Harnessing the Power of GPUs for Problems in Real Algebraic Geometry

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Pavel Emeliyanenko

Saarbrücken 2012

Tag des Kolloquiums:
09.11.2012

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Professor Dr. Mark Groves

Berichterstatter:
Professor Dr. Kurt Mehlhorn, Max-Planck-Institut für Informatik, Saarbrücken
Professor Chee K. Yap Ph.D., Courant Institute, New York University, New York, USA
Dr. Michael Sagraloff, Max-Planck-Institut für Informatik, Saarbrücken
Mitglieder des Prüfungsausschusses:
Professor Dr. Raimund Seidel (Vorsitzender)
Professor Dr. Kurt Mehlhorn
Professor Chee K. Yap Ph.D.
Dr. Michael Sagraloff

Akademischen Mitarbeiter:
Dr. Tobias Mömke

Acknowledgements

Foremost, I would like to thank Prof. Kurt Mehlhorn for letting me being a part of his research group where he allows a great deal of autonomy in the research and provides freedom for experimentation. I believe that truly inspirational ideas are only possible in such sort of environments.

I wish to express my great thanks to Michael Sagraloff, my thesis supervisor, for guiding me through all these years and, in particular, for helping me develop abstract mathematical way of thinking which should be of great value in my future career development. I am also indebted to my colleagues Eric Berberich, Michael Kerber and Alexander Kobel for valuable input and collaboration. Last but not least I am thankful to the whole Cluster of Excellence (MMCI) for providing me a financial support over these years.

Zusammenfassung

Diese Arbeit beschäftigt sich mit neuen parallelen Algorithmen, die das Leistungspotenzial der Grafik-Prozessoren (GPUs) zur exakten Berechnungen mit ganzzahlige Polynomen nutzen. Solche symbolische Berechnungen sind von großer Bedeutung zur Lösung vieler Probleme aus der reellen algebraischen Geometrie. Für die effiziente Implementierung eines Algorithmus auf massiv-parallelen Hardwarearchitekturen, wie z.B. GPU, ist vor allem auf eine hohe Datenparallelität zu achten. Unter Verwendung von Ergebnissen aus der strukturierten Matrix-Theorie konnten wir die entsprechenden Operationen mit Polynomen auf der Grafikkarte leicht übertragen. Außerdem zeigt eine Komplexitätanalyse im PRAM-Rechenmodell, dass die von uns entwickelten Verfahren eine deutlich bessere Komplexität aufweisen als dies für die klassischen Verfahren der Fall ist.

Neben dem theoretischen Ergebnis liegt ein weiterer Schwerpunkt dieser Arbeit in der praktischen Implementierung der betrachteten Algorithmen, wobei wir auf der Besonderheiten der Grafikhardware achten. Im Rahmen dieser Arbeit haben wir hocheffiziente modulare Arithmetik entwickelt, von der wir erwarten, dass sie sich für andere GPU Anwendungen, insbesondere der Public-Key-Kryptographie, als nützlich erweisen wird. Darüber hinaus betrachten wir Algorithmen für die Lösung eines Systems von Polynomgleichungen, Topologie Berechnung der algebraischen Kurven und deren Visualisierung welche in vollem Umfang von der GPU-Leistung profitieren können. Zahlreiche Experimente belegen dass wir zur Zeit die beste Verfahren zur Verfügung stellen.

Diese Dissertation ist in englischer Sprache verfasst.

Abstract

This thesis presents novel parallel algorithms to leverage the power of GPUs (Graphics Processing Units) for exact computations with polynomials having large integer coefficients. The significance of such computations, especially in real algebraic geometry, is hard to undermine. On massively-parallel architectures such as GPU, the degree of data-level parallelism exposed by an algorithm is the main performance factor. We attain high efficiency through the use of structured matrix theory to assist the realization of relevant operations on polynomials on the graphics hardware. A detailed complexity analysis, assuming the PRAM model, also confirms that our approach achieves a substantially better parallel complexity in comparison to classical algorithms used for symbolic computations.

Aside from the theoretical considerations, a large portion of this work is dedicated to the actual algorithm development and optimization techniques where we pay close attention to the specifics of the graphics hardware. As a byproduct of this work, we have developed high-throughput modular arithmetic which we expect to be useful for other GPU applications, in particular, open-key cryptography. We further discuss the algorithms for the solution of a system of polynomial equations, topology computation of algebraic curves and curve visualization which can profit to the full extent from the GPU acceleration. Extensive benchmarking on a real data demonstrates the superiority of our algorithms over several state-of-the-art approaches available to date.

This thesis is written in English.

Contents

1 Introduction

1.1 Problem statement

The goal of thesis is to develop a general framework as well as concrete algorithms to speed-up symbolic computations on modern parallel architectures. By symbolic or algebraic computation one usually understands operations carried out by the computer to manipulate mathematical expressions in *symbolic* form rather then working with specific *numeric* quantities or approximations represented by those symbols. With the development of computer algebra systems, symbolic algorithms have become increasingly more important by enabling a new level of computational complexity which was previously beyond the reach of mathematicians. That is why, symbolic algorithms have been widely adopted in many different areas of science and engineering. Many such algorithms have emerged at the dawn of personal computer age, and have not been undergoing major change since then. Currently, we are nearing the point where traditional computer platforms will no longer be able to satisfy the constantly increasing demands in computational power posed by new scientific problems. This, in turn, has motivated us to search for alternative architectures which could have greater performance potential for scientific computing. Among them, the most promising one is the massively-threaded architecture of GPUs (Graphics Processing Units) or graphics accelerators. The choice of this target platform is not surprising since, over the past years, the GPUs have evolved into fully-programmable general-purpose processors with incredible computational horsepower, and presently offer the best parallel performance per processing unit cost ratio.

Developing algorithms for a new platform is not an easy task to accomplish. It often requires new insights into seemingly well-studied problems since design decisions that underlie the original algorithms might not be relevant anymore. For instance, the GPU's execution model is exclusively based on data-level parallelism where the same program is run by a large number (several tens or even hundred thousand) of light-weight threads that do not possess large private memory spaces (except for a small register set and slow local memory) and, loosely speaking, cannot execute disjoint code paths without penalties. For its part, it prompts an algorithm developer to seek for alternative ways to decompose a problem at hand into a set of primitive operations that can be executed concurrently because function-level (or coarse-grained) parallel solutions, designed for traditional workstation networks or multi-core machines, are no longer applicable. Furthermore, unlike conventional CPUs, graphics processors do not have large cache capacities, which means that memory access is not as “transparent” as on the modern CPUs. Instead, off-chip memory latencies are hidden as long as the GPU can keep its functional units busy while waiting on the results of memory access. As a result, careful management of limited-size on-chip memory together with optimization of external memory access have become indispensable parts of the GPU algorithm development.

In this thesis, we foremost concentrate on symbolic algorithms dealing with *polynomials* in one or more variables which, for example, include: polynomial multiplication, greatest common divisors (GCDs), resultants and subresultants. These algorithms build up a basic computer algebra tool-box of contemporary mathematical software such as Maple or Mathematica, and have numerous applications in geometric modeling (Hof89, Sed83), algebraic geometry (CLO98, BPR06), robotics (SK08, Man92), or computer graphics (Vin08). The following small motivating example illustrates why symbolic computations can be very expensive. Suppose, we wish to compute an intersection of two algebraic surfaces of degree 4 defined by the equations:

$$\begin{aligned}
&(10y^2 + z^2 - 112)^2 - 5z^4y^4 + xy^4 - 1 = 0,\\
&50(x^2y^2 + y^2z^2 + x^2z^2) + (x^2 + y^2 + z^2 - 1)^2 = 0.
\end{aligned}$$

Such sort of computations, for instance, are quite often required in geometric modeling. The intersection is given by an algebraic curve in 3D whose projection onto the xy-plane satisfies the equation:

$$\begin{aligned}
R(x,y) := {}& 1 + 100x^2 + 736x^2y^2 - 202x^4 - 46700x^4y^2 - 89220x^2y^4 + 100x^6 - 42420y^6 -\\
& 936x^6y^2 + 379384x^2y^6 + 219186x^4y^4 - 10x^8y^4 - 1102310x^4y^8 + 234160x^4y^6 +\\
& 575661x^2y^8 - 480x^6y^4 + 4160x^6y^6 - 100x^6y^8 - 5300x^4y^{10} - 2374640x^2y^{10} +\\
& 2600x^6y^{10} + 67650x^4y^{12} + 25x^8y^8 - 5300y^{12}x^2 + 2600y^{14}x^2 + 160786y^8 -\\
& 1345660y^{12} + 25y^{16} - 100y^{14} + x^8 - 13510xy^{12} - 2xy^4 - 100x^3y^4 + 1020y^{10}x +\\
& 2702x^5y^4 + 1020x^3y^8 - 100xy^6 + 1852xy^8 - 26520y^10x^3 + 100y^2 + 648y^4 +\\
& 361020y^{10} - 13510x^5y^8 + 4264x^3y^6 = 0.
\end{aligned}$$

We further proceed by analyzing the “topological structure” of the curve $R(x,y)$. This, in turn, involves computing the resultant of R and R'_y which, in our case, is a dense polynomial of degree 132 and 500-bit coefficients.[1] Next, we slightly modify the equations for the original surfaces (increasing the degree), so that they become:

$$\begin{aligned}
&(10y^2 + z^2 - 112)^2 - 5z^5y^5 + xy^5 - 1 = 0,\\
&50(x^2y^2 + y^2z^2 + x^2z^2) + (x^2 + y^2 + z^2 - 1)^3 = 0.
\end{aligned}$$

Repeating the same computations, we find that $R(x,y)$ is now a polynomial of total degree 60 with 53-bit coefficients, while the resultant of R and R'_y has ultimately become a huge expression: that is, a dense polynomial of degree 2028 with 4500-bit coefficients! The above calculations show that symbolic expressions can grow very fast with respect to initial parameters and, as a result, quickly become unmanageable by a traditional algorithm tool-box. This was a major source of inspiration for the present work.

The main theoretical contribution of this thesis is the development of matrix algebra-based algorithms to facilitate the realization of relevant operations with polynomials on the graphics processor. Using this as a background, we have been able to realize *modular*

[1] For basic definitions on polynomials, see Section 2.1.2.

approaches to expedite polynomial GCD and resultant computations on the GPU. On average, our implementation gives about 100x speed-up over the analogous CPU-based algorithms. Moreover, a general framework for computations on the GPU, developed within this thesis, can also be utilized for other computationally-intensive tasks which do not necessarily originate from computer algebra. This, in particular, holds for various structured matrix problems arising in many theoretical and applied fields. Additionally, GPU-optimized modular arithmetic, can be found handy, for instance, in cryptography applications. The GPU algorithms have been implemented using CUDA framework (CUD10), and finally integrated in CGAL library.[1] To directly profit from the GPU implementation, we have designed novel algorithms for the solution of a system of bivariate polynomial equations (BES11), topology computation of algebraic curves (BEKS11a) and curve visualization (EBS09). Here, the main challenge was to restrict the set of required symbolic operations to those provided by the GPU while, at the same time, guarantee the correctness of the results. Our benchmarks confirm that, presently, we offer the best solutions, both in terms of accuracy and the running time, for these fundamental problems; see Sections 5.1.4 and 5.3.3 for comparison with other state-of-the-art algorithms. Besides, the detailed complexity analysis shows that our approach for the the solution of a system of polynomial equations is also the best in terms of the asymptotic complexity known for this problem so far; see Sections 5.2.1 and 5.2.2.

1.2 Related work

Unlike computer graphics, image processing or simulation, symbolic computing is a relatively new application domain of graphics accelerators and, at the time of writing, only a few results have been reported in the literature.

In (MP10), an algorithm was proposed to multiplying the polynomials using the FFT (Fast Fourier Transform) over a finite field on the GPU. The paper discusses a CUDA implementation of Cooley-Tukey and Stockham FFT algorithms which mainly differ in the specifics of data movements performed between the FFT stages. In this sense, Stockham's FFT is more preferable for the GPU realization. Our impression is that the realization of both FFT variants suffers a lot from the fact that that authors have considered only radix-2 transforms which result in low arithmetic intensity of computations, and have not decomposed the problem in a way most suitable for GPU processing. For example, Stockam's algorithm is realized in three kernel calls Stockham's algorithm is realized in three steps each of which is a sequence of calls to a particular GPU kernel. The first step performs only matrix transpositions. in the second one the results are scaled by twiddle factors and, in the last step, a set of primitive radix-2 "butterflies" are computed. As a result, the first kernel only performs memory operations, while the arithmetic intensity of the others two is far too low to amortize the cost of expensive memory operations. Instead, it would make sense to consider the FFTs of higher radices (8 or even 16) to provide reasonable thread workload and use a hierarchical FFT approach as discussed in Section 4.2.4. As a last remark, the paper only deals with the polynomial multiplication in a prime field, while the techniques to extend the algorithm for integer polynomials, such as binary segmentation or Chinese remaindering, are not considered.

[1] Computational Geometry Algorithms Library, www.cgal.org.

