,3,8} \\ & & & r_{1,4,6} & r_{1,4,7} & r_{1,4,8} \\ & & & r_{1,5,6} & r_{1,5,7} & r_{1,5,8} \\ & & & r_{1,6,6} & r_{1,6,7} & r_{1,6,8} \\ & & & & r_{1,7,7} & r_{1,7,8} \\ & & & & & r_{1,8,8} \end{bmatrix},$$

and continuing this pattern results in the last two matrices.

$$\tilde{R}_4 = \begin{bmatrix} r_{1,1,1} & r_{1,1,2} & r_{1,1,6} & r_{1,1,7} \\ & r_{1,2,2} & r_{1,2,6} & r_{1,2,7} \\ & & r_{1,3,6} & r_{1,3,7} \\ & & r_{1,4,6} & r_{1,4,7} \\ & & r_{1,5,6} & r_{1,5,7} \\ & & r_{1,6,6} & r_{1,6,7} \\ & & & r_{1,7,7} \end{bmatrix}, \quad \tilde{R}_5 = \begin{bmatrix} r_{1,1,1} & r_{1,1,6} \\ & r_{1,2,6} \\ & r_{1,3,6} \\ & r_{1,4,6} \\ & r_{1,5,6} \\ & r_{1,6,6} \end{bmatrix}.$$

The first Givens rotation in every case will introduce a zero to the first column with non-zero entries below the diagonal, using the last two non-zero entries in this column, and the entire rows on which these entries reside. The aforementioned two entries are used to construct the Givens matrix, which is then applied to the entirety of the two rows. This process is described in detail in Chapter 3. Note that in all cases $R_{2\ldots5}$ the entries required for this update are identical, these values being $r_{1,5,6}$ and $r_{1,6,6}$. This means that the same rotation matrix will be applied to all of the upper triangular factors $\tilde{R}_{2\ldots5}$.

The submatrix on which the first rotation will be applied for R_2 is shown below.

Marked on this submatrix are the submatrices on which the first rotation must be applied for R_3, shown in red, R_4, shown in blue, and R_5, shown in green.

$$\begin{bmatrix} r_{1,5,6} & r_{1,5,7} & r_{1,5,8} & r_{1,5,9} \\ r_{1,6,6} & r_{1,6,7} & r_{1,6,8} & r_{1,6,9} \end{bmatrix} \tag{6.1}$$

It is therefore apparent that in computing the first rotation for R_2, the rotations for $R_{3\ldots5}$ are also computed, and are contained in a submatrix of the result. In the second iteration the rotations diverge. The rotation performed for R_2 is independent, whereas the result of the rotation for R_3 will contain the results of the rotations for R_4 and R_5. In the third iteration the rotations performed for R_2 and R_3 will be independent, and the result of the rotation for R_4 will contain the result for R_5. In the final iteration all of the rotations will be independent.

This can be seen as a modification of the Sameh and Kuck method by the order in which these computations take place. In the matrices below, the symbol $\bullet$ represents a value that is unchanged, and the symbol $\sim$ represents a non-zero value that has been changed by the Givens rotations. The entries i_c represent the zeros that are introduced by each Givens rotation, in a similar way to that shown when describing the Sameh and Kuck method in Chapter 3. In these entries i is the iteration in which the rotation occurs, and c represents the specific computation within that iteration. Note that in each computation i_c, the first matrix in which each unique computation occurs requires the full result, with the subsequent matrices using a submatrix of this result as is shown in Equation 6.1.

$$\tilde{R}_2 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & & & \sim & \sim & \sim & \sim \\ & & & & 1_1 & \sim & \sim & \sim \\ & & & & & 2_1 & \sim & \sim \\ & & & & & & 3_1 & \sim \\ & & & & & & & 4_1 \end{bmatrix}, \tilde{R}_3 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet & \bullet \\ & & & \sim & \sim & \sim \\ & & & 2_2 & \sim & \sim \\ & & & 1_1 & 3_2 & \sim \\ & & & & 2_1 & 4_2 \\ & & & & & 3_1 \end{bmatrix},$$

$$\tilde{R}_4 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet \\ & & \sim & \sim \\ & & 3_3 & \sim \\ & & 2_2 & 4_3 \\ & & 1_1 & 3_2 \\ & & & 2_1 \end{bmatrix}, \tilde{R}_5 = \begin{bmatrix} \bullet & \bullet \\ & \sim \\ & 4_4 \\ & 3_3 \\ & 2_2 \\ & 1_1 \end{bmatrix}.$$

Many duplicates can be seen for the computations, with the last non-zero diagonal in

all matrices performing similar computations to those present the last diagonal of matrix $\tilde{R}_2$. The results for each of the matrices $\tilde{R}_{3\ldots5}$ being a submatrix of the result for $\tilde{R}_2$. The computations on the second to last diagonals of $\tilde{R}_4$ and $\tilde{R}_5$ are the same as those on the second to last diagonal of $\tilde{R}_3$. The same is true of the third to last diagonals of $\tilde{R}_4$ and $\tilde{R}_5$, and the only unique computations are those where $i = 4$, which is the final iteration. These identical operations mean that simultaneously processing all of the deletions for all of the matrices would be computationally wasteful. Instead an iterative scheme where values are copied from partially completed matrices would be more efficient.

Figure 6.1 gives an overview of this algorithm. In iteration 1 of this algorithm a single Givens rotation is computed, that being computation 1_1 using the notation from the matrices shown above. Values from this operation can then be used in both of the Givens rotations in the second iteration. In the third iteration values can be copied from the rotations in the previous iterations and passed on to the matrix R_{k+1}. This can continue until all values are computed.

Example

This example uses the vectors f and g defined below. These vectors represent the coefficients of polynomials with an AGCD of degree 2. These were formed by randomly generating vectors of length 4 to represent the coefficients of polynomials of degree 3, and another vector of length 3 represents a GCD of degree 2. The vectors representing the polynomial coefficients were convolved with the GCD, and random noise was added to ensure the polynomials represented by these vectors are coprime.

$$f = [0.2859, 0.0643, 0.4355, 0.2200, 0.5833, 0.1864, 0.2441],$$

$$g = [0.0003, 0.1959, 0.2221, 0.3568, 0.4812, 0.1926, 0.2054].$$

Forming a Sylvester matrix with these vectors, and computing the upper triangular factor through QR decomposition, gives the 12×12 upper triangular matrix R_1 shown in Matrix 1. This will be used to compute the upper triangular factors of $S_{2\ldots n}$.

For each triangular matrix R_k, only entries in the lower right section of this matrix, containing the last n rows of the last n columns, will have changed from those in R_1. The submatrices consisting of this lower right section will be denoted as $\check{R}_k$. While these matrices are incomplete they will be represented as $\check{R}_{k,i}$ where i represents the iteration. The last n rows of the last n columns of R_1 will be copied to form the submatrix $\check{R}_1$,

$$\check{R}_1 = \begin{bmatrix} 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 & -12.9799 \\ 0 & 0.6583 & 0.7763 & -0.1310 & -3.0065 & -14.0045 \\ 0 & 0 & 0.2002 & 0.2093 & -1.0077 & 0.1921 \\ 0 & 0 & 0 & 0.9568 & -0.9891 & -2.7582 \\ 0 & 0 & 0 & 0 & -0.0002 & 0.0005 \\ 0 & 0 & 0 & 0 & 0 & 0.0002 \end{bmatrix}.$$

Iteration 1

The first iteration will compute the first Givens rotation for $\check{R}_2$. To preform this rotation rows must be extracted from R_1 and $\check{R}_1$. While it would be possible to extract both of these rows from R_1, as $\check{R}_1$ is a submatrix of R_1, it is useful to keep the pattern of extractions consistent throughout the algorithm. The matrices on which the Givens rotations will be performed will be denoted as $M_{k,i}$, where i is the number of the iteration in which that matrix is relevant. In this case, the first row of $M_{2,1}$ will be taken from the 6th row of R_1, from column 7 to column 11. The second row will be taken from the entries of the top row of $\check{R}_1$, excluding the last entry.

$$M_{2,1} = \begin{bmatrix} 0.9422 & -0.1825 & -0.0545 & -6.5186 & -18.1315 \\ 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 \end{bmatrix}.$$

Performing a Givens rotation on this matrix will provide the first row of the first updated factor $\check{R}_2$, and provisional values for the second row. Provisional values will be denoted as $\dot{r}$, whereas values yet to be computed will be denoted as $\cdot$.

$$\check{R}_{2,1} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & \dot{0.7616} & \dot{-0.5640} & \dot{1.8365} & \dot{0.2441} \\ 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdot \end{bmatrix}.$$

Iteration 2

In the second iteration the algorithm will continue processing the rows of $\check{R}_2$, and thus the matrix $M_{2,2}$ on which to apply the givens rotation must be constructed. The first row of $M_{2,2}$ will consist of the provisional entries for $\check{R}_2$ computed in the last iteration, that being row 2 of $\check{R}_{2,1}$. The second row of $M_{1,2}$ will be extracted from the values on row 2 of $\check{R}_1$.

$$M_{2,2} = \begin{bmatrix} 0.7616 & -0.5640 & 1.8365 & 0.2441 \\ 0.6583 & 0.7763 & -0.1310 & -3.0065 \end{bmatrix}.$$

A Givens rotation is performed on $M_{2,2}$, and the values substituted back in to $\check{R}_{2,1}$ to give $\check{R}_{2,2}$.

$$\check{R}_{2,2} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & 1.0067 & 0.0810 & 1.3037 & -1.7815 \\ 0 & 0 & \dot{0.9562} & \dot{-1.3001} & \dot{-2.4341} \\ 0 & 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdot \end{bmatrix}.$$

In the previous iteration final values were computed for the first row of $\check{R}_2$, and thus all

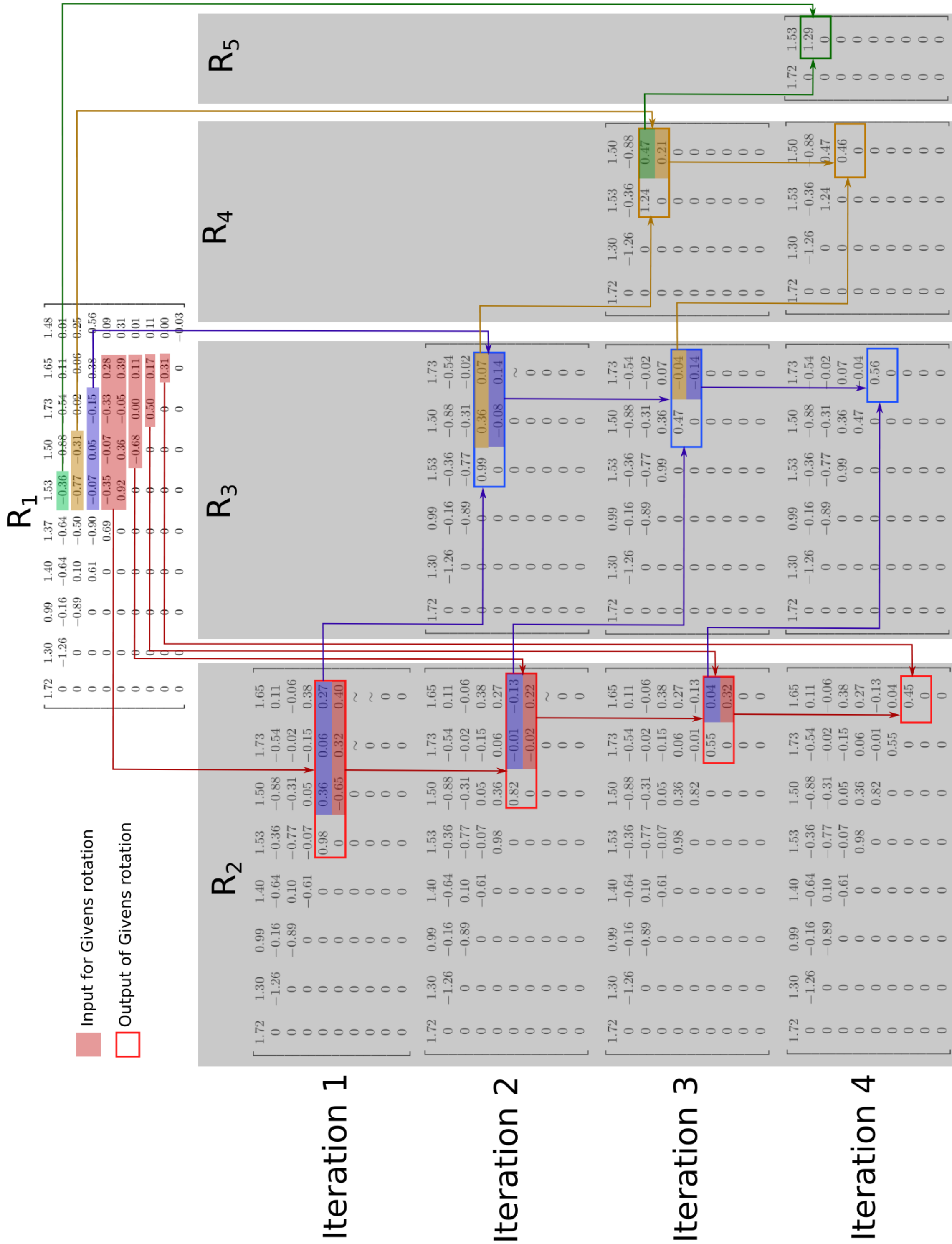


Figure 6.1: The structure of the matrix computations in this algorithm

$$R_1 = \begin{bmatrix}
-18.1874 & -2.2196 & -4.6050 & -1.0025 & -0.9800 & -0.1301 & -4.0187 & -4.9968 & -1.3858 & -1.0327 & -0.1584 & -0.0667 \\
0 & -18.0515 & -1.6701 & -4.5164 & -0.8896 & -0.9713 & -18.6954 & -3.4346 & -4.8640 & -1.2693 & -1.0210 & -0.1514 \\
0 & 0 & -17.5153 & -1.6105 & -4.4392 & -0.9142 & -6.5197 & -18.1357 & -3.3448 & -4.7960 & -1.3000 & -1.0403 \\
0 & 0 & 0 & -17.5153 & -1.6111 & -4.4397 & -0.0545 & -6.5197 & -18.1358 & -3.3456 & -4.7967 & -1.3005 \\
0 & 0 & 0 & 0 & -17.5137 & -1.6082 & -0.1825 & -0.0545 & -6.5190 & -18.1331 & -3.3418 & -4.7943 \\
0 & 0 & 0 & 0 & 0 & -17.5125 & 0.9422 & -0.1825 & -0.0545 & -6.5186 & -18.1315 & -3.3404 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 & -12.9799 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6583 & 0.7763 & -0.1310 & -3.0065 & -14.0045 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2002 & 0.2093 & -1.0077 & 0.1921 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9568 & -0.9891 & -2.7582 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.0002 & 0.0005 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0002
\end{bmatrix}$$

Matrix 1: The matrix R_1 for use in the example in Section 6.1.1.2

values required for the first Givens rotation of $\check{R}_3$ are now available. The first computation of $\check{R}_3$ can therefore be run in parallel with those for $\check{R}_2$ in this iteration. The matrix $M_{3,2}$ is constructed using row 5 from the original matrix R_1, and the first row of $\check{R}_{2,1}$, the final entries of which were computed in the last iteration.

$$M_{3,2} = \begin{bmatrix} -0.1825 & -0.0545 & -6.5190 & -18.1331 \\ 0.9917 & 0.0580 & -0.2426 & -6.2580 \end{bmatrix}.$$

Performing a Givens rotation on $M_{3,2}$ gives the final values for the first row of $\check{R}_3$ and provisional values for the second row.

$$\check{R}_{3,2} = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \\ 0 & 0.0\dot{4}31 & 6.4\dot{5}53 & 18.\dot{9}662 \\ 0 & 0 & \cdot & \cdot \\ 0 & 0 & 0 & \cdot \end{bmatrix}.$$

Iteration 3

In the third iteration enough values have been computed to perform three Givens rotations simultaneously, continuing the computation of $\check{R}_2$ and $\check{R}_3$, and starting the computation of $\check{R}_4$.

The values necessary for the next rotation of $\check{R}_2$ are retrieved from row 3 of $\check{R}_1$, and the preliminary values from $\check{R}_{2,2}$, in the same pattern as in the previous iterations.

$$M_{2,3} = \begin{bmatrix} 0.9562 & -1.3001 & -2.4341 \\ 0.2002 & 0.2093 & -1.0077 \end{bmatrix}.$$

A Givens rotation is performed on $M_{2,3}$, and the result is inserted into $\check{R}_{2,2}$ to give $\check{R}_{2,3}$,

$$\check{R}_{2,3} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & 1.0067 & 0.0810 & 1.3037 & -1.7815 \\ 0 & 0 & 0.9769 & -1.2296 & -2.5889 \\ 0 & 0 & 0 & 0.4\dot{7}13 & -0.\dot{4}876 \\ 0 & 0 & 0 & 0 & \cdot \end{bmatrix}.$$

$M_{3,3}$ is constructed by using the preliminary values from the first row of $\check{R}_{3,2}$ and from the final values computed in iteration 2 from the second row of $\check{R}_{2,2}$,

$$M_{3,3} = \begin{bmatrix} 0.0431 & 6.4553 & 18.9662 \\ 1.0067 & 0.0810 & 1.3037 \end{bmatrix}.$$

The result of the Givens rotation on $M_{3,3}$ is inserted into $\check{R}_{3,2}$ to give $\check{R}_{3,3}$,

$$\check{R}_{3,3} = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \\ 0 & 1.0076 & 0.3572 & 2.1142 \\ 0 & 0 & -6.4459 & -18.8931 \\ 0 & 0 & 0 & \cdot \end{bmatrix}.$$

As the first values have been computed for $\check{R}_3$ in the previous iteration, computation can start on $\check{R}_4$. As with the first computations for $\check{R}_2$ and $\check{R}_3$, the upper row of $M_{4,3}$ must be extracted from the original matrix R_1, this time from the 4$^{\text{th}}$ row, and the second row will be taken from the final values from $\check{R}_{3,2}$.

$$M_{4,3} = \begin{bmatrix} -0.0545 & -6.5197 & -18.1358 \\ 1.0084 & 0.0669 & 0.9412 \end{bmatrix}.$$

The Givens rotation is performed and the result is used to start the construction of $\check{R}_4$ with the matrix $\check{R}_{4,3}$ shown below.

$$\check{R}_{4,3} = \begin{bmatrix} 1.0098 & 0.4185 & 1.9182 \\ 0 & 6.5066 & 18.0587 \\ 0 & 0 & \cdot \end{bmatrix}.$$

Iteration 4

In this iteration, computation continues in the same pattern for $R_{2\ldots4}$, and a rotation can be performed for $\check{R}_5$.

As with previous iterations, matrices are constructed for $k = 2\ldots i$. In each case the preliminary values from $\check{R}_{k,i-1}$ are used for the first row, and the final values from $\check{R}_{k-1,i-1}$ are used for the second row.

$$M_{2,4} = \begin{bmatrix} 0.4713 & -0.4876 \\ 0.9568 & -0.9891 \end{bmatrix},$$

$$M_{3,4} = \begin{bmatrix} -6.4459 & -18.8931 \\ 0.9769 & -1.2296 \end{bmatrix},$$

$$M_{4,4} = \begin{bmatrix} 6.5066 & 18.0587 \\ 1.0076 & 0.3572 \end{bmatrix}.$$

The exception to this is $M_{5,4}$, as no provisional entries have been computed for R_5 at this point. $M_{5,4}$ will use values from the third row of R_1 for the first row, and the first row of $\check{R}_{4,3}$ will be used for the second row,

$$M_{5,4} = \begin{bmatrix} -6.5197 & -18.1357 \\ 1.0098 & 0.4185 \end{bmatrix}.$$

A Givens rotation is performed on each of these matrices, and the values inserted into

$\check{R}_{k,3}$ to give the matrices $\check{R}_{k,4}$.

$$\check{R}_{2,4} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & 1.0067 & 0.0810 & 1.3037 & -1.7815 \\ 0 & 0 & 0.9769 & -1.2296 & -2.5889 \\ 0 & 0 & 0 & 1.0666 & -1.1027 \\ 0 & 0 & 0 & 0 & 0.0004 \end{bmatrix},$$

$$\check{R}_{3,4} = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \\ 0 & 1.0076 & 0.3572 & 2.1142 \\ 0 & 0 & 6.5195 & 18.4955 \\ 0 & 0 & 0 & 4.0467 \end{bmatrix},$$

$$\check{R}_{4,4} = \begin{bmatrix} 1.0098 & 0.4185 & 1.9182 \\ 0 & 6.5841 & 17.9006 \\ 0 & 0 & -2.4106 \end{bmatrix},$$

$$\check{R}_{5,4} = \begin{bmatrix} 6.5974 & 17.9861 \\ 0 & 2.3623 \end{bmatrix}.$$

Iteration 5

In the final iteration the last entry will be computed for each triangular factor, including the only entry of $\check{R}_6$.

The matrices $M_{k,5}$ are constructed using the same method as previous iterations.

$$M_{2,5} = \begin{bmatrix} 0.0004 \\ -0.0002 \end{bmatrix},$$

$$M_{3,5} = \begin{bmatrix} 4.0467 \\ 1.0666 \end{bmatrix},$$

$$M_{4,5} = \begin{bmatrix} -2.4106 \\ 6.5195 \end{bmatrix},$$

$$M_{5,5} = \begin{bmatrix} 2.3623 \\ 6.5841 \end{bmatrix},$$

$$M_{6,5} = \begin{bmatrix} -18.6954 \\ 6.5974 \end{bmatrix}.$$

Givens rotations are performed on the matrices above to give the final entry for the last row of each triangular matrix, leaving the final forms for all the triangular factors of the subresultant matrices.

$$
\check{R}_2 = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & 1.0067 & 0.0810 & 1.3037 & -1.7815 \\ 0 & 0 & 0.9769 & -1.2296 & -2.5889 \\ 0 & 0 & 0 & 1.0666 & -1.1027 \\ 0 & 0 & 0 & 0 & 0.0004 \end{bmatrix},
$$

$$
\check{R}_3 = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \\ 0 & 1.0076 & 0.3572 & 2.1142 \\ 0 & 0 & 6.5195 & 18.4955 \\ 0 & 0 & 0 & 4.1849 \end{bmatrix},
$$

$$
\check{R}_4 = \begin{bmatrix} 1.0098 & 0.4185 & 1.9182 \\ 0 & 6.5841 & 17.9006 \\ 0 & 0 & 6.9509 \end{bmatrix},
$$

$$
\check{R}_5 = \begin{bmatrix} 6.5974 & 17.9861 \\ 0 & 6.9950 \end{bmatrix},
$$

$$
\check{R}_6 = \begin{bmatrix} 19.8253 \end{bmatrix}.
$$

6.1.2 Computation of Rank and Degree Estimation

While the computation of the upper triangular factors of the subresultant matrices requires a detailed algorithmic design in order to maximise the amount of parallelism, the parallelism potential of the tests used to estimate the degree of the AGCD described in Chapter 4 is more apparent.

The most computationally expensive part of these tests is that of the squaring and summation of all entries of the rows of every matrix to compute the row norms. This is a reduction algorithm, and can be performed efficiently in parallel with methods such as the warp shuffle prefix sum described in Chapter 5. The sums of all the rows of all the matrices can be computed simultaneously, and thus there is a significant amount of parallel potential.

Compared to the computation of the row norms, the computation of the minimum and maximum diagonal entries, and the minimum and maximum of the computed row norms, is relatively trivial. This could also be parallelised as a reduction algorithm, and again the computations on different triangular matrices can be performed simultaneously.

As the parallel potential of these algorithms is more apparent, these computations present more of an implementation problem than an algorithmic problem. Therefore the detail of how this section is implemented will be discussed in Section 6.2.3.

6.2 Implementation

This section will discuss the implementation of the algorithm presented in Section 6.1, starting with a discussion of how all of the subresultant matrices can be stored efficiently. This is necessary in order to make good use of the limited GPU memory. The implementation of an algorithm on a GPU is approached as a iterative process of development, profiling and optimisation. To reflect this, the implementation will be discussed in terms of an initial implementation, and improvements that were made to the initial implementation, to achieve a final optimised version. While the profiling that took place at each stage of optimisation will not be discussed, the profiling performed on the final implementation will be presented at the end of this section.

6.2.1 Efficient Storage of Upper Triangular Matrices

As discussed in Chapter 5, one of the challenges of implementing algorithms on a GPU is ensuring efficient utilisation of the device memory. It is therefore important to minimise the amount of memory each matrix takes up on the GPU. As the matrices being used have a predictable shape, with only zeros below the diagonal, the easiest way to minimise this footprint is to not store these zeros. Instead, each triangular matrix should be stored as a vector, with the zeros removed.

Additionally, as discussed in Chapter 3, when performing a QR column deletion only the entries on or below the principal diagonal at the point where the column is removed will change. This can be seen in Figure 6.1 where only values in the lower $n-k-1$ rows of each triangular factor have Givens rotations performed on them. While the entries above this point are still necessary for the computation of the rank, and thus the computation of the degree, the values of these entries can simply be extracted from the original upper triangular matrices when necessary.

Due to the method of storing triangular matrices, an efficient method of referencing the indices of these matrices is required. Therefore the mathematics of triangular numbers arises several times. The following sections will define the formulae for computing triangular numbers themselves, and the triangular root.

6.2.1.1 Triangular Numbers to Compute Matrix Indices

Triangular numbers can be used to provide the size of an efficiently stored upper triangular matrix, and are therefore useful at various stages of this algorithm. It is therefore necessary to gain an understanding of the mathematics of triangular numbers to understand how the efficiently stored matrices are utilised.

A triangular number t_n is the count of objects arranged into an equilateral triangle, with n elements on each side [79].

Figure 6.2 shows the first 6 triangular numbers, and how the width n relates to the number of elements t_n. From this it is easy to see the application in referencing indices of triangular matrices, where the non-zero elements are not stored. An upper triangular

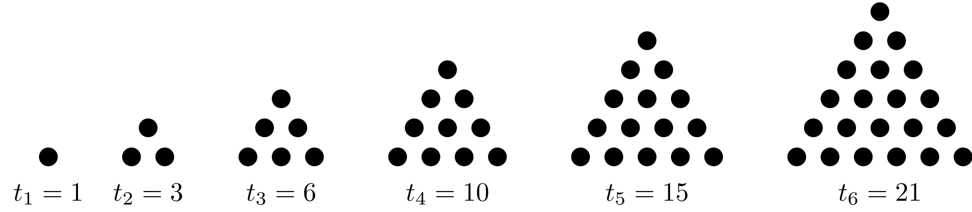


Figure 6.2: Diagram showing the progression of triangular numbers

matrix with n columns will have t_n non-zero elements. This can be seen in matrices $M_{1\ldots6}$ below. Each of these matrices M_n has n columns, with t_n non-zero elements, similar to Figure 6.2.

$$M_1 = \begin{bmatrix} \bullet \end{bmatrix}, M_2 = \begin{bmatrix} \bullet & \bullet \\ 0 & \bullet \end{bmatrix}, M_3 = \begin{bmatrix} \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet \\ 0 & 0 & \bullet \end{bmatrix}, M_4 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & \bullet \end{bmatrix},$$

$$M_5 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & 0 & \bullet \end{bmatrix}, M_6 = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & 0 & 0 & \bullet \end{bmatrix}.$$

Thus the computation of triangular numbers is the count of the number of elements above the diagonal in a square matrix.

$$\begin{aligned} T_n &= n + (n-1) + (n-2) + (n-3) \ldots 1 \\ &= \frac{n(n+1)}{2}. \end{aligned}$$

Triangular numbers can also provide the indices for a row in an efficiently stored upper triangular matrix. Given an $n \times n$ upper triangular matrix, the index of the ith row can be computed with $T_n - T_{n-(i-1)}$.

Take for example upper triangular matrix M_6 below. The non-zero entries of this matrix are denoted by their index within the efficiently stored vector form of the triangular matrix, starting at 0.

$$M_6 = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ & 6 & 7 & 8 & 9 & 10 \\ & & 11 & 12 & 13 & 14 \\ & & & 15 & 16 & 17 \\ & & & & 18 & 19 \\ & & & & & 20 \end{bmatrix}.$$

Consider the computation of the index of the start of the 4th row of M_6. In this case $n = 6$ and $i = 4$.

$$T_6 = \frac{6 \times 7}{2} = 21,$$

$$T_{6-(4-1)} = T_3 = \frac{3 \times 4}{2} = 6.$$

Thus with the two triangular numbers computed the index of this row can be computed.

$$T_6 - T_3 = 15.$$

As can be seen in M_6, the first entry on the 4th row is at the 15th index, therefore the correct index has been computed.

6.2.1.2 Triangular Roots

It is sometimes necessary to determine what row a given index within the efficiently stored matrix is on, or to identify if the index is located at one of the ends of its row. As was described in Section 6.2.1.1, triangular numbers can be used to find the index of a row. The inverse of this can be computed with a triangular root [80], where n must be computed from T_n.

The triangular root of an arbitrary integer a can be computed using the formula from [80]

$$n = \frac{\sqrt{8a+1} - 1}{2}.$$

If this formula results in an integer the number is triangular. The floor of the result from this computation can be used to find the row on which an index lies.

It is also possible to check whether an arbitrary number a is triangular or not, and thus whether it lies at the end of a row. If, and only if, $8a + 1$ is square then $a - 1$ is a triangular number.

6.2.2 Computation of the Upper Triangular Factors

Section 6.1.1 discussed where parallelisation can be found within the process of computing all upper triangular factors of the subresultant matrices. The algorithm discussed in that

section provides a significant amount of parallelisation, which makes it suitable to be implemented on a GPU. This section will detail how the algorithm was implemented on a GPU, and challenges that were overcome to this end.

6.2.2.1 Initial Implementation

As with many complex CUDA implementations, the first step is to get a simple, non-optimal, version of the algorithm working on a GPU. Thus the initial implementation did not account for efficient memory access patterns, and did not use any load balancing or advanced techniques. Therefore this initial implementation was not expected to perform well, and instead just served as a starting point on which to improve through an iterative process of optimisation.

A basic kernel was developed that took the original upper triangular matrix R_1 and distributed the columns of the matrix product of each Givens rotation across threads in a block, with different blocks processing different trials. Due to the lack of load balancing and work distribution, this meant that the original implementation was limited to images with a width of 1024, the maximum size of a block. Additionally, every thread recomputed the entries of the relevant Givens matrix, leading to a lot of unnecessary computation. The memory access pattern also had problems with strided memory access, which was the cause of the most significant slowdown in the original implementation.

As mentioned previously, a significant part of implementing any algorithm on a GPU is that of optimisation. This is an iterative process of profiling, identifying bottlenecks in performance, and addressing those bottlenecks. The following sections will detail some of the changes made through this process of iterative optimisation to reach a final implementation that was thoroughly tested to check its reliability and runtime.

6.2.2.2 Load Balancing

Processing of triangular matrices can lead to inherently unbalanced algorithms, as the number of non-zero elements is different on each row. This presents a significant problem when processing these matrices on a GPU, where the number of blocks and threads allocated to a task is decided before compilation and, unless dynamic parallelism is used by nesting kernel calls, does not change while the kernel is executing. Dynamic parallelism was decided to not be a good fit for this problem, as the number of nested kernel calls that would be required would lead to a large amount of overhead. Instead, a novel load balancing technique was used.

The aim of the load balancing technique is to minimise the amount of time that threads spend idling, to ensure that the work can be done in as few iterations as possible. This technique assigns all units of work a unique work index, and batches the work such that all batches, except for the final batch, will have no idle threads. As discussed in Section 6.2.2.3 the computations should be arranged so that operations on the same matrix occur in sequential threads.

```
1  // n is the degree of the polynomials
2  // i is the iteration of the algorithm
3
4  numPerMatrix ← (n/2)-i
5  totalWork ← i*numPerMatrix
6  batches ← ceil(totalWork / blockSize)
7
8  for batch ← 0 to batches
9      workUnit ← blockSize * batch + threadID
10     if (workUnit < totalWork)
11         matrixIndex ← workUnit % numPerMatrix
12         columnIndex ← workUnit / numPerMatrix
13         //DO WORK
```

Listing 1: Pseudocode showing the computation of the work index and how the load balancing is executed

Listing 1 shows the pseudocode for performing this load balancing technique. Lines 4-6 show the initial set up, where the width of the matrix product for each matrix is computed, as well as the total work, and how many batches to split the work in to. Line 9 shows how the work unit index is computed, and Line 10 performs a check to ensure no processing is done for the work units beyond the total amount of work. Finally, Lines 11-13 show the work for each index taking place.

Figure 6.3 shows how work is distributed, both before and after the load balancing, when processing a Sylvester matrix of width 18. Note that when using the load balancing technique the only idle threads that occur are those in the final iteration. By utilising threads that would otherwise be idle, the load balancing technique allows the algorithm to better utilise the device, and complete the computation in fewer iterations.

6.2.2.3 Avoiding Strided Memory

Strided memory, as discussed in Chapter 5, is where threads in the same warp access data in non-adjacent memory locations. This results in the necessary data reads being spread across multiple memory units, which is sub-optimal as it means that memory bandwidth will be wasted extracting unnecessary data, and more overall memory accesses are required. It is therefore important that memory access, wherever possible, is coalesced.