In his PhD thesis (Pan10, Chapter 6), W. Pan discusses the realization of Brown's subresultant PRS algorithm (Bro78) on the GPU. Unfortunately, similar to the previous work, his approach is restricted to a prime field, and therefore cannot be taken as a complete solution. Yet, the undoubted advantage of this algorithm is that it can handle *multivariate* polynomials through Kronecker substitution reducing the problem to the bivariate domain where, afterwards, a usual evaluation-interpolation scheme is employed. The interpolation is realized in terms of finite-field unidimensional FFTs developed in (MP10). The benefits of using the FFT relate to the fact that both evaluation and interpolation are essentially realized by the same procedure. However, the FFT-based solution induces an extra processing overhead which may not pay off for small inputs. Furthermore, it requires a very delicate treatment of "unlucky" homomorphisms (see Section 2.3.1 for definition).To alleviate this problem, W. Pan proposes to use random translations of the inputs, and provides detailed analysis under which conditions this operation succeeds. The core of the algorithm is the computation of pseudo-remainders on the GPU, controlled from the host machine. Again, we find that the main weakness of the algorithm is inadequate decomposition of the problem into primitive tasks. First, a GPU kernel performs one step of polynomial pseudo-division realized in just several arithmetic operations making the overall performance memory-bound. The second drawback of this solution is the need for too much control from the host side which causes additional global memory traffic and many kernel calls.

Another interesting work appears in (SS10) which deals with the resultant computation of bivariate polynomials on the GPU using Collins' algorithm (Col71). The algorithm realizes modular reduction as well as reconstruction of long integers entirely on the graphics card which could be of large technical hurdle. To compute univariate resultants, the authors propose to use a division-free PRS algorithm (Polynomial Remainder Sequences) which, in our opinion, is not a very suitable approach for parallelization. Unfortunately, the lack of further details does not allow us to reason about the efficiency of the implementation. For polynomial interpolation, the authors rely on recurrences to compute Newton polynomial bases, and then convert the resulting polynomial to monomial bases. From our perspective, this solution is somewhat "over-engineered" because it involves operations in a polynomial domain while, for comparison, our matrix-based approach is much easier to realize on the GPU. As reported in the paper, the algorithm restricts the maximal resultant degree to 512 and the same limit applies to the number of primes available for computations. The attained speed-up is about 60x over the resultant algorithm from Mathematica 6. In summary, the overall result makes a good impression of the focused effort to speed-up symbolic computations on the GPU and brings in some interesting ideas.

In (Fuj09), an algorithm was proposed to computing a GCD of fixed-length integers (1024 bits) on the graphics hardware, where the main application domain is a cryptography. Despite the fact that, this problem is not directly related to symbolic computations, it might still be worth an attention because polynomials can be mapped to large integers using Kronecker substitution (heuristic GCD algorithm). The author adopts a binary GCD algorithm which is exclusively based on shift-and-add operations, and thus does not use integer divisions which have no hardware support on the modern GPUs. The algorithm maps the computation of eight 1024-bit integer GCDs to one CUDA block. The realization requires some effort to implement the relevant operations on multi-precision integers where, seemingly, the main challenge lies in the implementation of parallel integer addition/subtraction. The author proposes to use a “carry skip method” to limit carry/borrow propagations. In summary, the paper demonstrates solid GPU programming techniques and discusses some ideas to improve the performance. The maximal speed-up of 11x over a CPU-based approach is attained when the number of computed GCDs is on the order of several thousand.[1] Yet, the main shortcoming of the paper is that no attempt has been made to extend the algorithm to handle larger integers.

From the above overview, we see that there have been several efforts to accelerate computations with polynomials and large integers on the graphics processor. However, neither of the above approaches is generic enough to be used in place of a corresponding CPU-based algorithm: either because only the *part* of the actual algorithm is realized on the GPU or the range of the inputs is subject to very strict limitations dictated by the graphics hardware. Thus, what is really missing, is a complete general approach and a set of useful subroutines that can be readily utilized to build a symbolic algorithm on the GPU “from ground up” since, in the long run, all such algorithms have a lot in common. In the present work, we shall try to achieve this goal.

However, we are not aiming at simply “showcasing” the performance of our approach on synthetic benchmarks. Thus, the further goal of this work is to develop actual algorithms which can take advantage of the GPU parallel processing. As noted earlier, we particularly concentrate on two fundamental problems from real algebraic geometry: namely, the solution of a system of polynomial equations and visualization of algebraic curves. The related works on these problems will be discussed separately in Sections 5.1.1 and 5.3.1, respectively. Additionally, we give a concise overview of an algorithm for the topology computation of real algebraic curves which is built on top of the above mentioned approach for the solution of systems of polynomial equations, and hence can also profit from the GPU acceleration. A detailed description and relevant benchmarks for the latter algorithm can be found in (BEKS11a, BEKS11b).

The rest of this thesis is structured as follows. In Chapter 1, we introduce a detailed mathematical machinery to deal with symbolic computations in computer environment and analyze the complexity of modular GCD and resultant algorithms. In Chapter 3 we present a theory of structured matrices which serves as a basis for efficient computation with polynomials on the graphics processor. Besides, we show that, with the help of linear algebra, we can improve upon the parallel complexity of polynomial algorithms by carefully exploiting data-level parallelism inherent in matrix computations. Chapter 4 begins with the introduction of the GPU architecture and CUDA programming model, and discusses the main aspects of the realization. In this chapter, we cover many topics spanning from parallel programming techniques, implementation of modular arithmetic to the realization of complete algorithms and the ways of exploiting block-level parallelism on the GPU. Finally, in Chapter 5 we consider two important applications of the developed algorithms: the solution of systems of polynomial equations and visualization of algebraic curves. In Chapter 6 we make some concluding remarks and sketch possible directions for future work.

[1] Unfortunately, the author did not mention what software was used for comparison.

2 Background

In this chapter, we establish the basic mathematical framework for symbolic computations and give an in-depth overview of some fundamental algorithms for polynomials. Additional information on this topic can be found in many textbooks on algorithms, computer algebra or algebraic geometry; for example, see (GCL92, vzGG03, Coh03, CLO98, Yap00, Knu97). This chapter has the following outline. We begin with the introduction of algebraic domains and some useful notation used throughout this thesis. Then, we turn our discussion to the modular techniques which is one of the most efficient ways to manipulate symbolic expressions in a computer environment. First, a general homomorphism approach will be presented along with the classical algorithms for polynomial interpolation and Chinese remaindering. Having all necessary prerequisites, we then discuss the concrete modular algorithms to computing a greatest common divisor (GCD) and a resultant of two polynomials. The latter algorithms constitute the main theoretical background of this work. At the end, the model of computation will be given to analyze the complexity of modular algorithms where we target both sequential and parallel platforms. We compare the derived complexity bounds with those known classical algorithms that are not based on homomorphism approach. Eventually, we shall see that the modular techniques is indeed a very powerful tool to speed-up symbolic computations.

2.1 Elementary concepts and computation with polynomials

2.1.1 Integral domains

Let $\mathbb{D}$ be an integral domain which is a commutative ring with unity that has no zero divisors.

Definition 2.1.1. For $a, b \in \mathbb{D}$, a is said to be a *divisor* of b if, for some $x \in \mathbb{D}$, it holds that $b = ax$. In this case, we write: $b \mid a$. •

In an integral domain, divisors of unity are called *units*. The elements $a, b \in \mathbb{D}$ are *associates* if $a \mid b$ and $b \mid a$. An element of $\mathbb{D}$ is called *irreducible* if its only divisors are units and associates.

Definition 2.1.2. For $a, b \in \mathbb{D}$, a *greatest common divisor* (GCD) of a and b denoted by $\gcd(a, b)$ is a largest possible $c \in \mathbb{D}$, such that $c \mid a$ and $c \mid b$. In other words, any other common divisor of a and b divides c. •

We call two elements $a, b \in \mathbb{D}$ relatively prime if $\gcd(a, b) = 1$. The next definition refines the notion of an integral domain.

Definition 2.1.3. A *Unique Factorization Domain* (UFD) is an integral domain $\mathbb{D}$ with an additional property that any non-zero $a \in \mathbb{D}$ is either unit or can be expressed as a finite product of irreducible elements: $a = p_1 p_2 \cdots p_n$, and this factorization is unique up to associates and reordering. •

An important property of a UFD is that, for any two elements a and b which are not zero at the same time, $\gcd(a, b)$ always exists, and it is unique up to associates. Examples of UFDs are the domain of integers $\mathbb{Z}$ whose only units are 1 and -1 as well as an arbitrary field (such as $\mathbb{Q}$ or $\mathbb{R}$) because all non-zero elements of a field are unit. Yet, the algebraic structure of a UFD alone only states the existence of the GCD but does not provide us with the way to compute it. For this purpose, we define a special function called *valuation* leading to further refinement of an integral domain:

Definition 2.1.4. A *Euclidean domain* is an integral domain $\mathbb{D}$ with valuation $v : \mathbb{D} \to \mathbb{N} \cup \{-\infty\}$ having the following properties. For $a, b \in \mathbb{D}$, it holds that:

$$\begin{aligned}
&v(a) = -\infty \textit{ if and only if } a = 0;\\
&v(ab) = v(a) + v(b) \geq \min(a, b);\\
&\textit{if } b \neq 0\textit{, there exist elements } q, r \in \mathbb{D} \textit{ such that } a = bq + r,\\
&\qquad \textit{where either } r = 0 \textit{ or } v(r) < v(b).
\end{aligned}$$

•

In the above definition, q is called a *quotient* of a divided by b which we denote by $\operatorname{quo}(a, b)$, while r is a *remainder* or $\operatorname{rem}(a, b)$. Then, the classical Euclidean algorithm to computing a GCD is a consequence of the following theorem:

Theorem 2.1.1: Let $\mathbb{D}$ be a Euclidean domain and elements $a, b, q, r \in \mathbb{D}$ ($b \neq 0$) satisfying: $a = bq + r$, with $r = 0$ or $v(r) < v(b)$, then $\gcd(a, b) = \gcd(b, r)$. See (GCL92, Theorem 2.3). ◊

Since a field $\mathbb{F}$ is by definition a commutative ring, where all non-zero elements are unit, $\mathbb{F}$ is also a Euclidean domain with the trivial valuation $v(a) = 1$ for all $a \in \mathbb{F} \setminus 0$. Another example of a Euclidean domain is a ring of integers $\mathbb{Z}$, where, for $a \in \mathbb{Z} \setminus 0$, the valuation is defined as $v(a) = |a|$. In addition, remark that any Euclidean domain is always a UFD but the converse is not necessarily true. Some examples will follow in the next section after we introduce the notion of a polynomial domain.

2.1.2 Polynomials

By $\mathbb{D}[x]$ we denote a *univariate polynomial domain* or the set of univariate polynomials $f(x)$ in the indeterminate x with coefficients f_k in some integral domain $\mathbb{D}$. A non-zero polynomial is defined as:

$$f(x) = \sum_{k=0}^{n} f_k x^k, \text{ with } f_n \neq 0 \text{ and } f_k \in \mathbb{D},$$

where each non-zero term $f_k x^k$ is called a *monomial* of f. The *degree* $\deg(f)$ of f is the largest integer n such that $f_n \neq 0$. An exception is a *zero polynomial* for which $f_k = 0$ for all k which is written in standard form as 0, and $\deg(0) = -\infty$. For a non-zero polynomial, f_n is called a *leading coefficient* denoted by $\mathrm{lcf}(f)$, while f_0 is a *constant term*. Given the smallest integer l for which $f_l \neq 0$, f_l is called a *trailing coefficient*. A polynomial is *monic* if its leading coefficient equals to 1.

By Gauss' lemma, if $\mathbb{D}$ is a UFD, then $\mathbb{D}[x]$ is also a UFD. Accordingly, the irreducible elements in $\mathbb{D}[x]$ are those which cannot be factored with respect to the coefficient domain $\mathbb{D}$. As noted earlier, a UFD is not necessarily a Euclidean domain: one example is the ring of polynomials $\mathbb{Z}[x]$. Indeed, although $\mathbb{Z}[x]$ is a UFD, the last property of Definition 2.1.4 does not hold for $\mathbb{Z}[x]$. However, for a field $\mathbb{F}$, $\mathbb{F}[x]$ is a Euclidean domain with valuation: $v(f(x)) = \deg(f)$. Using Definition 2.1.4, it can be easily verified for $f, g \in \mathbb{F}[x]$ that:

$$\deg(\mathrm{quo}(f,g)) = \deg(f) - \deg(g) \text{ if } \deg(f) \geq \deg(g) \text{ and } -\infty \text{ otherwise,}$$
$$\deg(\mathrm{rem}(f,g)) < \min(\deg(f), \deg(g)).$$

For a UFD $\mathbb{D}$, a polynomial $f \in \mathbb{D}[x]$ is said to be *square-free* if it has no square factors. In other words, there exist no such $g \in \mathbb{D}[x]$ that $g^2 \mid f$. A square-free part f^* of f can be extracted as follows: $f^* = f / \gcd(f, f')$. Indeed, suppose $f = h \cdot g^k$ for some $k \geq 2$ and g does not divide h, then by the product rule it holds that:

$$f' = kg^{k-1}kh + g^k h', \text{ therefore } \gcd(f, f') = g^{k-1} \text{ and } f^* = h \cdot g.$$

Square-free polynomials play an important role in many symbolic algorithms. The next definition is important to be able to compute the GCD in non-Euclidean domains.

Definition 2.1.5. For a UFD $\mathbb{D}$, the *content* of a non-zero polynomial $f \in \mathbb{D}[x]$, denoted by $\mathrm{cont}(f)$, is defined as a GCD of the coefficients of f. Moreover, f can be written in the form: $f = \mathrm{cont}(f) \cdot \mathrm{pp}(f)$, where $\mathrm{pp}(f) \in \mathbb{D}[x]$ is a *primitive part* of f. •

For convenience, we define $\mathrm{cont}(0) = 0$ and $\mathrm{pp}(0) = 0$. A polynomial f is called *primitive* if its non-zero coefficients are relatively prime, i.e., $\mathrm{cont}(f) = 1$. In particular, all polynomials over a field $\mathbb{F}$ are primitive.