In the initial implementation, memory access was strided, as contiguous threads worked on different values for k, and thus different triangular matrices. The original method was developed this way to make it simpler to compute the entries of the Givens matrices inside the main loop of the program without repeating this computation, and without leading to an unbalanced workload. However, this caused the memory access of the matrix product computations, for which the majority of the memory accesses take place, to be strided.

Section 6.2.2.2 describes a method of load balancing the algorithm, the matrix index and column index in Listing 1 shows the method for computing the indices for performing the matrix product computations in a coalesced form. This does however raise the issue of where to compute the entries of the Givens matrices. Initially these were just performed when they were needed by every thread, but to avoid unnecessary work the Givens matrices

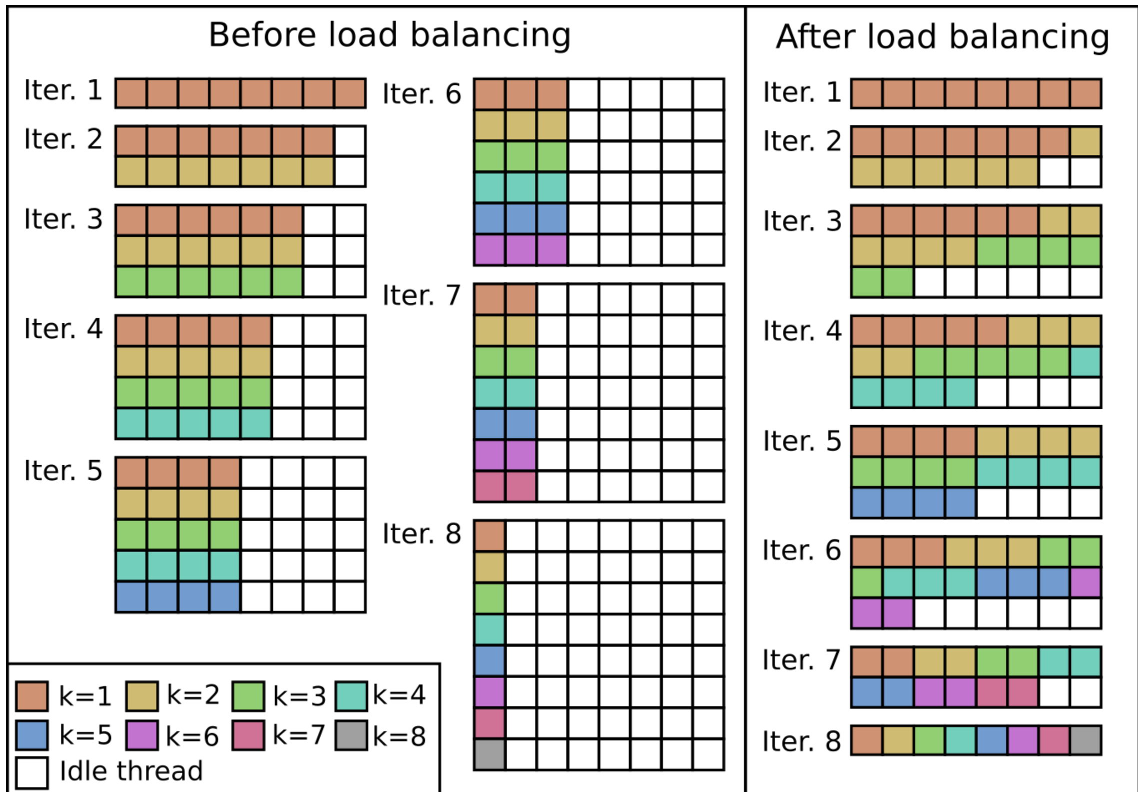


Figure 6.3: Diagram demonstrating how work is balanced before and after load balancing is applied

were instead computed in the first n work units of every iteration. While this does increase the amount of pointer arithmetic needed, as work units will need different rows for the computation of the Givens matrices, it also reduces the overall number of operations while still maintaining the balance of the implementation. This can be seen in Listing 2.

6.2.2.4 Batching

As discussed in Chapter 5, GPUs have their own dedicated memory. Unfortunately, given this memory is limited, this can place limitations on the amount of data that can be processed in a single kernel call in this implementation. When the degrees of the polynomials being processed are sufficiently large it is no longer possible to process all trials in the same kernel call, and instead they must be copied to the GPU in batches. When the polynomials to be processed reach a certain degree they will not be able to be processed at all by this implementation of the algorithm, as there will not be enough memory on the device to process even a single trial.

Figure 6.4 shows how the device memory used by this implementation of the algorithm scales for a single trial with increasing polynomial degrees. Three well known models of NVIDIA GPU are listed, along with their memory capacity. These GPUs are marked on the curve at the maximum degree of polynomial that they can process, assuming the polynomial degrees are equal. These degrees are only theoretical maximums, as actual

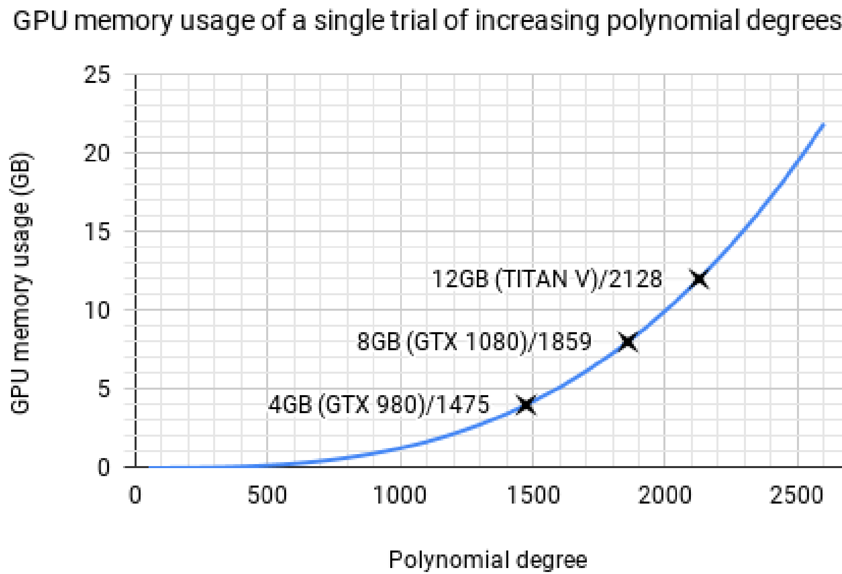


Figure 6.4: Graph showing theoretical GPU memory usage of computing a single trial of an AGCD computation on the GPU

GPU memory available will potentially be lower. This is due to memory overheads introduced by programs such as MATLAB. The curve is cubic, which could be problematic if the algorithm is required to process polynomials of high degree, or use a limited amount of GPU memory. This problem will be discussed further and addressed in Chapter 7.

6.2.2.5 Final Implementation

The preceding sections described how the initial implementation was improved in various ways. The pseudocode for the final implementation of the computation of $\check{R}_{2\ldots n}$ is provided in Listing 2, and a brief overview of the code will be provided in this section. While in the rest of this thesis the value of k, the index of the subresultant matrices, starts at 1, with R_1 being the upper triangular factor of the Sylvester matrix, in the pseudocode the value of k starts at 0. While not standard notation this change makes indexing the matrices in the code more concise and legible.

The kernel in Listing 2 computes one set of triangular factors per block, with different trials being computed in different blocks. The outer loop drives the iterative nature of the algorithm. As this counter increases, the implementation moves through the steps described in Section 6.2.2.2. This gives each thread in every batch a unique index, which can be used to perform the computation for the corresponding operation. Once the work unit index has been computed, the main body of the implementation can be broken down into two parts. Firstly the computation of the Givens matrices for this iteration, and secondly performing the Givens rotations with the computed Givens matrices.

The Givens matrices are computed first, on lines 18 to 30, to avoid unnecessary recomputation during the matrix products. This is performed in the first batch when the block

width is greater than or equal to n, but will overflow when this is not the case. Contiguous work units are used to compute the entries of Givens matrices for contiguous values of k.

A check is made on line 18 so that only the correct number of threads are used to compute the entries of the Givens matrices. This ensures that in the first iteration only a single Givens rotation is computed, while the other threads idle, but this increases on each iteration. When the number of threads in the block is less than the number of Givens matrices that need to be computed, this process must be batched. This occurs in the same loop as the processing of the Givens rotations. Because the Givens matrix computation and the rotations are performed in the same order, this means that there will never be a computation where the Givens matrix is required before the batch that computes it.

The values that make up each Givens matrix are stored in an array of structures in shared memory. Threads must be synchronised after this section to avoid any rotations being computed before the relevant Givens matrices are computed.

The second part of the main body, where the Givens rotation is performed, splits the work into columns. This happens on lines 34 to 50. Each work unit processes a single column of the matrix product, and stores this in the memory structure of the respective triangular factor. Again, threads must be synchronised after this computation to prevent race conditions.

An important part of both the computation of the Givens matrices and computation of the matrix product is finding the correct rows from which to extract the values, as shown in lines 22 to 26 and 42 to 46. This is due to the method in which the triangular factors are stored, as described in Section 6.2.1. Each matrix on which the Givens rotations are applied consists of two rows. The first row to be retrieved differs depending on whether there has been a previous rotation for R_k. When $\texttt{pass} = k$ the first row of this matrix will be extracted from the original upper triangular matrix R_1. When $\texttt{pass} > k$ then the first row will be retrieved from the preliminary values in the preceding iteration, which will have been stored in the memory locations for R_k. The second row will always be retrieved from the final values computed in the preceding iteration that are stored in R_{k-1}.

6.2.3 Estimating the Degree

As discussed in Section 6.1.2, the implementation of the degree estimation, through the two tests described in Chapter 4, is primarily an implementation problem. This section will discuss the implementation of these algorithms. Much of the challenge of this implementation comes from the storage method that was selected for the triangular matrices, as was discussed in Section 6.2.1.

Each factor of the subresultant matrices R_k will be considered as an upper section $\hat{R}_k$, consisting of the first m rows, and a lower section $\check{R}_k$, consisting of the last n rows. The previous section considered the computation of the matrices $\check{R}_{2\ldots n}$, and these are stored in their own memory structures. The entries of the upper section of each factor, $\hat{R}_{2\ldots n}$, therefore need to be extracted from entries of the upper m rows of the original factor R_1. The submatrices $\check{R}_k$ and $\hat{R}_k$ therefore require very different access patterns for the

```
1  // m is the degree of the larger polynomial
2  // n is the degree of the smaller polynomial
3  // R1 is the original R matrix for this block stored in upper triangular form
4  // Mats is an array of arrays where the triangular form of the lower right quarter
5  //      of all upper triangular factors of the subresultant matrices for this block
6  //      will be stored
7  // G_SM is an array of structures containing the Givens values c and s stored in
8  //      shared memory
9
10 gpu_parallel_for threadID ← 0 to blockWidth
11     for pass ← 0 to n
12         totalWork ← pass * (n - pass)
13         numBatches ← ceil(totalWork / blockWidth)
14         for batch ← 0 to numBatches
15             wu ← blockWidth * batch + threadID
16
17             //Calculate Givens values
18             if (wu < pass)
19                 k ← wu + 1
20                 rowIndex ← pass-k
21
22                 if (rowIndex = 0)
23                     val1 ← R1[trangle(2*n) - triangle(n-k) + k]
24                 else
25                     val1 ← Mats[k][triangle(n-k) - triangle(n-k-rowIndex)]
26                 val2 ← Mats[k-1][triangle(n-(k-1)) - triangle(n-(k-1)-rowIndex)]
27
28                 na ← sqrt(val1*val1 + val2*val2)
29                 G_SM[wu].c ← val1/na
30                 G_SM[wu].s ← val2/na
31
32             synchroniseThreads
33
34             //Perform Givens rotations
35             if (wu < totalWork)
36                 k ← wu / (n - pass) + 1
37                 i ← wu % (n - pass)
38
39                 rowIndex ← pass-k
40                 rowWidth ← n/2-k
41
42                 if (rowIndex = 0)
43                     val1 ← R1[triangle(2*n) - triangle(n-k) + k + i]
44                 else
45                     val1 ← Mats[k][triangle(n-k) - triangle(n-k-rowIndex) + i]
46                 val2 ← Mats[k-1][triangle(n-(k-1)) - triangle(n-(k-1)-rowIndex) + i]
47
48                 Mats[k][triangle(n-k)-triangle(n-k-(rowIndex))] ←
   val1*G_SM[k-1].c + val2*G_SM[k-1].s
49                 if (rowIndex < rowWidth-1) and (i!= 0)
50                     Mats[k][triangle(n-k)-triangle(n-k-(rowIndex+1))] ←
   val1*-G_SM[k-1].-s + val2*G_SM[k-1].c
51
52             synchroniseThreads
```

Listing 2: Pseudocode for computing the triangular factor of the subresultant matrices

computation of the row norms, and thus it is best to split this section into multiple kernels.

Due to the simplicity of the task, the computation of the minimum and maximum values of the diagonals, and the corresponding ratio, can occur in a single kernel. The computation of the upper and lower row norms will be split into two kernels, as there is greater complexity in these computations. The computation of the minimum and maximum norms, and the corresponding ratio, occurs in a separate kernel. The gradients for both sets of ratios can then be computed in another kernel. Once gradients have been computed the minimum gradient will simply be found on the CPU. As the kernels presented in this section are relatively simple implementations, when compared to the computation of the upper triangular factors of the subresultant matrices, they required less optimisation. Therefore the sections describing these kernels will simply discuss the final implementations.

6.2.3.1 Computing the Diagonal Ratios

A simple kernel was designed to compute the minimum and maximum diagonals for both the upper and lower sections of each triangular factor. While the presented approach is not optimal, the runtime is not particularly significant when compared to other sections of the implementation. Therefore it was not considered high priority for further optimisation.

The pseudocode for this kernel is shown in Listing 3. This kernel uses a single block to process each trial, with blocks processing multiple trials simultaneously.

The kernel uses the same load balancing method described in Section 6.2.2.2, with the work units representing different values of k, thus parallelising the values of k across threads.

At the start of each iteration the first thread extracts the first entry of the row with row index equal to the iteration number, and stores it in shared memory for efficient access. This requires the threads to be synchronised before and after. This is seen on lines 21 to 26.

Following a check on line 28 to make sure the index of the work unit has not passed the end of the matrix, each thread, of index k, then compares an entry on the diagonal of R_k, and the value retrieved from the diagonal of the R_1, against the stored minimum and maximum for the respective value of k. This is shown on lines 32 to 41.

Once the end of the main loop has been reached, the values of the minimum and maximum diagonals for each matrix R_k will have been computed. The ratios must now be computed, which is performed in the next loop. This is a simple case of batching and parallelising the ratio computations across threads as seen on lines 45 to 48.

6.2.3.2 Computing the Minimum and Maximum Squared Row Norms of $\hat{R}_k$

One of the tests described in Chapter 4 used the ratio of the row norms of the upper triangular factors of the subresultant matrices. To compute the 2-norm, as is used here, all values must be squared, summed, and the square root of this sum must be computed.

```
1  // mats is the array of triangular factors of the subresultant matrices
2  //      for this block
3  // r1 is the original upper triangular factor
4  // tempLower is the diagonal value extracted from the original upper
5  //      triangular factor, stored in shared memory
6  // sharedMin is an array of current minimum values found for all
7  //      triangular factors, stored in shared memory
8  // sharedMax is an array of current maximum values found for all
9  //      triangular factors, stored in shared memory
10 // diagRatios is an array to store the diagonal ratios of all triangular
11 //      factors
12 // triRowIndex(width, row) is a function that returns the row index of an
13 //      upper triangular matrix with specified width
14
15 gpu_parallel_for idx ← 0 to blockSize
16     for pass ← 0 to n
17         totalWork ← n-pass
18         for batch ← 0 to ceil(totalWork / blockWidth)
19             wu ← batch * blockWidth + threadID
20
21             synchroniseThreads
22
23             if wu = 0
24                 tempUpper ← abs(r1[triRowIndex(n*2, pass)])
25
26             synchroniseThreads
27
28             if wu<totalWork
29                 mat ← mats[wu]
30                 temp ← abs(mat[triRowIndex(n-wu, wu)])
31
32                 if pass = 0
33                     sharedMin[wu] ← temp
34                     sharedMax[wu] ← temp
35                 else
36                     sharedMin[wu] ← min(temp,sharedMin[wu])
37                     sharedMax[wu] ← max(temp,sharedMax[wu])
38
39                 if wu ≥ pass
40                     sharedMin[wu] ← min(tempUpper,sharedMin[wu])
41                     sharedMax[wu] ← max(tempUpper,sharedMax[wu])
42
43         synchroniseThreads
44
45     for batch ← 0 to < ceil(halfN / blockWidth)
46         wu ← batch * blockWidth + threadID
47         if wu < halfN
48             diagRatios[wu] ← sharedMax[wu] / sharedMin[wu]
```

Listing 3: Pseudocode for computing the ratio of diagonal values for all upper triangular factors of the subresultant matrices

However, as the maximum and minimum entries are required to be found for all triangular factors, the square root can occur after the maximum and minimum values have been identified. Leaving the computation of the square root until the relevant values are found leads to a more efficient algorithm, as fewer operations are required. Therefore this kernel, and its equivalent for the matrices $\check{R}_k$, will compute the squared row norms, rather than the row norms.

The kernel for the computation of the squared row norms of $\check{R}_k$ will compute this from the original upper triangular factor R_1. All the values that are required are in the upper half of this matrix $\hat{R}_1$, shown below.

$$\hat{R}_1 = \left[\begin{array}{ccccc|ccccc} r_{1,1,1} & r_{1,1,2} & r_{1,1,3} & r_{1,1,4} & r_{1,1,5} & r_{1,1,6} & r_{1,1,7} & r_{1,1,8} & r_{1,1,9} & r_{1,1,10} \\ & r_{1,2,2} & r_{1,2,3} & r_{1,2,4} & r_{1,2,5} & r_{1,2,6} & r_{1,2,7} & r_{1,2,8} & r_{1,2,9} & r_{1,2,10} \\ & & r_{1,3,3} & r_{1,3,4} & r_{1,3,5} & r_{1,3,6} & r_{1,3,7} & r_{1,3,8} & r_{1,3,9} & r_{1,3,10} \\ & & & r_{1,4,4} & r_{1,4,5} & r_{1,4,6} & r_{1,4,7} & r_{1,4,8} & r_{1,4,9} & r_{1,4,10} \\ & & & & r_{1,5,5} & r_{1,5,6} & r_{1,5,7} & r_{1,5,8} & r_{1,5,9} & r_{1,5,10} \end{array}\right]. \tag{6.2}$$

The matrix $\hat{R}_1$ shown above has been split into two sections. In R_2 the last column in each of these sections will have been removed in the process of the column deletions. Additionally, the final row of $\hat{R}_1$ will have been modified through Givens rotation to form the first row of $\check{R}_2$. Therefore $\hat{R}_2$ will be formed from $\hat{R}_1$, excluding the aforementioned columns and row. In R_3 the last two columns in each section are removed, and the final two rows will have been modified by Givens rotations and thus be a part of $\check{R}_3$, and so on for all values of k. It follows that the sum for each row of the upper matrices can be computed by the sum of entries of the first $n-(k-1)$ columns of each side of the matrix, and for each matrix, for $n-(k-1)$ rows.

This process can be described with the following equation.

$$\|r_{k,j,*}\|^2 = \sum_{i=j}^{m-k} r_{1,j,i}^2 + \sum_{i=m+1}^{m+n-k} r_{1,j,i}^2,$$

for $k = 1, \ldots, n-1$, and $j = 1, \ldots, m-k$.

This can be accomplished in parallel on a GPU by performing prefix sums on each row of both the left and right sections, then adding the results of the prefix sums of each side together.

Listing 4 provides pseudocode for this computation. This is separated into two sections, each acting on half of the matrix $\hat{R}_k$. The processing of the right section occurs first, as seen on lines 13 to 31, with the processing for the left section occuring on lines 33 to 52.

Unlike the other kernels that are presented in this chapter, this kernel does not distribute trials across blocks. Instead the blocks process different rows of the same trial, with different trials processed in sequence. This is to assist with balancing.

To this end the first step in the main loop of both of these sections is to compute the

row that will be acted on by the current iteration. The row is alternated on every other iteration, on even numbered iterations each block b will process the row of index b, and on odd iterations it will process the row of index $n - b - 1$. This is shown on lines 14 to 17 and 34 to 37. Alternating the rows in this way aids in balancing out the inherently unbalanced task of summing the upper triangular left half of the matrix, and storing the necessary upper triangular portion of both sides of the matrix.

Both sections perform a warp shuffle prefix sum of the squared values, shown on lines 26 and 47. The operation of warp shuffling was described in Chapter 5. The right side of $\hat{R}_1$ is processed first, with the upper triangular segment of the results from this section of the matrix being stored in the upper triangular portion of the submatrix that was summed. The results of the left section are then added to the results stored in this upper triangular section. The checks on lines 28 to 31 and 49 to 52 manage the output, to ensure only the correct memory locations are written to.

The result of this processing is an upper triangular matrix in which the cth column represents the squared row sums of the matrix $\hat{R}_{n-c}$. The matrix below demonstrates this for a situation where R_1 is of width 10. In this matrix $r_{k,i}$ represents row i of the triangular factor R_k.

$$\begin{bmatrix} \|r_{5,1}\|_2^2 & \|r_{4,1}\|_2^2 & \|r_{3,1}\|_2^2 & \|r_{2,1}\|_2^2 & \|r_{1,1}\|_2^2 \\ & \|r_{4,2}\|_2^2 & \|r_{3,2}\|_2^2 & \|r_{2,2}\|_2^2 & \|r_{1,2}\|_2^2 \\ & & \|r_{3,3}\|_2^2 & \|r_{2,3}\|_2^2 & \|r_{1,3}\|_2^2 \\ & & & \|r_{2,4}\|_2^2 & \|r_{1,4}\|_2^2 \\ & & & & \|r_{1,1}\|_2^2 \end{bmatrix}$$

Example

This example will use the matrix R_1 shown in Matrix 1. The upper part of this matrix $\hat{R}_1$ will further be split, with $\hat{R}_{1L}$ being the left side of this matrix, and $\hat{R}_{1R}$ being the right side. This is split as was shown in Equation 6.2.

$$\hat{R}_{1L} = \begin{bmatrix} -18.1874 & -2.2196 & -4.6050 & -1.0025 & -0.9800 & -0.1301 \\ 0 & -18.0515 & -1.6701 & -4.5164 & -0.8896 & -0.9713 \\ 0 & 0 & -17.5153 & -1.6105 & -4.4392 & -0.9142 \\ 0 & 0 & 0 & -17.5153 & -1.6111 & -4.4397 \\ 0 & 0 & 0 & 0 & -17.5137 & -1.6082 \\ 0 & 0 & 0 & 0 & 0 & -17.5125 \end{bmatrix},$$

```
// blockID is the block ID.
// n is half the width of the original upper triangular matrix R1
// allR1 is a matrix of all the original upper triangular matrices.
// scan_SM is the shared memory array used by the warp shuffle prefix sum
//      function. The results of the prefix sum are stored at the beginning of
//      this array.
// toAdd_SM is the shared memory location in which the running total is stored
// prefixSum is a function to perform a warp shuffle prefix sum, taking in
//      the width of the sum and the value for the thread.

gpu_parallel_for threadID ← 0 to blockSize
    //Right submatrix
    for pass ← 0 to num
        if (pass%2 = 0)
            row ← blockID
        else
            row ← n-blockID-1
        R1 ← allR1[pass]
        rowPtr ← R1+(triangle(2*n)-triangle(2*n-row))+(n-row)
        toAdd_SM ← 0
        synchroniseThreads
        batches ← ceil(n/blockDim.x)
        for batch ← 0 to batches
            width ← (batches-batch = 1) ? n % blockDim.x : blockID
            index ← (batch*blockID)+threadID
            scan_SM ← prefixSum(width, rowPtr[index]*rowPtr[index])
            synchroniseThreads
            if ((index ≥ row) and (index < n))
                rowPtr[index] ← scan_SM[threadID]+toAdd_SM
            synchroniseThreads
            if (batch<batches-1) and (threadID==0) toAdd_SM+= scan_SM[blockSize-1
    //Left submatrix
    for pass ← 0 to num
        if (pass%2 = 0)
            row ← blockID
        else
            row ← n-blockID-1
        R1 ← allR1[pass]
        rowPtr ← R1+(triangle(2*n)-triangle(2*n-row))
        toAdd_SM ← 0
        synchroniseThreads
        batches ← ceil(n/blockDim.x)
        for batch ← 0 to batches
            width ← (batches-batch = 1) ? m-row % blockDim.x : blockID
            index ← (batch*blockID)+threadID
            value ← index<width ? rowPtr[index] : 0
            scan_SM ← prefixSum(width, value*value)
            synchroniseThreads
            if ((index < n-row) and (index > 2*n))
                rowPtr[index+n] ← rowPtr[index+n] + scanShared[threadID]+toAdd_SM;
            synchroniseThreads
            if (batch<batches-1) and (threadID==0) toAdd_SM+= scan_SM[blockSize-1]
```

Listing 4: Pseudocode for computing the minimum and maximum squared row norms of $\hat{R}_k$

$$\hat{R}_{1R} = \begin{bmatrix} -4.0187 & -4.9968 & -1.3858 & -1.0327 & -0.1584 & -0.0667 \\ -18.6954 & -3.4346 & -4.8640 & -1.2693 & -1.0210 & -0.1514 \\ -6.5197 & -18.1357 & -3.3448 & -4.7960 & -1.3000 & -1.0403 \\ -0.0545 & -6.5197 & -18.1358 & -3.3456 & -4.7967 & -1.3005 \\ -0.1825 & -0.0545 & -6.5190 & -18.1331 & -3.3418 & -4.7943 \\ 0.9422 & -0.1825 & -0.0545 & -6.5186 & -18.1315 & -3.3404 \end{bmatrix}.$$

Starting with $\hat{R}_{1R}$, a prefix sum of the squared elements in every row is computed. While all elements of every row are used in the prefix sum, only the upper triangular portion of the result of the sum is required. This is shown in the matrix below, which will be known as $\dot{R}_{1R}$.

$$\dot{R}_{1R} = \begin{bmatrix} 16.1499 & 41.1180 & 43.0384 & 44.1049 & 44.1300 & 44.1344 \\ & 361.3145 & 384.9730 & 386.5841 & 387.6265 & 387.6494 \\ & & 382.5978 & 405.5994 & 407.2894 & 408.3716 \\ & & & 382.6097 & 405.6181 & 407.3094 \\ & & & & 382.5106 & 405.4959 \\ & & & & & 383.3257 \end{bmatrix}.$$

The entries of the left side of the matrix, $\hat{R}_{1L}$, are squared and summed in the same way.

$$\dot{R}_{1L} = \begin{bmatrix} 330.7815 & 335.7081 & 356.9142 & 357.9192 & 358.8796 & 358.8965 \\ & 325.8567 & 328.6459 & 349.0438 & 349.8351 & 350.7786 \\ & & 306.7857 & 309.3794 & 329.0859 & 329.9217 \\ & & & 306.7857 & 309.3814 & 329.0923 \\ & & & & 306.7297 & 309.3160 \\ & & & & & 306.6877 \end{bmatrix}.$$

By adding these matrices together $\dot{R}_{1L} + \dot{R}_{1R}$ results in the squared row sums of every matrix $\hat{R}_{1 \ldots n}$. This is shown in the table below.

k	6	5	4	3	2	1
Row 1	346.9315	376.8261	399.9526	402.0240	403.0095	403.0309
Row 2		687.1711	713.6188	735.6278	737.4617	738.4280
Row 3			689.3835	714.9788	736.3753	738.2933
Row 4				689.3955	714.9994	736.4017
Row 5					689.2403	714.8119
Row 6						690.0134

6.2.3.3 Computing the Squared Row Norms of $\check{R}_k$

The implementation for the computation of the squared row norms of $R_{1...n}$ uses warp shuffle prefix sums, similar to those used in the previous section. Similarly to the other kernels described in this chapter, with exception to the kernel for the computation of the squared row norms of $\hat{R}_k$, this kernel will parallelise the trials in different blocks, with each block computing all the norms for a single trial.

As this was the second most computationally expensive part of the computation of the degree, multiple implementations were considered and tested. While the row sums can be computed efficiently with a warp shuffle prefix sum, it becomes a more challenging problem when considering how to balance the multiple sums of different widths required by this algorithm.

The pseudocode for the final implementation selected is shown in Listing 5. In this implementation a prefix sum of the entire matrix is computed. Computing the sum of the whole matrix, rather than individual rows, helps to balance the algorithm, avoiding the problem of rows of unequal lengths.

Due to the sum of the whole matrix being computed, it is necessary to compute the prefix sum starting at the bottom row of every matrix and working upwards. Values in the bottom rows of the matrices tend to be significantly smaller than those on the upper rows, as was explained in Chapter 3. Therefore, due to the limitations of floating point numbers, the values of the sums of the lower rows would be lost in precision errors if they were added to the sum of the significantly larger values on the upper rows. The computation of the index and retrieval of the correct value can be seen on Lines 23 and 24, with the warp shuffle algorithm being called on Line 25.

Once the sum has been computed, final squared row norms are then computed by extracting the entries at the end of each row, using a triangular root calculation as discussed in Section 6.2.1. The values extracted from the sum for each row can then be subtracted from the row sum of the previous row in the matrix to find the final row sum. This final row sum is then stored at the end of the respective row in the subresultant matrix. This process is shown on Lines 26 to 33, with the result being written back to the matrix on Lines 34 to 39.

Example

For this example only a single matrix, $\check{R}_1$ from the matrix R_1 shown in Matrix 1, will be used. In the actual implementation the same process shown here will be repeated for all matrices $\check{R}_{1...n}$.

```
1  // n is the degree of the polynomial of lower degree.
2  // Mats is an array of the lower right corners of the triangular factors
3  //      stored in triangular form in 2D arrays.
4  // scan_SM is the shared memory array used by the warp shuffle prefix sum
5  //      function. The results of the prefix sum are stored at the beginning of
6  //      this array.
7  // Sums_SM stores the results for the sum of each row in shared memory.
8  // toAdd_SM is the shared memory location in which the running total is stored.
9  // prefixSum is the function that performs a warp shuffle prefix sum. It includes
10 //      the width of the sum and the value for that thread.
11 // triRoot is a function that returns the triangular root of a number. It
12 //      returns -1 if a triangular root does not exist.
13
14 gpu_parallel_for idx ← 0 to blockSize
15     for k ← 0 to n
16         if (idx = 0)
17             toAdd_SM ← 0
18         synchronise threads
19         width ← triangle(n-k)
20         batches ← ceil(width/blockSize)
21         for batch ← 0 to batches
22             wu ← (batch*blockSize) + idx
23             index ← width - wu-1
24             value ← wu<width ? mats[k][index] : 0
25             scan_SM ← prefixSum(blockSize, value*value)
26             if (wu<width)
27                 root ← triangularRoot(wu+1)
28                 if (root != -1)
29                     Sums_SM[root-1] ← scan_SM[idx] + toAdd_SM
30             synchronise threads
31             if ((batch != batches-1) and (idx = 0))
32                 toAdd_SM ← toAdd_SM + scan_SM[blockSize-1]
33             synchronise threads
34         for batch ← 0 to ceil((n-k)/blockSize)
35             wu ← batch*blockSize + idx
36             if (wu<n-k)
37                 row ← n-k-wu-1
38                 value ← (wu != 0) ? Sums_SM[wu]-Sums_SM[wu-1] : Sums_SM[wu]
39                 Mats[k][triangle(n-k)-triangle(n-k-(row+1)+(n-k-1)] ← val
```

Listing 5: Pseudocode for computing the minimum and maximum squared row norms of $\check{R}_k$

$$\check{R}_1 = \begin{bmatrix} 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 & -12.9799 \\ 0 & 0.6583 & 0.7763 & -0.1310 & -3.0065 & -14.0045 \\ 0 & 0 & 0.2002 & 0.2093 & -1.0077 & 0.1921 \\ 0 & 0 & 0 & 0.9568 & -0.9891 & -2.7582 \\ 0 & 0 & 0 & 0 & -0.0002 & 0.0005 \\ 0 & 0 & 0 & 0 & 0 & 0.0002 \end{bmatrix}.$$

The entries of this matrix are squared, this results in the matrix $\check{R}_1^{\circ 2}$.

$$\check{R}_1^{\circ 2} = \begin{bmatrix} 0.0958 & 0.5500 & 0.3739 & 0.0433 & 32.4763 & 168.4778 \\ 0 & 0.4334 & 0.6026 & 0.0172 & 9.0390 & 196.1260 \\ 0 & 0 & 0.0401 & 0.0438 & 1.0155 & 0.0369 \\ 0 & 0 & 0 & 0.9155 & 0.9783 & 7.6077 \\ 0 & 0 & 0 & 0 & 4.0 \times 10^{-8} & 2.5 \times 10^{-7} \\ 0 & 0 & 0 & 0 & 0 & 4.0 \times 10^{-8} \end{bmatrix}.$$

A prefix sum is performed from the bottom of this matrix up to the top, moving right across each row. Only the non-zero elements will be considered. This will result in the matrix $\dot{R}_1$.

$$\dot{R}_1 = \begin{bmatrix} 216.9517 & 217.5017 & 217.8756 & 217.9189 & 250.3952 & 418.8730 \\ & 11.0711 & 11.6737 & 11.6909 & 20.7299 & 216.8559 \\ & & 9.5415 & 9.5853 & 10.6008 & 10.6377 \\ & & & 0.9155 & 1.8938 & 9.5015 \\ & & & & 8.0 \times 10^{-8} & 3.3 \times 10^{-7} \\ & & & & & 4.0 \times 10^{-8} \end{bmatrix}.$$

The final row sums for $\check{R}_1$, $\dot{r}_i$, where i is the row number, are computed by subtracting r_{i+1} from r_i.

$$\begin{aligned} \dot{r}_1 &= 418.8730 - 216.8559 &&= 202.0171, \\ \dot{r}_2 &= 216.8559 - 10.6337 &&= 206.2222, \\ \dot{r}_3 &= 10.6337 - 9.5015 &&= 1.1322, \\ \dot{r}_4 &= 9.5015 - 3.3 \times 10^{7} &&= 9.5015, \\ \dot{r}_5 &= 3.3 \times 10^{-7} - 4.0 \times 10^{-8} &&= 2.9 \times 10^{-7}, \\ \dot{r}_6 & &&= 4.0 \times 10^{-8}. \end{aligned}$$

This process is repeated for all matrices $\check{R}_k$, to compute the squared row sums of every row of every upper triangular factor.

6.2.3.4 Computing the Ratios of the Row Norms

Sections 6.2.3.2 and 6.2.3.3 provided implementations for computing the squared row norms of each of the upper and lower portions of every triangular factor. The results from these kernels can now be used to compute the ratios of the row sums.

Listing 6 shows the pseudocode of this computation, which is very similar to that of the diagonal ratios shown in Section 6.2.3.1. Each block computes the ratios for a single trial. Each thread is responsible for an individual triangular matrix, moving down the columns to find the minimum and maximums of the rows. A single value from the upper matrix and a single value from the lower matrix are compared to stored values each iteration. The comparisons of minimum and maximum values are seen on Lines 13 to 34. Once all maximums and minimums have been computed the final ratios of the square roots of the minimum and maximum values is performed. The ratios can then be found by dividing the maximum value by the minimum value for every matrix, as is seen on Lines 36 to 38.

Similarly to the implementation presented in 6.2.3.1, this is not a well balanced approach. However, as the overall runtime is extremely low, it was not seen as a priority to further optimise this kernel.

6.2.3.5 Computing the Gradients

The final task to be completed on the GPU in this implementation is the computation of the gradients. The pseudocode for this can be seen in Listing 7. This is a very simple kernel. Trials are distributed to different blocks, and the ratios for each value of k within that trial processed by threads within the block. This kernel can be used for the gradients from both the diagonals and the row norms.

6.2.3.6 Computing the Degree

Once the final gradients have been computed, the gradients are copied back to host memory. Finding the index of the minimum gradient, and the modal minimum index, is performed on the CPU.

While it would be possible to speed up these operations on a GPU, the benefit of this would only be minor. The amount of data that needs to be copied back to the host at this point is very small, and comparatively few computations need to occur to find the minimum. Therefore it was not considered to be worth the development time to consider these sections for acceleration.

6.2.4 Profiling the Final GPU Implementation and Establishing Launch Parameters

The final profiling results give an overview of how efficiently the implementation is utilising the GPU. This section will discuss the profiling results for the final algorithm for five different degrees of polynomial when processing 25 trials.

```
1  // minMaxArray is an array of structures containing minimum and maximum
2  //      values, stored in shared memory
3  // ratios is an array to store the computed ratios of each subresultant
4  //      matrix
5  // triRowIndex(width, row) is a function to compute the index of a row
6  //      in a triangular matrix with the specified width and row
7
8  gpu_parallel_for 0 to blockWidth
9      for batch ← 0 to ceil(n / blockWidth)
10         wu ← (batch * blockWidth) + threadID
11         index ← (n-1)-wu
12         for row ← 0 to n
13             if (wu<n) {
14                 mm ← minMaxArray[index]
15                 rowIndex ← orig + triRowIndex(n*2, row) + (n - row)
16                 if row = 0
17                     mm.min ← rowPtr[wu]
18                     mm.max ← mm.min
19                 else if wu ≥ row
20                     mm.min ← min(orig[rowIndex + wu], mm.min)
21                     mm.max ← max(orig[rowIndex + wu], mm.max)
22
23                 if (wu ≥ row) {
24                     mat ← mats[n-1-wu]
25                     rowIndex2 ← triRowIndex(wu+1, row+1)-1
26
27                     mm.min ← min(mat[rowPtr2], mm.min)
28                     mm.max ← max(mat[rowPtr2], mm.max)
29
30             synchroniseThreads
31
32             if (wu<n) minMaxArray[index] ← mm
33
34             synchroniseThreads
35
36         if wu < n
37             MinMax mm ← minMaxArray[index]
38             ratios[wu] ← sqrt(mm.max) / sqrt(mm.min)
```

Listing 6: Pseudocode for computing the ratios of the row norms for all upper triangular factors of the subresultant matrices

```
1  // numGrads is the number of gradients to be computed (number of ratios -1)
2  // gradients is an array in which to store the gradients for this block
3  // ratios is an array from which to get the ratios for this block
4
5  gpu_parallel_for 0 to blockWidth
6      batches ← ceil(numGrads / blockWidth)
7      for batch ← 0 to batches
8          wu ← batch * blockWidth + threadID
9          if (wu ≥ numGrads) break
10         if wu<numGrads
11             k ← wu % numGrads
12             gradients[k] ← log10(ratios[k+1]) - log10(ratios[k])
```

Listing 7: Pseudocode for computing the gradients

Profiling was performed using an NVIDIA GTX 780 GPU, with 3GB of memory. The polynomial degrees tested were 250, 500, 750, 1000 and 1250. This was a wide enough range to test the batching in the algorithm from a single batch, at degree 250, to batches consisting of only a single trial at a degree of 1250.

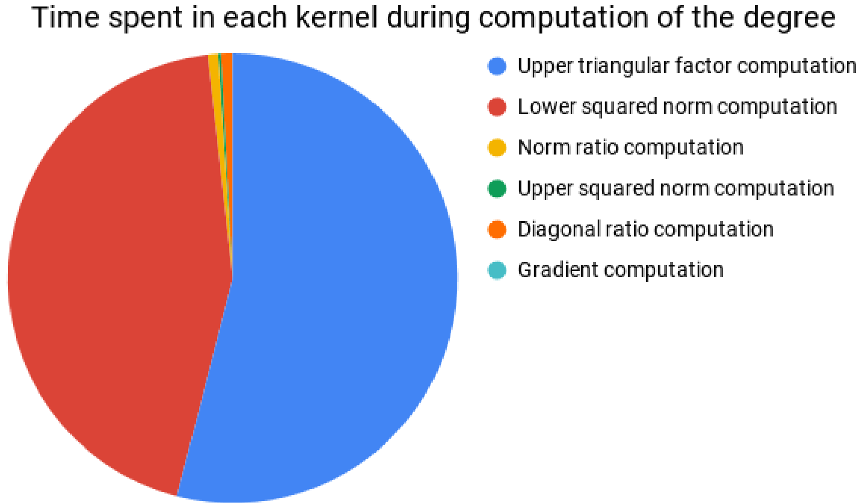


Figure 6.5: Time spent in each kernel of the GPU implementation

Figure 6.5 shows a pie chart of time spent in each kernel of the implementation. The relative runtimes were very similar for all polynomial degrees, with the computation of the matrices $\check{R}_{2 \ldots n}$ and the squared row sums of $\check{R}_{1 \ldots n}$ both taking significantly more time than the other kernels. The total runtimes of these two kernels ranged from 96.7% for polynomial degrees of 250, to 99.3% for polynomial degrees of 1250. Due to the discrepancy between the runtimes of the two most expensive kernels and the rest of the kernels, this section will focus on profiling the two expensive kernels.

While the kernels developed here all require a predetermined grid size, the load balancing described in Section 6.2.2.2 means that the block sizes can be scaled arbitrarily. Blocks can contain a maximum of 1024 threads. Typically block sizes of multiples of 32 are ideal, as warps will process 32 threads at a time. Therefore experiments were conducted using blocks of various multiples of 32. It was found that for polynomials of most degrees, setting the block size to be 1024 resulted in the greatest performance, but for polynomials of low degree this was excessive, and leads to a significant number of idle threads. The best compromise was found to be the width of R_1, rounded to the nearest multiple of 32 and capped at 1024.

The theoretical device occupancy available in both major kernels is 50%. The limiting factor here is the number of registers required per thread. By default the kernel for the computation of $\check{R}_{2 \ldots n}$ requires 54 registers per thread, while the kernel for the squared row norms requires 42 registers per thread. It is possible to force the compiler to limit the number of registers used per thread, either with the `maxrregcount` argument of the

compiler, or with the `__launch_bounds__` modifier in the code.

Theoretical occupancy only improves beyond 50% at the block sizes required when the registers per thread are reduced to 32. The trade off with limiting registers in this way means that local variables will instead be stored in memory, leading to increased latency when accessing them. When limiting the kernels to 32 registers the runtime of the computation of $\check{R}_{2 \ldots n}$ is significantly slower in all cases, and while the squared row sum kernel does see improvements at polynomials of lower degrees, performance decreases for polynomials of higher degree. Due to this the number of registers were kept at their default values for both kernels.

With these launch parameters it was found that actual device occupancy achieved was 50% for higher degrees, and only reducing to 48.7% for the lower degrees.

6.3 Results

The implementation discussed in this chapter presents the first GPU accelerated algorithm and implementation for the computation of the degree of the AGCD. All tests in this section were performed on a system with a 6 core Intel i7 6850k CPU and an NVIDIA Titan V GPU with 12GB of GPU memory. All tests were run 10 times and an average taken of these results to give the final results. The standard deviation in all cases was less than 10% of the mean, typically being less than 4% of the mean.

6.3.1 Reliability testing

The results of the algorithm were tested against that of the original MATLAB implementation. Each implementation was tested with a variety of polynomials of equal degree, the coefficients of which were generated with the MATLAB `rand` command, and convolved with a GCD, with added noise to ensure the polynomials were coprime. The results were checked at each stage of the algorithm to ensure all results from the GPU implementation were equal for all values to the results from the CPU implementation. This was the case for all polynomials tested, at every stage of the computation.

Table 6.1 shows the algorithms reliability for various levels of noise. It should be noted that in this table the algorithms used were the full computations, investigating every subresultant matrix, as opposed to the reduced computations as described in Chapter 4. These results were gathered through random vectors generated in MATLAB. A GCD was constructed, with a degree of roughly 10% that of the polynomials. Random vectors were generated for the polynomials, and these were convolved with the GCD. Random noise of the levels specified in the table was generated and added to the convolved polynomials. The algorithm was run over 25 pairs of polynomials with the same GCD to receive the estimate for the number of successful trials, and this entire process was repeated 10 times to receive the final results presented here. In all cases these results were identical to those of the serial algorithm.

SNR (dB)	120	100	80
$n = 500$	92.8%	76.0%	30.0%
$n = 1000$	94.0%	71.6%	11.2%
$n = 1500$	80.0%	38.4%	3.2%
$n = 2000$	75.2%	35.2%	0.0%

Table 6.1: Rate of successful trials for varying polynomial degrees and noise levels

The results show that for polynomials with degrees of up to 2000, when the signal to noise ratio (SNR) is as high as 120 dB, the degree could be reliably computed in all cases. When the noise in increased to an SNR of 100 dB the degree is computed less reliably. However, even at degrees of 1500 and 2000 the degree could still be computed reliably 30-35% of the time, which is likely enough to identify the degree given enough trials. When the SNR decreases to 80 dB, only the polynomials of degree 500 are likely to return the correct degree, with the other results providing poor reliability for the degree computation.

6.3.2 Runtime testing

The runtime results showed a significant improvement in terms of performance over the CPU implementations, though some issues did arise due to the necessary batching discussed in 6.2.2.4, and the large amount of memory required to store matrices.

6.3.2.1 Scalability testing

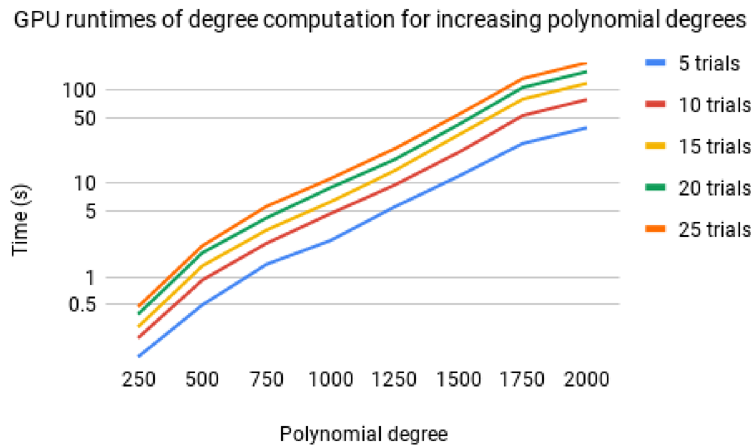


Figure 6.6: Runtimes of the GPU method for increasing polynomial degrees and varying numbers of trials

Figure 6.6 shows the runtimes for increasing degrees of the convolved polynomials, those being the rows or columns of the inexact image, and varying numbers of trials. The curve for each number of trials is relatively smooth between degrees of 250 and 1500. At

a degree of 1750 the implementation starts to encounter performance issues. This is due to the batching technique described in Section 6.2.2.4. At this degree only the data for a single trial can be stored on the GPU memory at a time. This results in a greater overhead from the number of kernel calls and memory operations, as well as under-utilisation of the device, which results in worse performance. As the polynomial degrees increase again to 2000, there is a decrease in the gradient of the curve, suggesting an improvement of the scaling of the algorithm. This is due to the utilisation of the device being improved compared to degrees of 1750. While for both of these degrees only a single trial can be performed per kernel call, the increased number of operations required for each Givens rotation for a degree of 2000 results in greater overall device utilisation.

6.3.2.2 Runtime comparisons

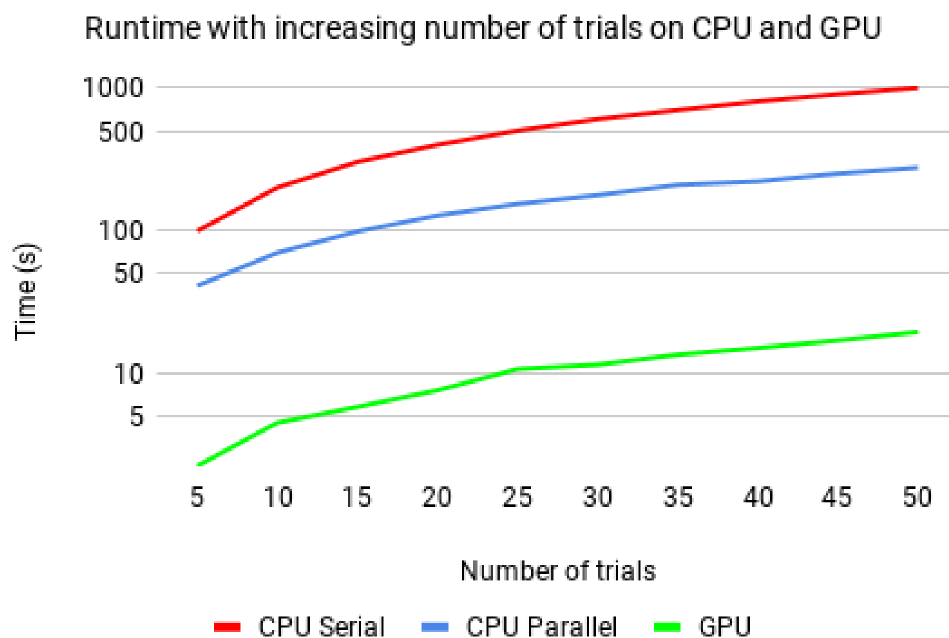


Figure 6.7: Runtimes of the GPU implementation compared CPU methods for increasing polynomial trials

Figure 6.7 shows the comparative performance between the CPU methods, both serial and parallel, and the GPU method. This is shown on a log scale to make the comparison of the scalability of the algorithms more obvious, and to allow the runtimes of the GPU algorithm to be more apparent. For these tests convolved polynomials of degree 1000 were used, and tested with an increasing number of trials. Similar performance issues to those observed in Figure 6.6 can be seen here, with the most obvious irregularities being at 10 and 25 trials. This is for the same reason as described previously, as the batching process described in 6.2.2.4 takes effect, causing extra overhead, and potentially causing overall device utilisation to decrease.

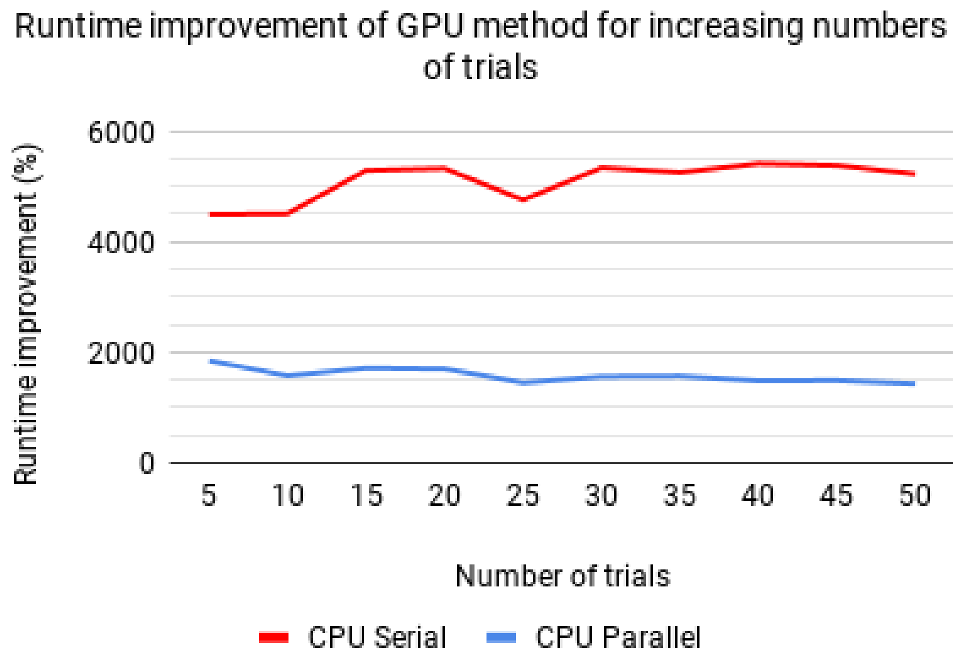


Figure 6.8: Runtime improvement of the GPU method compared to the CPU methods for increasing numbers of trials

Figure 6.8 shows the actual speedup of the GPU method, compared to that of the CPU methods, with relation to the results shown in Figure 6.7. The graph here shows a fairly constant improvement across all numbers of trials. This is to be expected, as the number of operations scales linearly with the number of trials. Compared to the serial algorithm speedups ranging between 44.97 to 53.38 times were observed, with speedups compared to the CPU parallel implementation ranging between 14.32 and 18.48 times. It should be noted that the polynomial degree of 1000 is the degree at which runtime improvement of the GPU implementation over the serial implementation is at a maximum, as will be shown in Figure 6.10, though similar results with regards to scaling can be observed for all degrees. This is due to the relationship between the number of trials and overall number of operations being linear, and all three implementations are shown to scale relatively linearly in relation to this.

Figure 6.9 shows the runtimes of the three implementations for increasing polynomial degrees. For these tests the number of trials was fixed at 25. While it may seem counter-intuitive that the CPU parallel implementation is initially slower than the serial implementation, this is due to the overhead of the parallel pool creation in MATLAB, as was discussed in Chapter 4. This takes approximately the same amount of time regardless of the problem size, so this will have the largest relative impact on the performance when the runtime is at its lowest, thus the impact is seen most clearly at lower degrees. Due to this overhead, the CPU parallel algorithm appears to scale better than the other implementations. This is due to the overhead becoming less significant relative to the

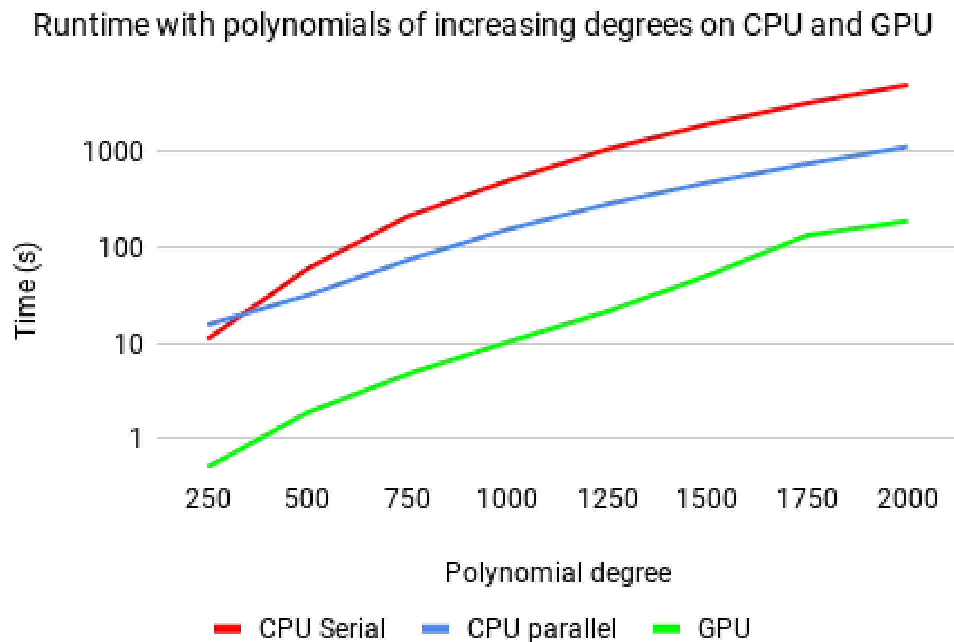


Figure 6.9: Runtimes of the GPU implementation compared CPU methods for increasing polynomial degrees

overall runtime of the implementation. The same performance issues discussed previously can be seen at a degree of 1750. As discussed this is due to the GPU only being able to run a single polynomial in each batch, as at a degree of 1750 there is not enough space in memory for the more than one trial at a time.

Figure 6.10 shows the relative improvement of the GPU implementation compared to the CPU implementations, with regards to the runtimes for the increasing polynomial degrees shown in Figure 6.9. The most notable feature of this graph is the inconsistency of the speedup of the GPU implementation compared to the CPU serial implementation. The low performance when the polynomial degrees are larger can be explained with the batching discussed previously. The relative improvement in performance for degrees of 2000 when compared to those of 1750 was discussed in relation to Figure 6.6. While both of these polynomial degrees can only be processed a single trial at a time by the GPU, the computations of the test where the polynomial degrees are 2000 better utilise the GPU. The decreased performance for lower degrees can be explained by the better utilisation of the GPU for larger degrees, as more computation is required.

Compared to the CPU parallel implementation the speedup decreases as the polynomial degree increases. This is due to the overhead of the parallel pool initialisation, as the runtime for this remains constant for all degrees.

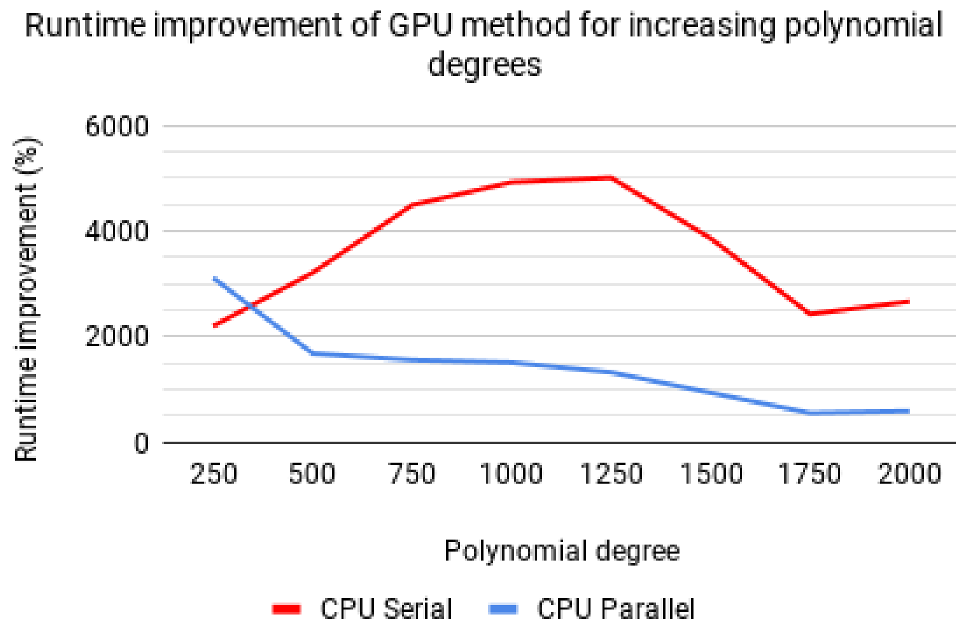


Figure 6.10: Runtime improvement of the GPU method compared to the CPU methods for increasing numbers of polynomial degrees

6.4 Issues and Potential Improvements

While Section 6.3 shows a GPU implementation that has significant performance improvements compared to the CPU methods, some issues were highlighted. The most significant of which is the memory usage of such an implementation.

Figure 6.11 shows the number of trials that can be run in a single kernel call on GPUs with different amounts of memory. While using a GPU with a larger amount of memory does allow for polynomials of larger degrees, and helps to demonstrate where there are performance issues, the cubic nature of the memory scaling, as described in Section 6.2.2.4, demonstrates why this is potentially unsustainable. An implementation using less memory would be desirable, as it would gain performance increases from avoiding the need for batching computations, as well as the ability to process large polynomials on GPUs with limited memory.

6.5 The Reduced Algorithm

Chapter 4 discussed a reduced form of the algorithm, where only the first half of the triangular factors are computed. This section will discuss this reduced algorithm when applied on a GPU, and test it against the full algorithm.

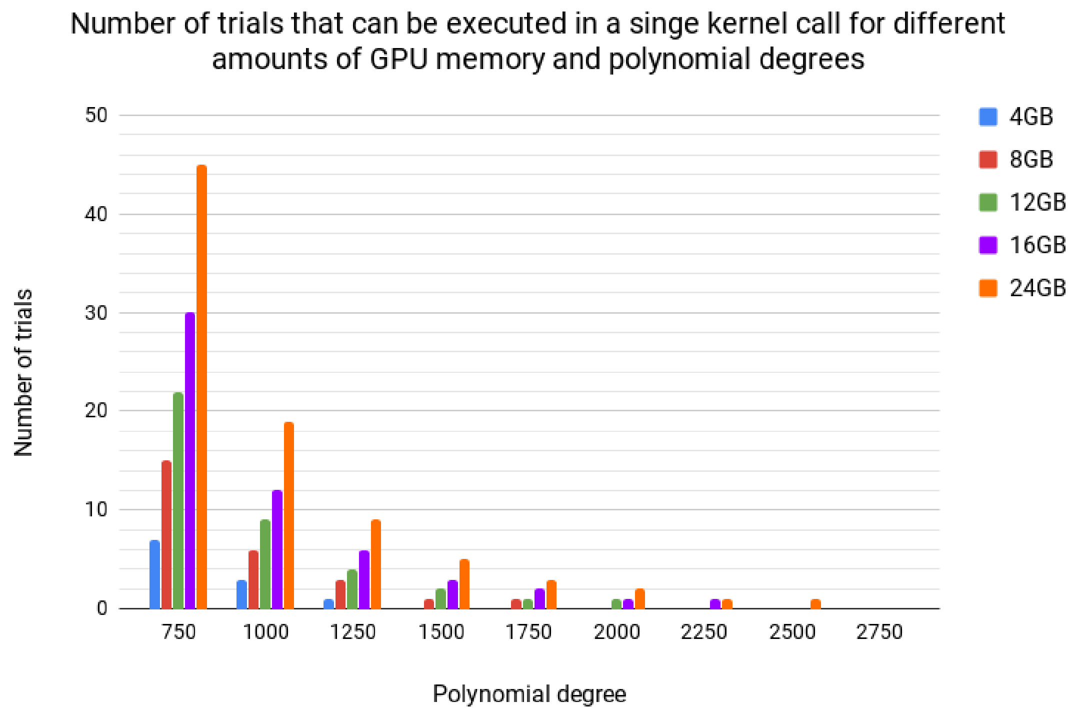


Figure 6.11: The number of trials that can be executed in a single kernel call by GPUs with different amounts of memory

6.5.1 Implementation

The implementation of this modification is relatively straightforward. The only changes that must be made are to the memory allocation and indexing, where reduced memory is needed for nearly all constructs within the algorithm, and adjustments to the load balancing and work distribution sections of the algorithms, to ensure the correct number of matrices are computed.

6.5.2 Results

The reduced version of the algorithm was tested with the same parameters as were used in the runtime testing described in Section 6.3.2.2. The degrees provided by these algorithms show the exact same increase in reliability as described in Chapter 4. As results are exactly the same, as expected, this section will instead concentrate on the runtime comparisons against the full GPU implementation.

Figure 6.12 shows the runtimes for both implementations on the GPU for increasing numbers of trials. Unexpectedly, the full implementation shows better performance at lower degrees than the reduced implementation for lower numbers of trials, and maintains a very close runtime to the reduced implementation for degrees of 1000. Comparatively the improvement in runtime between the two CPU implementations shown in Figure 6.13 is more significant. While the difference between the runtimes in the GPU implementations

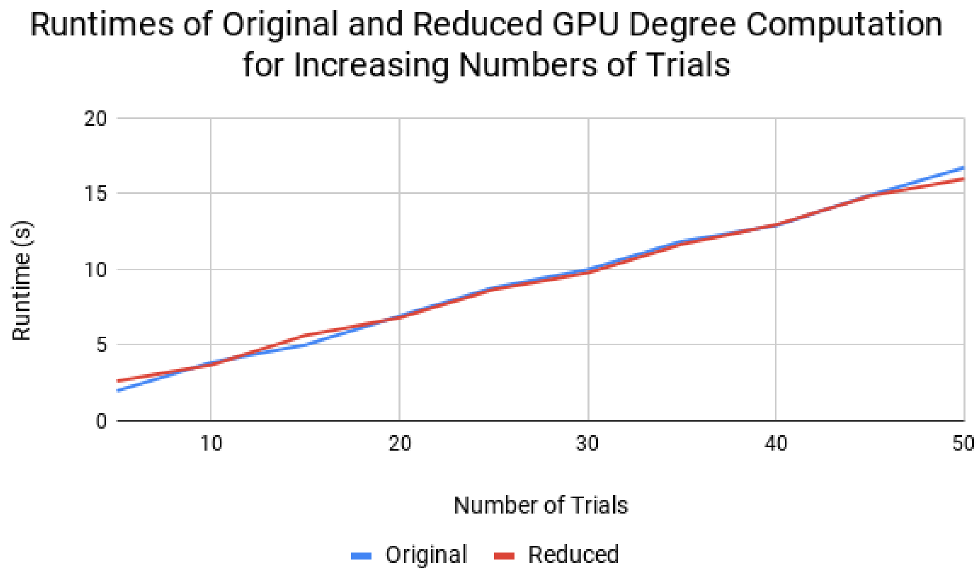


Figure 6.12: Runtimes of the GPU original and reduced implementations for an increasing number of trials

was expected to be less significant than those of the CPU implementations, this shows closer runtimes than were expected. This is likely due to the increased complexity of indexing values in the GPU reduced implementation, and the balance of the workload in the reduced algorithm being worse.

When comparing the results for increasing polynomial degrees of the original and reduced algorithms, as shown in Figure 6.14, the difference is more significant than that shown in the results in Figure 6.12. While the two implementations have very similar runtimes they start to separate at a degree of around 1000, with the reduced algorithm having a slight, but noticeable, advantage over the original implementation. When compared to the CPU implementation runtimes shown in Figure 6.15, the improvements of the GPU reduced algorithm are more slight than those in the CPU reduced algorithm.

While the improvements in reliability shown by the reduced algorithm in Chapter 4 are still present, the runtime improvements of this method when applied on GPUs in the reduced implementation are less significant. The reduced implementation does provide better performance than the original implementation. However, the improvements in both memory usage and performance are very slight. It may therefore be better to concentrate on developing a more general form of the algorithm, from which the gradient results can be limited after computation as is required by the specific situation.