Definition 2.1.6. For an integral domain $\mathbb{D}$, let $\mathbb{F}$ be the *fraction field* of $\mathbb{D}$, in other words, a smallest field such that $\mathbb{D} \subset \mathbb{F}$.[1] Then, for a polynomial $f \in \mathbb{D}[x]$ and a fixed $a \in \mathbb{F}$ we define the *polynomial evaluation map* $\rho_a : \mathbb{D}[x] \to \mathbb{F}$ as

$$\rho_a(f) = f(a), \; \textit{where } f(a) = \sum\nolimits_{i=0}^{\deg(f)} f_i a^i. \qquad \bullet$$

Using the evaluation map, we can now define a *root* of a polynomial as follows. Let $\overline{\mathbb{F}}$ be an algebraic closure of $\mathbb{F}$. A root α of a polynomial $f \in \mathbb{F}[x]$ (or $f \in \mathbb{D}[x]$ where $\mathbb{D} \subset \mathbb{F}$) is an element of $\overline{\mathbb{F}}$ such that $f(\alpha) = 0$. In particular, by the Fundamental Theorem of Algebra, any polynomial $f \in \mathbb{D}[x]$ can be expressed as:

$$f(x) = \mathrm{lcf}(f) \prod\nolimits_{i=1}^{k} (x - \alpha_i)^{e_i}, \text{ with } \alpha_i \in \overline{\mathbb{F}} \text{ and } \sum\nolimits_{i=1}^{k} e_i = \deg(f),$$

[1] To remove any confusions, by $\subset$ we denote a *set* inclusion not a *class* inclusion because a field, having more algebraic structure, can be attributed as a subclass of an integral domain.

where $e_i \geq 1$ is called the *multiplicity* of a root α_i.

The above definitions can be naturally extended to a *multivariate* polynomial domain denoted by $\mathbb{D}[x_1, \ldots, x_d]$ or $\mathbb{D}[\mathbf{x}]$, where $\mathbf{x}$ is a vector of indeterminates $\mathbf{x} = (x_1, \ldots, x_d)$ and $d \geq 1$. Let $\mathbf{c} = (c_1, \ldots, c_d) \in \mathbb{N}^d$ be the *exponent vector*,[1] then each element f of $\mathbb{D}[\mathbf{x}]$ is a finite sum of the following form:

$$f(\mathbf{x}) = \sum_{\mathbf{c} \in \mathbb{N}^d} f_{\mathbf{c}} \mathbf{x}^{\mathbf{c}}, \text{ where } f_{\mathbf{c}} = f_{(c_1,\ldots,c_d)} \in \mathbb{D}, \;\; \mathbf{x}^{\mathbf{c}} = x_1^{c_1} \cdots x_d^{c_d}. \tag{2.1}$$

Using this representation, it is common to say that $f_{\mathbf{c}}$'s are *scalar* coefficients of $f(\mathbf{x})$ defined over an integral domain $\mathbb{D}$. Similarly, the content of a non-zero polynomial $f \in \mathbb{D}[\mathbf{x}]$ is the GCD of *scalar* coefficients of f, and the primitive part of f is defined by analogy to univariate case. If all $f_{\mathbf{c}}$ in the above sum are zero, (2.1) defines a zero polynomial, denoted by 0. For each monomial $f_{\mathbf{c}}\mathbf{x}^{\mathbf{c}}$ of f, we define the *total degree* as $\sum_{i=1}^{d} c_i$, and the total degree of f, denoted by $\deg(f)$, is the maximum total degree of its monomials. To give an example, a *bivariate* polynomial $f \in \mathbb{Z}[x, y]$ of total degree 9 can be of the following form:

$$f(x, y) = -y^7x^2 + 2xy^5 - (8x^3 + 11x^2)y^3 + (3x^5 + 8)y^2 - 16x^3y + x^6 - 1.$$

A polynomial $f \in \mathbb{D}[\mathbf{x}]$ is called *homogeneous* if each of its monomials has the same total degree. The next definition is required if we wish to apply symbolic algorithms to multivariate polynomials.

Definition 2.1.7. Let $\mathbf{u} = (u_1, \ldots, u_d)$ and $\mathbf{v} = (v_1, \ldots, v_d)$ be two elements of $\mathbb{N}^d$. Then, the *lexicographical ordering* of exponent vectors is defined as follows: $\mathbf{u} = \mathbf{v}$ if $u_i = v_i$ for all $i = 1, \ldots, d$; otherwise let j be the smallest index for which $u_j \neq v_j$, then $\mathbf{u} < \mathbf{v}$ if $u_j < v_j$, and $\mathbf{u} > \mathbf{v}$ otherwise. •

Using Definition 2.1.7, we can generalize the notion of leading/trailing coefficients to polynomials in $\mathbb{D}[\mathbf{x}]$. Suppose, the coefficients of $f \in \mathbb{D}[\mathbf{x}]$ are arranged in lexicographically increasing order of their exponent vectors. Then, the coefficient of the first term (of the monomial with the maximal total degree) is a leading coefficient, while the coefficient of the last one is a trailing coefficient.

Definition 2.1.8. For a polynomial $f \in \mathbb{D}[\mathbf{x}]$, by $|f|_k$, where $k \in \mathbb{N} \cup \{\infty\}$, we define the *k-norm* on f as a function $\mathbb{D}[\mathbf{x}] \to \mathbb{R}$:

$$|f|_k = \Big(\sum\nolimits_{\mathbf{c} \in \mathbb{N}^d} |f_{\mathbf{c}}|^k\Big)^{1/k} \;\; \textit{and} \;\; |f|_\infty = \max_{\mathbf{c} \in \mathbb{N}^d} |f_{\mathbf{c}}|, \;\; \textit{where} \;\; f = \sum\nolimits_{\mathbf{c} \in \mathbb{N}^d} f_{\mathbf{c}}\mathbf{x}^{\mathbf{c}}.$$ •

In particular, $|f|_2$ is often referred to as a *Euclidean* norm, while $|f|_\infty$ is called the *height* of f. We shall use the norms on polynomials later when we talk about modular computations.

Multivariate polynomials are usually expressed in two ways: using *distributive* (2.1) and *recursive* representations. In the latter case, one indeterminate variable is declared

[1] Here, $\mathbb{N}^d$ denote a Cartesian product of d copies of $\mathbb{N}$.

Algorithm 2.1 Extended Euclidean algorithm

```
procedure EGCD($a, b \in \mathbb{D}$)    ▷ computes $c_1, c_2 \in \mathbb{D}$, s.t., $a \cdot c_1 + b \cdot c_2 = \gcd(a, b)$
    $c \leftarrow a,\ d \leftarrow b$
    $c_1 \leftarrow 1,\ d_1 \leftarrow 0$
    $c_2 \leftarrow 0,\ d_2 \leftarrow 1$
    while $d \neq 0$ do
        $q \leftarrow \mathrm{quo}(c, d)$
        $r \leftarrow c - q \cdot d,\ c \leftarrow d,\ d \leftarrow r$
        $r_1 \leftarrow c_1 - q \cdot d_1,\ c_1 \leftarrow d_1,\ d_1 \leftarrow r_1$
        $r_2 \leftarrow c_2 - q \cdot d_2,\ c_2 \leftarrow d_2,\ d_2 \leftarrow r_2$
    od
    return ($c,\ c_1,\ c_2$)    ▷ returns $c := \gcd(a, b)$, $c_1$ and $c_2$
end procedure
```

outermost while the rest are hidden in the coefficient domain. In other words, a polynomial can be considered as an element of $\mathbb{D}[x_2, \ldots, x_d][x_1]$ since the latter one is naturally isomorphic to $\mathbb{D}[x_1, x_2, \ldots, x_d]$:

$$f(x_1, \ldots, x_d) = \sum_{i=0}^{\deg_{x_1}(f)} f_i(x_2, \ldots, x_d)x_1^i, \text{ where } f_i(x_2, \ldots, x_d) \in \mathbb{D}[x_2, \ldots, x_d]. \qquad (2.2)$$

Similarly, the coefficients $f_i(\ldots)$ can be rewritten in the same manner by declaring another indeterminate variable, for example x_2, outermost. Accordingly, $\deg_{x_i}(f)$ is the degree of f considered as univariate polynomial with coefficients in $\mathbb{D}[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d]$. It is easy to see that $\deg_{x_i}(f) < \deg(f)$. For multivariate polynomial domains, if $\mathbb{D}$ is a UFD, then $\mathbb{D}[\mathbf{x}]$ is also a UFD. However, for a field $\mathbb{F}$, $\mathbb{F}[\mathbf{x}]$ is a UFD but *not* a Euclidean domain if the number of indeterminates is *greater* than one. To illustrate this, consider a bivariate polynomial domain $\mathbb{Z}_m[x, y]$ defined over a finite field $\mathbb{Z}_m$. Using recursive representation, we can treat each element $f \in \mathbb{Z}_m[x, y]$ as a univariate polynomial with coefficients in $\mathbb{Z}_m[x]$. Now, for f written in such a way, we see that the third property in Definition 2.1.4 is *violated* because, otherwise, we would be dealing with general *rational functions* instead of polynomials. This, in turn, implies that $\mathbb{Z}_m[x, y]$ is not a Euclidean domain.

2.1.3 Classical GCD algorithms

We now consider some classical algorithms to computing a GCD. Recall that, if $\mathbb{D}$ is a Euclidean domain, application of Theorem 2.1.1 results in a classical Euclidean scheme for GCD computations, see (GCL92, Algorithm 2.1). Its further development, known as Extended Euclidean Algorithm or EGCD, in addition to GCD, also computes such $s, t \in \mathbb{D}$ that for any two elements $a, b \in \mathbb{D}$ it holds that: $\gcd(a, b) = as + bt$. One important application of EGCD is computing a *modular inverse* in a finite field. Indeed, suppose a and b are relatively prime, then $as+bt = 1$, and hence s is a modular inverse of a modulo b, see also Section 2.2.2. The pseudocode for EGCD algorithm is given in Algorithm 2.1, see also (GCL92, Algorithm 2.2), (Knu97, p. 342). Note that the Euclidean algorithm as well as its extension can be applied equally well to univariate polynomials with coefficients in a field $\mathbb{F}$ since $\mathbb{F}[x]$ is a Euclidean domain.

Algorithm 2.2 Primitive GCD algorithm in a UFD $\mathbb{D}[x]$

```
 1: procedure PGCD(f, g ∈ $\mathbb{D}$[x])                     ▹ computes gcd(f, g) ∈ $\mathbb{D}$[x]
 2:     if f = 0 then return g                                ▹ handle special cases
 3:     elif g = 0 then return f fi
 4:     a ← pp(f), b ← pp(g)
 5:     if deg(a) < deg(b) then swap(a, b) fi                 ▹ ensure that deg(a) ≥ deg(b)
 6:     while b ≠ 0 do
 7:         r ← prem(a, b)
 8:         a ← b, b ← pp(r)                ▹ extract primitive part of the pseudo-remainder
 9:     od
10:     a ← a · gcd(cont(a), cont(b))     ▹ multiply by a content gcd in the coefficient domain
11:     return (a)
12: end procedure
```

Naturally, we aim to extend the above algorithms to polynomials with coefficients over a UFD which would also enable us to compute a GCD in multivariate domains. In a UFD, it is usually impossible to carry out the division process (unless the divisor is monic), and hence the Euclidean algorithm does not work. Instead, we can rely on the process of *pseudo-division* formalized in the following definition:

Definition 2.1.9. Given a polynomial domain $\mathbb{D}[x]$ over a UFD $\mathbb{D}$, then for all $f, g \in \mathbb{D}[x]$ with $g \neq 0$ and $\deg(f) \geq \deg(g)$, there exist polynomials $q, r \in \mathbb{D}[x]$ satisfying:

$$\mathrm{lcf}(g)^{\delta+1} f = g \cdot q + r, \quad \text{where} \quad \deg(r) < \deg(g),\ \delta = \deg(f) - \deg(g).$$

The polynomials r and g are called *pseudo-quotient* and *pseudo-remainder* denoted by $\mathrm{pquo}(f, g)$ and $\mathrm{prem}(f, g)$, respectively. •

The next theorem is the basis for GCD computations in non-Euclidean domains:

Theorem 2.1.2: Let $\mathbb{D}[x]$ be a polynomial domain over a UFD $\mathbb{D}$. Then, for $f, g \in \mathbb{D}[x]$ with $g \neq 0$ and $\deg(f) \geq \deg(g)$, it follows that: $\gcd(f, g) = \gcd(g, \mathrm{pp}(r))$, where $r = \mathrm{prem}(f, g)$ and $\mathrm{pp}(r)$ is a primitive part of r. See (GCL92, Theorem 2.10). ◊

The above theorem leads us to Algorithm 2.2, known as *primitive GCD* algorithm, which works over a UFD $\mathbb{D}$. Important is the fact that this algorithm can be applied equally well to multivariate polynomials (defined over some UFD). Unfortunately, one of its main drawbacks is the need for computing a primitive part in each step (line 8) which progressively becomes harder as the coefficients of 'a' grow. In fact, there is a common problem shared by all symbolic algorithms, known as *expression swell*, where the intermediate results of computations tend to grow exponentially in size while the final result is relatively small. Besides, performing the arithmetic operations in a polynomial domain is inefficient in the computer environment since these operations are not natively supported on a computer platform. The tools introduced in the next section are supposed to overcome these difficulties.