6.6 Conclusion

The implementation demonstrated in this chapter provides a significant improvement over current CPU implementations of this algorithm. Unfortunately the scalability of this

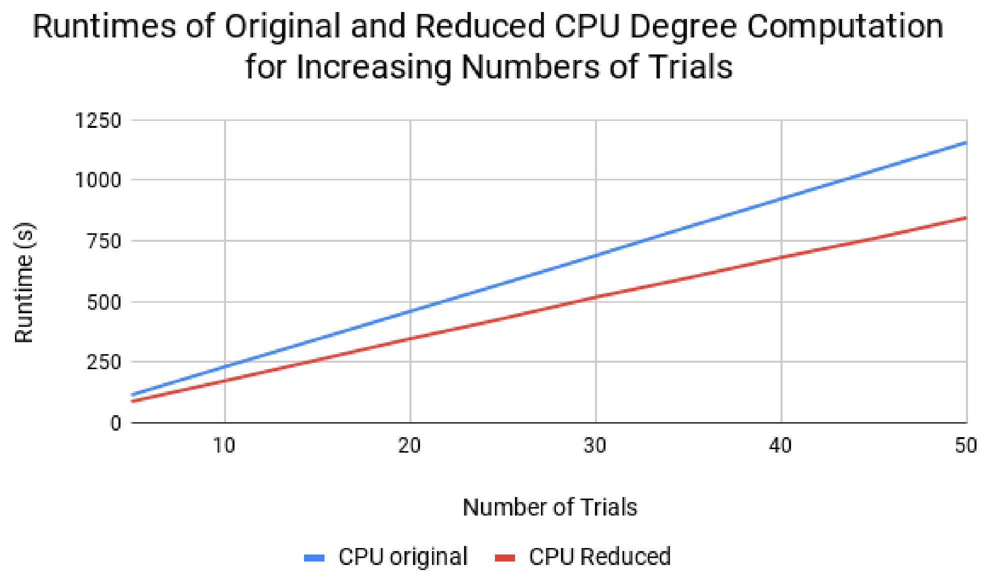


Figure 6.13: Runtimes of the CPU original and reduced implementations for an increasing number of trials

algorithm is somewhat limited by the need to store numerous large matrices in a limited amount of memory. This scalability will cause issues when there is need to estimate the degree of an AGCD of polynomials of very large degree on GPUs with a limited amount of memory.

The next chapter will investigate a method with which the high memory requirements described in this chapter can be overcome, and thus gaining improvements to both runtime and scalability of the degree computation algorithm.

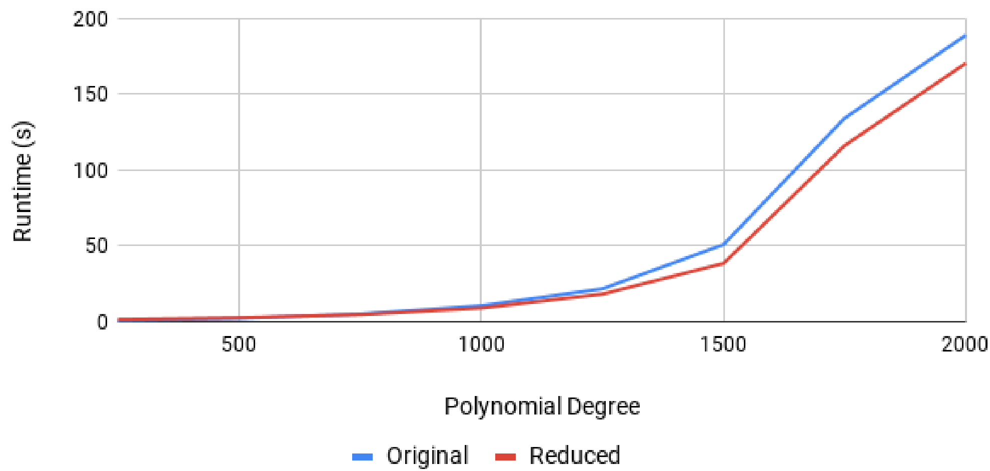


Figure 6.14: Runtimes of the GPU original and reduced implementations for increasing polynomial degrees

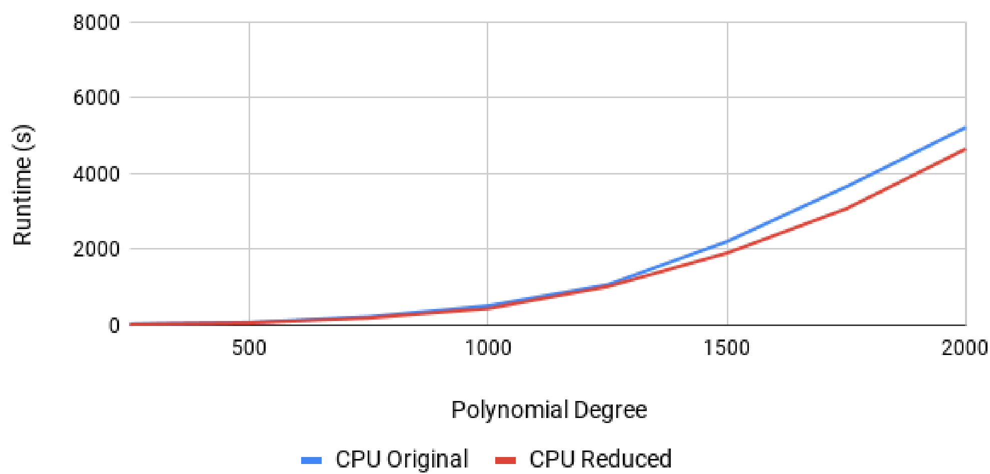


Figure 6.15: Runtimes of the GPU original and reduced implementations for increasing polynomial degrees

Chapter 7

A Low GPU Memory Approach to Degree Computation

Chapter 6 proposed an efficient GPU parallel implementation of the degree computation algorithm. While this implementation provided significant improvements to the runtime of the computation of the degree of an AGCD, it also suffers from scalability issues caused by the need for a large amount of GPU memory. This is due to the processing of numerous large matrices. This high memory usage leads to a limit in scalability, and, in some situations, a limit in the utilisation of the device. Due to this, it is beneficial to investigate methods in which a low memory version of this algorithm could be implemented, providing improved scalability and performance.

This chapter discusses a low memory implementation of the algorithm described in Chapter 6, which enables computations on polynomials of higher degrees, and eliminating the need to split the processing of trials into batches. This resulted in an implementation that is both more scalable and faster than the original algorithm proposed in Chapter 6.

Section 7.1 will discuss the changes made to the algorithm presented in Chapter 6 to reduce the memory footprint. Section 7.2 will discuss how these changes were implemented in the low memory implementation, including pseudocode. This implementation will be profiled, similar to the profiling that occurred in Chapter 6. Section 7.3 will compare the memory usage of this implementation to that of the implementation presented in Chapter 6, and the effect this has on the scalability of the algorithm. Section 7.4 will then test the performance of the low memory implementation to evaluate the modifications to the algorithm for reliability, runtime and scalability.

7.1 Algorithm Changes

The issue of high memory usage in the original algorithm arises from the large number of triangular matrices that must be stored. The entirety of each upper triangular factor is required for the tests described in Chapter 4 to compute the rank of the corresponding subresultant matrices, and thus to compute the degree of the AGCD.

When developing a kernel for a GPU it is generally recommended for the function to be focused, as this will minimise register usage so device occupancy can be maximised. Due to this, in the implementation discussed in Chapter 6, the algorithm was split into several kernels. This led to an efficient algorithm in all ways other than memory utilisation, which in turn led to performance issues.

To solve this problem it is possible to perform the tests for computing the rank, described in Chapter 4, into the kernel for the computation of the upper triangular matrices. This means that the minimums and maximums of both the sum of the squared entries on each row and diagonal values from each row can be computed straight after the final rows have been computed in each iteration. Once these values have been processed the computed rows can simply be discarded.

Unfortunately this does lead to some complications, primarily that of increased shared memory usage. This can cause some limitations in the number of blocks that can be processed simultaneously, which can lead to decreased occupancy. Despite these limits however, the performance impact of such a complex kernel was found to not be as significant as the impact caused by the batching used in Chapter 6.

The original kernel for the QR column deletions described in Chapter 6 takes input of the original upper triangular matrix R_1, and outputs the submatrices $\check{R}_{2\ldots n}$ containing the last n rows of $R_{2\ldots n}$. These matrices can then be analysed using the tests in Chapter 4 to compute the degree estimation. Storing these subresultant matrices is the main cause of the high memory usage in the original GPU implementation.

The algorithm presented in this chapter will instead process the rows of these submatrices as they are computed. This means that the QR deletion kernel presented in this chapter will instead output the minimum and maximum row norms and diagonals of the matrices $\check{R}_{2\ldots n}$. The computation of the squared row norms of the upper half of the triangular factors $\hat{R}_{1\ldots n}$ does still need to be processed in a separate kernel, as do the final computations of the ratios and gradients.

7.1.1 Example

For this example the matrix R_1 from Matrix 1, defined in Chapter 6, will be used. This enables direct comparisons to be drawn with the example in Chapter 6. This entire matrix will be stored on the GPU, and is the only full matrix that will ever be stored in this implementation.

Iteration 1

The first step of this iteration is the computation of the squared row norm of the first row of R_1, extracted from the full matrix. The ith row of $\check{R}_k$ will be denoted as $r_{k,i}$.

$$r_{1,1} = \begin{bmatrix} 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 & -12.9799 \end{bmatrix}.$$

Squaring and summing this vector will give the squared row norm for this row.

$$\|r_{1,1}\|_2^2 = 202.0171.$$

The absolute diagonal of this row $|r_{1,1,1}|$ and the norm computed above will be stored as the current minimum and maximum of the diagonals and row norms respectively. The table below shows the current values for these minimum and maximum values.

k	1	2	3	4	5	6
$\min(\lvert r_{k,i}\rvert)$	0.3095					
$\max(\lvert r_{k,i}\rvert)$	0.3095					
$\min(\lVert r_{k,i}\rVert_2^2)$	202.0171					
$\max(\lVert r_{k,i}\rVert_2^2)$	202.0171					

Once this row has been tested the first rotation for the matrix $\check{R}_2$ will be computed. As with the previous algorithm, the matrix $M_{k,j}$ represents the matrix constructed for the rotation on the kth factor, and the jth iteration of the algorithm.

In the first iteration the matrix $M_{2,1}$ will be computed from rows 6 and 7 from R_1.

$$M_{2,1} = \begin{bmatrix} 0.9422 & -0.1825 & -0.0545 & -6.5186 & -18.1315 \\ 0.3095 & 0.7416 & -0.6115 & -0.2081 & -5.6988 \end{bmatrix}.$$

Performing a givens rotation on this matrix results in the matrix $\bar{R}_{2,1}$.

$$\bar{R}_{2,1} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \\ 0 & 0.7616 & -0.5640 & 1.8365 & 0.2441 \end{bmatrix}.$$

The first row of this matrix consists of final values for $\check{R}_2$, and the second row consists of values that require further processing.

Iteration 2

In iteration 2 the second row of $\check{R}_1$ and the first row of $\check{R}_2$ will be assessed to find the diagonal and squared row norm values.

The second row of $\check{R}_1$ will be extracted from R_1,

$$r_{1,2} = \begin{bmatrix} 0.6583 & 0.7763 & -0.1310 & -3.0065 & -14.0045 \end{bmatrix},$$

and the first row of $\check{R}_2$ will be extracted from $\bar{R}_{2,1}$, using the final values on the first row computed in the previous iteration.

$$r_{2,1} = \begin{bmatrix} 0.9917 & 0.0580 & -0.2426 & -6.2580 & -19.0044 \end{bmatrix}.$$

The squared row norms of each of these matrices is computed

$$\|r_{1,2}\|_2^2 = 206.2182,$$
$$\|r_{2,1}\|_2^2 = 401.3755.$$

The absolute diagonal and squared row norm of $r_{1,2}$ will be compared to the stored minimum and maximums, and the absolute values for $r_{2,1}$ will be stored as the minimums and maximums.

k	1	2	3	4	5	6
$\min(\lvert r_{k,i}\rvert)$	0.3095	0.9917				
$\max(\lvert r_{k,i}\rvert)$	0.6583	0.9917				
$\min(\|r_{k,i}\|_2^2)$	202.0171	401.3755				
$\max(\|r_{k,i}\|_2^2)$	206.2182	401.3755				

Once these rows have been assessed the algorithm continues with the next Givens rotations of $\check{R}_2$ and $\check{R}_3$.

The matrix $M_{2,2}$ will be constructed using the second row of $\bar{R}_{2,1}$ and row 7 of R_1,

$$M_{2,2} = \begin{bmatrix} 0.7616 & -0.5640 & 1.8365 & 0.2441 \\ 0.6583 & 0.7763 & -0.1310 & -3.0065 \end{bmatrix},$$

and the matrix $M_{3,2}$ will be constructed from row 5 of R_1 and the first row of $\bar{R}_{2,1}$,

$$M_{3,2} = \begin{bmatrix} -0.1825 & -0.0545 & -6.5190 & -18.1331 \\ 0.9917 & 0.0580 & -0.2426 & -6.2580 \end{bmatrix}.$$

Performing Givens rotations on these matrices will give the matrices $\bar{R}_{2,2}$ and $\bar{R}_{3,2}$ respectively.

$$\bar{R}_{2,2} = \begin{bmatrix} 1.0067 & 0.0810 & 1.3037 & -1.7815 \\ 0 & 0.9562 & -1.3001 & -2.4341 \end{bmatrix},$$
$$\bar{R}_{3,2} = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \\ 0 & 0.0431 & 6.4553 & 18.9662 \end{bmatrix}.$$

Note that at this point the matrix $\bar{R}_{2,1}$ is no longer necessary, so $\bar{R}_{2,2}$ can be stored in the same memory structure.

Iteration 3

The row norms of the third row of $\check{R}_1$, the second row of $\check{R}_2$, and the first row of $\check{R}_1$ must now be computed. The row vector $r_{1,3}$ is extracted from R_1, and the vectors $r_{2,2}$ and $r_{3,1}$ will be extracted from their respective $\bar{R}_{k,2}$ matrices.

$$r_{1,3} = \begin{bmatrix} 0.2002 & 0.2093 & -1.0077 & 0.1921 \end{bmatrix},$$
$$r_{2,2} = \begin{bmatrix} 1.0067 & 0.0810 & 1.3037 & -1.7815 \end{bmatrix},$$
$$r_{3,1} = \begin{bmatrix} 1.0084 & 0.0669 & 0.9412 & -2.8729 \end{bmatrix}.$$

The squared norms of these vectors are computed,

$$\|r_{1,3}\|_2^2 = 1.1362,$$
$$\|r_{2,2}\|_2^2 = 5.8934,$$
$$\|r_{3,1}\|_2^2 = 10.1608.$$

These norms, and the respective diagonals, are compared against the stored minimums and maximums, with those for $k = 3$ simply being stored, as there is nothing to compare them against.

k	1	2	3	4	5	6
$\min(\|r_{k,i}\|)$	0.2002	0.9917	1.0084			
$\max(\|r_{k,i}\|)$	0.6583	1.0067	1.0084			
$\min(\\|r_{k,i}\\|_2^2)$	1.1362	5.8934	10.1608			
$\max(\\|r_{k,i}\\|_2^2)$	206.2182	401.3755	10.1608			

The next three Givens rotations can now be computed. Firstly the matrices on which the Givens rotations are applied are constructed,

$$M_{2,3} = \begin{bmatrix} 0.9562 & -1.3001 & -2.4341 \\ 0.2002 & 0.2093 & -1.0077 \end{bmatrix},$$
$$M_{3,3} = \begin{bmatrix} 0.0431 & 6.4553 & 18.9662 \\ 1.0067 & 0.0810 & 1.3037 \end{bmatrix},$$
$$M_{4,3} = \begin{bmatrix} -0.0545 & -6.5197 & -18.1358 \\ 1.0084 & 0.0669 & 0.9412 \end{bmatrix}.$$

Performing Givens rotations on each of these matrices results in the matrices $\bar{R}_{2\ldots4,3}$,

$$\bar{R}_{2,3} = \begin{bmatrix} 0.9769 & -1.2296 & -2.5889 \\ 0 & 0.4713 & -0.4876 \end{bmatrix},$$
$$\bar{R}_{3,3} = \begin{bmatrix} 1.0076 & 0.3572 & 2.1142 \\ 0 & -6.4459 & -18.8931 \end{bmatrix},$$
$$\bar{R}_{4,3} = \begin{bmatrix} 1.0098 & 0.4185 & 1.9182 \\ 0 & 6.5066 & 18.0587 \end{bmatrix}.$$

Again, at this point the entries of the matrices $\bar{R}_{2,2}$ and $\bar{R}_{3,2}$ are no longer required, and thus the matrices computed here can be written into the same memory locations.

Further iterations

This process will continue for the next three iterations. In each iteration the row sums of the final entries of the jth row of $\check{R}_1$, and the first row of the matrices $\bar{R}_{2\ldots j+1,j}$, will be tested to find the minimum and maximum diagonals and squared row norms. Then the Matrices $M_{k,j}$ will be constructed, a Givens rotation will be performed on them, and the matrices $\bar{R}_{k,j}$ will be computed. These matrices can be stored in the memory locations for $\bar{R}_{k,j-1}$.

A final step is required once the matrices $\bar{R}_{k,6}$ have been computed in iteration 6. In the final step the squared row norms and diagonals of the final rows of $\check{R}_k$ must be calculated, and these will be compared against the stored values. Once this step is complete the final minimums and maximums of $\check{R}_k$ will have been computed as is shown below.

k	1	2	3	4	5	6
$\min(\lvert r_{k,i}\rvert)$	0.0002	4×10^{-4}	1.0076	1.0098	6.5974	19.8253
$\max(\lvert r_{k,i}\rvert)$	0.9568	1.0067	6.5195	6.9509	6.9950	19.8253
$\min(\lVert r_{k,i}\rVert_2^2)$	4×10^{-8}	1.6×10^{-7}	5.6127	4.8743	48.9300	393.0425
$\max(\lVert r_{k,i}\rVert_2^2)$	206.2182	401.3755	384.5874	363.7819	367.0255	393.0425

7.2 Implementation

The algorithmic changes discussed in Section 7.1 mean that the implementation discussed in Chapter 6 must be modified. In this section these changes will be described.

7.2.1 New Memory Structure

The original implementation, described in Chapter 6, allocated memory for all matrices $\check{R}_{2\ldots n}$, which were then computed in the QR update kernel. These matrices were then analysed to compute the minimum and maximum values from these stored matrices.

As was established in Chapter 6, each Givens rotation performed on the triangular matrix R_k makes use of two rows. The first row will either come from the work in progress rows computed for R_k in the previous iteration, or a row from R_1 if it is the first Givens rotation performed for that particular value of k. The second row is extracted from the final values of R_{k-1} that had been computed in the previous iteration. This rotation then provides a final row and a work in process row that can be used in the next iteration.

Each iteration must therefore store a final row and a work in process row for each matrix $\check{R}_{2\ldots n}$. While it would be possible to store each of these sets of rows in an array of size $({}^{n}/{}_{2})^2$, by moving the pointers of each row after each iteration, this would lead to unnecessary complexity in the handling of pointers. Because this method uses so little memory, redundant memory locations are no longer a major concern. Therefore each of the sets of rows can be stored in a triangular matrix, with the nth row of this matrix storing the row values for $\check{R}_n$. This means that each matrix has the correct width of row storage for it's maximum row width, and that the row pointers do not need to change between iterations. In addition to the row storage there must be memory locations to store n minimum and maximum values for both diagonals and row norms, as well as the computed ratios.

In the first implementation of this new method it was noted that the large number of structures that need to be defined, and memory allocated for, were causing a significant amount of overhead when compared to the implementation described in Chapter 6. This was addressed by allocating the memory used in several chunks, for example all of the structures for storing the rows of the computations in all blocks are allocated together. The pointers of individual memory structures within these chunks were found using pointer arithmetic. By reducing the number of GPU memory calls in this way, the overhead caused by the increased number of structures was drastically reduced.

7.2.2 Kernel Changes

As discussed in Section 7.1, the main change in this implementation is moving the computation of the row norms into the same kernel as the upper triangular factor computations.

In each iteration of the kernel, the row sum is processed first, with the first iteration computing the squared row norm of the first row of $\check{R}_1$ using a row prefix sum. The final result of this sum is stored in the minimum and maximum memory locations for $\check{R}_1$. The kernel then moves on to process the first rotation for $\check{R}_2$, storing the two rows generated by this rotation.

In the second iteration the square row norms for the second row of $\check{R}_1$ and the first row of $\check{R}_2$ are computed. The result for $\check{R}_1$ being compared against the stored minimum and maximum values, and the result for $\check{R}_2$ being stored as both the minimum and maximum values for this $\check{R}_2$. Once the row sums have been computed the first row of $\check{R}_3$ and the next rows of $\check{R}_1$ and $\check{R}_2$ are computed.

Additionally, while the squared row norms of $\check{R}_{1\ldots n}$ are computed, the minimum and maximum diagonal values can be computed at the same time. This is accomplished by

simply testing these values against a stored minimum and maximum at the same time as the computed squared row sums are checked against their stored minimum and maximum.

This pattern continues, with each iteration analysing the final rows computed in the previous iteration, and comparing the row sums to the stored minimum and maximum values, before moving on to compute the next work in progress and final rows.

Both the prefix sum to compute the squared row norms and the storage of the Givens matrices use shared memory, which can cause an issue with high shared memory usage. This will reduce the number of blocks that can be processed simultaneously for large problems. However, neither section requires the contents of shared memory to persist between iterations. To mitigate the amount of shared memory needed, the same shared memory locations are used for both sections of the kernel, simply overwriting the data used in the previous step.

Due to the length of this kernel, the pseudocode has been split into several sections. Listing 8 shows the overall structure of the kernel, including the outer loop and computation of values that will be used throughout. Line 33 refers to the pseudocode in Listing 9. Line 38 refers to the pseudocode listed in Listing 10. Listing 8 also defines the values used commonly in all sections. Similarly to Chapter 6, the value of k will start at 0 in the pseudocode, which will make the indexing of the relevant matrices and rows simpler.

Listing 8 shows the relatively simple overall structure of the low memory implementation. The number of iterations in the main loop of the kernel is increased by 1 when compared to the QR deletion kernel presented in Chapter 6. This is to allow for the computation of the row norms of $\check{R}_1$ to be processed in the first iteration, and this is why the computation of the squared row sums occurs first. The check before the QR deletion section of the kernel enables the final iteration to simply calculate the row sums of the previous iteration without needing to compute more Givens rotations.

Listing 9 shows the code for finding the minimum and maximum squared row norms and diagonal values from $\check{R}_{1\ldots n}$. Firstly, the code gets batched to allow for load balancing as is seen on Line 7. This is performed in the same way as the implementations shown in Chapter 6. If $k = 1$ (or 0 in the pseudocode) the value must be extracted from the original upper triangular matrix, as shown in the example in Chapter 6, otherwise the rows to be summed are the final rows computed in the Givens rotations in the previous iteration. The selection of the relevant rows is seen on Lines 14 to 19. A warp shuffle prefix sum is then performed on the squares of the entries of all of these rows, saving the running sum to shared memory to allow the sums to be batched, as is seen on Lines 26 to 33. Once this sum has been performed, the first work unit for each value of k checks the diagonal value against the stored minimum and maximum values, and replaces these as necessary. The final work unit for each value of k also does the same with the squared row sums, subtracting the sum of R_{k-1} from the sum of R_k before performing the check. These checks can be seen on Lines 36 to 51.

The execution of the Givens rotations remains very similar to the method presented in Chapter 6. The pseudocode for this is shown in Listing 10. First the load balancing

```
// R1 is an array storing the original upper triangular matrix
//      in triangular form
// n is the width of R1
// finalRows is an array storing the values from the final rows
//      of each triangular factor after each Givens Rotation
// wipRows is an array storing the values from the work in progress
//      rows of each triangular factor after each Givens Rotation
// diagsMax stores the maximum diagonal values found for the lower
//      section of the triangular factors
// diagsMin stores the minimum diagonal values found for the lower
//      section of the triangular factors
// normsMaxstores the maximum squared norm values found for the
//      lower section of the triangular factors
// normsMin stores the minimum squared norm values found for the
//      lower section of the triangular factors
// sharedG is an array in shared memory that stores the values that
//      make up the Givens matrices
// scanShared is an array in shared memory (with the same pointer as
//      sharedG) that is used during the warp shuffle prefix sum
// toAdd is a single location in  shared memory that stores the running
//      sum to allow batching the warp shuffle algorithm
// warpShufflePrefixSum(width, value, scanShared) is a function that
//      performs a warp shuffle prefix sum with specified with, using
//      the provided value in that thread and the given shared memory
// getTriMatrixIndex(width, row) is a function that retrieves the index
//      of a particular row in a triangular matrix with specified width


gpu_parallel_for thread ← 0 to blockWidth
    for pass ← 1 to n + 1
        newFinalWidth ←  n - pass
        oldFinalWidth ←  n - pass + 1

        minMaxRowSumsAndDiagonals

        synchroniseThreads

        if (pass < n)
            performGivensRotations
```

Listing 8: Pseudocode showing the overall structure of the new kernel

```
1  // FIND MIN AND MAX ROW ROW SUMS AND DIAGS
2  totalVals ← oldFinalWidth * pass
3  batches ← ceil(totalVals / blockWidth)
4
5      if (threadID = 0) toAdd ← 0
6
7      for batch ← 0 to batches
8          width ← (batches - batch = 1) ? totalVals%blockWidth : blockWidth
9          wu ← batch*blockWidth + threadID
10         k ← wu/oldFinalWidth
11         i ← wu%oldFinalWidth
12
13         //Get relevant row
14         if (k==0)
15             tempFinal ← R1
16             tempFinalIndex ← getTriMatrixIndex(origN, n+pass-1)
17         else
18             tempFinal ← finalRows
19             tempFinalIndex ← getTriMatrixIndex(n, k)
20
21         if (wu<totalVals) tempVal ← tempFinal[tempFinalIndex + i]
22         else tempVal ← 0
23
24         //Compute Prefix sum and output to SM
25         synchroniseThreads
26         warpShufflePrefixSum(width, tempVal*tempVal, scanShared)
27         synchroniseThreads
28         if (wu<totalVals) {
29             if (i = 0) tempDiags[k] ← abs(tempVal)
30             if (i = oldFinalWidth-1) tempSums[k] ← scanShared[threadID] + toAdd
31         }
32         synchroniseThreads
33         if ((batch!= batches-1) && (threadID==0)) *toAdd+= scanShared[blockWidth-1]
34
35     //Find min and max values
36     for batch ← 0 to ceil(n/blockWidth)
37         k ← batch * blockWidth + threadID
38         sum ← tempSums[k]
39         if (k>0) sum -← tempSums[k-1]
40
41         if (k < n)
42             if (k = pass-1)
43                 diagsMax[k] ← tempDiags[k]
44                 diagsMin[k] ← tempDiags[k]
45                 normsMax[k] ← sum
46                 normsMin[k] ← sum
47             else
48                 diagsMax[k] ← max(diagsMax[k], tempDiags[k])
49                 diagsMin[k] ← min(diagsMin[k], tempDiags[k])
50                 normsMax[k] ← max(normsMax[k], sum)
51                 normsMin[k] ← min(normsMin[k], sum)
```

Listing 9: Pseudocode showing the process of computing the minimum and maximum squared row norms and diagonals in the new kernel

technique is used again, batching the computations and providing a unique work index for each computation. The first work units in every iteration will compute the entries of the Givens matrices for this iteration. After synchronising the threads the rotations are then performed on the relevant rows, firstly by finding the relevant rows and entries for the Givens matrix, and then performing the matrix product.

The main difference in the computation of the QR deletion, compared to that from Chapter 6, is the selection of rows on which the Givens rotation is performed. The new row computations can be seen on Lines 32 to 46. Note that in this implementation the structures for the final and work in progress rows are used, with the code selecting a specific row from these structures, rather than requesting the index of the matrix. Additionally, the selection of the second row on which to perform the givens matrix is no longer constant for all values of k, and for the first triangular factor R_1 this must instead be extracted from the original matrix R_1. This is because the full triangular factors are no longer stored for all of the subresultant matrices, and so the first factor must be treated differently, as this is stored in its full form.

7.2.3 GPU Profiling and and Establishing Launch Parameters

Similarly to the implementation proposed in Chapter 6, this implementation was thoroughly profiled at various stages of development to test its performance. The final profiling results are discussed in this section.

In the profiling performed in Chapter 6, it was shown that the execution time of the implementation was primarily split between two kernels. Here those kernels were combined into one, resulting in the low memory implementation spending 97.7% of the runtime in the primary kernel at a polynomial degree of 250, and 99.5% of the runtime in the primary kernel at a degree of 1250.

When the default launch parameters are used this kernel requires 85 registers per thread. Using the default launch parameters limits the theoretical device occupancy to only 25%, and the block size to 512. As was discussed in Chapter 6, the use of registers can be lowered using `__launch_bounds__`. The usage of this feature requires the maximum block size and the minimum number of multiprocessors for the kernel to be specified. By requesting a maximum block size of 1024, with a minimum multiprocessor count of 1, the kernel restricts the registers per thread to 64. This means the kernel can achieve a theoretical occupancy of 50%. While not ideal, this is the same occupancy as achieved in the implementation in Chapter 6.

While restrictions with the launch bounds feature in the previous implementation caused a significant decrease in performance in most cases, the opposite is true for this implementation. Even for polynomial degrees as low as 250, restricting the registers in this way reduces the duration of the primary kernel from roughly 92ms down to roughly 57ms. An improvement from 545ms down to 457ms was seen at a polynomial degree of 500. Due to the size of the blocks required, polynomials of degrees greater than those discussed here would not launch before the bounds had been set.

```
1  // COMPUTE GIVENS VALUES
2  totalWork ← newFinalWidth * pass
3  maxWorkPerThread ← ceil(totalWork / blockWidth)
4  for batch ← 0 to maxWorkPerThread
5      wu ← (batch * blockWidth) + threadID
6      if (wu < totalWork)
7          if (batch < ceil(n / blockWidth))
8              if (wu < pass)
9                  tempK ← pass-wu
10                 tempRowIndex ← pass-tempK
11
12                 if (tempRowIndex = 0)
13                     x ← orig[getTriMatrixIndex(origN, (n-tempK)) + tempK]
14                 else
15                     x ← wipRow[getTriMatrixIndex(tempK,  n-1)]
16
17                 if (tempK = 1)
18                     y ← orig[getTriMatrixIndex(origN, n+pass-1)]
19                 else
20                     y ← finalRows[getTriMatrixIndex(tempK-1, n)]
21
22                 na ← sqrt(x*x+y*y)
23                 sharedG[pass-1-wu].c←x/na
24                 sharedG[pass-1-wu].s←y/na
25         synchroniseThreads
26
27         k ← pass - (wu / newFinalWidth)
28         i ← wu % newFinalWidth
29         rowIndex ← pass-k
30         fullRow ←  n - k
31
32         //RETRIEVE RELEVENT GIVENS VALUES
33         if (wu < totalWork)
34             if (rowIndex==0)
35                 row1 ← orig
36                 index1 ← getTriMatrixIndex(origN, n-k) + k
37             else
38                 row1 ← wipRow
39                 index1 ← getTriMatrixIndex, k, n-1)
40
41             if (k==1)
42                 row2 ← orig
43                 index2 ← getTriMatrixIndex(origN, n+pass-1)
44             else
45                 row2 ← finalRow
46                 index2 ← getTriMatrixIndex(k-1, n)
47
48             c ← sharedG[k-1].c
49             s ← sharedG[k-1].s
50             temp1 ← row1[index1]
51             temp2 ← row2[index2]
52         synchroniseThreads
53         //COMPUTE AND WRITE NEW VALUES
54         if (wu < totalWork)
55             rowIndex1 ← getTriMatrixIndex(k, n)
56             finalRow[rowIndex + i] ← temp1 * c + temp2 * s
57             if ((rowIndex < fullRow-1) && (i!= 0))
58                 rowIndex2 ← getTriMatrixIndex(k, n-1)
59                 wipRow[rowIndex2 + i - 1] ← temp1 * -s + temp2 * c
60         synchroniseThreads
```

Listing 10: Pseudocode showing the process of performing Givens Rotations in the new kernel

The achieved occupancy of the kernel ranges from 48.7% for the lower polynomial degrees, to 50% for higher degrees.

7.3 Memory Usage

As discussed throughout this chapter, the main aim of this implementation was to reduce the memory usage of the GPU implementation discussed in Chapter 6. In the original implementation, single trials of the AGCD algorithm could take up the entire memory of the device when the degree of the polynomial is sufficiently large. Figure 7.1 is included in this chapter for easy reference, it is identical to Figure 6.4. As can be seen from this figure, when the polynomials are of degree 2128 the algorithm already uses more memory than is available on the high end NVIDIA Titan V, which has 12GB of GPU memory. Most commercial GPUs at the time of writing have memory between 4GB and 8GB.

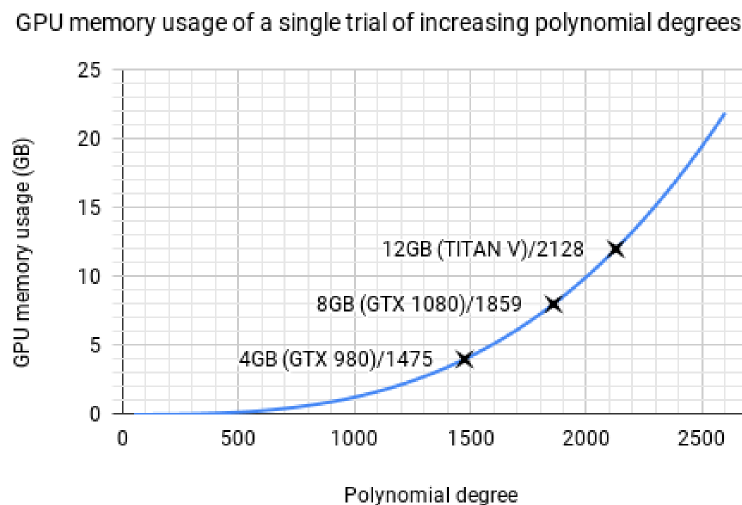


Figure 7.1: GPU memory usage of the original method from Chapter 6

The results shown in Chapter 6 showed that when the polynomials convolved with an AGCD were of high degree the results returned by the full computation can be unreliable. However, it may still be desired to attempt the computation on degrees beyond 2000, when little noise is present, or when some degrees of AGCD can be ruled out.