2.2 Modular techniques

The modular or homomorphism approach is a traditional way to avoid computational problems associated with symbolic algorithms. Although, in most general form, it can be formulated for polynomials in $\mathbb{D}[x_1, \ldots, x_d]$ for some integral domain $\mathbb{D}$, we focus our attention on a special case where polynomials are defined over the domain of integers $\mathbb{Z}$ since, in most situations, one can reduce a problem at hand to an equivalent problem in $\mathbb{Z}[x_1, \ldots, x_d]$. A modular approach originates in the works of Collins (Col71), which investigates in computing resultants of multivariate polynomials, and Brown (Bro71, Bro78) dealing with multivariate GCD computations and subresultants.

In vague terms, this method decomposes a problem stated in one algebraic domain into a set of similar problems over a (much simpler) domain. For instance, performing computations in a finite field $\mathbb{Z}_m$ for a small prime m is a lot more efficient than working with arbitrary large integers in $\mathbb{Z}$ because multi-precision arithmetic is not natively supported by the computer hardware. In contrast, elements of $\mathbb{Z}_m$ can be operated upon as single-precision integers if the prime m fits in a machine word. In addition, with finite field arithmetic we can prevent the coefficient growth of intermediate results.

Certainly, we must concern ourselves with how to recover a solution in the original domain given the set of "homomorphic images". This is a task of Chinese remaindering and interpolation algorithms which we discuss in this section in detail.

2.2.1 Homomorphisms

We begin with the definition of a *ring morphism*, that is a function between two rings which preserves the operations of addition and multiplication.

Definition 2.2.1. Let R and R' be two rings. A mapping $\phi : R \to R'$ is called a *ring morphism* if

$$\begin{aligned} &\phi(a+b) = \phi(a) + \phi(b) \ \ \textit{for all} \ \ a, b \in R, \\ &\phi(ab) = \phi(a) \cdot \phi(b) \ \ \textit{for all} \ \ a, b \in R, \\ &\phi(1) = 1. \end{aligned} \qquad \bullet$$

When the defining function is *surjective* (i.e. onto), $\phi : R \to R'$ is called an *epimorphism*. Such type of morphisms is especially interesting since it allows us to reconstruct the preimage of an operation under certain conditions. Throughout the discussion, we shall agree to use the term *homomorphism* identifying it with epimorphism which is a common practice in mathematical literature.[1] Adopting this terminology, R' is then called a *homomorphic image* of R.

The two types of homomorphisms introduced below are of primary importance for efficient symbolic computations. For polynomials defined over the ring of integers $\mathbb{Z}$, a *modular homomorphism* is a map:

$$\phi_m : \mathbb{Z}[x_1, \ldots, x_d] \to \mathbb{Z}_m[x_1, \ldots, x_d],$$

where $m \in \mathbb{Z}$ is a fixed integer, usually chosen to be a prime number. In other words, a modular homomorphism is an operation where all scalar coefficients of $f \in \mathbb{Z}[x_1, \ldots, x_d]$ are reduced modulo m. For example, given a polynomial in $\mathbb{Z}[x, y]$:

$$f(x,y) = y^4 - 11y^3 + 212x^2y^2 - 3x^2y - x^4,$$

[1] To be precise, a homomorphism is simply a synonym for the term morphism, however it is often used referring particularly to *surjective* morphism or epimorphism.

then the result of applying modular homomorphism for $m = 7$ is:

$$\phi_7(f(x,y)) = y^4 + 3y^3 + 5x^2y^2 + 4x^2y + 6x^4.$$

Another type of homomorphism needed for the efficient computations with multivariate polynomials is an *evaluation homomorphism*:

$$\phi_{x_i-\alpha} : \mathbb{D}[x_1,\ldots,x_d] \to \mathbb{D}[x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d],$$

where $\alpha \in \mathbb{D}$ for some integral domain $\mathbb{D}$. In fact, the effect of $\phi_{x_i-\alpha}$ is nothing but evaluating f at a fixed element $\alpha \in \mathbb{D}$ which is often called an *evaluation point*:

$$\phi_{x_i-\alpha}(f(x_1,\ldots,x_d)) = f(x_1,\ldots,x_{i-1},\alpha,x_{i+1},\ldots,x_d).$$

Formally speaking, evaluation homomorphism can be regarded as computing a *residue* of $f \in \mathbb{D}[x_1,\ldots,x_d]$ "modulo $(x_i - \alpha) \in \mathbb{D}[x_i]$" which also explains the notation. To see this, let f be an element of $\mathbb{D}'[x_i]$ with coefficients in $\mathbb{D}' = \mathbb{D}[x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_d]$. Then, we can write the following identity:

$$f(x_i) - f(\alpha) + f(\alpha) = \sum\nolimits_{j>0} f_j(x_i^j - \alpha^j) + f(\alpha) \equiv f(\alpha) \bmod (x_i - \alpha),$$

since $x_i - \alpha$ divides $x_i^j - \alpha^j$ for $j > 0$. Together, the composition of modular and evaluation homomorphisms is a very powerful technique to facilitate symbolic operations. Yet, before developing the actual algorithms, we need to study how to "lift" the results back to the original domain after performing the computations on homomorphic images. This is the next step in our discussion.

2.2.2 Chinese remaindering

Since we identify a homomorphism with a surjective mapping, it is necessary to have several (different) homomorphic images to be able to uniquely reconstruct the result in the original domain. The idea of "inverting" modular homomorphisms is formalized in the theorem below. In what follows, we will adopt the standard notation that for any $a, b, m \in \mathbb{Z}$, $a \equiv b \pmod{m}$ is equivalent to writing $a = b + r \cdot m$ for some $r \in \mathbb{Z}$.

Theorem 2.2.1 (Chinese Remainder Theorem for integers): Let $m_1, m_2, \ldots, m_k \in \mathbb{Z}$ be pairwise relatively prime integers, that is, $\gcd(m_i, m_j) = 1$ for $i \neq j$, and let $r_i \in \mathbb{Z}_{m_i}$ be the set of corresponding residues $(1 \le i \le k)$. Then, there exists a unique $r \in \mathbb{Z}$ such that any integral solution $a \in \mathbb{Z}$ of the following congruence system:

$$a \equiv r_i \pmod{m_i} \quad (1 \le i \le k),$$

satisfies $a \equiv r \pmod{M}$, where $M = \prod_{i=1}^k m_i$ and $0 \le r < M$. ◇

The proof can be found in many sources: for instance, see (GCL92, Section 5.6), (Yap00, Lecture IV). In the above theorem, the set of residues $(r_1,\ldots,r_k)$ is often referred to as RNS (Residue Number System) representation of an integer r, while the product M is called a *dynamic range* of RNS defined by the moduli set $(m_1,\ldots,m_k)$.

Algorithm 2.3 Incremental Chinese remainder algorithm (CRA)

```
1: procedure CRA_INCREMENTAL($\{r_1,\dots,r_k\}, \{m_1,\dots,m_k\}$)   ▷ set of moduli $m_i \in \mathbb{Z}$ and residues: $r_i \in Z_{m_i}$
2:     $M \leftarrow m_1,\ r \leftarrow r_1$                           ▷ initially dynamic range is a single modulus
3:     for $i = 2$ to $k$ do                                           ▷ iterate over all residues
4:         $c \leftarrow M^{-1} \bmod m_i$
5:         $r' \leftarrow r \bmod m_i,\ s \leftarrow (r_i - r')c \bmod m_i$
6:         $r \leftarrow r + s \cdot M,\ M \leftarrow M \cdot m_i$     ▷ add next residue to the result
7:     od
8:     return $r$                                                      ▷ return r satisfying $0 \le r < M$
9: end procedure
```

Let us denote $s_i = M/m_i = \prod_{j=1,j\neq i}^{k} m_j$, and $s_i^* = s_i^{-1} \bmod m_i$, where s_i^{-1} is a *modular inverse* of s_i modulo m_i. As noted in Section 2.1.3, the modular inverse can be computed using the Extended Euclidean algorithm (see Algorithm 2.1 in Section 2.1.3). Thus, it can be immediately verified that a unique $r \in \mathbb{Z}$ satisfying the congruence relations of Theorem 2.2.1 is:

$$r = \left(\sum_{i=1}^{k} s_i(r_i s_i^* \bmod m_i)\right) \bmod M. \qquad (2.3)$$

The above formula can readily be used to recover the integer r. However, we would like to avoid computing a residue modulo M since M is usually a large number. The first solution to this problem is to apply the Chinese remaindering algorithm (CRA) *incrementally*, that is to add one residue at a time to the final product, as outlined by Algorithm 2.3. In practical applications, the moduli set is usually fixed, hence it is wise to precompute the numbers $m_1^{-1} \bmod m_2$, $(m_1m_2)^{-1} \bmod m_3$, etc., so that computing the modular inverse in line 4 of the algorithm can be avoided. A major strength of Algorithm 2.3 lies in the fact that we can monitor the progress by gradually expanding the dynamic range of RNS, and stop as soon as a result satisfies some predefined criterion. This serves as a basis for *probabilistic* modular algorithms, see (dKMW05, Mon05). On the other hand, Algorithm 2.3 extensively uses manipulations with large integers (lines 5–6) and is not suited for parallel implementation. Directly applying the formula (2.3), we can also develop a "divide-and-conquer" Chinese remainder algorithm. However, the latter algorithm requires a significant amount of precomputation work to be efficient, see (vzGG03, Alg. 10.22).

The next approach, called Mixed Radix Conversion (MRC), partially solves the above mentioned problems by gently decoupling the arithmetic operations in a field $\mathbb{Z}_m$ from the operations in $\mathbb{Z}$. In addition, the computations are arranged in a neatly structured way allowing for a parallel processing. The idea is to associate the "search-for" integer r with a set of *Mixed-radix* (MR) digits $\{\gamma_i\}$ as follows:

$$r = \gamma_1 M_1 + \gamma_2 M_2 + \dots + \gamma_k M_k,$$

where $M_1 = 1$, $M_i = m_1 m_2 \dots m_{i-1}$ ($i = 2,\dots,k$). The mixed-radix digits can be computed using the algorithm (Yas91) in the following way ($i = 1,\dots,N$):

$$\begin{aligned}
\gamma_1 &= r_1, \gamma_2 = (r_2 - \gamma_1)c_2 \bmod m_2,\\
\gamma_3 &= ((r_3 - \gamma_1)c_3 - (\gamma_2 M_2 c_3 \bmod m_3)) \bmod m_3, \ \dots\\
\gamma_i &= ((r_i - \gamma_1)c_i - (\gamma_2 M_2 c_i \bmod m_i) - \dots\\
&\qquad -(\gamma_{i-1} M_{i-1} c_i \bmod m_i)) \bmod m_i. \qquad (2.4)
\end{aligned}$$

Algorithm 2.4 Mixed Radix Conversion

```
procedure MRC_ALGORITHM($\{r_1, \dots, r_k\}$, $\{m_1, \dots, m_k\}$)   ▷ set of moduli $m_i \in \mathbb{Z}$ and residues: $r_i \in Z_{m_i}$
    $\gamma_1 \leftarrow r_1$   ▷ initialize the first MR digit
    for $i = 2$ to $k$ do
        $\gamma_i \leftarrow (r_i - \gamma_1)c_i \bmod m_i$   ▷ $c_i$'s should be precomputed
        $M_i \leftarrow m_1 c_i \bmod m_i$
    od
    for $i = 2$ to $k - 1$ do   ▷ in iteration i, the digits $(\gamma_1, \dots, \gamma_i)$ are already computed
        for $j = i + 1$ to $k$ do   ▷ update the MR digits $(\gamma_{i+1}, \dots, \gamma_k)$ using $\gamma_i$
            $\gamma_j \leftarrow (\gamma_j - \gamma_i M_j) \bmod m_j$
            $M_j \leftarrow M_j m_i \bmod m_j$   ▷ add the next modulus to each $M_j$
        od
    od
    return $(\gamma_1, \dots, \gamma_k)$   ▷ return a set of MR digits: $\gamma_i \in Z_{m_i}$
end procedure
```

where $c_i = (m_1 m_2 \dots m_{i-1})^{-1} \bmod m_i$. One can see that it is possible to extract some data-level parallelism from these computations. We shall exploit this fact later when we discuss the realization of Chinese remaindering on the graphics hardware, see Section 4.3.4. The serial algorithm to computing MR digits is given in Algorithm 2.4. Here, we assume that c_i's can be precomputed in advance, while the M_i's are computed iteratively. Observe that all the arithmetic operations are restricted to a finite field which is a main advantage of this algorithm. Having the set of MR digits computed, the associated integer r can be recovered by simply evaluating Horner's scheme on the digits:

$$r = \gamma_1 + m_1(\gamma_2 + m_2(\gamma_3 + m_3(\dots))).$$

2.2.3 Polynomial interpolation

We next consider the analogous process of inverting an evaluation homomorphism which is known as polynomial interpolation. For our purposes, it shall suffice to consider only a *univariate* interpolation problem where one homomorphism $\phi_{x_i-\alpha}$ is inverted at a time, e.g.:

$$\begin{tikzcd} \mathbb{D}[x_3] \arrow[r, "\phi^{-1}_{x_2-\alpha}"] & \mathbb{D}[x_2, x_3] \arrow[r, "\phi^{-1}_{x_1-\alpha}"] & \mathbb{D}[x_1, x_2, x_3]. \end{tikzcd}$$

Remark, however, that it is possible to compose several inversions together: this idea is exploited in sparse multivariate interpolation algorithms, see (BO88, ZV02). In fact, polynomial interpolation has a lot in common with the integer Chinese remaindering, if we recall that a homomorphism $\phi_{x_i-\alpha}$ is equivalent to computing the residue of a polynomial modulo $(x_i - \alpha)$. It can be seen as a special case of the following theorem.