The issue of memory usage in the implementation in Chatper 6 is not just a problem when processing polynomials of very high degrees. The results in Chapter 6 show the issues for polynomials of intermediate degree. When the computation needs to be batched, the overall performance of the implementation suffers. There are multiple causes of these performance issues. Firstly, batching causes the kernel and memory calls to be sent to the GPU in multiple batches. While no extra data is copied to the GPU, and no extra computation is performed, the additional kernel and memory calls required will cause a significant amount of extra overhead. Secondly, when the work is split this could lead to non-optimal utilisation of the device, leading to more idle cores and an overall less efficient

implementation.

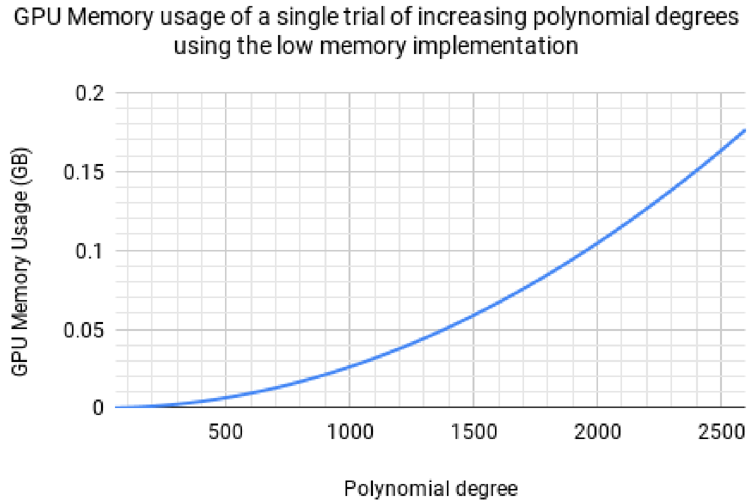


Figure 7.2: GPU memory usage of the low memory method

The solutions described in this chapter provide significantly improved memory utilisation, which will in turn lead to better device utilisation overall. Figure 7.2 shows the memory needed for a single trial for polynomials of degrees up to 2500. It should be noted that this is a theoretical maximum, and a real world application may require more GPU memory than is shown here, as some applications, such as MATLAB, add overhead to the GPU memory. The results here show that even at a polynomial degree of 2500, this algorithm would only require around 0.17GB of memory, compared to roughly 20GB in the previous implementation.

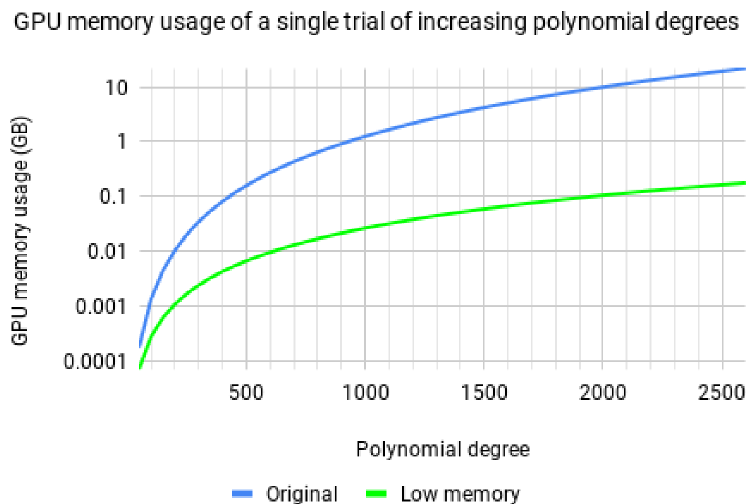


Figure 7.3: Comparison of the memory usage between the two methods

Figure 7.3 shows a comparison between the implementation from Chapter 6 and the

implementation from this chapter on a log scale. This graph shows that both algorithms scale their memory usage similarly, but the original method uses over 100 times more memory than the implementation presented in this chapter.

The limiting factor of the low memory implementation of the algorithm presented in this chapter is no longer that of the global GPU memory, but of shared memory. Current NVIDIA GPUs have a shared memory limit of 48kB. This means that the total number of double precision values that can be stored simultaneously in shared memory is 6144. The kernel in this implementation requiring the most shared memory is that of the primary kernel, where the computation and analysis of $\check{R}_{1\ldots n}$ is performed. In this kernel, while space is saved by reusing the same shared memory locations for both the prefix sum and the entries of the Givens matrices, the total shared memory required by this kernel is still very high.

In the current implementation the total number of double precision values required is equal to the total shared memory needed for the block size, plus $2n$ entries to store temporary sums and diagonals found, plus an extra value for storing the running total of the batched prefix sum. Therefore the total number of double precision entries required a, for a matrix of width n is

$$a = s + 2n + 1,$$

where s is the memory required for each batch of the prefix sum,

$$s = q + \frac{q}{32} \quad , \quad q = \min(n, 1024).$$

This limits the degree n of the polynomials being processed (assuming polynomials of equal degree) to 2543. This limit is already higher than that of the algorithm presented in the previous implementation, and could still be improved further. By moving some of the shared memory locations into global memory, particularly those in which the temporary diagonals and row sums are stored, the shared memory usage could be reduced. However, this would also impact performance, as global memory has higher memory latency than shared memory. This modification could be made if images larger than this limit in either dimension are required to be processed.

7.4 Results

The memory utilisation improvements discussed in Section 7.3 provide significant improvements to the computational performance of the algorithm, with significant benefits to both runtime and scalability when compared to the method presented in Chapter 6.

As with Section 6.3, the results here will be broken down into two sections. Section 7.4.1 will briefly discuss the reliability testing, where the algorithm is checked to ensure the exact same results are offered by this implementation as the original GPU and CPU versions. This section is brief, as it was found that the algorithm performed identically in

this regard to all previous implementations. Section 7.4.2 will discuss the runtime testing, where the runtime is compared to the CPU and GPU methods, and scalability is tested. All tests in this section were performed on a system with a 6 core Intel i7 6850k CPU and an NVIDIA Titan V GPU. All tests were run 10 times and an average taken of these results to give the final results. The standard deviation in all cases was less than 11% of the mean, typically being less than 6% of the mean.

7.4.1 Reliability Testing

Similarly to the testing described in Chapter 6, the algorithm was tested for reliability. The results at each stage of the execution were examined, and the results checked for consistency against existing algorithms. At all stages of the algorithm the results of the low memory implementation were exactly the same as those from the CPU and original GPU methods.

Using the same polynomials as were tested in Chapter 6 provided the same results as the implementation in this chapter. Thus, for more information about the reliability of this implementation Table 6.1 can be referenced.

The reduced form of the algorithm, presented in Chapter 4 and tested in Chapter 6, was not implemented with the new low memory techniques. The reasons behind this were explained in Chapter 6. However, the reliability advantages of such an algorithm can still be gained by limiting the size of AGCDs being considered, and only considering gradients up to a set point. This algorithm is therefore still able to achieve the reliability shown in Tables 4.1 and 4.2 of Chapter 4.

7.4.2 Runtime Testing

This section will test the runtime of the new method, and compare it against the CPU and original GPU versions. To test the scalability of this implementation the tests will measure the runtime of varying degrees of the convolved polynomials and of varying numbers of trials.

Figure 7.4 shows the runtimes of the implementation presented in this chapter, compared to that presented in Chapter 6. The original method times are shown with dotted lines, while the method from this chapter are shown with solid lines. The tests with the same numbers of trials are shown in the same colour. When the degree of the convolved polynomials is as low as 250 there is very little difference in the performance between the implementations, but as the degree increases the improvement appears more apparent.

For polynomials of degrees between 250 and 1250 both methods scale relatively consistently, though notable runtime improvements from the low memory implementation over the previous implementation can already be observed. After this point, the low memory algorithm continues to scale well, while the original implementation begins to encounter performance issues. This, as discussed in Chapter 6, was largely due to the batching method implemented, which enabled the processing of large polynomials by processing

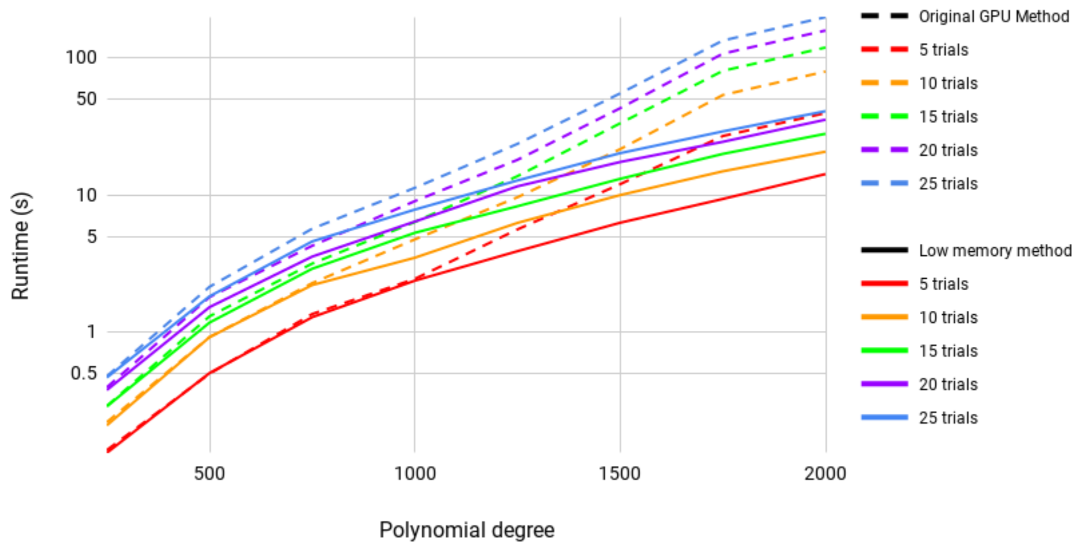


Figure 7.4: Runtimes of the low memory implementation for increasing polynomial degrees and various numbers of trials, with runtimes from the original implementation for comparison

batches of trials in sequence. This batching occurs at the expense of device utilisation, and increased overheads. By reducing the memory of each individual trial, and thus reducing the need for batching, the low memory implementation allows for a greater number of trials to be processed in the same kernel call. This allows the algorithm to scale significantly more consistently for all degrees of polynomial tested.

Figure 7.5 shows how the algorithm performs across an increasing number of trials for polynomials of degree 1000. This was shown on a log scale, to make it easier to see the GPU implementations in comparison to the CPU implementations. The performance issues caused by the batching are clearly visible on the line representing the original GPU method, while the low memory version has a smooth curve much more similar to that of the CPU implementations. The low memory implementation does appear to only be marginally faster than the original GPU method here, and that is due to the choice of a polynomial degree of 1000. Batching at this degree in the original GPU method provides for a smoother curve, as the smaller problem sizes for individual trials, compared to larger degrees, allows for greater device utilisation.

Figure 7.6 shows the relative speedup of the low memory implementation across varying numbers of trials when compared to the CPU implementations and the original GPU implementation. The speedup against the GPU and CPU parallel implementations is relatively consistent for all numbers of trials, which is to be expected as it was shown in Section 6.3 that the degree of the convolved polynomials affects runtime more than the number of trials. There is a significant increase in the speedup when compared to the

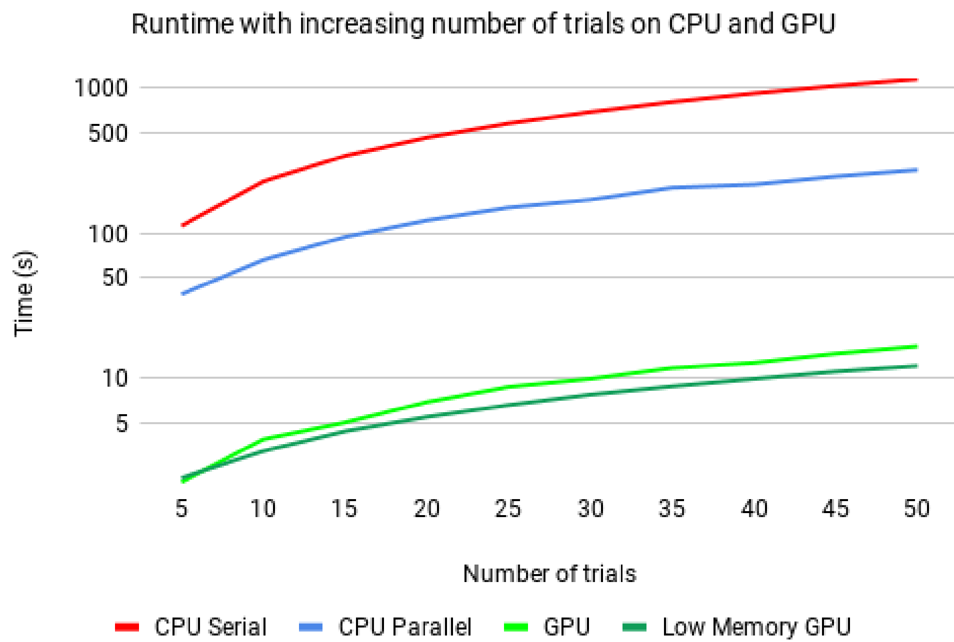


Figure 7.5: Runtimes of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of trials

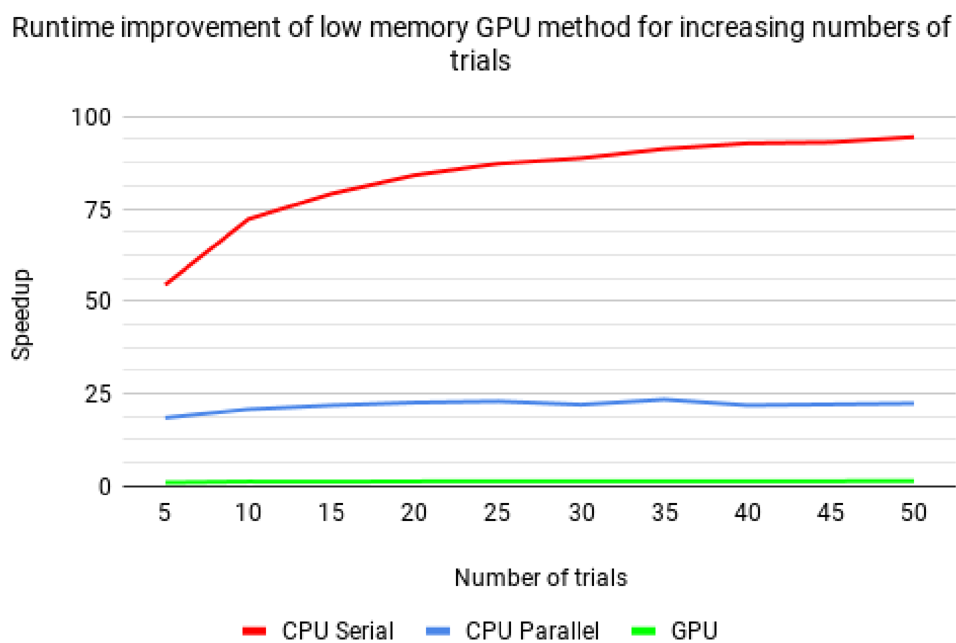


Figure 7.6: Runtime improvement of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of trials

serial implementation, with the low memory implementation being 57.45 times faster at 5 trials, and eventually levelling off to being between 90 to 95 times faster when 35 or more trials are processed. This increase in relative performance is due to the utilisation of the device being better when more data is able to be processed simultaneously.

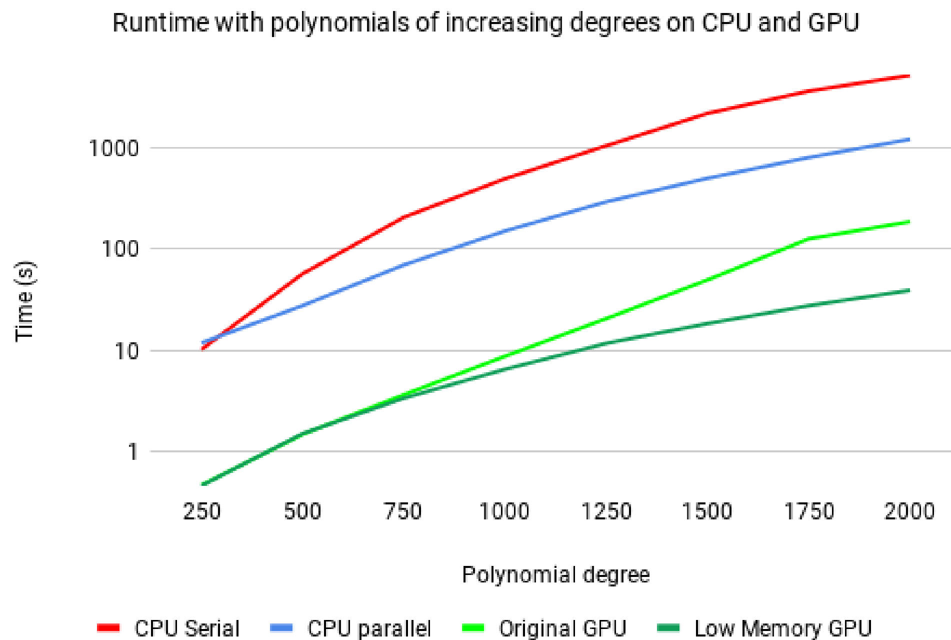


Figure 7.7: Runtimes of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing polynomial degrees

Figure 7.7 shows the runtimes of 25 trials for polynomials of increasing degree on a log scale. The improvement from the removal of batching is clearly visible here, as the original and low memory methods start off being very close together, but, as the batching causes the original method to encounter performance issues, the low memory version keeps a curve very similar to that of the CPU implementations.

Figure 7.8 shows the speedup of the low memory implementation over the original GPU and CPU implementations for increasing polynomial degrees. There is only a modest improvement in performance over the GPU implementation in Chapter 6, between 1.01 times at a degree of 250, to 4.77 times at a degree of 2000. However, the trend here does appear to be an increase in relative performance when compared to all three other implementations. This increase starts to level off at degrees of 1750-2000, though performance is still improving for greater numbers of trials. This suggests that the algorithm not only scales better than other algorithms, but will also continue to at least scale in line with them at higher degrees.

Comparing this to the speedup for the original GPU method, shown in Figure 6.8, it can be seen that, while the original GPU implementation had quite inconsistent speedups against the CPU parallel version, here the speedup is consistently increasing as the number

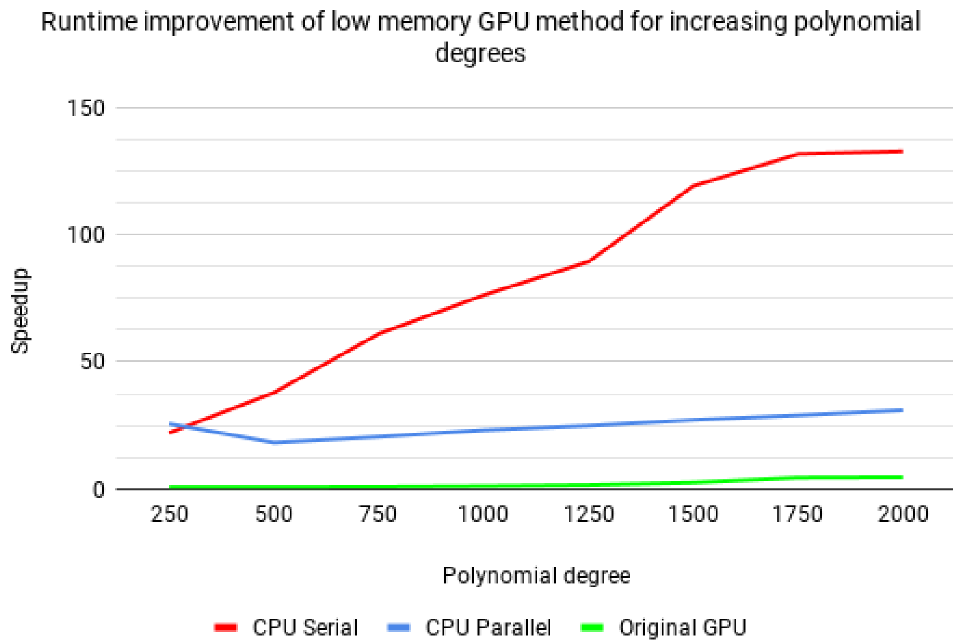


Figure 7.8: Runtime improvement of the low memory GPU method compared to the original GPU method, and the CPU methods for increasing numbers of polynomial degrees

of trials increases. Even when the speedup is at its lowest, at a degree of 500, a speedup of 18.38 times is still present. The high speedup that can be seen before this point at a degree of 250 can be explained, as was discussed in Chapter 6, as the initial overhead from the parallel pool creation. The overhead of the parallel pool initialisation remains constant for all degrees, and thus the impact is larger relative to the overall runtime when the overall runtime is lower.

7.4.3 Testing on Limited Hardware

While the results shown in this section so far have exceeded the original implementation presented in Chapter 6, the low memory implementation proposed in this chapter has benefits outside of runtime performance. The results shown prior to this point have made use of a high end NVIDIA Titan V GPU with a large 12GB of memory. However, the low memory footprint of the new algorithm allows even low end or dated consumer GPUs to take advantage of the accelerated algorithm. This section will test how the algorithm performs when testing on such hardware.

The implementation was tested on a system with an NVIDIA GTX 780 GPU with 3GB of GPU memory, and an Intel 4790k CPU. While these were high end consumer level components at the time of release, they are now somewhat dated, and more comparable to newer low to mid level hardware available today. The Titan V on the other hand is a recent, high end, professional level GPU. While the Titan V, released in 2017, has 5120 CUDA cores with a maximum clock speed of 1455 MHz, the GTX 780, released in

2013, has only 2304 CUDA cores with a maximum clock speed of 900 MHz. While these specifications do not give the full picture of the difference in expected performance of the two GPUs, they do give an idea of the difference between the two devices.

The same tests performed in Section 7.4.2 were again performed on the system with the GTX 780. As the rest of these results will show, the low memory implementation performed well, even on the more limited hardware, still managing to outperform the CPU parallel implementation by a significant margin. These tests demonstrate that while the best GPU performance comes from high end GPUs, even a less powerful, more affordable GPU can provide significant benefits over a purely CPU implementation.

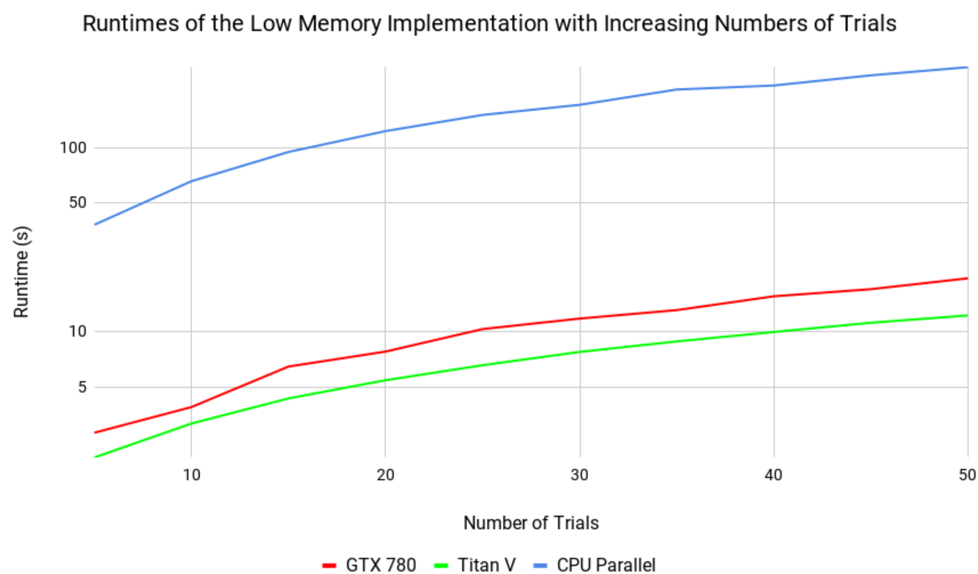


Figure 7.9: Runtimes of the low memory GPU method on an NVIDIA GTX 780 compared to the NVIDIA Titan V and CPU serial implementation for increasing numbers of trials

Figure 7.9 shows the runtimes of the low memory method on the more limited hardware of the NVIDIA GTX 780. The CPU method has been included in this graph as a point of comparison. Even on less powerful hardware, it is shown that the low memory implementation scales well in this regard, and is much closer to the runtimes of the implementation on the Titan V than the more modern high end CPU.

The same remains true when increasing the polynomial degree. The two GPUs have almost identical runtimes when processing polynomials of lower degrees, due to under-utilisation of the high end Titan V. The two curves start to separate, with both levelling off so the devices are scaling similarly, with the GTX 780 taking roughly 1.74 times longer than the Titan V. It is also clear that the memory issues of the implementation described in Chapter 6 have been solved. The inconsistencies seen in the curves in the implementation presented in Chapter 6, due to the need for batching, are not present here, even with the significantly reduced memory capacity of the GTX 780.

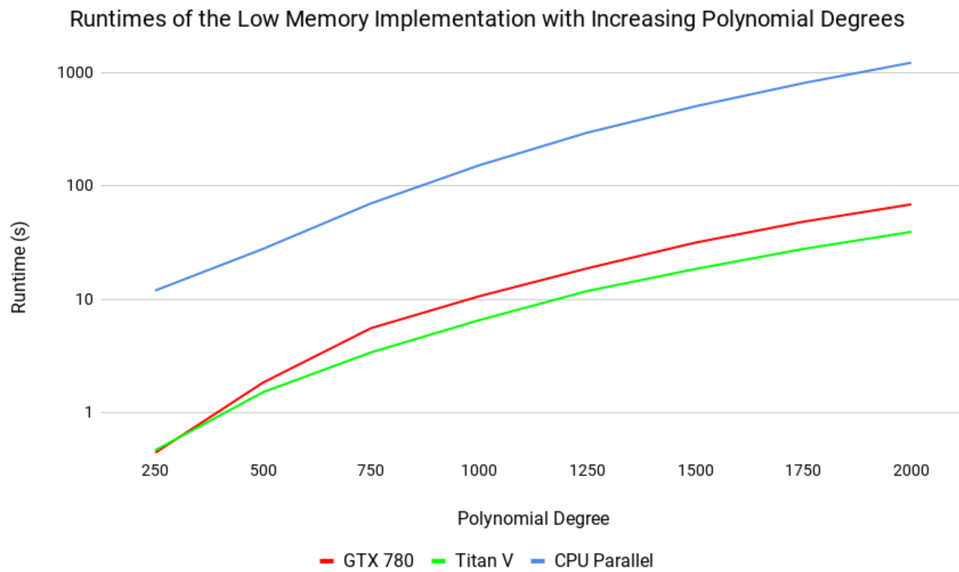


Figure 7.10: Runtimes of the low memory GPU method on an NVIDIA GTX 780 compared to the NVIDIA Titan V and CPU parallel implementation for increasing polynomial degrees

7.5 Profiling of the Accelerated Blind Image Deconvolution Algorithm

The algorithm proposed in this chapter provides a fast method of computing the degree of the AGCD, as was used in the BID algorithm described in Chapter 4. Given the extent that this algorithm has been accelerated, it is now worth going back to the original profiling of the full blind image deconvolution algorithm, with the new degree computation method, and seeing where further optimisations can be made.

The profiling in this section was performed using the same image and PSF as were used in Chapter 4. The same CPU, an Intel i7 6850K, was used, alongside an NVIDIA Titan V.

Figure 7.11 shows the new profiling results of the full deconvolution algorithm after acceleration of the degree computation. When compared to the profiling in Figure 4.10 of Chapter 4, it is clear that the acceleration was successful. The new implementation reduces the runtime of the overall program from 42.45 seconds to 13.50 seconds. The degree computation algorithm, shown here in the function `DegreesRowColumnStandalone-CudaLowMem` is reduced to 2.82 seconds, with the CUDA function itself taking only 0.12 seconds.

With this acceleration it becomes clear that the next most expensive section of the algorithm is that of the optimal column selection, as part of the AGCD coefficient computation. The largest proportion of self time remaining is that in `qrdelete`, which is called from the above function `CalculateOptimalColumn`. The next most expensive

Profile Summary
Generated 12-Aug-2019 11:50:19 using performance time.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
RunStandaloneWithImgAndPSFCudaLowMem	1	13.499 s	0.822 s	
DeblurCudaLowMem	1	9.643 s	0.029 s	
SAlgorithmStandalone	2	6.746 s	0.035 s	
CalculateOptimalColumn	2	5.887 s	0.083 s	
qrdelete	1024	5.782 s	4.845 s	
gpuDevice	1	2.892 s	0.116 s	
DegreesRowColumnStandaloneCudaLowMem	2	2.818 s	0.598 s	
GPUDevice.GPUDevice>GPUDevice.select	1	2.776 s	0.002 s	
selectDevice	1	2.769 s	2.766 s	
optimal_linprog	50	1.553 s	0.048 s	
linprog	50	1.476 s	0.049 s	
planerot	261632	0.937 s	0.937 s	
...DualSimplex>LinprogDualSimplex.run	50	0.673 s	0.059 s	
optimoptions	100	0.623 s	0.008 s	
createSolverOptions	100	0.615 s	0.048 s	
KthSylvesterM	50	0.587 s	0.587 s	

Figure 7.11: Results of profiling the MATLAB implementation of the image deconvolution algorithm using the low memory GPU degree computation

algorithms below this involve linear programming, for which parallel methods have been widely researched, and construction of the Sylvester matrix, which does not present a significant research challenge.

Since the optimal column selection takes a significant amount of the remaining runtime, and requires a novel solution, it should be the next focus for acceleration.

7.6 Conclusion

The low memory degree computation described in this chapter represents a significant improvement over the original algorithm proposed in Chapter 6. With the changes described the new implementation is able to process the computation of the degree of an AGCD faster than the original GPU implementation, and the improved scalability allows for polynomials of higher degrees to be processed on hardware with limited amounts of memory.

Section 7.5 shows the updated profile of the MATLAB implementation after this acceleration is integrated into the serial implementation. The speedups demonstrated here are a significant improvement over the original implementation, though other sections of the deconvolution must still be optimised. The next chapter will discuss the optimisation and acceleration of the computation of the optimal column to be used in the structured matrix methods involved in the computation of the coefficients of the AGCD.

Chapter 8

GPU Acceleration of the Computation of the Coefficients of an AGCD

Chapters 6 and 7 proposed an accelerated method for the computation of a degree of an AGCD. While the results in these chapters did show a significant improvement in the runtime of the overall algorithm, there still exist sections of the BID algorithm in which optimisations can be made. The most significant section of the remaining runtime of the BID algorithm is the computation of an optimal column for use in the modified SNTLN method proposed by Winkler and Hasan [56]. This will be used to compute the coefficients of the AGCD, as was originally discussed in Chapter 4. In this chapter, the optimal column computation, and how this can be accelerated using GPUs, will be discussed.

Section 8.1 will give an overview of the algorithm, including areas within the algorithm that optimisations were investigated to reduce unnecessary computation. Section 8.2 will discuss the details of implementing the algorithm discussed in Section 8.1, including the pseudocode of the algorithm, and the final profile of the implemented algorithm. Section 8.3 will detail how the implementation was tested, for both the runtime improvements and to ensure reliability, and testing the algorithm on lower end hardware.

8.1 Algorithm

The algorithm with which the AGCD coefficients are computed was discussed briefly in Chapter 4. In this section the algorithm will be discussed in more detail, and optimisations that can be made to this algorithm will be discussed. The aim of this computation is to find the optimal column c_o of the pth subresultant matrix $S_p(f, g)$, with which the coefficients will be computed. The subresultant matrix $S_p(f, g)$ is constructed using the polynomials $f(x)$ and $g(x)$, of degrees m and n respectively, where p is the degree of the AGCD. The construction of these matrices was discussed in Chapter 3.

In the degree computation there were multiple trials of different pairs of polynomials

$f(x)$ and $g(x)$, each selected from the rows, or columns, of the input image. In this algorithm only one pair of polynomials is considered. The original BID algorithm, presented by Winkler, selected the polynomials from the first trial in which the degree computation provided the modal value for the degree of the AGCD, as this gives a reasonable chance of finding a low rank estimation of S_p assuming that the computed degree is correct. While this provides good results, the selection of these polynomials can be improved.

It is proposed in this thesis that the pair of polynomials used should be those that returned the minimum combined gradients from the tests described in Chapter 4. Lower gradients would suggest that the entries in the lower rows of R_k are closer to zero, and thus the subresultant matrix $S_k(f, g)$ is closer to being rank deficient.

As was discussed in Chapter 4, the optimal column is column c_p from the Sylvester matrix S_p in which the residual of the approximate equation $A_p x \approx c_p$ is minimised. This section will give an overview of how that residual is computed.