Theorem 2.2.2 (Chinese Remainder Theorem for polynomials): Let $f_1, \dots, f_k \in \mathbb{F}[x]$ be pairwise relatively prime polynomials with coefficients in a field $\mathbb{F}$, and $g_1, \dots, g_k \in \mathbb{F}[x]$ be arbitrary polynomials. Then, there exists a unique $g \in \mathbb{F}[x]$ such that any solution $a \in \mathbb{F}[x]$ of the following system of congruences:

$$a \equiv g_i \pmod{f_i} \ \ (1 \le i \le k),$$

satisfies $a \equiv g \, (\bmod \ F)$ with $F = \prod_{i=1}^{k} f_i$ and $\deg(g) < \deg(F)$. See (Sho09, Thm 16.19).$\diamond$

Algorithm 2.5 Newton interpolation algorithm

```
 1: procedure NEWTON_INTERP({$\alpha_0, \ldots, \alpha_n$}, {$y_0, \ldots, y_n$})   ▹ set of distinct points $\alpha_i \in \mathbb{D}$ and values $y_i \in \mathbb{D}$
 2:     for i = 0 to n do
 3:         $f_{[i,i]} = y_i$                                                  ▹ initialization
 4:     od
 5:     for p = 1 to n do                                                ▹ compute Newton coefficients
 6:         for i = 0 to n − p do
 7:             $f_{[i,i+p]} = \left(f_{[i,i+p-1]} - f_{[i+1,i+p]}\right)/(\alpha_i - \alpha_{i+p})$
 8:         od
 9:     od
10:     $f \leftarrow f_{[0,n]}$                                           ▹ reconstruct the interpolating polynomial
11:     for i = n − 1 to 0 by −1 do
12:         $f \leftarrow f \cdot (x - \alpha_i) + f_{[0,i]}$
13:     od
14:     return f                                ▹ polynomial $f \in \mathrm{Quo}(\mathbb{D})[x]$, such that $f(\alpha_i) = y_i$
15: end procedure
```

Precisely, interpolation can be defined as follows: given $n+1$ evaluation points $\alpha_0, \ldots, \alpha_n \in \mathbb{D}$ and corresponding values $y_0, \ldots, y_n \in \mathbb{D}$, find a polynomial $f \in \mathbb{D}[x]$ of degree at most n, such that:

$$f(\alpha_i) = y_i \text{ for } 0 \le i \le n.$$

This, in turn, leads to a linear system with $n + 1$ equations and $n + 1$ unknowns:

$$V\mathbf{a} = \mathbf{y}, \text{ with } V_{ij} = \alpha_i^j \text{ for } i, j = 0, \ldots, n,$$

where $\mathbf{a}$ is a vector of polynomial coefficients and V is a *Vandermonde* matrix. Because the determinant of the Vandermonde matrix equals to: $\det V = \prod_{0 \le i < j \le n}(\alpha_j - \alpha_i)$, it follows that the interpolation problem has a solution if the elements $\{\alpha_i\}$ are pairwise distinct.

We now discuss the Newton interpolation algorithm which expresses the interpolating polynomial f in terms of divided differences (similar to mixed-radix representation):

$$f(x) = f_{[0,0]}(0) + f_{[0,1]}(x - \alpha_0) + \cdots + f_{[0,n]} \prod_{i=0}^{n-1}(x - \alpha_i),$$

where $f_{[i,j]} \in \mathrm{Quo}(\mathbb{D})$ are Newton coefficients or *divided differences*, and $\mathrm{Quo}(\mathbb{D})$ defines a quotient field of an integral domain $\mathbb{D}$. The divided differences can be recursively computed as follows (Knu97):

$$f_{[i,i+p]} = \left(f_{[i,i+p-1]} - f_{[i+1,i+p]}\right)/(\alpha_i - \alpha_{i+p}), \text{ and } f_{[i,i]} = y_i.$$

Using these recursions, one can compute the terms $f_{[i,j]}$ in a "triangle" fashion: that is, start from $f_{[0,0]}, f_{[1,1]}, f_{[2,2]}$, etc., then compute $f_{[0,1]}, f_{[1,2]}, f_{[2,3]}$ and so on, until $f_{[0,n]}$. The pseudocode for polynomial interpolation is outlined in Algorithm 2.5. The algorithm admits further optimizations: for example, we can rearrange the computations in such a way to reduce the number of divisions which is frequently used for interpolation in a finite field. A possible weakness of this method is that it relies on operations in a polynomial domain (line 12) to reconstruct the final result which is not always desirable. By analogy to the integer CRA, we can also process the evaluation points incrementally. Yet, this is not always efficient in practice as one needs to deal with inverses modulo a polynomial which are expensive to precompute because interpolation usually precedes integer

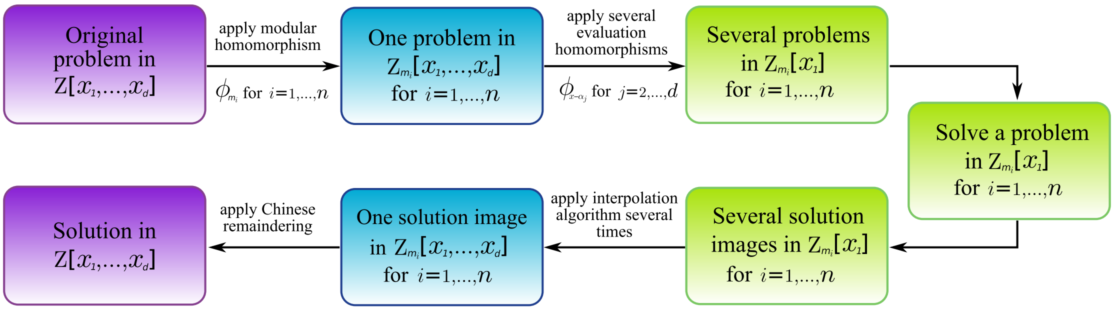
Original problem in Z[x_1,...,x_d]
apply modular homomorphism
ϕ_{m_i} for i=1,...,n
One problem in Z$_{m_i}$[x_1,...,x_d] for i=1,...,n
apply several evaluation homomorphisms
$\phi_{x-\alpha_j}$ for j=2,...,d
Several problems in Z$_{m_i}$[x_1] for i=1,...,n
Solve a problem in Z$_{m_i}$[x_1] for i=1,...,n
Several solution images in Z$_{m_i}$[x_1] for i=1,...,n
apply interpolation algorithm several times
One solution image in Z$_{m_i}$[x_1,...,x_d] for i=1,...,n
apply Chinese remaindering
Solution in Z[x_1,...,x_d]

Figure 2.1: Homomorphism diagram for the solution of a symbolic problem

Chinese remaindering. A “divide-and-conquer” version of the polynomial interpolation algorithm can be found in (vzGG03, Alg. 10.11).

Another alternative to Newton method is to directly solve the Vandermonde system. In that case, we would immediately obtain the coefficients of f without the need for polynomial arithmetic. In fact, as we see in Section 3.2.2, this idea leads to a rather efficient algorithm operating in $\mathbb{Z}_m$, provided that we can exploit the structure of the Vandermonde matrix.

2.2.4 Homomorphism diagram

Having discussed the main steps of the modular approach, it is illustrative to visualize a whole “solution route” using a diagram in Figure 2.1. Given a multivariate domain $\mathbb{Z}[x_1,\dots,x_d]$, we first apply a modular homomorphism for a set of moduli m_i projecting the original domain to $\mathbb{Z}_{m_i}[x_1,\dots,x_d]$ for $i = 1,\dots,n$. The actual number of moduli we need depends on a concrete problem at hand, it will be clarified later how to estimate it. Next, a sequence of evaluation homomorphisms is applied recursively further projecting $\mathbb{Z}_{m_i}[x_1,\dots,x_d]$ onto $Z_{m_i}[x_1]$. The latter one is a Euclidean domain since Z_{m_i} is a field. Clearly, the operations in $Z_{m_i}[x_1]$ can be performed efficiently by the computer. Once the problem is solved over $Z_{m_i}[x_1]$, we take the way back by lifting a solution first to $\mathbb{Z}_{m_i}[x_1,\dots,x_d]$ and eventually to $\mathbb{Z}[x_1,\dots,x_d]$ by applying polynomial interpolation and Chinese remaindering, respectively. Although, it might seem that we have taken a very long “path” to the solution, a homomorphism approach can substantially reduce the computational cost in many cases as we shall see in short.

However, the things are not as transparent with the homomorphism approach as they look like: there are important issues that we have not addressed yet. For example, it is not always easy to find the right number of homomorphic images while theoretical bounds could be quite pessimistic (if known at all) rendering the whole algorithm inefficient. Another problem relates to detecting the so-called “unlucky” homomorphisms which occur when the input polynomials (partially) lie in the kernel of a homomorphism transformation we have applied. Using such homomorphisms to recover a final solution can lead to incorrect results. These and related problems will be discussed in the following sections.

2.3 Modular GCD computations

We begin our study of modular algorithms with the GCD computation of two polynomials. The GCD has a fundamental importance in algebraic manipulation and appears as a subproblem of many sophisticated symbolic algorithms: for instance, to compute a square-free factorization of a polynomial,[1] for comparison of algebraic numbers or solving systems of non-linear (polynomial) equations.

2.3.1 Setting

Observe that, we are interested in a GCD algorithm which operates in some UFD $\mathbb{D}$ to ensure that this solution is applicable both for univariate and multivariate polynomial domains. We start with some theoretical considerations which form the basis for a modular GCD algorithm. To simplify the notation, we state the results for univariate polynomials with coefficients in a UFD $\mathbb{D}$. It should be noted that these results extend naturally to multivariate polynomials in $\mathbb{D}[\mathbf{x}]$ if we impose a lexicographical order of polynomial terms. However, observe that multivariate polynomials shall *not* be viewed in recursive representation. The following theorem provides the necessary conditions to computing a GCD using homomorphism approach:

Theorem 2.3.1: For UFDs $\mathbb{D}$ and $\mathbb{D}'$, let $\phi : \mathbb{D} \to \mathbb{D}'$ be a homomorphism of rings. This also induces a natural homomorphism from $\mathbb{D}[x]$ to $\mathbb{D}'[x]$. Suppose $f, g \in \mathbb{D}[x]$ and $h = \gcd(f, g)$ with $\phi(\mathrm{lcf}(h)) \neq 0$. Then, it holds that:

$$\deg(\gcd(\phi(f), \phi(g))) \geq \deg(\gcd(f, g)). \qquad \diamond$$

Proof Since h is a GCD, f and g can be written as: $f = p \cdot h$ and $g = q \cdot h$ for some $p, q \in \mathbb{D}[x]$. By definition, a ring morphism preserves multiplication, thus we have:

$$\phi(f) = \phi(p) \cdot \phi(h), \text{ and } \phi(g) = \phi(q) \cdot \phi(h),$$

which implies that $\phi(h)$ is a common factor of both $\phi(f)$ and $\phi(g)$. As a result, $\phi(h)$ must divide a GCD of homomorphic images. Finally, since $\phi(\mathrm{lcf}(h)) \neq 0$, we conclude that:

$$\deg(\gcd(\phi(f), \phi(g))) \geq \deg(\phi(h)) := \deg(\gcd(f, g)). \qquad \blacksquare$$

The condition $\phi(\mathrm{lcf}(h)) \neq 0$ can be verified by checking that the homomorphic images of f and g do not decrease in degree. In Theorem 2.3.1, those ϕ's for which $\deg(\gcd(\phi(f), \phi(g))) > \deg(\phi(h))$ are often referred to as *unlucky* homomorphisms. Apparently, such homomorphisms cannot be used to recover a GCD of the original polynomials. Yet, one can show that for specific $f, g \in \mathbb{D}[x]$, there is only a finite number of unlucky homomorphisms. For the time being, we assume that we can somehow identify and discard homomorphisms of this type. We return to this issue after presenting the overall approach. The modular algorithm (Bro71) relies on the fact that:

$$\phi(\gcd(f, g)) = c \cdot \gcd(\phi(f), \phi(g)), \text{ where } \phi : \mathbb{D} \to \mathbb{D}' \text{ and } c \in \mathbb{D}',$$

[1] For definition of a square-free polynomial, see Section 2.1.2.

provided that the condition of Theorem 2.3.1 is satisfied and ϕ is not an unlucky homomorphism. Here, the constant c is not a problem because we can extract the primitive parts $\tilde{f}$ and $\tilde{g}$ of f and g, compute a GCD $\tilde{h}$ of respective homomorphic images, and normalize the result to ensure that:

$$\mathrm{lcf}(\tilde{h}) = \phi(\gcd(\mathrm{lcf}(f), \mathrm{lcf}(g))) \text{ with } \tilde{h} = \gcd(\phi(\tilde{f}), \phi(\tilde{g})).$$

This can be easily achieved as we know that $\mathrm{lcf}(h) \mid \gcd(\mathrm{lcf}(f), \mathrm{lcf}(g))$, where $h = \gcd(f, g)$. Repeating this process for a number of homomorphic images, and then combining them together yields a polynomial whose primitive part is precisely $\gcd(\tilde{f}, \tilde{g})$, and the rest is obvious.

2.3.2 The algorithm

In what follows, we further restrict ourselves by assuming that polynomials are defined over the domain of integers $\mathbb{Z}$ since, as noted in Section 2.2, most practical problems, in the long run, can be reduced to computations with integer polynomials. For polynomials $f, g \in \mathbb{Z}[x_1, \ldots, x_d]$, the modular GCD algorithm proceeds with the following steps:

(a) calculate the number of homomorphisms required;

(b) find a homomorphism ϕ with $\deg(\phi(f)) = \deg(f)$ and $\deg(\phi(g)) = \deg(g)$; this is equivalent to: $\mathrm{lcf}(f), \mathrm{lcf}(g) \notin \ker\phi$;

(c) compute $\gcd(\phi(f), \phi(g))$ using the image algorithm;

(d) if the chosen homomorphism is not unlucky, add the image to the final result; otherwise, go back to step **(b)**;

(e) terminate the algorithm if the number of images suffices to reconstruct the result; otherwise, go back to step **(b)**.