8.1.1 Algorithm Overview

Starting with the Sylvester subresultant matrix S_p, each column c_t must be tested, where $t = 1 \ldots w$ and $w = m + n - 2p$, the width of the subresultant matrix S_p. This test involves the use of QR decomposition to find the orthogonal factor $Q_{p,t}$, of the matrix $S_{p,t}$, which is the subresultant matrix S_p with column t removed. Similarly to the degree computation, performing the full QR decomposition of each matrix $S_{p,t}$ is unnecessary for the computation of the residual. This can instead be computed through QR deletions of each column from the factors Q_p and R_p of the subresultant matrix S_p,

$$S_p = Q_p R_p.$$

The computation of the factors $Q_{p,t}$ and $R_{p,t}$ can then be achieved through QR deletion of column t from the factors Q_p and R_p, to give the factors $Q_{p,t}$ and $R_{p,t}$ of the matrix $S_{p,t}$, for all values of t. The process of performing QR column deletion is discussed in Chapter 3. As discussed in that chapter, when removing a single column from the original QR decomposition, the QR updates via Householder reflections and Givens rotations reduce to the same computation. Thus there are no performance benefits in this case of using Householder transformations, and Givens rotations shall be used instead.

$$S_{p,t} = Q_{p,t} R_{p,t} \quad , \quad t = 1 \ldots w,$$

All of these QR deletions can be performed in parallel, and all computations within these deletions that occur on the same row of $R_{p,t}$, or column of $Q_{p,t}$, can also be performed in parallel. This is a significant amount of work with potential parallelism. Unfortunately this leads to an unbalanced workload, as fewer rotations are needed to update the factors of S_p for greater values of t. Additionally, on each iteration for each matrix, the matrix products of the rotations will decrease in size. Methods to balance this workload will be discussed in Section 8.2.

Unlike the degree computation, in which the orthogonal matrix Q can be discarded, the orthogonal factors $Q_{p,t}$ are needed for the computation of the residual, and therefore the factor Q_p is required to compute $Q_{p,t}$. Additionally, the upper triangular factor R_p is required for the computation of $Q_{p,t}$, and thus both factors are required in this algorithm.

Once the computation of all orthogonal matrices $Q_{p,t}$ is complete, the product of the transpose of each of the orthogonal factors $Q_{p,t}$ and the tth column of S_p, c_t, is computed. The resulting vector will be known as h_t.

$$h_t = Q_{p,t}^T c_t.$$

The residual of this matrix is computed through the norm of the last p elements of h_t, denoted as $h_{t,(w-p)\ldots w}$. The optimal column is computed by finding the column which has the minimum residual associated to it.

$$r_t = \left\| h_{t,(w-(p-1))\ldots w} \right\|_2 .$$

The index o, of the optimal column c_o, can be computed by finding the value of t in with the lowest residual r_t.

$$o = \arg\min_t (r_t) \quad , \quad t = 1 \ldots (m+n-2d).$$

8.1.2 Optimisations

Section 8.1.1 gave an overview of the optimal column calculation from a high level, considering the overall computations that must be completed. This section will discuss the areas of this computation in which optimisations can be made to reduce unnecessary computations.

The main optimisation that can be made to this algorithm is by reducing the portion of the product of $Q_{p,t}^T$ and $c_{p,t}$ that must be computed. It was noted that only the norm of the final p entries of the vector h_t are required to compute the residual.

In the matrix product shown below the blue highlighted area of h_t represents the values that are needed in order to compute the residual for this value of k. The green highlighted area represents the values of $Q_{p,t}^T$ that are required to compute these values.

$$h_t = Q_{p,t}^T c_{p,t},$$

$$\begin{bmatrix} h_{p,1} \\ h_{p,2} \\ \vdots \\ h_{p,m-(p-1)} \\ \vdots \\ h_{p,m} \end{bmatrix} = \begin{bmatrix} q_{1,1} & q_{2,1} & \cdots & q_{m,1} \\ q_{1,2} & q_{2,2} & \cdots & q_{m,2} \\ \vdots & \vdots & \vdots & \vdots \\ q_{1,m-(p-1)} & q_{2,m-(p-1)} & \cdots & q_{m,m-(p-1)} \\ \vdots & \vdots & \vdots & \vdots \\ q_{1,m} & q_{2,m} & \cdots & q_{m,m} \end{bmatrix} \begin{bmatrix} c_{t,1} \\ c_{t,2} \\ \vdots \\ \vdots \\ \vdots \\ c_{t,m} \end{bmatrix}.$$

By removing the non-highlighted area of $Q^T_{p,t}$, a significant part of each matrix product can be avoided. The new equation is shown below. The reduced forms of h_t and $Q_{p,t}$ used for this computation will be henceforth known as $\check{h}_t$ and $\check{Q}_t$. And thus the computation of $\check{h}_t$ is

$$\check{h}_t = \check{Q}^T_{p,t} c_{p,t},$$

$$\begin{bmatrix} h_{p,m-p} \\ \vdots \\ h_{p,m} \end{bmatrix} = \begin{bmatrix} q_{1,m-p} & q_{2,m-p} & \cdots & q_{m,m-p} \\ \vdots & \vdots & \vdots & \vdots \\ q_{1,m} & q_{2,m} & \cdots & q_{m,m} \end{bmatrix} \begin{bmatrix} c_{t,1} \\ c_{t,2} \\ \vdots \\ \vdots \\ \vdots \\ c_{t,m} \end{bmatrix}.$$

Therefore column o is computed with

$$o = \arg\min_t (\|\check{h}_t\|_2) \quad , \quad t = 1 \ldots (m+n-2p).$$

While only the last p columns of each matrix $Q_{p,t}$ are required for the computation of the residual, the full update must still be computed. During the QR update the rotations are applied to the columns of the orthogonal matrix Q, on two columns at a time. Sequential rotations are performed on overlapping pairs of columns. This means that, even if only the last p columns are needed from the updated orthogonal matrix, the full update must still be computed.

The subresultant matrix $S_t(f,g)$ is of dimension $(w+p-1)\times w$, and its factors Q_t and R_t are of dimensions $(w+p-1)\times(w+p-1)$ and $(w+p-1)\times w$ respectively. Take, for example, the matrix $S_3(f,g)$ constructed from the polynomials $f(x)$ and $g(x)$, which are of degree 6, and have a GCD of degree $p=3$. This subresultant matrix is split into its factors, Q_3 and R_3.

$$S_3(f,g) = \begin{bmatrix} f_0 & 0 & 0 & 0 & g_0 & 0 & 0 & 0 \\ f_1 & f_0 & 0 & 0 & g_1 & g_0 & 0 & 0 \\ f_2 & f_1 & f_0 & 0 & g_2 & g_1 & g_0 & 0 \\ f_3 & f_2 & f_1 & f_0 & g_3 & g_2 & g_1 & g_0 \\ f_4 & f_3 & f_2 & f_1 & g_4 & g_3 & g_2 & g_1 \\ f_5 & f_4 & f_3 & f_2 & g_5 & g_4 & g_3 & g_2 \\ f_6 & f_5 & f_4 & f_3 & g_6 & g_5 & g_4 & g_3 \\ 0 & f_6 & f_5 & f_4 & 0 & g_6 & g_5 & g_4 \\ 0 & 0 & f_6 & f_5 & 0 & 0 & g_6 & g_5 \\ 0 & 0 & 0 & f_6 & 0 & 0 & 0 & g_6 \end{bmatrix},$$

$$
Q_3 = \begin{bmatrix}
q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} & q_{1,5} & q_{1,6} & q_{1,7} & q_{1,8} & q_{1,9} & q_{1,10} \\
q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} & q_{2,5} & q_{2,6} & q_{2,7} & q_{2,8} & q_{2,9} & q_{2,10} \\
q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} & q_{3,5} & q_{3,6} & q_{3,7} & q_{3,8} & q_{3,9} & q_{3,10} \\
q_{4,1} & q_{4,2} & q_{4,3} & q_{4,4} & q_{4,5} & q_{4,6} & q_{4,7} & q_{4,8} & q_{4,9} & q_{4,10} \\
q_{5,1} & q_{5,2} & q_{5,3} & q_{5,4} & q_{5,5} & q_{5,6} & q_{5,7} & q_{5,8} & q_{5,9} & q_{5,10} \\
q_{6,1} & q_{6,2} & q_{6,3} & q_{6,4} & q_{6,5} & q_{6,6} & q_{6,7} & q_{6,8} & q_{6,9} & q_{6,10} \\
q_{7,1} & q_{7,2} & q_{7,3} & q_{7,4} & q_{7,5} & q_{7,6} & q_{7,7} & q_{7,8} & q_{7,9} & q_{7,10} \\
q_{8,1} & q_{8,2} & q_{8,3} & q_{8,4} & q_{8,5} & q_{8,6} & q_{8,7} & q_{8,8} & q_{8,9} & q_{8,10} \\
q_{9,1} & q_{9,2} & q_{9,3} & q_{9,4} & q_{9,5} & q_{9,6} & q_{9,7} & q_{9,8} & q_{9,9} & q_{9,10} \\
q_{10,1} & q_{10,2} & q_{10,3} & q_{10,4} & q_{10,5} & q_{10,6} & q_{10,7} & q_{10,8} & q_{10,9} & q_{10,10}
\end{bmatrix},
$$

$$
R_3 = \begin{bmatrix}
r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & r_{1,5} & r_{1,6} & r_{1,7} & r_{1,8} \\
0 & r_{2,2} & r_{2,3} & r_{2,4} & r_{2,5} & r_{2,6} & r_{2,7} & r_{2,8} \\
0 & 0 & r_{3,3} & r_{3,4} & r_{3,5} & r_{3,6} & r_{3,7} & r_{3,8} \\
0 & 0 & 0 & r_{4,4} & r_{4,5} & r_{4,6} & r_{4,7} & r_{4,8} \\
0 & 0 & 0 & 0 & r_{5,5} & r_{5,6} & r_{5,7} & r_{5,8} \\
0 & 0 & 0 & 0 & 0 & r_{6,6} & r_{6,7} & r_{6,8} \\
0 & 0 & 0 & 0 & 0 & 0 & r_{7,7} & r_{7,8} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & r_{8,8} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}.
$$

Note that the last two rows of R_3 are zero. This means that no QR updates will occur on these rows to compute the updated decomposition. The last rows on which a Givens rotation must be performed to are $w-1$ and w, in this case these are rows seven and eight, introducing the final zero on row eight.

$$
R_{3,t} = \begin{bmatrix}
r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & r_{1,5} & r_{1,6} & r_{1,7} \\
0 & r_{2,2} & r_{2,3} & r_{2,4} & r_{2,5} & r_{2,6} & r_{2,7} \\
0 & 0 & r_{3,3} & r_{3,4} & r_{3,5} & r_{3,6} & r_{3,7} \\
0 & 0 & 0 & r_{4,4} & r_{4,5} & r_{4,6} & r_{4,7} \\
0 & 0 & 0 & 0 & r_{5,5} & r_{5,6} & r_{5,7} \\
0 & 0 & 0 & 0 & 0 & r_{6,6} & r_{6,7} \\
0 & 0 & 0 & 0 & 0 & 0 & r_{7,7} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}.
$$

When computing the updated orthogonal matrix, in this case $Q_{p,t}$, the transpose of the Givens matrix for each rotation will be applied to columns of Q_p with the same indices

of the rows the rotation was applied to in R_p. In this case the rotation that was applied to rows seven and eight in R_3 will be applied to columns seven and eight of Q_3 to compute $Q_{3,t}$.

This means that, when $t = 1$, and the first column of S_3 is deleted, giving $S_{3,1}$, and the factor $Q_{3,1}$, the first 8 columns of $Q_{3,1}$ will have changed from the values in Q_3.

$$Q_{3,1} = \begin{bmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} & q_{1,5} & q_{1,6} & q_{1,7} & q_{1,8} & q_{1,9} & q_{1,10} \\ q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} & q_{2,5} & q_{2,6} & q_{2,7} & q_{2,8} & q_{2,9} & q_{2,10} \\ q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} & q_{3,5} & q_{3,6} & q_{3,7} & q_{3,8} & q_{3,9} & q_{3,10} \\ q_{4,1} & q_{4,2} & q_{4,3} & q_{4,4} & q_{4,5} & q_{4,6} & q_{4,7} & q_{4,8} & q_{4,9} & q_{4,10} \\ q_{5,1} & q_{5,2} & q_{5,3} & q_{5,4} & q_{5,5} & q_{5,6} & q_{5,7} & q_{5,8} & q_{5,9} & q_{5,10} \\ q_{6,1} & q_{6,2} & q_{6,3} & q_{6,4} & q_{6,5} & q_{6,6} & q_{6,7} & q_{6,8} & q_{6,9} & q_{6,10} \\ q_{7,1} & q_{7,2} & q_{7,3} & q_{7,4} & q_{7,5} & q_{7,6} & q_{7,7} & q_{7,8} & q_{7,9} & q_{7,10} \\ q_{8,1} & q_{8,2} & q_{8,3} & q_{8,4} & q_{8,5} & q_{8,6} & q_{8,7} & q_{8,8} & q_{8,9} & q_{8,10} \\ q_{9,1} & q_{9,2} & q_{9,3} & q_{9,4} & q_{9,5} & q_{9,6} & q_{9,7} & q_{9,8} & q_{9,9} & q_{9,10} \\ q_{10,1} & q_{10,2} & q_{10,3} & q_{10,4} & q_{10,5} & q_{10,6} & q_{10,7} & q_{10,8} & q_{10,9} & q_{10,10} \end{bmatrix}.$$

In the above matrix the values boxed in red will have changed from the matrix Q_3, while the entries boxed in blue are those required for the computation of the residual. Note that only one column exists in the overlap between the two boxed areas, that being column eight in this case. More generally the index of this column is w, the second column of the last rotation in the update algorithm. The same is true for all values of t, with column w being the only column necessary for the computation of the residual that will have changed in the updated matrices.

As this is the only column that will have changed and is necessary for the computation of the residual, only a single column needs to be stored for each updated orthogonal matrix. Entries for the columns $(w+1)\ldots(w+p-1)$ can be extracted from the original orthogonal matrix Q_3. Additionally, only the second row of each rotation performed on Q_3 is required to compute the next rotation, and the required column is computed in the second column of the last rotation. Thus, the computations for the first column of each rotation are unnecessary. Through avoiding this computation the number of operations required to compute the necessary column is almost halved.

The product of the matrix $Q_{3,t}$ and the column c_t can then be computed in a series of warp shuffle row sums, computing each entry of the residual vector in sequence. These entries can be squared and added to a running sum to compute the squared norm of the residual. As the aim is to find the minimum of the norms, the squared norm will suffice, and the square root is not required.

8.1.3 Example

This section will give an example of the computation with the optimisations described in Section 8.1.2.

The normalised vectors f and g on which this example will be based are shown below.

The polynomials, of which these vectors represent the coefficients, have degrees of $m = n = 4$, and have an AGCD of degree $p = 2$.

$$f = [0.3156, 1.1968, 2.7722, 2.4429, 0.3910],$$
$$g = [0.3717, 1.2175, 2.7722, 1.8053, 0.3156].$$

The Sylvester matrix $S_p(f, g)$ is constructed,

$$S_p(f,g) = \begin{bmatrix} 0.3156 & & & 0.3717 & & \\ 1.1968 & 0.3156 & & 1.2175 & 0.3717 & \\ 2.7722 & 1.1968 & 0.3156 & 2.7722 & 1.2175 & 0.3717 \\ 2.4429 & 2.7722 & 1.1968 & 1.8053 & 2.7722 & 1.2175 \\ 0.3910 & 2.4429 & 2.7722 & 0.3156 & 1.8053 & 2.7722 \\ & 0.3910 & 2.4429 & & 0.3156 & 1.8053 \\ & & 0.3910 & & & 0.3156 \end{bmatrix},$$

and the QR decomposition of this matrix is computed,

$$Q_p = \begin{bmatrix} -0.0806 & -0.0899 & -0.0756 & 0.1273 & 0.0089 & -0.0613 & 0.9796 \\ -0.3056 & -0.2203 & -0.1243 & 0.1392 & 0.0132 & -0.8980 & -0.1293 \\ -0.7079 & -0.3321 & -0.1907 & 0.3913 & -0.0064 & 0.4277 & -0.1275 \\ -0.6238 & 0.3646 & 0.3012 & -0.6161 & 0.0055 & -0.0263 & 0.0838 \\ -0.0998 & 0.8233 & -0.0880 & 0.5460 & -0.0361 & -0.0696 & -0.0144 \\ 0 & 0.1496 & -0.9016 & -0.3533 & 0.1960 & 0.0376 & -0.0094 \\ 0 & 0 & -0.1765 & -0.0938 & -0.9798 & -0.0055 & 0.0071 \end{bmatrix},$$

$$R_p = \begin{bmatrix} -3.9163 & -2.9167 & -1.2467 & -3.5219 & -2.8848 & -1.2993 \\ 0 & 2.6135 & 2.9793 & -0.3042 & 2.0580 & 2.8728 \\ 0 & 0 & -2.2151 & -0.1922 & 0.1132 & -1.6314 \\ 0 & 0 & 0 & 0.3617 & -0.3056 & 0.2416 \\ 0 & 0 & 0 & 0 & 0.0089 & -0.0513 \\ 0 & 0 & 0 & 0 & 0 & -0.0001 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The updated factors of $S_p(f, g)$ must now be computed, after each column is removed. As was discussed in Section 8.1.2 only the last column that is changed in $Q_{p,t}$ must be retained.

The first step in computing the update for $Q_{p,1}$ is to remove the first column from R_p to give $\tilde{R}_{p,1}$.

$$\tilde{R}_{p,1} = \begin{bmatrix} -2.9167 & -1.2467 & -3.5219 & -2.8848 & -1.2993 \\ 2.6135 & 2.9793 & -0.3042 & 2.0580 & 2.8728 \\ 0 & -2.2151 & -0.1922 & 0.1132 & -1.6314 \\ 0 & 0 & 0.3617 & -0.3056 & 0.2416 \\ 0 & 0 & 0 & 0.0089 & -0.0513 \\ 0 & 0 & 0 & 0 & -0.0001 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The first Givens matrix G_1 is computed from the first elements in the first two columns of $\tilde{R}_{p,1}$. The process for this is discussed in Chapter 3.

$$G_1 = \begin{bmatrix} -0.7448 & 0.6673 \\ -0.6673 & -0.7448 \end{bmatrix}.$$

This rotation is then applied to $\tilde{R}_{p,1}$. As was discussed in the previous section, only the result on the bottom row of the rotation on the upper triangular factor must be computed.

The updated row of $\tilde{R}_{p,1}$ is computed, ignoring the top row of the matrix product. The entries of the result of this matrix product that do not need to be computed are denoted here as $\bullet$.

$$\begin{bmatrix} -0.7448 & 0.6673 \\ -0.6673 & -0.7448 \end{bmatrix} \begin{bmatrix} -2.9167 & -1.2467 & -3.5219 & -2.8848 & -1.2993 \\ 2.6135 & 2.9793 & -0.3042 & 2.0580 & 2.8728 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & -1.3869 & 2.5768 & 0.3924 & -1.2725 \end{bmatrix}.$$

The result of this matrix product and the next row of $\tilde{R}_{p,1}$, in this case row 3, are used to compute the next row of the upper triangular factor. Firstly the Givens matrix G_2 is computed from the first non-zero elements in these rows.

$$G_2 = \begin{bmatrix} -0.5307 & -0.8476 \\ 0.8476 & -0.5307 \end{bmatrix}.$$

This matrix is then used in the computation of the next row, again ignoring the upper row of the matrix product.

$$\begin{bmatrix} -0.5307 & -0.8476 \\ 0.8476 & -0.5307 \end{bmatrix} \begin{bmatrix} -1.3869 & 2.5768 & 0.3924 & -1.2725 \\ -2.2151 & -0.1922 & 0.1132 & -1.6314 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & 2.2860 & 0.2726 & -0.2127 \end{bmatrix}.$$

This process repeats, resulting in the next Givens matrix G_3.

$$G_3 = \begin{bmatrix} 0.9877 & 0.1563 \\ -0.1563 & 0.9877 \end{bmatrix}.$$

And the next row from the matrix product,

$$\begin{bmatrix} 0.9877 & 0.1563 \\ -0.1563 & 0.9877 \end{bmatrix} \begin{bmatrix} 2.2860 & 0.2726 & -0.2127 \\ 0.3617 & -0.3056 & 0.2416 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet \\ 0 & -0.3444 & 0.2718 \end{bmatrix}.$$

The process repeats again resulting in G_4.

$$G_4 = \begin{bmatrix} -0.9997 & 0.0257 \\ -0.0257 & -0.9997 \end{bmatrix}.$$

$$\begin{bmatrix} -0.9997 & 0.0257 \\ -0.0257 & -0.9997 \end{bmatrix} \begin{bmatrix} -0.3444 & 0.2718 \\ 0.0089 & -0.0513 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet \\ 0 & 0.0443 \end{bmatrix}.$$

As was noted in Section 8.1, the upper triangular factor is not needed to compute the residual. Therefore, in the last iteration, only the Givens matrix, and not the matrix product, needs to be computed.

$$G_5 = \begin{bmatrix} 1.0000 & -0.0015 \\ 0.0015 & 1.0000 \end{bmatrix}.$$

With all of the Givens matrices now computed, work can begin on applying the Givens rotations to the columns of Q_p to compute the necessary column of the updated matrix, which will be the first column of $\check{Q}_{p,1}$. The first of these rotations is computed by finding the matrix product of the first two columns of Q_p with the transpose of the matrix G_1. As with the computations for the upper triangular matrices, only half of the result of the matrix product is required. In this case only the second column of the result is computed.

$$\begin{bmatrix} -0.0806 & -0.0899 \\ -0.3056 & -0.2203 \\ -0.7079 & -0.3321 \\ -0.6238 & 0.3646 \\ -0.0998 & 0.8233 \\ 0 & 0.1496 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -0.7448 & -0.6673 \\ 0.6673 & -0.7448 \end{bmatrix} = \begin{bmatrix} \bullet & 0.1207 \\ \bullet & 0.3680 \\ \bullet & 0.7197 \\ \bullet & 0.1447 \\ \bullet & -0.5465 \\ \bullet & -0.1114 \\ \bullet & 0 \end{bmatrix}.$$

The next product is computed from the matrix consisting of the computed column in the previous iteration and the next column of Q_p, in this case this is the third column.

$$\begin{bmatrix} 0.1425 & 0.1273 \\ 0.3779 & 0.1392 \\ 0.7112 & 0.3913 \\ -0.0371 & -0.6161 \\ -0.4165 & 0.5460 \\ 0.3840 & -0.3533 \\ 0.0937 & -0.0938 \end{bmatrix} \begin{bmatrix} -0.5307 & 0.8476 \\ -0.8476 & -0.5307 \end{bmatrix} = \begin{bmatrix} \bullet & 0.1425 \\ \bullet & 0.3779 \\ \bullet & 0.7112 \\ \bullet & -0.0371 \\ \bullet & -0.4165 \\ \bullet & 0.3840 \\ \bullet & 0.0937 \end{bmatrix}.$$

This pattern continues until all Givens rotations have been applied.

$$\begin{bmatrix} 0.1207 & -0.0756 \\ 0.3680 & -0.1243 \\ 0.7197 & -0.1907 \\ 0.1447 & 0.3012 \\ -0.5465 & -0.0880 \\ -0.1114 & -0.9016 \\ 0 & -0.1765 \end{bmatrix} \begin{bmatrix} 0.9877 & -0.1563 \\ 0.1563 & 0.9877 \end{bmatrix} = \begin{bmatrix} \bullet & 0.1035 \\ \bullet & 0.0784 \\ \bullet & 0.2754 \\ \bullet & -0.6027 \\ \bullet & 0.6044 \\ \bullet & -0.4090 \\ \bullet & -0.1073 \end{bmatrix},$$

$$\begin{bmatrix} 0.1035 & 0.0089 \\ 0.0784 & 0.0132 \\ 0.2754 & -0.0064 \\ -0.6027 & 0.0055 \\ 0.6044 & -0.0361 \\ -0.4090 & 0.1960 \\ -0.1073 & -0.9798 \end{bmatrix} \begin{bmatrix} -0.9997 & -0.0257 \\ 0.0257 & -0.9997 \end{bmatrix} = \begin{bmatrix} \bullet & -0.0116 \\ \bullet & -0.0152 \\ \bullet & -0.0006 \\ \bullet & 0.0100 \\ \bullet & 0.0206 \\ \bullet & -0.1854 \\ \bullet & 0.9822 \end{bmatrix},$$

In the final computation the necessary column of $Q_{p,1}$ is computed, which will be the first column of $\check{Q}_{p,1}$.

$$\begin{bmatrix} -0.0116 & -0.0613 \\ -0.0152 & -0.8980 \\ -0.0006 & 0.4277 \\ 0.0100 & -0.0263 \\ 0.0206 & -0.0696 \\ -0.1854 & 0.0376 \\ 0.9822 & -0.0055 \end{bmatrix} \begin{bmatrix} 1.0000 & -0.0015 \\ 0.0015 & 1.0000 \end{bmatrix} = \begin{bmatrix} \bullet & -0.0613 \\ \bullet & -0.8980 \\ \bullet & 0.4277 \\ \bullet & -0.0263 \\ \bullet & -0.0696 \\ \bullet & 0.0373 \\ \bullet & -0.0040 \end{bmatrix}.$$

This column will be known as q_t.

$$q_t = \begin{bmatrix} -0.0613 \\ -0.8980 \\ 0.4277 \\ -0.0263 \\ -0.0696 \\ 00373 \\ -0.0040 \end{bmatrix}.$$

The full matrix $\check{Q}_{p,t}$ is constructed using this column vector and the last $p-1$ columns of Q_p. In this case this submatrix of Q_p only consists of a single column.

$$\check{Q}_{p,t} = \begin{bmatrix} -0.0613 & 0.9796 \\ -0.8980 & -0.1293 \\ 0.4277 & -0.1275 \\ -0.0263 & 0.0838 \\ -0.0696 & -0.0144 \\ 0.0373 & -0.0094 \\ -0.0040 & 0.0071 \end{bmatrix}.$$

The product of the transpose of $\check{Q}_{p,t}$ and column c_t is computed, giving the vector $\check{h}_1$.

$$\begin{aligned} \check{h}_1 &= \begin{bmatrix} -0.0613 & -0.8980 & 0.4277 & -0.0263 & -0.0696 & 0.0373 & -0.0040 \\ 0.9796 & -0.1293 & -0.1275 & 0.0838 & -0.0144 & -0.0094 & 0.0071 \end{bmatrix} \begin{bmatrix} -0.0806 \\ -0.3056 \\ -0.7079 \\ -0.6238 \\ -0.0998 \\ 0 \\ 0 \end{bmatrix}, \\ &= \begin{bmatrix} 0.1343 \times 10^{-4} \\ 0 \end{bmatrix}. \end{aligned}$$

The norm of h_1 is computed, giving the residual r_t,

$$\begin{aligned} r_t &= \|h_t\|_2 , \\ r_1 &= 0.1343 \times 10^{-4}. \end{aligned}$$

This process is repeated for every column in S_p, to give the final values of the residuals r_t where $t = 1 \ldots (m + n - d)$. In this case this results in 6 residuals.

$$
\begin{aligned}
r_1 &= 0.1343 \times 10^{-4}, \\
r_2 &= 0.1573 \times 10^{-4}, \\
r_3 &= 0.8272 \times 10^{-4}, \\
r_4 &= 0.1582 \times 10^{-4}, \\
r_5 &= 0.1154 \times 10^{-4}, \\
r_6 &= 0.6676 \times 10^{-4}.
\end{aligned}
$$

The optimal column index o is equal to the value of t with the minimum residual. In this case this r_5 is the minimum residual, therefore $o = 5$.

8.2 Implementation

With these optimisations to the algorithm it is still important to devise an efficient implementation that makes use of the GPU. The implementation presented in this section takes the form of three kernels. Some of the techniques used and developed in previous methods, such as the load balancing, were used again here. Inspiration was also taken from the low memory structures described in Chapter 7.

This implementation splits the algorithm into three kernels. To keep the kernels simple it was decided to split the QR update into two separate kernels. The first of these kernels processes the computation of the Givens matrices through the computation of the upper triangular matrix $R_{p,t}$ after the QR updates. While the matrix $R_{p,t}$ is unnecessary for the computation of the residuals, these computations are necessary in order to compute all of the Givens matrices required for the computation of the updated orthogonal matrices $Q_{p,t}$, as was shown in Section 8.1.3.

The second kernel computes the necessary column from the orthogonal matrix $Q_{p,t}$, with the entries of the Givens matrices computed in the previous kernel. As discussed in Section 8.1.1 this is only required to output a single column of the updated orthogonal matrix, as the remainder of these columns can be extracted from the original orthogonal matrix Q_p.

The final kernel computes the residual of the product of $Q_{p,t}$ and the column c_t. Each entry of the vector is computed in sequence, the result is squared, and added to a running sum to compute the squared residual.

8.2.1 Computing the Givens Matrices

The first kernel, the pseudocode of which is shown in Listing 11, involves the computation of the Givens matrices that are required to compute the updated orthogonal factor. To aid balance this was split into a separate kernel from the computation of the orthogonal matrix. The computation of the Givens matrices is performed in a very similar way to

```
1  // R1 is the original upper triangular matrix R1 stored in triangular form
2  // rWidth is the width of the original R1 matrix
3  // givensVals is a two dimensional array of structures for storing the Givens
4  //      values necessary for all updates arranged by value of k then by row.
5  // GV creates a structure to store the entries of the givens matrix
6  // allWipR is a two dimensional array in which work in progress rows of the
7  //      each updated matrix are stored
8  // getTriMatrixIndex(width, row) is a function to get the index of a specified
9  //      row of a matrix with the specified width
10
11 gpu_parallel_for 0 to blockWidth
12     t ← blockID
13
14     for i ← t to rWidth-1
15         iWidth ← rWidth-1-i
16         totalWork ← iWidth
17         batches ← ceil(totalWork / blockWidth)
18         for batch ← 0 to batches
19             wu ← batch*blockWidth + threadID
20             if (wu ≤ i)
21                 row ← i-t
22
23                 if (row=0)
24                     x ← R1[getTriMatrixIndex(rWidth, row+t)+2]
25                 else
26                     x ← allWipR[t][1]
27                 y ← R1[getTriMatrixIndex(rWidth, row+t+1)]
28
29                 na ← sqrt(x*x+y*y)
30
31                 c ← x/na
32                 s ← y/na
33                 givensVals[t][row] ← GV(c,s)
34
35             synchroniseThreads
36
37             col ← wu
38             if (wu < totalWork) && (i < rWidth-1)
39                 row ← i - t
40                 v ← givensVals[t][row]
41
42                 if (row = 0)
43                     val1 ← R1[getTriMatrixIndex(rWidth, row+t) + col + 1]
44                 else
45                     val1 ← allWipR[t][col + 1]
46                 val2 ← R1[getTriMatrixIndex(rWidth, row+1+t) + col]
47
48             synchroniseThreads
49
50             if (wu < totalWork) && (i < rWidth-1)
51                 allWipR[t][col] ← val1 * -v.s + val2 * v.c
52
53             synchroniseThreads
```

Listing 11: Pseudocode showing the computation of the entries of the necessary Givens matrices

the method of computing the upper triangular matrix in Chapter 7. The computation is batched, with computations on the same row of $R_{p,t}$ happening in the same iteration, which can be seen on Lines 14 to 18. Each column deletion is computed in a separate block, with the rows of the update computed in sequence in each block. This leads to an unbalanced workload between blocks. However, to follow the scheme used in the implementations discussed in Chapters 6 and 7 would have meant that all of the computation would occur in a single block. This was found to limit the utilisation of the device by a greater amount than the unbalanced workload.

Each batch begins with the computation of the Givens entries on Lines 19 to 35. This happens in much the same way as in the implementation in Chapter 7. Once these are computed, the program moves on to applying the rotation to the affected rows, which occurs on Lines 37 to 51. Unlike the computation of the upper triangular matrix in Chapter 7, only a single row is required to be stored by this kernel. This row will be used to compute the entries of the Givens matrices in the next iteration, as the entries of the upper triangular matrix are not required beyond the computation of the entries of the Givens matrices. This means that the upper row of each matrix product does not need to be computed, and the lower row of each product does not need to be retained after the next row has been computed. This nearly halves the number of operations required for the matrix products, and drastically reduces the amount of memory needed to compute the entries of the Givens matrices.

8.2.2 Computing Column w of the Orthogonal Matrices

The second kernel, the pseudocode of which is shown in Listing 12, uses the entries of the Givens matrices computed in the previous kernel, and applies the rotations to the orthogonal matrix Q_p to compute the updated orthogonal matrices $Q_{p,t}$. As was discussed in Section 8.1.2, only the column at index w is required to be computed in this implementation.

Similarly to the previous kernel, the computation for each column deletion occurs in a separate block. This leads to a balanced workload within the block, and thus a minimal number of idle threads. The kernel batches the computation on Lines 17 to 20, finds the relevant Q matrices on Lines 21 to 30, and applies the relevant rotation from the Givens matrix entries computed in the previous kernel on Line 32. Also similarly to the previous kernel, only the second column of each matrix product needs to be computed. This is because the second column will be used in the next rotation, while the first column is not needed for any further computations. In the final iteration the entries of the second column are retained, as this is column w that is required for the computation of the residuals.