If the algorithm is applied to univariate polynomials (that is, $d = 1$), it suffices to use only a modular homomorphism $\phi_m : \mathbb{Z} \to \mathbb{Z}_m$ for sufficiently many primes m. For multivariate polynomials, we again start with a modular homomorphism, and then, in step **(c)**, invoke the algorithm recursively using the set of evaluation homomorphisms until the problem is reduced to GCD computations in $\mathbb{Z}_m[x]$. In the latter case, we can either use the Euclidean scheme since $\mathbb{Z}_m[x]$ is a Euclidean domain or we can adapt the primitive GCD algorithm for reasons of efficiency. We will discuss this in Section 2.3.3 in greater detail.

We next turn to the question of how to choose the right number of homomorphic images. In case of *modular homomorphisms*, it depends on the size of the scalar coefficients of a GCD. For that, one can use the bounds on polynomial divisors because for $h = \gcd(f, g)$, it holds that $h \mid f$ and $h \mid g$. In essence, obtaining such bounds is a large topic on its own, and not much is known in the multivariate case. Here, we give only basic estimates and postpone the further discussion to Section 4.5, where we discuss the realization of a univariate GCD algorithm on the GPU. The following result is due to (Mig74) which was generalized to multivariate domains in (HS06, Cor04). For non-zero polynomials $f, h \in \mathbb{Z}[x_1, \ldots, x_d]$ of maximal degree n in each variable separately, with $h \mid f$, we have

$$|h|_\infty \leq 2^{(n+1)^d+1}|f|_2.$$

Naturally, since $\gcd(f, g)$ is the divisor of both f and g, we can apply the bound separately to both polynomials and pick up the minimal one. Still, this estimate is usually quite pessimistic unless we do some preprocessing of input polynomials. This is why, one usually invokes the algorithm *incrementally*. In other words, we keep on adding new images to the result $h \in \mathbb{Z}[\mathbf{x}]$ until the latter one does *not* change anymore from one image to the next. In all likelihood, this should happen as soon as h satisfies the following a-priory bound (GCL92, p.308):

$$|h|_\infty \leq 2^{\min(\deg(f),\deg(g))} \min(|f|_\infty, |g|_\infty)| \gcd(\mathrm{lcf}(f), \mathrm{lcf}(g))|.$$

When the above inequality holds, we finally check if $h \mid f$ and $h \mid g$ indicating that a GCD is computed. In the literature, this test is often referred to as *trial division*. It works very well in practice because it makes the algorithm output-sensitive. For *evaluation homomorphisms*, the number of images required depends on the degree of a GCD polynomial in each variable separately. These can be trivially estimated as:

$$\deg_{x_i}(h) \leq \min(\deg_{x_i}(f), \deg_{x_i}(g)), \text{ for } i = 1, \ldots, d$$

since a GCD must divide both polynomials irrespective of which x_i is chosen to be the outermost variable in the polynomial representation.

Now, it remains to take care of unfinished business with unlucky homomorphisms. In the original work (Bro71), it is argued that, for any input polynomials there are finitely many primes and evaluation points that induce unlucky homomorphisms, see also (Yap00, § 4). Furthermore, Brown shows that, for $f, g \in \mathbb{Z}[x_1, \ldots, x_d]$, the probability of a prime m being unlucky is bounded by d/m, see (Bro71, Section 4.4). Likewise, the probability that an evaluation point for the outermost variable x_d chosen at random from the elements of $\mathbb{Z}_m$ is unlucky is at most u/m, where

$$u \leq \max_{i=1}^{d-1}(\deg_{x_i}(f) + \deg_{x_i}(g)) \cdot \max(\deg_{x_d}(f), \deg_{x_d}(g)) \cdot (d-1),$$

see (Bro71, Section 4.6). For realization, one usually selects the primes that fit in a single machine word on the target architecture. Therefore, in practice, the probability of encountering an unlucky prime is always negligibly small. Simple calculations show that the same also holds for evaluation points. Assuming that unlucky homomorphisms occur rarely on the average, we can use the degree anomaly check based on Theorem 2.3.1 to identify them. More precisely, investigations in (Bro71) show that it suffices to keep those homomorphic images for which $\deg(\gcd(\phi(f), \phi(g)))$ is *minimal* and discard the remaining ones.

Example 2.3.1. To illustrate how this works, let us consider the GCD computation for polynomials

$$\begin{aligned} f = {} & -y^4 + (140x + 2)y^3 - (4900x^2 + 139x)y^2 - (140x^3 + 142x + 2)y + \\ & 4900x^3 + 5040x^2 + 141x + 1, \text{ and} \\ g = {} & -2y^3 + (210x + 3)y^2 - (4900x^3 + 139x)y - 70x^2 - 71x - 1. \end{aligned}$$

At the beginning, we choose the following three primes: 7, 11 and 13 to work with. The prime set can be enlarged later if needed. Since $\gcd(f, g) := h$ must divide both

Algorithm 2.6 Modular GCD algorithm with trial division: part I

```
procedure GCD_INT($f, g \in \mathbb{Z}[x_1, \dots, x_d]$)          ▷ computes $\gcd(f, g) \in \mathbb{Z}[x_1, \dots, x_d]$
    $a \leftarrow \mathrm{cont}(f),\ b \leftarrow \mathrm{cont}(g),\ F \leftarrow f/a,\ G \leftarrow g/b$          ▷ normalize input polynomials
    $c \leftarrow \gcd(a, b),\ q \leftarrow \gcd(\mathrm{lcf}(F), \mathrm{lcf}(G))$
    $k \leftarrow \min(\deg_{x_d}(F), \deg_{x_d}(G)),\ (M, H) \leftarrow (0, 0)$
    $\mu \leftarrow 2^k \cdot |q| \cdot \min(|f|_\infty, |g|_\infty)$          ▷ compute "intuitive" bound on the size of GCD coefficients
    while true do
        $m \leftarrow$ NEXT_PRIME()          ▷ choose a prime that does not divide q
        while $(m \mid q)$ $m \leftarrow$ NEXT_PRIME() od
        $\tilde{F} \leftarrow \phi_m(F),\ \tilde{G} \leftarrow \phi_m(G)$          ▷ compute homomorphic images modulo m
        $\tilde{q} \leftarrow \phi_m(q),\ \tilde{H} \leftarrow$ GCD_MOD($\tilde{F}, \tilde{G}, m$), $l \leftarrow \deg_{x_d}(\tilde{H})$          ▷ compute a GCD in $\mathbb{Z}_m[x_1, \dots, x_d]$
        $\tilde{H} \leftarrow \tilde{q} \cdot \mathrm{lcf}(\tilde{H})^{-1} \cdot \tilde{H}$          ▷ normalize the result by forcing $\mathrm{lcf}(\tilde{H}) = \tilde{q}$
        if $(l < k)$ then          ▷ tests for unlucky homomorphisms
            $(M, H) \leftarrow (m, \tilde{H}),\ k \leftarrow l$          ▷ degree check fails: discard previous results
        elif $(l = k)$ then
            for all scalar coefficients $H_e \in \mathbb{Z}$ of $H$ do          ▷ otherwise, add the image $\tilde{H}$ to the result H
                $H_e \leftarrow$ CRA_INCREMENTAL($\{H_e, \tilde{H}_e\}, \{M, m\}$)
            $M \leftarrow M \cdot m$
        fi
        if $(M > \mu)$ then          ▷ perform division check as soon as GCD coefficients are large enough
            $Q \leftarrow \mathrm{pp}(H)$
            if $(Q \mid F)$ and $(Q \mid G)$ then
                return $(c \cdot Q)$          ▷ add the GCD of contents to the result
            fi
        fi
    od
end procedure
```

polynomials, we know that $\deg_x(h) \le 3$ and $\deg_y(h) \le 3$. Thus, it suffices to use 4 evaluation points to substitute the variable x in the original polynomials. To calculate the GCD in $\mathbb{Z}_7[x, y]$, we take the polynomials:

$$f_7(x, y) = -y^4 + 2y^3 + xy^2 - (2x + 2)y + x + 1,\ g_7(x, y) = -2y^3 + 3y^2 + xy - x - 1,$$

where a GCD of the leading coefficients is $q_7 := \gcd(\mathrm{lcf}(f_7), \mathrm{lcf}(g_7)) = -1$. We shall account for this later. Evaluating these polynomials at $x = \{1, 2, 3, 4\}$ and computing the GCDs in $\mathbb{Z}_7[y]$, yields:

$$\begin{aligned}
\gcd(f_7(1, y), g_7(1, y)) &= \gcd(-y^4 + 2y^3 + y^2 + 3y + 2, -2y^3 + 3y^2 + y - 2) = y - 1,\\
\gcd(f_7(2, y), g_7(2, y)) &= \gcd(-y^4 + 2y^3 + 2y^2 + y + 3, -2y^3 + 3y^2 + 2y - 3) = y - 1,\\
\gcd(f_7(3, y), g_7(3, y)) &= \gcd(-y^4 + 2y^3 + 3y^2 - y - 3, -2y^3 + 3y^2 + 3y + 3) = y - 1,\\
\gcd(f_7(4, y), g_7(4, y)) &= \gcd(-y^4 + 2y^3 - 3y^2 - 3y - 2, -2y^3 + 3y^2 - 3y + 2) = y - 1.
\end{aligned}$$

Clearly, the result does not depend on x for $m = 7$, hence interpolation gives us the first image of a GCD in $\mathbb{Z}_7[x, y]$: $h_7(x, y) = q_7 \cdot (y - 1) = -y + 1$. Repeating the calculations for $m = 11$, we have:

$$\begin{aligned}
f_{11}(x, y) &= -y^4 - (3x - 2)y^3 - (5x^2 - 4x)y^2 + (3x^2 + x - 2)y + 5x^3 + 2x^2 - 2x + 1,\\
g_{11}(x, y) &= -2y^3 + (x + 3)y^2 - (5x^2 - 4x)y - 4x^2 - 5x - 1,
\end{aligned}$$

Algorithm 2.7 Modular GCD algorithm with trial division: part II

```
procedure GCD_MOD($f, g \in \mathbb{Z}_m[x_1, \dots, x_d], m \in \mathbb{Z}$)    ▷ computes $\gcd(f, g) \in \mathbb{Z}_m[x_1, \dots, x_d]$
    if ($d = 1$) then    ▷ proceed with univariate GCDs
        return GCD_UNIVARIATE($f, g, m$)
    fi    ▷ consider f, g as polynomials in $\mathbb{Z}_m[x_1, \dots, x_{d-1}]$ with coeffs. in $\mathbb{Z}_m[x_d]$:
    let $(\tilde{f}, \tilde{g}) \leftarrow (f, g)$ where $\tilde{f}, \tilde{g} \in \mathbb{Z}_m[x_1, \dots, x_{d-1}][x_d]$
    $a \leftarrow \mathrm{cont}(\tilde{f}),\ b \leftarrow \mathrm{cont}(\tilde{g}),\ F \leftarrow f/a,\ G \leftarrow g/b$    ▷ divide out the contents $a, b \in \mathbb{Z}_m[x_d]$
    $c \leftarrow$ GCD_UNIVARIATE($a, b, m$), $q \leftarrow$ GCD_UNIVARIATE($\mathrm{lcf}(F), \mathrm{lcf}(G), m$)    ▷ here $c, q \in \mathbb{Z}_m[x_d]$
    $k \leftarrow \min(\deg_{x_d}(F), \deg_{x_d}(G)),\ (S, n) \leftarrow (\{\emptyset\}, 0),\ \delta \leftarrow k + \deg(q)$
    while true do
        $\alpha \leftarrow$ NEXT_POINT($m$)    ▷ choose an evaluation point $\alpha \in \mathbb{Z}_m$ such that $q(\alpha) \neq 0$
        while ($q(\alpha) = 0$) $\alpha \leftarrow$ NEXT_POINT($m$) od
        $\tilde{F} \leftarrow \phi_{x_d-\alpha}(F),\ \tilde{G} \leftarrow \phi_{x_d-\alpha}(G)$    ▷ compute homomorphic images modulo $x_d - \alpha$
        $\tilde{q} \leftarrow q(\alpha),\ \tilde{H} \leftarrow$ GCD_MOD($\tilde{F}, \tilde{G}, m$), $l \leftarrow \deg_{x_{d-1}}(\tilde{H})$    ▷ invoke the algorithm recursively
        $\tilde{H} \leftarrow \tilde{q} \cdot \mathrm{lcf}(\tilde{H})^{-1} \cdot \tilde{H}$    ▷ normalize the result so that $\mathrm{lcf}(\tilde{H}) = \tilde{q}$
        if ($l < k$) then    ▷ test for unlucky homomorphisms
            $S \leftarrow \{(\alpha, \tilde{H})\},\ (k, n) \leftarrow (l, 1)$    ▷ degree check fails: discard previous results
        elif ($l = k$) then
            $S \leftarrow S \cup \{(\alpha, \tilde{H})\},\ n \leftarrow n + 1$    ▷ update S to include the next evaluation point
        fi
        if ($n = \delta$) then    ▷ interpolate as soon as we reach the degree bound
            $H \leftarrow$ NEWTON_INTERP($S$)    ▷ reconstruct polynomial
            $Q \leftarrow \mathrm{pp}(H)$    ▷ here H is a multivariate polynomial with coeffs. in $\mathbb{Z}_m[x_d]$
            if ($Q \mid F$) and ($Q \mid G$) then
                return ($c \cdot Q$)    ▷ multiply by the GCD of the contents
            fi
        fi
    od
end procedure
```

with $q_{11} = \gcd(\mathrm{lcf}(f_{11}), \mathrm{lcf}(g_{11})) = -1$. Again, we proceed by computing the series of GCDs in $\mathbb{Z}_{11}[y]$:

$$\begin{aligned}
\gcd(f_{11}(1,y), g_{11}(1,y)) &= \gcd(-y^4 - y^3 - y^2 + 2y - 5, -2y^3 + 4y^2 - y + 1) = y - 5,\\
\gcd(f_{11}(2,y), g_{11}(2,y)) &= \gcd(-y^4 - 4y^3 - y^2 + y + 1, -2y^3 + 5y^2 - y - 5) = y + 2,\\
\gcd(f_{11}(3,y), g_{11}(3,y)) &= \gcd(-y^4 + 4y^3 - 5y + 5, -2y^3 - 5y^2 + 3) = y - 2,\\
\gcd(f_{11}(4,y), g_{11}(4,y)) &= \gcd(-y^4 + y^3 + 2y^2 - 5y + 4, -2y^3 - 4y^2 + 2y + 3) = y + 5.
\end{aligned}$$