It is important to note that the upper triangular matrix used here is input into the algorithm in column major form. This is because the computations occur across the columns of the orthogonal matrices, and thus storing in column major form allows the memory access to be coalesced, while the upper triangular matrices are stored in row major form for the same reason. If these matrices are in different forms than is required

```
1  // Q1 is the original orthogonal matrix
2  // QWidth is the width of Q1
3  // allQCols is a 2 dimensional array that stores work in progress columns of the
4  //      all the updated orthogonal matrices, the final values in these arrays
5  //      will represent the necessary column for the residual computation
6  // degree is the previously computed degree of the AGCD
7  // allGivensVals is a 2 dimensional array storing all the Givens values needed
8  //      to compute all the updated rows of Q
9
10 gpu_parallel_for 0 to blockWidth
11     t ← blockID
12
13     givensVals ← allGivensVals[t];
14     qCol ← allQCols[t];
15
16     for col ← t to qWidth - degree
17         batches ← ceil(qWidth/blockWidth)
18         givensIndex ← col - t
19         for batch ← 0 to batches
20             wu ← blockWidth * batch + threadID
21             if (wu < qWidth)
22                 row ← wu
23                 v ← givensVals[givensIndex]
24
25                 if (col ≤ t)
26                     val1 ← Q1[qWidth*col + row]
27                 else
28                     val1 ← qCol[row]
29
30                 val2 ← Q1[qWidth * (col+1) + row]
31
32                 qCol[row] ← val1 * -v.s + val2 * v.c
33         synchroniseThreads
```

Listing 12: Pseudocode showing the computation of the necessary columns of the updated Q matrices

before being input this just requires a transpose operation. The overhead of performing these operations is negligible compared to the runtime reduction from the efficient memory access.

8.2.3 Computing the Residuals

The pseudocode for the final kernel is shown in Listing 13. This kernel computes both the matrix product of $\check{Q}_{p,t}$ with $\check{h}_t$, and the squared norms of the results of these products. The kernel returns the residual r_t for each value of t. Similarly to the other kernels, each block works on the computation for an individual value of t.

The outer loop, initiated on Line 21, iterates over the columns of each orthogonal matrix, and Lines 25 to 31 show this computation being batched.

As was discussed previously in this chapter, only a single column will have been stored for each orthogonal matrix. This means that in the first iteration each block will use column w, computed for $Q_{p,t}$, and all successive iterations will use columns retrieved from the original orthogonal matrix Q_p. The only exception is for the last column w, for which no update needs to be computed, and thus all columns are retrieved from the orthogonal matrix Q_p. This can be seen on Lines 33 to 38.

The column c_t is retrieved from the subresultant matrix S_p on Line 39. The subresultant matrix, much like the orthogonal matrix, is input in column major format to ensure memory access is coalesced. The products of the relevant entries of these columns are summed, using a warp shuffle prefix sum algorithm on Lines 41 to 45. This prefix sum computes the sum of the product of a row from $\check{Q}_{p,t}^T$ and the column c_t, to give a final value for a single entry of the column vector $\check{h}_t$. The prefix sum is batched in case the columns have more entries than the maximum block size. The sum computed for the column vector $\check{h}_t$ is squared and added to the running sum for this column vector on Lines 48 to 52. Once all p iterations are complete the squared residuals will have been computed, and a minimum of these residuals can be found.

8.2.4 Finding the Minimum Residual

Once the squared residuals have been computed, they are copied back to host memory, where processing to find the minimum residual can occur on the CPU. Similarly to the final steps of the degree computation, it would be possible to compute the minimum on the GPU. However, minimal computation remains at this point, and only a small amount of data needs to be copied back to host memory. Due to this there is unlikely to be a significant speed improvement by completing this computation to the GPU.

8.2.5 GPU profiling

In this section the final profiling results of this implementation will be discussed. The pie chart in Figure 8.1 shows the amount of time spent in each of the kernels described in the preceding sections. It is clear that the computation of the entries of the Givens matrices

```
// Q1 is the original orthogonal matrix
// allQCols is a 2 dimensional array in which the necessary columns of the
//      orthogonal matrix required for the computation of the residual
//      are stored
// qWidth is the width of Q1
// degree is the previously computed degree of the AGCD
// rWidth is the width of R1
// Sp is the Sylvester matrix from which Q1 and R1 were computed
// residualSqSums is an array of the residuals computed from the final columns
//      of the updated Qs
// scanShared is the shared memory used for the prefix sum
// toAdd is a shared memory location where the running sum of all the prefix
//      sums is stored to allow them to be batched
// warpShufflePrefixSum(width, value, shared) is the warp shuffle prefix sum
//      algorithm taking in the width of the sum, a single value per thread,
//      and the shared memory used for the sum

gpu_parallel_for 0 to blockWidth
    int t ← blockID

    for col ← 0 to degree
        if (threadID=0) toAdd ← 0
        synchroniseThreads

        batches ← ceil(qWidth / blockWidth)
        for batch ← 0 to batches
            if (batches - batch = 1)
                width ← qWidth % blockWidth
            else
                width ← blockWidth
            wu ← batch * blockWidth + threadID

            if ((col=0) && (t<rWidth-1)) {
                qVal ← allQCols[t][wu]
            }
            else {
                qVal ← Q1[((qWidth - degree) + col) * qWidth + wu]
            }
            sVal ← Sp[t * qWidth + wu]

            val ← 0
            if (wu<qWidth)
                val ← kthCol[wu] * qCol[wu]

            blockPrefixSum(width, val, scanShared)

            if (wu = qWidth - 1)
                outVal ← toAdd[0] + scanShared[threadID]
                if (col=0)
                    residualSqSums[blockID] ← outVal * outVal
                else
                    residualSqSums[blockID]+= outVal * outVal

            synchroniseThreads
            if (threadID = blockWidth - 1) toAdd+= scanShared[blockWidth - 1]
            synchroniseThreads
```

Listing 13: Pseudocode showing the computation of residuals of the updated Q matrices

takes the majority of the execution time. This is expected, due to the unbalanced nature of the kernel, and the fact that both Givens matrices and the updated upper triangular factors must be computed, while the kernel for the computation of the updated orthogonal matrix uses the Givens matrices computed previously.

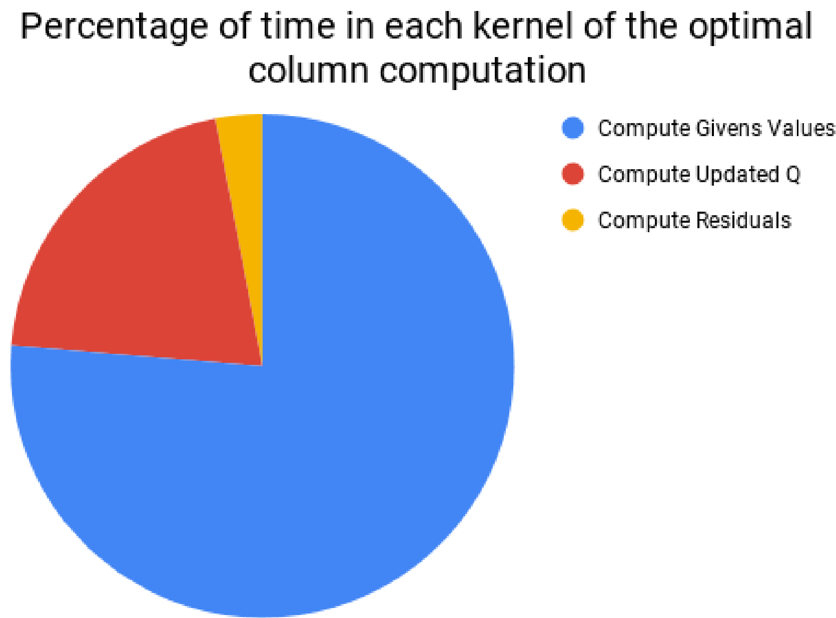


Figure 8.1: Pie chart showing the percentage of time spent in each kernel

The kernel for the computation of the entries of the Givens matrices, as discussed, takes a significant amount of time to compute relative to the other kernels. This was to be expected, due to the increased workload compared to the computation of the updating of orthogonal matrices, and the unbalanced workload. The main limiting factor, however, is the occupancy. An occupancy of only 50% was achieved. The limiting factor in this case was, similarly to the kernels in the previous chapters, the register usage. The kernel uses 62 registers per thread. This amount means that it is not worth limiting the number of threads in the block, as this would mean a significant number of variables would have to be stored in memory, which would impact performance. The kernel also sees low compute throughput. This is unfortunately due to the unbalanced nature of the algorithm. Further research would be required here to find a way to improve balance in this algorithm.

The computation of the updated orthogonal matrices is a well balanced kernel. This, combined with the register count of only 32 per thread, allows the kernel to achieve an occupancy of 99.9%, with significantly higher compute throughput compared to that of the computation of the Givens matrices. This implies that this kernel is well optimised, and not needing any further consideration regarding its optimisation.

The final kernel achieves a high occupancy of 79%. Considering that this kernel makes up only a small amount of the overall runtime it was not considered a priority for further optimisation.

8.3 Results

Similarly to Chapters 6 and 7, the tests in this section cover two areas. Firstly the reliability is tested, where the code was assessed to ensure that it provided the exact same result as that of the serial implementation at every stage of the computation. Secondly the runtime was investigated, to test how significant the improvement over CPU serial and parallel implementations is. The optimisations described in Section 8.1.2 were also applied to the CPU implementations.

8.3.1 Reliability testing

In the degree computation the results provided by the algorithm could be tested against a known degree, and against the output of the serial algorithm. In this situation however, the optimal column is not known during the construction of the test data, and thus the algorithm cannot be tested against a prior known value. Instead the algorithm was tested against the original serial implementation, and a serial implementation that makes use of the optimisations described in Section 8.1.2, to ensure the same results are reached. The algorithm was inspected at every stage, the computation of the Givens matrix entries, the computation of the matrices $Q_{p,t}$, the computation of the product, and the final residuals.

The algorithm was thoroughly tested with polynomials of a wide range of degrees, and AGCDs of a wide range of degrees. The output from each kernel was examined. In all tested cases the algorithm provided the exact same result as both of the serial implementations. However, as will be seen in Chapter 9, when the degree of the AGCD is high, and the noise is also high, the computation of the coefficients is less reliable.

8.3.2 Runtime testing

The GPU implementation was compared against the CPU serial and parallel implementations to test its runtime, and the amount of speedup that was gained by parallelising this algorithm. All three implementations were timed from before the polynomials were normalised, until an optimal column index was computed. While the results in Chapters 6 and 7 included the overhead of the initialisation of the parallel pool in the times for the CPU parallel implementation, they were not included in this case. It is a reasonable assumption that if an optimal column is to be computed using a CPU parallel implementation, then the parallel pool would have already been initialised as part of the degree computation. It was therefore decided that it was unnecessary to include this overhead in the times shown here.

All results, as with the results shown in the previous chapters, were computed on a system with a 6 core Intel i7 6850k CPU, and an NVIDIA Titan V GPU. The CPU parallel implementation parallelised the computation of the residual for each column, including the column deletion, the matrix product and the computation of the norm, across all 6 cores of the CPU.

The tests for all algorithms considered degrees of the polynomials convolved with the AGCDs from 200 up to 2000, at increments of 200. The degree of the AGCD in each case was equal to roughly 10% of the convolved polynomial degree. The vectors tested were generated using the MATLAB `rand` command, and noise was added after the convolution. Ten sets of vectors of these degrees were generated in each test, and the time taken for each set was averaged to give the final runtimes as are presented here.

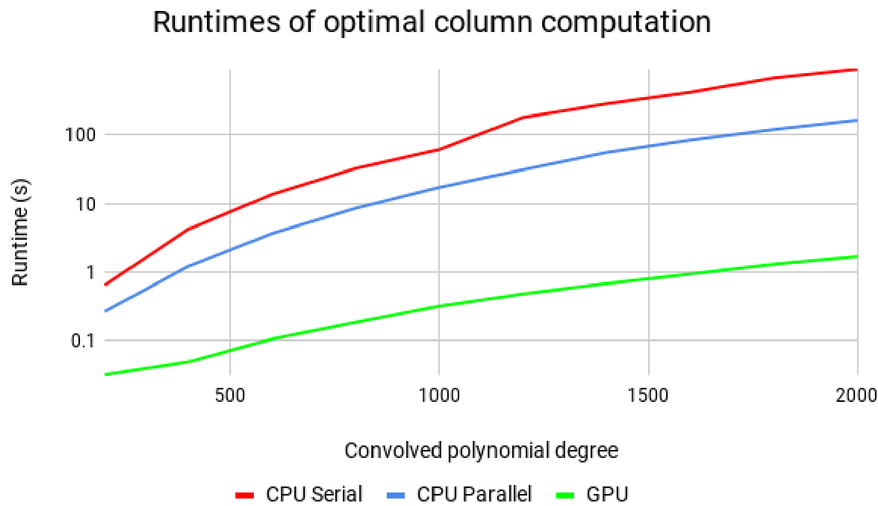


Figure 8.2: Runtimes of the GPU method and the CPU serial and parallel methods

Figure 8.2 shows the runtimes of the GPU implementation against that of the CPU serial and parallel implementations on a log scale. It is immediately clear that the GPU implementation is significantly faster than the CPU implementations, with runtimes staying between one and two orders of magnitude below even the CPU parallel implementation.

It is noticeable that the CPU serial implementation has a significant increase in runtime at a degree of 1200. Profiling of this algorithm did not reveal the reason behind this increase, and the CPU parallel implementation does not have such an increase. The CPU parallel runtimes before this point were between 2.41 and 3.81 times faster than the serial implementation, and after this point the runtimes were between 4.97 and 5.78 times faster than the serial implementation. Given the CPU on which the algorithm was tested has 6 physical cores, the runtimes for degrees between 1200 and 2000 are more in line with what was expected, and it is a reasonable assumption that the algorithm would continue to scale in this way. Before this point the lower than expected runtimes could potentially be explained by underlying optimisations within MATLAB for smaller computations.

The CPU parallel and GPU implementations, on the other hand, scale relatively consistently for increasing degrees, demonstrating the scalability of this algorithm. For degrees as large as 2000, the GPU implementation takes only 1.68 seconds, while the CPU implementations took 912.21 seconds and 163.85 seconds for the serial and parallel implementations respectively.

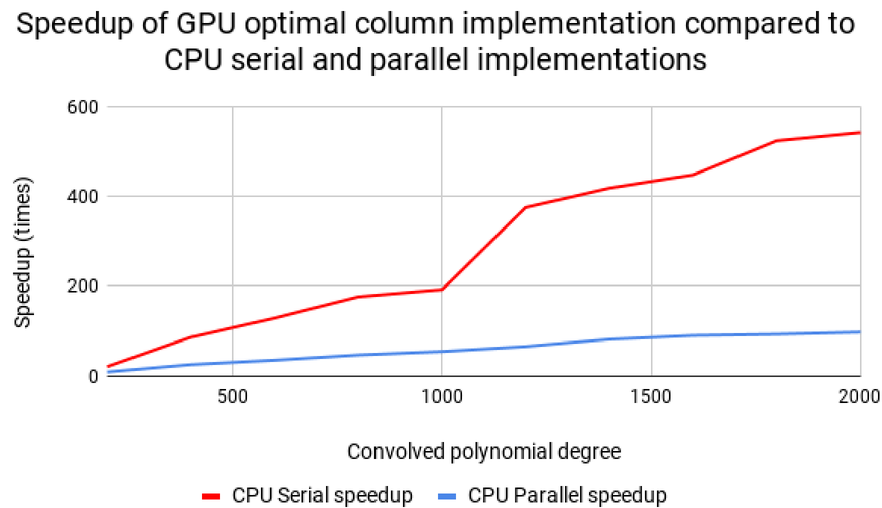


Figure 8.3: Speedup of the GPU method over those of the CPU serial and parallel methods

Figure 8.3 shows the speedup of the GPU implementation against the CPU implementations. The most obvious feature here is the large increase in the speedup against the CPU serial implementation at a degree of 1200, as was described during the discussion of the runtimes. The cause of this increase is unknown, as was discussed previously.

At the lower end of the convolved degrees the improvements over the CPU implementations were only modest. At a degree of 200 runtimes of 20.20 and 8.25 times faster were recorded for the CPU serial and parallel implementations respectively. This scales smoothly until the 1000 mark, as previously discussed, where the runtimes compared against the CPU serial and parallel implementations were 192.01 and 53.85 times faster respectively. After this point the CPU serial implementation starts to take longer, and the runtime improvement of the GPU implementation against the CPU serial implementation increases to 375.59 times faster at a degree of 1200. The rate of improvement then levels off again, and at the highest degree tested, 2000, the runtimes were 542.68 and 97.47 times faster than the CPU serial and parallel implementations respectively.

The most notable thing about these runtimes is that, without exception, higher degrees lead to a higher relative improvement for the GPU accelerated implementation. This suggests that the algorithm would continue to scale well, and provide even greater improvements with regards to runtime if the degree is increased further.

8.3.3 Testing on Limited Hardware

This implementation, similar to that in Chapter 7, does not use much of the GPU memory. This means that the algorithm would be able to be used on lower end hardware. To test this the algorithm was run on a system with a 4 core Intel i7 4790k CPU, and an NVIDIA GTX 780 GPU, with 3GB of GPU memory. This GPU was compared against the NVIDIA Titan V used in the other tests in Chapter 7.

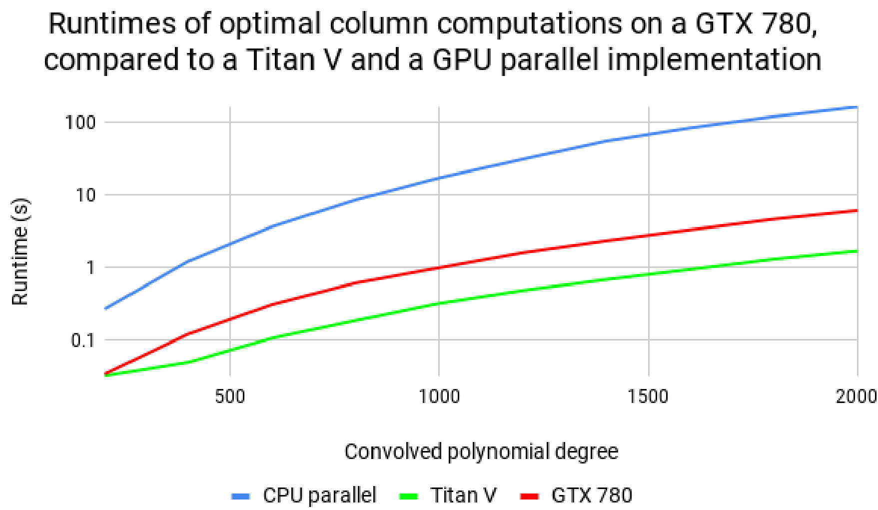


Figure 8.4: Runtime of the GTX 780 against the Titan V, as well as the CPU parallel implementation as a comparison

Figure 8.4 shows the runtimes of the implementation on these two GPUs, with the CPU parallel implementation as a comparison. The difference in runtime between the two GPUs is significantly less than the difference between the slower GPU, the GTX 780, and the CPU parallel implementation. The GPUs have similar runtimes at the lower end of the polynomial degrees, with runtimes of 0.033 and 0.031 seconds for the Titan V and the GTX 780 respectively. The low difference between these runtimes can be explained by the poor utilisation of the Titan V for a computation this small. As the degree increases the difference between the runtimes provided by the GPUs becomes more apparent. At a degree of 800 the difference starts to level off, and for degrees between 800 and 2000 the Titan V stays between 3.10 and 3.61 times faster than the GTX 780.

Despite this difference in runtimes, running the implementation on a GTX 780 is still significantly faster than the CPU parallel implementation. The runtime of the GPU implementation, even when implemented on a dated GPU, starts at 7.90 times faster than the CPU parallel implementation. This runtime continues to scale well, with the relative runtimes continuing to improve up to the maximum degree tested, where the GPU algorithm was 27.03 times faster. This implies, similar to the tests using the Titan V, that the algorithm would continue to scale well for all degrees.

8.4 Conclusion

This chapter demonstrated the acceleration of the computation of an optimal column of a subresultant matrix as part of a modified SNTLN method, in order to calculate the coefficients of an AGCD. The optimisations and acceleration discussed in this chapter significantly reduce the runtime of the computation of the coefficients, by concentrating on the most expensive part of this algorithm that required a novel solution. With the

optimisations and acceleration proposed in this chapter, and those from Chapters 6 and 7, the BID algorithm originally discussed in Chapter 4 has been significantly accelerated, and the scalability and low memory usage of this algorithm suggests that even larger images could be processed efficiently.

The next chapter will take the accelerated implementations presented in these chapters, and the improvements discussed in Chapter 4, and investigate the improvement that these have made, both in terms of runtime and reliability, to the original BID algorithm.

Chapter 9

Full Image Deconvolution and Results

Chapters 6, 7, and 8 have shown how aspects of the BID algorithm presented in Chapter 4 have been optimised by using a GPU. While each of these chapters has discussed the impact these optimisations have had on the degree and coefficient algorithms, they have not discussed the impact these optimisations have had overall. Additionally, the improvements made to the degree estimation algorithm in Chapter 4, primarily by reducing the range of gradients that the algorithm will consider, have not been demonstrated within the context of the image deconvolution algorithm.

This chapter will investigate these improvements and optimisations, to give an idea of the overall impact the research presented in this thesis has had on the BID algorithm in its entirety.

After deconvolution, it was found that the processed images appeared significantly darker than the input images, this is due to the convolved images being darker around the border area. To correct for this, the ratio between the norms of the blurred image, excluding the border area, and the deconvolved image was found. This ratio was used to scale the pixel values of the output images so that they appear to be the correct brightness.

9.1 Standard Test Images

For this section, eight standard test images were selected to test the algorithm. These images were tested at two resolutions, 256×256 and 512×512. These images were convolved with a PSF of dimensions 25×25 for the smaller images, and 51×51 for the larger images. The images were converted to double matrices, with values scaled between 0 and 1, and random, uniformly distributed noise with an SNR of roughly 120dB was added to both the PSF before convolution, and the blurred image after convolution. All of these images were tested with the best performing GPU accelerated implementation, using the implementations from Chapters 7 and 8, and the best performing CPU implementations, those being the CPU parallel implementations discussed in Chapter 4.

While the algorithms are performing the exact same computations, the method with which the trials are parallelised means that the two implementations may select different rows and columns from the convolved images. This means that the results of the deconvolution in the two implementations may differ.

The results of the tests performed in this section are analysed from both qualitative and quantitative perspectives. The qualitative analysis will discuss how the images look, and the clarity of detail to the human eye. The quantitative approach is to compute a numerical measure of the error in the deconvolved image compared to the exact image, and compare this to the error in the blurred image again compared to the exact image. The convolved image will have the border area trimmed so it is the same size as the exact image, and therefore provide a more accurate comparison.

9.1.1 Error Computation

To compute the error e between an exact image $\mathcal{I}$ and another image $\mathcal{A}$ the images must first be normalised. The pixel values of $\mathcal{A}$ are scaled by the norm of $\mathcal{I}$, and the pixel values of $\mathcal{I}$ are scaled by the norm of $\mathcal{A}$.

$$\tilde{\mathcal{I}} = \frac{\mathcal{I}}{\|\mathcal{A}\|_2} \quad , \quad \tilde{\mathcal{A}} = \frac{\mathcal{A}}{\|\mathcal{I}\|_2}.$$

The pixel values of the image $\tilde{\mathcal{A}}$ are then subtracted from the pixel values of $\tilde{\mathcal{I}}$ to give the difference between the two images.

$$\mathcal{E} = \tilde{\mathcal{A}} - \tilde{\mathcal{I}}.$$

The error relative to the original image is then computed by finding the ratio between the Frobenius norm of $\mathcal{E}$ and the Frobenius norm of the normalised exact image $\tilde{\mathcal{I}}$.

$$e = \frac{\|\mathcal{E}\|_F}{\left\|\tilde{\mathcal{I}}\right\|_F}.$$

9.1.2 Error and Runtime Results

Eight standard test images were used as the subjects of these tests. The exact forms of these images are shown in Figure 9.1.

Due to the relative size of these images and the PSFs with which they were convolved, the degraded images at both sizes look very similar. Therefore only the larger blurred images will be shown here. Figure 9.2 shows the convolved images. The black boundary areas, discussed in Chapter 2, are shown in these images. These boundary areas are necessary for the deconvolution to be successful in the current implementation.

Figure 9.3 shows the runtime and error results of the deconvolution of the smaller 256×256 images using the CPU parallel algorithm, and Figure 9.4 shows the same images deconvolved using the GPU accelerated algorithm. The results of the deconvolutions

Figure 9.1: The eight exact images used in this experiment

(a) (b) (c) (d) (e) (f) (g) (h)

Figure 9.2: Blurred images from Figure 9.1

Figure 9.3: 256×256 images restored on the CPU

Figure 9.4: 256×256 images restored with the GPU

		CPU		GPU	
Image	Blurred error	Time (s)	Deconvolved error	Time (s)	Deconvolved error
a	0.2113	34.85	0.0590	1.49	0.0597
b	0.1782	34.56	0.0619	1.49	0.0609
c	0.1667	34.86	0.0614	1.52	0.0615
d	0.2204	34.82	0.0563	1.47	0.0567
e	0.2024	34.95	0.0577	1.52	0.0588
f	0.1894	34.87	0.0600	1.52	0.0608
g	0.2088	34.86	0.0589	1.49	0.0597
h	0.2052	34.90	0.0593	1.48	0.0605

Table 9.1: Time taken and error before and after the deconvolution on the CPU and GPU for the 256×256 images

demonstrate that at this level of noise, with the PSF sizes specified, the image deconvolution is consistently successful for all tests. Both implementations appear to give very similar results, such that it is hard to identify the difference between the output of the two implementations by looking at them. Both implementations result in deconvolved images in which the only difference from the exact image appears to be the presence of noise, which, as was described in Chapter 4, is a result of the noise being added to the convolved image.

Table 9.1 shows the results of the deconvolutions for both implementations. The average error for the blurred images across all of these tests is equal to 0.1978. The average errors in the deconvolved images from the CPU and GPU algorithms are 0.0593 and 0.0598 respectively. These show a significant decrease in the error present in the deconvolved images, with both algorithms, as expected, providing similar decreases in the error present in the images.

The GPU implementation shows a significant decrease in runtime compared to the CPU parallel implementation, from an average of 34.83 seconds per image down to an average of 1.49 seconds per image. This decrease demonstrates that the acceleration has been successful. While the majority of the algorithm, including the linear programming sections, remains identical between the two implementations, the AGCD degree computation and the computation of an optimal column have been accelerated. These two sections now make up only a very small amount of the runtime of the GPU accelerated BID algorithm, with the majority of the remaining runtime consisting of linear programming and least squares equality problems.

Figures 9.5 and 9.6 show the blurred images and deconvolution results of the larger 512×512 images. Chapter 4 discussed that the effect of increasing the degree of the AGCD. While there was a decrease in the reliability of the computation of the degree at high convolved degrees, this only had a significant impact with high levels of noise. This is still the case here, with the degree being reliably computed, however the coefficient computation struggles to maintain reliability for larger PSFs. This is seen in the increased

Figure 9.5: 512×512 images restored on the CPU

Figure 9.6: 512×512 images restored with the GPU

Image	Blurred error	CPU		GPU	
		Time (s)	Deconvolved error	Time (s)	Deconvolved error
a	0.2165	180.38	0.2735	5.78	0.2586
b	0.1794	162.67	0.2874	5.48	0.2571
c	0.1712	163.21	0.2860	5.57	0.2848
d	0.2298	162.71	0.2567	24.69	0.2524
e	0.2165	162.97	0.2709	5.55	0.2734
f	0.2066	162.47	0.2807	5.53	0.2810
g	0.2149	162.43	0.2772	5.66	0.2753
h	0.2165	162.47	0.2720	25.00	0.2540

Table 9.2: Time taken and error before and after the deconvolution on the CPU and GPU for the 512×512 images

presence of noise in both sets of results compared to those of the original images.

Table 9.2 shows the results for runtimes and errors from these two sets of images. The errors in both cases appear larger than the errors caused by the blurring. The average error of the blurred images in this case was 0.2064, while the average errors in the deconvolved images are 0.2756 and 0.2671 for the CPU and GPU accelerated algorithms respectively. Despite this increase in error the images appear be restored compared to the blurred image. The increase in error is likely due to the increase in noise present in the images, compared to the smaller images, due to the algorithm not coping as well with the larger PSF size.

The average runtime for the CPU implementation is 164.19 seconds, while the GPU accelerated implementation is 10.41 seconds. This is a significant decrease in time provided by the GPU acceleration. It is notable that image (a) in the CPU implementation, and images (d) and (h) in the GPU accelerated implementation, take roughly 19 seconds more than the other images. In these cases the algorithm struggles to compute the coefficients of the AGCD to be within the threshold of permutations for the source polynomials. This results in the least squares equality function taking a significant amount of time longer than would otherwise be required. When this is not the case the average time of the CPU parallel algorithm is 162.70 seconds for the CPU implementation, and 5.59 seconds for the GPU accelerated algorithm.

When the degree of the AGCD is increased further, the deconvolution typically fails. While the degree is typically still estimated correctly, the computation of the coefficients fails. This often results in the output of a black image or only noise, with no visible trace of the exact image, when the deconvolution is attempted.

9.1.3 Reliability Improvements

Chapter 4 detailed two methods in which the computation was improved over the original method. These improvements provided more reliable computation, in the first case for the AGCD degree computation, in the second case for the AGCD coefficient computation. The

tests discussed in the previous section were attempted without each of these improvements, and the results of these tests will be discussed here.

9.1.3.1 Degree Computation Improvements

The AGCD degree computation, detailed in Chapter 4, considered a reduced set of the subresultant matrices, to be able to compute the degree of the AGCD more reliably, and with more confidence.

When considering all subresultant matrices for the computation of the degree, deconvolution of the smaller images in the tests above were still successful. However, when the image size and PSF size were increased the computed degree was incorrect in at least one of the dimensions of the PSF in all situations. While some trials for most convolved images returned the correct degree, the modal result of the trials returned the incorrect degree. When the computed degree is incorrect by even a small amount this results in a failed deconvolution. With the reduced degree estimation the degree was computed correctly in all tests.

9.1.3.2 Coefficient Computation Improvements

Chapters 4 and 8 detailed the selection of the trials from the degree computation that would be taken forward to be used for the SNTLN algorithm. In the original implementation the first of these trials which returned the correct degree was selected. It was suggested that the selection of the trial to be used in the algorithm for the computation of the coefficients could be improved, by using the gradients computed for each trial during the degree computation. The solution used was to find trials that, where possible, the tests using both the norms and diagonals returned the correct degree, and of these that have the minimum combined gradient.

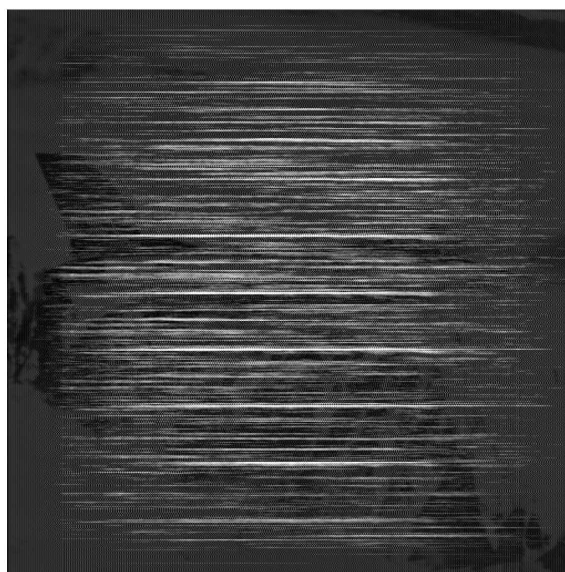


Figure 9.7: Failed deconvolution from incorrect trial selection

The tests detailed previously in this section were attempted for both cases. When using the original implementation the correct coefficients were identified for all of the smaller images. When considering the larger images the deconvolution was successful in several cases, however there were multiple images in which the coefficients were incorrectly computed, and the deconvolution was unsuccessful. An example of this is shown in Figure 9.7, where image (d) from 9.2 was deconvolved to result in the image shown here. It was found that by selecting the trial using the method proposed here the images on which deconvolution had previously failed were successfully deconvolved.

9.2 Large Images

The runtime and reliability improvements presented in this thesis make processing larger images feasible. While there are limitations in the current GPU implementations that restrict the size of the image due to shared memory, the processing of images as large as 2000×2000 is still possible. In this section the GPU accelerated algorithm will be tested on a large image, to investigate the noise level and AGCD degree that can be present for a successful deconvolution to occur on an image of this size. While section 9.1 demonstrated the algorithm being used on a variety of images, the tests presented in this section demonstrate the effect of the algorithm on a single image, which was selected for the variety in high frequency and low frequency features present in this image. The images will be tested with a variety of PSF dimensions and noise levels to investigate how the noise and size of the PSF will effect the result.

Figure 9.8: 2000×2000 image that will be used for the tests in this section

The image shown in Figure 9.8 will be blurred with Gaussian PSFs of dimensions 25×25, 51×51, 75×75, and 101×101. Additionally, the image will be tested at three levels of noise. In this section these noise levels will be referred to as low, medium, and

high levels of noise. Each of these noise distributions will add noise of specified upper and lower bounds to the PSF, and to the convolved image, with the noise uniformly distributed between the specified bounds. The low level of noise is defined as having an SNR of roughly 80dB. Medium level of noise is defined as having an SNR of roughly 100dB. Finally high levels of noise are defined as having an SNR of roughly 120dB, which is the same level as noise as was used in Section 9.1.

All tests in this section were performed using the same 6 core Intel i7 6850k CPU, and an NVIDIA Titan V GPU as were used to test the algorithm in the previous chapters. To give a baseline for the runtime of the algorithm, the deconvolution of a PSF of size 25×25 was performed using the CPU parallel implementation, which performed the deconvolution in 5662.29 seconds.

PSF Size	Time (s)	Blurred error	Deconvolved error
25×25	174.16	0.2267	0.0109
51×51	197.91	0.2737	0.0419
75×75	182.06	0.2759	0.0320
101×101	187.57	0.2789	0.5989

Table 9.3: Time taken and errors for the deconvolution of large images at low levels of noise

Figures 9.9, 9.10, 9.11, and 9.12 show the blurred images and the deconvolution results when a low level of noise is added. Table 9.3 gives the runtimes and error measures for these deconvolutions. The images appear less blurred than those in Section 9.1, as the size of the PSF is lower relative to the size of the image. Despite this, it is apparent that many of the high frequency areas of the image have been lost, even for small PSFs. This is most obvious in the tiles on the roof of the building, and the trees in the background of the image.

The first three images were successfully deconvolved, with only a small amount of noise differentiating the results from the exact images in Figure 9.1. This is reflected in the error measures shown in Table 9.3, where the error has been reduced by an order of magnitude in all cases. The final image, however, shows a large amount of noise. This is due to the coefficient computation struggling to cope with the large PSF, even at this low level of noise. Despite the large amount of noise that is present in the deconvolved image it is clear that some of the high frequency components have been restored, with the most obvious change being around the leaves at the top of the image.

Figures 9.13, 9.14, and 9.15 show the blurred images and results of the deconvolution for a medium level of noise, with Table 9.4 giving the numerical values of runtime and error. When testing the algorithm on large images, with low levels of noise, the results for four sizes of PSF were shown. In this case the 101×101 PSF did not result in a successful deconvolution, as the degree of the AGCD was not correctly estimated. All three of these images demonstrate a successful deconvolution, with a reduction in the

(a) Blurred image (b) Restored image

Figure 9.9: The blurred and deconvolved images for a PSF of dimensions 25×25 with low levels of noise

(a) Blurred image (b) Restored image

Figure 9.10: The blurred and deconvolved images for a PSF of dimensions 51×51 with low levels of noise

(a) Blurred image (b) Restored image

Figure 9.11: The blurred and deconvolved images for a PSF of dimensions 75×75 with low levels of noise

(a) Blurred image (b) Restored image

Figure 9.12: The blurred and deconvolved images for a PSF of dimensions 101×101 with low levels of noise

(a) Blurred image (b) Restored image

Figure 9.13: The blurred and deconvolved images for a PSF of dimensions 25×25 with medium levels of noise

(a) Blurred image (b) Restored image

Figure 9.14: The blurred and deconvolved images for a PSF of dimensions 51×51 with medium levels of noise

PSF Size	Time (s)	Blurred error	Deconvolved error
25×25	174.04	0.2511	0.0193
51×51	179.86	0.2656	0.0299
75×75	182.08	0.2759	0.2217

Table 9.4: Time taken and errors for the deconvolution of large images at low levels of noise

level of error present for all PSFs. The PSF of size 75×75 did result in a high level of noise, though not as high as was present in the deconvolution of the 101×101 PSF image of the low noise tests. Similarly to the low noise tests, the first two PSFs reduced the error by an order of magnitude when deconvolved. However, the last image shows only a small reduction in error compared to the blurred image, though it does appear to be deconvolved successfully in Figure 9.15. This is due to the presence of noise added by the deconvolution.

PSF Size	Time (s)	Blurred error	Deconvolved error
25×25	174.38	0.2511	0.0211
51×51	179.67	0.2656	0.6085

Table 9.5: Time taken and errors for the deconvolution of large images at low levels of noise

Figures 9.16 and 9.17, and Table 9.5 show the results for the deconvolution at high levels of noise. Increasing the noise again resulted in the 75×75 PSF being too large to deconvolve, and thus only two results are shown here. The smaller PSF was successfully deconvolved, with the error again being an order of magnitude lower than the blurred image. The larger PSF, however, resulted in a similar deconvolution to the 101×101 PSF of the low noise tests. While it is apparent that the edges within the image have been sharpened, and high frequency features restored, a large amount of noise has been added to the image during the deconvolution. This is reflected in the error measure for this image, which is significantly higher than the error of the blurred image.

The tests on larger images demonstrated that the BID algorithm can perform well at low levels of noise. However, when either the noise increases too much, or the PSF is too large, the computation of the coefficients starts to produce worse results. Increasing the size of the PSF and noise level further causes the computation of the degree of the AGCD to fail in at least one dimension, which results in the deconvolution failing.

The algorithm took an average of 181.30 seconds to run. This is an runtime improvement of 31 times faster than the CPU parallel implementation. This could be improved further by investigating other parts of the algorithm that could benefit from parallelism, as will be described further in Chapter 10.

(a) Blurred image (b) Restored image

Figure 9.15: The blurred and deconvolved images for a PSF of dimensions 75×75 with medium levels of noise

(a) Blurred image (b) Restored image

Figure 9.16: The blurred and deconvolved images for a PSF of dimensions 25×25 with high levels of noise

9.3 Conclusion

This chapter demonstrated how the algorithms can be applied in the full BID algorithm. Section 9.1 showed the algorithm being applied to a number of images with a high level of noise added. While a significant amount of noise is present when deconvolving images with a large PSF, the deconvolution was successful in that the images were clearer than before the deconvolution. Section 9.2 demonstrated how reliable the algorithm was when processing large images. As the results in this section showed, the algorithm can successfully deconvolve very large images, provided the size of the PSF, or the noise present in the image, are low enough.

This chapter represents a culmination of the work in all previous chapters, and demonstrated the effectiveness of accelerating the BID algorithm using GPUs. The final chapter will discuss the key findings presented in this thesis, and provide insights on directions the research presented here could be taken further.

(a) Blurred image (b) Restored image

Figure 9.17: The blurred and deconvolved images for a PSF of dimensions 51×51 with high levels of noise

Chapter 10

Conclusion

This thesis has presented a significantly accelerated algorithm for the computation of an AGCD as part of the BID algorithm presented in Chapter 4. Throughout this work a number of advancements and findings were made. This chapter will discuss the findings made throughout this thesis, and the potential avenues for future research.

10.1 Key Findings

This section will outline the key findings of this thesis, and discuss the impact they have on the overall algorithm. The findings have been split into several sections. These sections will be the findings that could be applied to the original algorithm, those necessary for the acceleration of the computation of the degree, those necessary for the optimisation of the coefficient computation, and finally those related to how the algorithm has been improved as a whole through this acceleration.

10.1.1 Changes to the Original Algorithm

Chapter 4 presented the original image deconvolution algorithm, first proposed by Winkler [12]. In addition to describing the method of blind image deconvolution that the work in this thesis is based on, modifications to this algorithm were proposed to improve the reliability of the computation of the AGCD.

Prevention of selection of the same rows or columns for degree trials The computation of the degree of the AGCD involved sampling an arbitrary number of row and column vectors from the image. By avoiding selecting the same rows or columns of pixels from an image wherever possible, the algorithm avoided encountering problematic rows more than once. This led to more trials producing the correct result for the degree estimation, and could potentially mean that fewer trials are required for the degree computation.

Improved runtime and reliability of degree computation by reducing the number of subresultant matrices considered The original BID algorithm,

proposed by Winkler and described in Chapter 4, computed the degree of an AGCD based on all subresultant matrices. While this provided reliable results in many cases, increases in noise and the degrees of the convolved polynomials led to a rapid falloff in the reliability of the algorithm. Winkler noted that by limiting the considered modes to a predetermined range, such as half of the degree of the convolved polynomial, the algorithm could be made more reliable. In Chapter 4 a reduced implementation was considered, where only half of the subresultant matrices were considered, thus assuming the degree of the PSF is less than half of the overall image. By making this assumption, and limiting the degrees that each trial could return, decisions on the degree can be made with more confidence, as more trials return the expected answer. Performance improvements were also observed, though these were only modest improvements, as the subresultant matrices eliminated were the smaller matrices, and thus did not take as long to process anyway.

Removing unnecessary computation of the row deletion for each upper triangular factor The original implementation of the computation of the degree of an AGCD required the deletion of two columns and a row of from each subresultant matrix S_k in order to compute the subresultant matrix S_{k+1}. This row deleted from S_k also meant performing a row deletion on the QR factors of S_k to compute the factors of S_{k+1}. It was noted in Chapter 4 that the removal of this row was mathematically unnecessary when considering only the rank of S_{k+1}, and its upper triangular factor R_{k+1}. In both of these matrices the removal of the two columns from S_k would result in the bottom row of S_{k+1} having no non-zero entries, and therefore the bottom row of R_{k+1} would also have no non-zero entries. Therefore these rows would have no effect on the rank. By removing the need to compute this row deletion the need to compute the orthogonal factors $Q_{1\ldots n}$ is removed entirely, and thus the computation of the upper triangular factors can be made significantly more efficient.

Removing unnecessary computation in the optimal column algorithm Chapter 7 described multiple methods in which the optimal column computation, as part of the modified SNTLN algorithm, could be accelerated. While some of these optimisations were specific to parallelisation, two in particular were able to be applied to the serial implementation. The first of these is the reduction of the QR update algorithm, in which only a single column of each rotation needs to be computed, almost halving the number of operations in the matrix products. Additionally, in the matrix product required to compute the residual, a reduced matrix product can be computed instead. This significantly reduces the number of operations needed for this computation.

10.1.2 Parallelisation of the Degree Computation

The main focus of this thesis has been on accelerating the computation of the degree of an AGCD. This was shown in Chapters 6 and 7. The results in Chapter 6 showed significant improvements in the overall runtime of the degree computation, though suggested potential issues with scalability due to the limits in memory on the GPU. Chapter 7 improved on this implementation by drastically reducing the memory needed for each trial. This led to significant improvements in performance and scalability, and made it possible to process polynomials of high degree, even on hardware with a limited amount of GPU memory.

Fast processing of triangular factors of subresultant matrices for degree computation on a GPU The implementation proposed in Chapter 6 showed promising results for the performance improvements of a GPU algorithm for the computation of the degree of a AGCD. While runtime was improved over the CPU parallel implementation for all polynomial degrees, and therefore all sizes of image, the algorithm did show some scaling problems as the memory operations and kernel calls had to be batched. When the degree of the convolved polynomials is high enough the polynomials could not be processed, due to running out of memory. This was the first attempt at parallelising the degree computation on a GPU, and the runtime improvements gained here could still be considered a success, despite the issues with scaling. The algorithm presented in Chapter 6 showed an improvement of up to 57 times faster than the CPU serial implementation, and up to 25 times faster than the CPU parallel implementation, though these results were inconsistent due to the batching algorithm.

An efficient memory structure for storing and referencing triangular matrices Chapter 6 presented a method for efficiently storing and referencing the rows of triangular matrices. This was required due to the limited memory provided by GPUs. As there are no non-zero elements below the principal diagonal, only the entries on and above the principal diagonal are stored. Triangular numbers and triangular roots can then be used to reference the rows of the matrices. While the use of efficiently stored matrices increased the complexity of retrieving entries of the matrices, the use of these memory structures enabled the algorithm to scale better on GPUs with a limited amount of memory.

Novel method of load balancing The algorithm implemented in Chapter 6 has an inherently unbalanced workload. As a result a load balancing method was necessary in order to efficiently process the computations. The method implemented was very successful at batching the work, and keeping idle threads to a minimum. Due to how effective this method was it was successfully utilised in most kernels developed for this thesis.

Minimal runtimes improvements from reducing the number of subresultant matrices As discussed in Section 10.1.1, reducing the number of subresultant

matrices in the algorithm introduced in Chapter 4 improved both reliability and runtime of the degree computation algorithm. This was again demonstrated on a GPU in Chapter 6. While the improvements in the reliability of the degree computation remained, the improvement to the runtime of the algorithm by excluding the smaller subresultant matrices was minimal. This was due to the decreased amount of work on the GPU, and the work being less balanced for such an implementation. As a result, when developing the low memory implementation the reduced algorithm was not considered. Instead a more general approach was decided upon, where all subresultant matrices were computed, and an arbitrary limit can be set on the gradients to be considered after the computation.

Use of warp shuffle prefix sums for balanced computation of row norms of triangular matrices The necessity to compute the row norms of all subresultant matrices meant a method needed to be developed to efficiently compute a number of unbalanced prefix sums. The efficient memory structures designed for the storage of upper triangular matrices meant that the computation of these norms was split into two kernels, and separate methods of balancing the prefix sums for each were developed. The first implementation parallelised the rows across blocks, with threads alternating between a row from the upper half of the matrix and a row from the bottom half in each iteration. This meant that over the course of the iterations the blocks ended up balancing. The other implementation computed the sums of the entire matrix, then subtracted the sum at the end of rows from that of previous row in the sum. This meant that the work was balanced within the sum itself. Both of these techniques proved to be effective at balancing the inherently unbalanced row sums.

Efficient utilisation of GPU memory for the computation of the degree of an AGCD The implementation proposed in Chapter 7 demonstrated a new, memory efficient, method of computing the degree of an AGCD using GPUs. This new implementation moved the computation of the row sums into the same kernel as the upper triangular factor computation. This led to a more complex kernel, leading to issues with increased shared memory and register usage, but resulted in improvements to the overall scalability of the algorithm.

Low memory implementation showed significant improvements in execution time over the previous implementation The low memory implementation described in Chapter 7 provided significant improvements to the runtime over the previous iteration. While there were multiple contributing factors to this improvement, the most significant reason was removing the need to batch the trials, thus reducing the overhead of the extra kernel and GPU memory calls, as well as being able to process more data simultaneously. The results in Chapter 7 demonstrated that this algorithm performed up to 5 times faster than the original GPU implementation, with runtimes 133 and 31 times faster than the CPU serial and parallel implementations respectively.

10.1.3 Parallelisation of the Computation of the Coefficients

Chapter 8 demonstrated an efficient method of computing the most expensive part of the modified SNTLN algorithm presented by Winkler in [21]. This implementation considered the computation of an optimal column as part of the modified SNTLN algorithm. This algorithm, drawing on inspiration from the algorithm described in Chapter 7, successfully parallelised the algorithm for the computation of the optimal column.

Reduction in the number of operations Chapter 8 presented a reduced algorithm, in which only one of the two rows, or columns, of the matrix products for the Givens rotations needs to be computed. This represents a significant optimisation over the original implementation, and helps minimise the runtime of both the CPU and GPU implementations.

Runtime improvements over the original algorithm The implementation presented in Chapter 8 demonstrated significant runtime improvements over the CPU parallel implementation. The implementation of this algorithm led to speedups of up to 97 times faster than the CPU parallel algorithm being observed.

Efficient use of memory The algorithm presented in Chapter 8 uses a similar memory structure to that seen in Chapter 7. The algorithm scales well, with efficient use of global and shared memory meaning that this will continue to be the case for polynomials of very large degrees.

10.1.4 Overall Improvements to the Blind Image Deconvolution Algorithm

Chapter 9 demonstrates the application of the research presented throughout this thesis to the BID algorithm proposed by Winkler. This gives insight into the performance and reliability improvements that were achieved by this algorithm.

A significant improvement in runtime over CPU implementations of the BID algorithm The GPU accelerated algorithm was tested against the CPU parallel implementation, The GPU algorithm in most cases was around 30 times faster than the CPU parallel implementation, demonstrating a significant decrease in runtime. The most computationally expensive sections of the remaining algorithm were those of the least squares equality and linear programming problems, which could be accelerated to decrease this runtime even further.

Improvements to the reliability of degree computation The improvements to the original BID algorithm described in Chapter 4, with regards to the number of subresultant matrices being considered, and the selection of a trial with which to compute the coefficients, were demonstrated to have a notable affect on the reliability of the image deconvolution. The increase in reliability in the degree computation algorithm allowed for deconvolution to be attempted at higher levels

of noise, and for larger PSFs. The improvements made to the selection of a trial for the optimal column algorithm meant the computation of the coefficients was more robust, leading to a more reliable deconvolution.

10.2 Future Research Opportunities

The findings of this thesis demonstrate how the GPU acceleration of this BID algorithm provides a significant decrease in the runtime, and the improvements proposed in Chapter 4 demonstrated improvements to the reliability of the algorithm. Despite these improvements there is still further research that could improve the runtime or reliability of this algorithm. Additionally, the application of the accelerated AGCD algorithms have focussed on the BID algorithm. Research could be undertaken into how this accelerated algorithm could be applied in other scenarios.

This section outlines some areas in which further research could be beneficial. These areas have been split into two sections. The first section will detail research that could be undertaken to improve the runtime performance of this algorithm, while the second section details how these accelerated algorithms could be applied in situations beyond the current BID algorithm.

10.2.1 Further Improvements to the GPU Algorithms

While the algorithm has been significantly accelerated, there are potential avenues through which the runtime could be reduced even further. This may be necessary when processing very large images, or videos, when greater time constraints are in place.

Further acceleration of the BID algorithm The acceleration presented in this thesis focused on two key areas of the algorithm in which the majority of the runtime was spent. These being the computation of the degree, and the optimal column computation as part of SNTLN. While these two sections now take significantly less time, other areas of the algorithm have not been modified. Some areas such as linear programming already have heavily optimised libraries available on GPUs. It would be desirable to implement these sections on a GPU as well, to further accelerate the overall algorithm.

Introducing pipelining into the QR update algorithm Pipelining, in relation to the parallelisation of the QR decomposition using Givens rotations, was discussed in Chapter 5. This technique involved the processing of partial rows, thus enabling further parallelisation of the decomposition. This could potentially be implemented into both accelerated sections of the AGCD algorithm. Research into how this could be integrated could potentially speed up the already accelerated sections of the algorithm even further, particularly for large images and polynomial degrees.

Reducing the complexity of the data structures The algorithm for the degree computation used efficient memory structures to store the triangular matrices, due to the shortage of GPU memory. Unfortunately this led to an increase in the number of operations that must be computed, as triangular numbers and triangular roots must frequently be computed. The implementation introduced in Chapter 7 presented a low memory implementation. As the algorithm is no longer limited by the amount of GPU memory, simpler data structures could potentially be used. This would avoid the need for such complex indexing computations, and could potentially lead to improved performance.

Investigate the effect of moving more data into global memory The algorithm presented in Chapter 7 is limited by the amount of shared memory available per block. While the amount of shared memory available may increase in future hardware, it would be desirable for this limit to be reduced, in order for polynomials of greater degree to be processed. By moving some of the data from shared memory to global memory this limit can be reduced, though there will be a impact on the performance. The extent of this performance impact for polynomials of higher degrees should be investigated.

Utilisation of multiple GPUs The degree computation split the computation into several trials, which, for most kernels, were computed into separate blocks. This separation would allow the implementation to easily be distributed to multiple GPUs. The effect of this, and the improvement of the scaling from splitting the computation between multiple devices, could be investigated.

Removing unnecessary computations in the optimal column algorithm The computation of the optimal column in Chapter 8 demonstrated several methods to eliminate unnecessary work. This could be extended further in the computation of the matrix product and squared residual. A significant proportion of elements in every column of the subresultant matrices will be zero, and the position of the non-zero elements is easily calculable given the column index. This means that the matrix product can be reduced further. This would also lead to fewer elements in the resulting vector, and therefore fewer operations to compute the squared norm. The impact this reduction could have on the algorithm could be investigated.

A Gram-Schmidt optimal column implementation As was discussed in Chapter 3, one of the advantages of the Gram-Schmidt orthogonalisation is the ability to compute a single column of the orthogonal matrix. This could potentially be of use in the optimal column computation, as only a single column of the required submatrix of each updated orthogonal matrix differs from the original matrix. While the Gram-Schmidt method has issues with numerical stability it would be worth investigating if this would harm the reliability of the results, and if a Gram-Schmidt method could provide performance improvements over the Givens method currently used.

10.2.2 Applications of the Algorithm

This thesis focused on the computation of the AGCD as part of a polynomial BID algorithm for an image with a separable PSF. The AGCD algorithm, however, could be used more generally. This section will focus on the potential applications of this algorithm in other situations.

Application to bivariate polynomials The algorithm accelerated in this thesis can only be applied to separable PSFs. Winkler suggested an extension of this algorithm to a non-separable PSF by investigating bivariate polynomials in the Fourier domain. It would be desirable to extend the implementations here to accelerate these computations as well. Therefore, a key area for further investigation would be to research how the algorithms developed through this thesis could be modified to accelerate such an algorithm.

Application of the accelerated AGCD algorithm in other fields As was noted in Chapter 1, the computation of an AGCD is not unique to BID algorithms, and can be found in a number of other fields, such as control theory [16] and the computation of multiple roots of a polynomials [17, 18, 19]. An area for further investigation would be how the algorithm presented here could be harnessed in these scenarios, and the benefit such implementations would have.

10.3 Conclusion

This chapter has given an overview of the key findings, and areas for further research that were identified throughout this thesis. The algorithm implemented, while being the first such implementation to parallelise polynomial methods of BID, is also the first accelerated algorithm for the computation of an AGCD. This acceleration led to a significantly faster blind deconvolution of a blurred image, while retaining the accuracy and low error that the original algorithm presented by Winkler was notable for.

The key findings described in Section 10.1 describe a number of novel features of this research, that could potentially be implemented in other algorithms. Additionally the acceleration of the AGCD algorithm could also have potential implications beyond the act of image restoration. This research has therefore been successful in accelerating not only the BID algorithm, but aspects of the implementations presented in throughout this chapter could have applications in a wide variety of topics.

Bibliography

[1] J. R. Winkler. Polynomial computations for blind image deconvolution. *Linear Algebra and its Applications*, 502:77 – 103, 2016. Structured Matrices: Theory and Applications.

[2] W. H. Richardson. Bayesian-based iterative method of image restoration. *Journal of the Optical Society of America*, 62(1):55–59, 1972.

[3] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 79:745, 1974.

[4] B. Kursat Gunturk and X. Li. *Image Restoration: Fundamentals and Advances*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2017.

[5] A. Katsaggelos and K.T. Lay. Maximum likelihood blur identification and image restoration using the EM algorithm. *Signal Processing, IEEE Transactions on*, 39:729 – 733, 04 1991.

[6] A. C. Likas and N. P. Galatsanos. A variational approach for Bayesian blind image deconvolution. *IEEE Transactions on Signal Processing*, 52(8):2222–2233, Aug 2004.

[7] R. G. Lane and R. H. T. Bates. Automatic multidimensional deconvolution. *Journal of the Optical Society of America*, 4(1):180–188, Jan 1987.

[8] K. Panchapakesan, D. G. Sheppard, M. W. Marcellin, and B. R. Hunt. Blur identification from vector quantizer encoder distortion. *IEEE Transactions on Image Processing*, 10(3):465–470, March 2001.

[9] R. Nakagaki and A. K. Katsaggelos. A vq-based blind image restoration algorithm. *IEEE Transactions on Image Processing*, 12(9):1044–1053, Sep. 2003.

[10] L. Xu, J. Ren, C. Liu, and J. Jia. Deep convolutional neural network for image deconvolution. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1790–1798. Curran Associates, Inc., 2014.

[11] C. J. Schuler, H. C. Burger, S. Harmeling, and B. Scholkopf. A machine learning approach for non-blind image deconvolution. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2013.

[12] J. R. Winkler. The sylvester resultant matrix and image deblurring. In *Curves and Surfaces*, pages 461–490, Cham, 2015. Springer International Publishing.

[13] Z. Li, Z. Yang, and L. Zhi. Blind image deconvolution via fast approximate gcd. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 155–162, New York, NY, USA, 2010. ACM.

[14] B. Liang and S. U. Pillai. Blind image deconvolution using a robust 2-d gcd approach. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems. Circuits and Systems in the Information Age ISCAS '97*, volume 2, pages 1185–1188 vol.2, June 1997.

[15] S. Unnikrishna Pillai and B. Liang. Blind image deconvolution using a robust gcd approach. *IEEE Transactions on Image Processing*, 8(2):295–301, Feb 1999.

[16] P. Stoica and T. Söderström. Common factor detection and estimation. *Automatica*, 33(5):985 – 989, 1997.

[17] J. R. Winkler, X. Y. Lao, and M. Hasan. The computation of multiple roots of a polynomial. *Journal of Computational and Applied Mathematics*, 236:3478–3497, 2012.

[18] J. R. Winkler. Structured matrix methods for the computation of multiple roots of a polynomial. *Journal of Computational and Applied Mathematics*, 272:449–467, 2014.

[19] Z. Zeng. Computing multiple roots of inexact polynomials. *Mathematics of Computation*, 74(250):869–903, 2005.

[20] J. Rosen, H. Park, and J. Glick. Structured total least norm for nonlinear problems. *SIAM Journal on Matrix Analysis and Applications*, 20(1):14–30, 1998.

[21] J. R. Winkler, Madina Hasan, and X. Lao. Two methods for the calculation of the degree of an approximate greatest common divisor of two inexact polynomials. *Calcolo*, 49(4):241–267, Dec 2012.

[22] E. W. Weisstein. Convolution From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/Convolution.html`. Last visited on 15/7/2019.

[23] Mathworks MATLAB documentation - deconvlucy. `https://uk.mathworks.com/help/images/ref/deconvlucy.html`. Last visited on 09/7/2019.

[24] Mathworks MATLAB documentation - deconvwnr. `https://uk.mathworks.com/help/images/ref/deconvwnr.html`. Last visited on 09/7/2019.

[25] Mathworks MATLAB documentation - deconvreg. `https://uk.mathworks.com/help/images/ref/deconvreg.html`. Last visited on 09/7/2019.

[26] Mathworks Matlab documentation - deconvblind. https://uk.mathworks.com/help/images/ref/deconvblind.html. Last visited on 09/7/2019.

[27] D. Kundur and D. Hatzinakos. Blind image deconvolution. *IEEE Signal Processing Magazine*, 13(3):43–64, 1996.

[28] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[29] P. Campisi and K. Egiazarian, editors. *Blind image deconvolution. Theory and applications*. CRC Press, Taylor & Francis Group, 2007.

[30] J. Sun, W. Cao, Z. Xu, and J. Ponce. Learning a convolutional neural network for non-uniform motion blur removal. *CoRR*, abs/1503.00593, 2015.

[31] A. M. Deshpande and S. Patnaik. A spatially variant motion blur removal technique for single image deblurring. In *2014 Annual IEEE India Conference (INDICON)*, pages 1–6, Dec 2014.

[32] X Zhang, R Wang, X Jiang, W Wang, and W Gao. Spatially variant defocus blur map estimation and deblurring from a single image. *Journal of Visual Communication and Image Representation*, 35:257 – 264, 2016.

[33] M. Temerinac-Ott, O. Ronneberger, R. Nitschke, W. Driever, and H. Burkhardt. Spatially-variant lucy-richardson deconvolution for multiview fusion of microscopical 3d images. In *2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 899–904, March 2011.

[34] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[35] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[36] T. S. Srensen, T. Schaeffter, K. . Noe, and M. S. Hansen. Accelerating the nonequispaced fast fourier transform on commodity graphics hardware. *IEEE Transactions on Medical Imaging*, 27(4):538–547, April 2008.

[37] L. Domanski, P. Vallotton, and D. Wang. Two and three-dimensional image deconvolution on graphics hardware. In *18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation: Interfacing Modelling and Simulation with Mathematical and Computational Sciences, Proceedings*, pages 1010–1016, 2009. Cited By :7.

[38] C. L. Matson, K. Borelli, S. Jefferies, C. C. Beckner Jr., E. K. Hege, and M. Lloyd-Hart. Fast and optimal multiframe blind deconvolution algorithm for high-resolution ground-based imaging of space objects. *Appl. Opt.*, 48(1):A75–A92, Jan 2009.

[39] J. T. Klosowski and S. Krishnan. Real-time image deconvolution on the GPU. In *Parallel Processing for Imaging Applications*, volume 7872, page 78720H. International Society for Optics and Photonics, 2011.

[40] D. Krishnan and R. Fergus. Fast image deconvolution using Hyper-Laplacian priors. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1033–1041. Curran Associates, Inc., 2009.

[41] T. Goto, S. Otake, and S. Hirano. Blind image restoration for blurred images implemented on GPU. *2017 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW)*, pages 213–214, 2017.

[42] J. R. Winkler and H. Halawani. The Sylvester and Bézout resultant matrices for blind image deconvolution. *Journal of Mathematical Imaging and Vision*, 60:1284–1305, 2018.

[43] S. Barnett. *Polynomials and Linear Control Systems.* Marcel Dekker, Inc., New York, NY, USA, 1983.

[44] J. R. Winkler and X. Lao. The calculation of the degree of an approximate greatest common divisor of two polynomials. *Journal of Computational and Applied Mathematics*, 235(6):1587 – 1603, 2011.

[45] Z. Zeng and B. H. Dayton. The approximate gcd of inexact polynomials. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 320–327. ACM, 2004.

[46] J. R. Bunch and Christopher P. Nielsen. Updating the singular value decomposition. *Numerische Mathematik*, 31(2):111–129, Jun 1978.

[47] A. Schönhage. Quasi-GCD computations. *Journal of Complexity*, 1(1):118 – 137, 1985.

[48] M. Noda and T. Sasaki. Approximate GCD and its application to ill-conditioned equations. *Journal of Computational and Applied Mathematics*, 38(1):335 – 351, 1991.

[49] V. Hribernig and H. J. Stetter. Detection and validation of clusters of polynomial zeros. *Journal of Symbolic Computation*, 24(6):667 – 681, 1997.

[50] R. M. Corless, P. M. Gianni, B. M. Trager, and S. M. Watt. The singular value decomposition for polynomial systems. In *Proceedings of the 1995 International Symposium*

on Symbolic and Algebraic Computation, ISSAC '95, pages 195–207, New York, NY, USA, 1995. ACM.

[51] R. M. Corless, S. M. Watt, and Lihong Zhi. QR factoring to compute the GCD of univariate approximate polynomials. *IEEE Transactions on Signal Processing*, 52(12):3394–3402, Dec 2004.

[52] J. Rosen, H. Park, and J. Glick. Total least norm formulation and solution for structured problems. *SIAM Journal on Matrix Analysis and Applications*, 17(1):110–126, 1996.

[53] C. J. Zarowski, Xiaoyan Ma, and F. W. Fairman. Qr-factorization method for computing the greatest common divisor of polynomials with inexact coefficients. *IEEE Transactions on Signal Processing*, 48(11):3042–3051, 2000.

[54] D. Bini and P. Boito. Structured matrix-based methods for polynomial in-gcd: analysis and comparisons. pages 9–16, 01 2007.

[55] J. R. Winkler and J. D. Allan. Structured total least norm and approximate gcds of inexact polynomials. *Journal of Computational and Applied Mathematics*, 215(1):1 – 13, 2008.

[56] J. Winkler and M. Hasan. An improved non-linear method for the computation of a structured low rank approximation of the sylvester resultant matrix. *Journal of Computational and Applied Mathematics*, 237:253268, 01 2013.

[57] A. Björck. *Numerical Methods for Least Squares Problems.* Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 1996.

[58] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[59] W. Ford. Chapter 17 - implementing the QR decomposition. In W. Ford, editor, *Numerical Linear Algebra with Applications*, pages 351 – 378. Academic Press, Boston, 2015.

[60] D. Bau III L. N. Trefethen. *Numerical linear algebra.* Society for Industrial and Applied Mathematics, 1997.

[61] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the gram-schmidt qr factorization. *Mathemathics of Computation*, 30:772–795, 10 1976.

[62] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, January 1978.

[63] M. McGraw-Herdeg. Benchmarking the NVIDIA 8800 GTX with the CUDA development platform. 2007.

[64] L. Marcellino and G. Navarra. A gpu-accelerated svd algorithm, based on QR factorization and givens rotations, for dwi denoising. In *2016 12th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*, pages 699–704, Nov 2016.

[65] M. Hofmann and E. J. Kontoghiorghes. Pipeline givens sequences for computing the QR decomposition on a erew pram. *Parallel Computing*, 32(3):222 – 230, 2006.

[66] F. Rotella and I. Zambettakis. Block householder transformation for parallel QR factorization. *Applied Mathematics Letters*, 12(4):29 – 34, 1999.

[67] A. Kerr, D. Campbell, and M. Richards. QR Decomposition on GPUs. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 71–78, New York, NY, USA, 2009. ACM.

[68] R. Andrew and N. Dingle. Implementing QR factorization updating algorithms on GPUs. *Parallel Computing*, 40(7):161 – 172, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.

[69] J. R. Winkler and M. Hasan. A non-linear structure preserving matrix method for the low rank approximation of the sylvester resultant matrix. *Journal of Computational and Applied Mathematics*, 234(12):3226 – 3242, 2010.

[70] NVIDIA CUDA toolkit documentation - hardware implementation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation`. Last visited on 19/7/2019.

[71] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[72] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1063–1072, May 2014.

[73] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 1st edition, 2010.

[74] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Pearson Education, 2013.

[75] NVIDIA developer blog - cooperative groups: Flexible cuda thread programming. `https://devblogs.nvidia.com/cooperative-groups/`. Last visited on 8/9/2019.

[76] T. Soyata. *GPU Parallel Program Development Using CUDA*. Chapman and Hall/CRC, 2018.

[77] NVIDIA CUDA C programming guide - launch bounds. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#launch-bounds`. Last visited on 04/6/2019.

[78] T. McKercher J. Cheng, M. Grossman. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2014.

[79] E. W. Weisstein. Triangular Number. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/TriangularNumber.html`. Last visited on 03/6/2019.

[80] C.Foster. Triangular roots. Applied Probability Trust, 2012.