Interpolating the individual coefficients modulo $m = 11$, one obtains: $h_{11}(x, y) = q_{11}(y - 4x - 1) = -y + 4x + 1$. Note that, h_7 and h_{11} both have the same degree in the variable y, thus we can argue that either none or both primes yield unlucky homomorphisms at this step. We further apply coefficient-wise Chinese remaindering giving a polynomial: $h_{77}(x, y) = -y - 7x + 1$. Next, the computations modulo $m = 13$ produce the following result: $h_{13} = -y + 5x + 1$ which has the same degree in the variable y as previously computed h_7 and h_{11}. By combining h_{13} with h_{77}, one obtains: $\tilde{h}(x, y) = -y + 70x + 1$. Finally, by performing the division check, we verify that $\tilde{h}(x, y)$ is indeed a GCD. ♣

The first part of the modular GCD algorithm, GCD_INT dealing with modular homomorphisms, is outlined in Algorithm 2.6. For each homomorphic image, GCD_INT invokes the algorithm GCD_MOD (see Algorithm 2.7) which, in its turn, recursively reduces the GCD computation to the univariate case (GCD_UNIVARIATE). Both algorithms essentially follow the same lines of code with only few differences. Namely, in Algorithm 2.7 we invoke GCD_MOD recursively. In addition, the interpolation points are not processed incrementally for reasons explained in Section 2.2.3. This also helps us to avoid the explicit construction of many intermediate polynomials which is undesirable. Instead, we collect the evaluation points and perform interpolation in one step. Lastly, we remark that, for the degree anomaly check in Algorithm 2.6, we use the degree of $\tilde{H}$ in the variable x_d (line 10); while in Algorithm 2.7 we use the degree of $\tilde{H}$ in the variable x_{d-1}, see line 13. Now, the only missing part is the efficient GCD algorithm in $\mathbb{Z}_m[x]$ which we discuss next.

2.3.3 Computing univariate GCDs

Clearly, since $\mathbb{Z}_m$ is a field, we can employ the Euclidean algorithm to compute a GCD in $\mathbb{Z}_m[x]$. However, the Euclidean scheme heavily uses divisions which are equivalent to computing expensive modular inverses in $\mathbb{Z}_m[x]$, and we would like to avoid this. First, observe that the primitive GCD algorithm (PGCD from Section 2.1.3) applied to $f, g \in \mathbb{D}[x]$ generates a sequence of polynomials which, in general terms, can be written as:

$$\begin{aligned}&\alpha_i F_{i-1} = Q_i F_i + \beta_i F_{i+1} \text{ with } \deg(F_{i+1}) < \deg(F_i) \quad (i = 2, \dots, k),\\ &\operatorname{prem}(F_{k-1}, F_k) = 0,\end{aligned}$$

where $F_1 = f$, $F_2 = g$, $Q_i = \operatorname{pquo}(F_{i-1}, F_i)$, and $\alpha_i, \beta_i \in \mathbb{D}$. This is called a *polynomial remainder sequence* (PRS) generated by f and g which also arises later in our discussion in relation to the resultant computations. Particularly, for the primitive GCD algorithm we have: $\alpha_i = \operatorname{lcf}(F_i)^{\delta_i+1}$, where $\delta_i = \deg(F_{i-1}) - \deg(F_i)$, and $\beta_i = \operatorname{cont}(\operatorname{prem}(F_{i-1}, F_i))$. This is known as the *primitive* PRS. By choosing such β_i in each step of the algorithm, we can keep the size of the remainders minimal. In $\mathbb{Z}_m[x]$, coefficient growth is no longer a problem and, in fact, all polynomials are primitive, hence we can trivially set $\beta_i = 1$. The resulting sequence is called the *Euclidean* PRS.

Now, it becomes clear how to efficiently compute a GCD in $\mathbb{Z}_m[x]$. We run the primitive GCD algorithm with $\beta_i = 1$, and in the end normalize the GCD polynomial by converting it to monic form. There is no need to multiply the result by the GCD of the contents. This concludes our discussion of a modular GCD algorithm. The next section is devoted to the computation of resultants which is another fundamental symbolic algorithm.

2.4 Resultants

In this section, we introduce the concept of the resultant of two polynomials. As mentioned in the introduction, resultants have many applications, for instance, in the topological study of algebraic curves, non-linear systems solving or computer graphics. Roughly speaking, the resultant is an *elimination tool* which provides us an algebraic criterion to decide whether a system of polynomial equations has a common solution expressed in terms of the coefficients of these polynomials. It should be noted that we discuss here what is known as the *Sylvester resultant* or the resultant of two polynomials, while, in most general form, the resultant can be defined for an arbitrary (finite) number of polynomials, see (CLO98, Chapter 3). We begin with the definition and basic properties, and then discuss in detail a modular approach to computing resultants.

2.4.1 Definition and main properties

To avoid excessive notation, we shall agree to write the definitions in the domain $\mathbb{D}[x]$ (as long as no ambiguity occurs) whereas it is assumed that $\mathbb{D}$ itself can be a polynomial domain, thereby extending the definitions to the multivariate case.

Definition 2.4.1. Let $f, g \in \mathbb{D}[x]$ be non-zero polynomials of degrees p, q, respectively. *Sylvester's matrix*[1] S associated with f and g is $(p+q) \times (p+q)$ matrix defined as:

$$S = \begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & & \vdots \\ 0 & \dots & 0 & f_p & f_{p-1} & \dots & f_0 \\ g_q & g_{q-1} & \dots & g_0 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & & \vdots \\ 0 & \dots & 0 & g_q & g_{q-1} & \dots & g_0 \end{bmatrix},$$

where the first q rows are formed of coefficients of f, and the last p rows are formed of coefficients of g. We shall also write $S^{(x)}$ to emphasize that the matrix S is associated with polynomials f and g in the indeterminate x. •

Definition 2.4.2. The *resultant* of $f, g \in \mathbb{D}[x]$, denoted by $\text{res}(f, g)$, is the determinant of Sylvester's matrix S:

$$\text{res}(f, g) = \det(S) \in \mathbb{D}.$$

Similarly, we may write $\text{res}_x(f, g)$ to include the indeterminate x in the definition. •

For example, given two bivariate polynomials $f, g \in \mathbb{D}[x, y]$, $\text{res}_x(f, g)$, defined as the determinant of $S^{(x)}$, is a univariate polynomial in the variable y. Below, we outline the basic properties of resultants. Some of them can be readily verified using the properties of determinants, see also (Yap00, § 4), (Coh03, Chapter 7). Let f and g be two polynomials as in Definition 2.4.1, then:

- $\text{res}(f, g)$ is a polynomial homogeneous in the coefficients of f and g;
- $\text{res}(f, g) = (-1)^{pq} \cdot \text{res}(g, f)$;
- $\text{res}(f \cdot h, g) = \text{res}(f, g) \cdot \text{res}(h, g)$, where $h \in \mathbb{D}[x]$;
- $\text{res}(c \cdot f, g) = c^q \cdot \text{res}(f, g)$ with $c \in \mathbb{D}$;
- $\text{res}(x - c, g) = g(c)$ with $c \in \mathbb{D}$.

The following theorems display fundamental properties of the resultants.

[1] The matrix is named for Joseph Sylvester (1814–1897), an English mathematician who first studied its properties in relation to the solution of a system of polynomial equations.

Theorem 2.4.1: Let $f, g \in \mathbb{D}[x]$ be polynomials of degrees $p, q > 0$, respectively. Then, there exist non-zero polynomials $s, t \in \mathbb{D}[x]$ with $\deg(s) < q$ and $\deg(t) < p$, such that

$$s \cdot f + t \cdot g = \mathrm{res}(f, g),$$

where s and t are often called *resultant cofactors* of f and g. ◇

Proof Let S be Sylvester's matrix associated with f and g. It is immediate to verify that, if we multiply S by the column vector $\mathbf{v} = [x^{p+q-1}, \ldots, x, 1]^T$, we obtain a vector formed of polynomials f and g multiplied by consecutive powers of x, in other words:

$$(S \cdot \mathbf{v})^T = [x^{q-1}f, x^{q-2}f, \ldots, f, x^{p-1}g, x^{p-2}g, \ldots, g]^T.$$

Next, using Cramer's rule for the last component of $\mathbf{v}$ (which equals to 1) yields:

$$\det \begin{bmatrix} f_p & f_{p-1} & \ldots & f_0 & 0 & \ldots & 0 & x^{q-1}f \\ 0 & \ddots & \ddots & & \ddots & & & \vdots \\ 0 & \ldots & 0 & f_p & f_{p-1} & \ldots & f_1 & f \\ g_q & g_{q-1} & \ldots & g_0 & 0 & \ldots & 0 & x^{p-1}g \\ 0 & \ddots & \ddots & & \ddots & & & \vdots \\ 0 & \ldots & 0 & g_q & g_{q-1} & \ldots & g_1 & g \end{bmatrix} = \det(S).$$

The result follows if we expand the determinant on the left-hand side by minors along the last column. By the same token, it implies that the polynomials s and t can be represented as the determinants of special matrices obtained from Sylvester's matrix by replacing the last column with the column vectors $[x^{q-1}, \ldots, x, 1, 0, \ldots, 0]^T$ and $[0, \ldots, 0, x^{p-1}, \ldots, x, 1]^T$ of size $(p + q)$, respectively. ■

The next result can be seen as a consequence of the theorem above.

Theorem 2.4.2: For polynomials $f, g \in \mathbb{D}[x]$, defined over a UFD $\mathbb{D}$, $\mathrm{res}(f, g) = 0$ if and only if f and g have a non-trivial common factor in $\mathbb{D}[x]$. For proof, see (BPR06, Prop 4.15). ◇

Finally, this last theorem establishes a more deeper connection between the resultant and the roots of polynomials.

Theorem 2.4.3: If polynomials $f, g \in \mathbb{D}[x]$ of degrees $p, q > 0$, are written in the form $f = \mathrm{lcf}(f) \prod_{i=1}^{p}(x - \alpha_i)$ and $g = \mathrm{lcf}(g) \prod_{i=1}^{q}(x - \beta_i)$, then:

$$\mathrm{res}(f, g) = \mathrm{lcf}(f)^q \prod_{i=1}^{p} g(\alpha_i) = (-1)^{pq}\,\mathrm{lcf}(g)^p \prod_{i=1}^{q} f(\beta_i) = \mathrm{lcf}(f)^q\,\mathrm{lcf}(g)^p \prod_{i=1}^{p}\prod_{j=1}^{q}(\alpha_i - \beta_j).$$

For proof, see (Yap00, Thm. 15), (BPR06, Thm. 4.16). ◇

Algorithm 2.8 Resultant computation in a Euclidean domain $\mathbb{F}[x]$

```
procedure RESULTANT_PRS($f, g \in \mathbb{F}[x]$)                ▷ returns $\mathrm{res}(f, g) \in \mathbb{F}[x]$
    $R \leftarrow 1,\ a \leftarrow f,\ b \leftarrow g$
    $p \leftarrow \deg(a),\ q \leftarrow \deg(b)$
    while ($q \neq 0$) do                                         ▷ compute remainder sequence
        $r \leftarrow \mathrm{rem}(a, b),\ s \leftarrow \deg(r)$
        $R \leftarrow R \cdot (-1)^{pq} \cdot \mathrm{lcf}(b)^{p-s}$
        $a \leftarrow b,\ b \leftarrow r,\ p \leftarrow q,\ q \leftarrow s$
    od
    return ($R \cdot \mathrm{lcf}(b)^{p}$)                        ▷ multiply by the last coefficient
end procedure
```

2.4.2 Computing resultants

Classical algorithms to computing resultants are based on the following theorems.

Theorem 2.4.4: Let $\mathbb{F}[x]$ be a Euclidean domain, and $f, g \in \mathbb{F}[x]$ polynomials of degrees p, q, respectively. Then, denoting by r a remainder in the Euclidean division of f by g, it holds that:

$$\mathrm{res}(f, g) = (-1)^{pq}\,\mathrm{lcf}(g)^{p-s}\,\mathrm{res}(g, r), \text{ where } s = \deg(r).$$

For proof, see (BPR06, Lem. 4.17). ◊

It follows that, the resultant can be calculated by repeatedly applying Theorem 2.4.4. Pseudocode for the resultant computation based on the Euclidean scheme is given in Algorithm 2.8. To extend the resultant algorithm to multivariate polynomials, we can utilize a pseudo-division property:

Theorem 2.4.5: Suppose $f, g \in \mathbb{D}[x]$ are polynomials of degrees $p, q > 0$, respectively, with $p \geq q$, and $\mathbb{D}$ is a UFD. Then, for $r = \mathrm{prem}(f, g)$, we have:

$$\mathrm{res}(f, g) = (-1)^{pq}\,\mathrm{res}(g, r)/\,\mathrm{lcf}(g)^{\delta q - p + q}, \text{ where } s = \deg(r), \text{ and } \delta = p - q + 1.$$

For proof, see (Coh03, Thm. 7.12). ◊

Unwinding the recursive definition in Theorem 2.4.5 leads us to the polynomial remainder sequence (PRS) which we have already met in Section 2.3.3. As we have seen, one can obtain a different PRS depending on the choice of the parameters α_i and β_i which, in turn, affects the degree and coefficient growth during the computation of resultant. In that sense, a good compromise between two extreme cases, the Euclidean and primitive PRS, is a *subresultant PRS* which is often used to compute resultants.

Definition 2.4.3. For $f, g \in \mathbb{D}[x]$, the *subresultant PRS* is the sequence of polynomials F_i starting with $F_1 = f$, $F_2 = g$, and defined as ($i = 2, \ldots, k$):

$$\alpha_i F_{i-1} = Q_i F_i + \beta_i F_{i+1}, \textit{ with } \deg(F_{i+1}) < \deg(F_i), \ \ \mathrm{prem}(F_{k-1}, F_k) = 0,$$

where

$$\begin{aligned}
&\alpha_i = \mathrm{lcf}(F_i)^{\delta_i + 1},\ \delta_i = \deg(F_{i-1}) - \deg(F_i),\\
&\beta_2 = (-1)^{\delta_2 + 1},\ \beta_i = -\,\mathrm{lcf}(F_{i-1}) \cdot \psi_i^{\delta_i}\ \ (i = 3, \ldots, k),\ \textit{and}\\
&\psi_2 = -1,\ \psi_i = (-\,\mathrm{lcf}(F_{i-1}))^{\delta_{i-1}} \cdot \psi_{i-1}^{1-\delta_{i-1}}\ \ (i = 3, \ldots, k).
\end{aligned}$$

•

We skip the pseudocode for subresultant PRS algorithm as it is not relevant in the context of our present discussion, and refer the interested reader to (Coh03, Chapter 7). We next consider a homomorphism approach which provides us a better way to calculating resultants especially in the multivariate domains.

2.4.3 Modular algorithm

As noted earlier, the modular resultant algorithm was first introduced by Collins in (Col71). From a high-level perspective, the modular GCD and resultant algorithms closely resemble each other with the main difference being that for the latter one there is nothing like "division check" is available to enable early termination of the algorithm. A ramification of this is that the resultant algorithm uses a-priori bounds for the height and degree of the resultant.[1] Luckily, the bounds on resultants are easily obtainable thanks to linear algebra analysis. In (Mon05), the modular algorithm has been made output-sensitive which is a significant improvement when the aforementioned bounds are inaccurate. Yet, the price for this is that there is a small probability that the algorithm returns incorrect results. In addition, this approach requires a very delicate treatment of "unlucky" primes. In this overview, we focus on Collins' original approach, not considering its improvements, which is sufficient for our needs. This will also facilitate the complexity analysis of the modular algorithms as provided in Section 2.5. We start by formulating the necessary conditions to compute resultants using a modular approach.

Theorem 2.4.6 (Collins): Let $f, g \in \mathbb{D}[x]$ be polynomials of degrees $p, q > 0$ over some integral domain $\mathbb{D}$, and $\phi : \mathbb{D} \to \mathbb{D}'$ be a homomorphism of rings. This also induces a natural homomorphism from $\mathbb{D}[x]$ to $\mathbb{D}'[x]$. Suppose that, $\deg(\phi(f)) = p$ and $\deg(\phi(g)) = r$, $0 \le r \le q$, then

$$\phi(\mathrm{res}(f, g)) = \phi(\mathrm{lcf}(f)^{q-r}) \cdot \mathrm{res}(\phi(f), \phi(g)). \qquad \diamond$$

Proof Let S be Sylvester's matrix associated with f and g, and $\tilde{S}$ be Sylvester's matrix for homomorphic images $\tilde{f} := \phi(f)$ and $\tilde{g} := \phi(g)$, respectively. If $r = q$, then, by definition, $\phi(S) = \tilde{S}$ and

$$\phi(\mathrm{res}(f, g)) = \phi(\det(S)) = \det(\phi(S)) = \det(\tilde{S}) = \mathrm{res}(\tilde{f}, \tilde{g}),$$

which shows the first part. If $r < q$, $\tilde{S}$ is of size $(r + p) \times (r + p)$ and can be obtained from $\phi(S)$ by deleting its first $q - r$ rows and columns since the coefficients $\tilde{g}_q, \ldots, \tilde{g}_{q-r+1}$ vanish. Next, observe that, the first $q-r$ columns of $\phi(S)$ contain $\phi(\mathrm{lcf}(f))$ on the diagonal and are zero below it. Hence, by the property of determinants:

$$\phi(\mathrm{res}(f, g)) = \phi(\det(S)) = \det(\phi(S)) = \phi(\mathrm{lcf}(f)^{q-r}) \cdot \mathrm{res}(\phi(f), \phi(g))$$

which proves the claim. ■

Similar to the case of GCD (cf. Theorem 2.3.1), the above theorem states that, to reconstruct the result from homomorphic images, we must not take those homomorphisms ϕ

[1] For definition of the height of a polynomial, see Section 2.1.2.

Algorithm 2.9 Modular resultant algorithm: part I

```
procedure RES_INT($f, g \in \mathbb{Z}[x_1, \ldots, x_d]$)        ▷ returns $\mathrm{res}_{x_d}(f, g) \in \mathbb{Z}[x_1, \ldots, x_{d-1}]$
    $\mu \leftarrow$ RES_HEIGHT_BOUND($f, g, x_d$)        ▷ estimate the height of the resultant w.r.t. variable $x_d$
    $(p, q) \leftarrow (\deg_{x_d}(f), \deg_{x_d}(g))$,  $(R, M) \leftarrow (0, 0)$
    while true do
        $(\tilde{F}, \tilde{G}) \leftarrow (0, 0)$
        while $(\deg_{x_d}(\tilde{F}) < p)$ or $(\deg_{x_d}(\tilde{G}) < q)$ do        ▷ obtain the next prime modulus
            $m \leftarrow$ NEXT_PRIME()        ▷ compute homomorphic images modulo m:
            $\tilde{F} \leftarrow \phi_m(f)$,  $\tilde{G} \leftarrow \phi_m(g)$        ▷ here $\tilde{F}, \tilde{G} \in \mathbb{Z}_m[x_1, \ldots, x_d]$
        od
        $\tilde{R} \leftarrow$ RES_MOD($\tilde{F}, \tilde{G}, m$)        ▷ compute resultant in $\mathbb{Z}_m[x_1, \ldots, x_d]$
        if $(M = 0)$
            $(R, M) \leftarrow (\tilde{R}, m)$        ▷ initialize at the 1st iteration
        else
            for all scalar coefficients $R_{\mathbf{e}} \in \mathbb{Z}$ of R do        ▷ add the image $\tilde{R}$ to the result R
                $R_{\mathbf{e}} \leftarrow$ CRA_INCREMENTAL($\{R_{\mathbf{e}}, \tilde{R}_{\mathbf{e}}\}, \{M, m\}$)
            $M \leftarrow M \cdot m$        ▷ add the prime number to the set
        fi        ▷ terminate if coefficients are large enough
        if $(M \geq \mu)$ then return R fi
    od
end procedure
```

for which $\phi(\mathrm{lcf}(f))$ and $\phi(\mathrm{lcf}(g))$ vanish simultaneously. In essence, the result of Theorem 2.4.6 is even stronger as it asserts that we can reconstruct the resultant even in the situation where one of the leading coefficients (but not both) vanishes under homomorphism. Moreover, in contrast to GCD computations, there is no "degree anomaly" problem (cf. Theorem 2.3.1) provided that the conditions of the above theorem are satisfied.

For polynomials $f, g \in \mathbb{Z}[x_1, \ldots, x_d]$, the modular resultant algorithm can be summarized as follows:

(a) calculate the number of homomorphisms required;

(b) find a homomorphism ϕ such that $\deg(\phi(f)) = \deg(f)$ and $\deg(\phi(g)) = \deg(g)$ which is equivalent to: $\mathrm{lcf}(f), \mathrm{lcf}(g) \notin \ker \phi$;

(c) compute $\mathrm{res}(\phi(f), \phi(g))$ using the image algorithm;

(d) terminate the algorithm if the number of images suffices to reconstruct the result; otherwise, go back to step **(b)**.

Again, in case of multivariate polynomials, we invoke the algorithm recursively in step **(c)** until it eventually comes to computing the resultants over $\mathbb{Z}_m[x]$. In the latter case, we can apply the Euclidean PRS (Algorithm 2.8) or, even better, use pseudo-division to compute the resultants, see Theorem 2.4.5 in Section 2.4.2.

To make a concrete algorithm out of this, we have to bound the number of evaluation and modular homomorphisms required. Suppose that for given polynomials $f, g \in \mathbb{Z}[x_1, \ldots, x_d]$, we wish to compute the resultant $R := \mathrm{res}_{x_d}(f, g)$ with respect to the variable x_d. For evaluation homomorphism, the degree bound on R applies, which is easily computable from Sylvester's matrix. Let $S^{(x_d)}$ be Sylvester's matrix associated with f and g such that $\mathrm{res}_{x_d}(f, g) = \det S^{(x_d)}$. We can bound $d := \deg_{x_{d-1}}(R)$ by summing up the

Algorithm 2.10 Modular resultant algorithm: part II

```
procedure RES_MOD($f, g \in \mathbb{Z}_m[x_1, \dots, x_d]$, m)          ▷ returns $\mathrm{res}_{x_d}(f, g) \in \mathbb{Z}_m[x_1, \dots, x_{d-1}]$
    if ($d = 1$) then
        return RES_UNIVARIATE($f, g, m$)                              ▷ invoke univariate resultant algorithm
    fi
    $\delta \leftarrow$ RES_DEGREE_BOUND($f, g, x_{d-1}$) + 1              ▷ bound for the degree of the resultant in $x_{d-1}$
    $(p, q) \leftarrow (\deg_{x_d}(f), \deg_{x_d}(g)),\ (S, n) \leftarrow (\{\emptyset\}, 0)$
    while true do
        $(\tilde{F}, \tilde{G}) \leftarrow (0, 0)$
        while ($\deg_{x_d}(\tilde{F}) < p$) or ($\deg_{x_d}(\tilde{G}) < q$) do     ▷ find a proper evaluation point $\alpha \in \mathbb{Z}_m$
            $\alpha \leftarrow$ NEXT_POINT(m)                            ▷ compute homomorphic images modulo $(x_{d-1} - \alpha)$:
            $\tilde{F} \leftarrow \phi_{x_{d-1}-\alpha}(f),\ \tilde{G} \leftarrow \phi_{x_{d-1}-\alpha}(g)$              ▷ here $\tilde{F}, \tilde{G} \in \mathbb{Z}[x_1, \dots, x_{d-2}, x_d]$
        od
        $\tilde{R} \leftarrow$ RES_MOD($\tilde{F}, \tilde{G}, m$)                  ▷ compute the resultant in $\mathbb{Z}_m[x_1, \dots, x_{d-2}, x_d]$
        if ($n < \delta$) then
            $S \leftarrow S \cup \{(\alpha, \tilde{R})\},\ n \leftarrow n + 1$               ▷ collect the next evaluation point
        fi
        if ($n = \delta$) then                         ▷ the number of points suffices for interpolation
            $R \leftarrow$ NEWTON_INTERP(S)                                  ▷ reconstruct polynomial
            return R
        fi
    od
end procedure
```

maximal degrees along the rows and columns of the matrix $S^{(x_d)}$, see also (Mon05):

$$d_1 = \sum_{i=1}^{p+q} \max_{j=1}^{p+q} \deg_{x_{d-1}}(S^{(x_d)}_{i,j}),\ d_2 = \sum_{j=1}^{p+q} \max_{i=1}^{p+q} \deg_{x_{d-1}}(S^{(x_d)}_{i,j}),\ \text{ and } d \le \min(d_1, d_2), \qquad (2.5)$$

where $p = \deg_{x_d}(f)$ and $q = \deg_{x_d}(g)$. In addition, if $d = 2$, one can apply Bezout's theorem, saying that the number of roots of R (and hence the degree of R) is bounded by: $\deg_x(f)\deg_y(g) + \deg_y(f)\deg_x(g)$. From our experiences, Bezout's bound is almost tight for dense polynomials while the bounds derived from Sylvester's matrix give better results when polynomials are sparse. Therefore, in practice, the combination of both types of estimates is preferred.

In case of a modular homomorphism, one usually applies Hadamard's inequality to bound the size of a matrix determinant. The next bound is due to (GR74), where the authors extended Hadamard's bound to matrices with polynomial entries, see also (Mon05). First, we form a matrix $\hat{S}$ whose entries are the 1-norms of respective entries of $S^{(x_d)}$, then it follows that:

$$|R|_\infty < \prod_{i=1}^{p+q} \sqrt{\sum_{j=1}^{p+q} (\hat{S}_{i,j})^2}, \text{ where } \hat{S}_{i,j} = |S^{(x_d)}_{i,j}|_1\ (i, j = 1 \dots p + q). \qquad (2.6)$$

It should be noted that, the above bounds are not sharp, although they usually work well in practice. To derive sharp estimates, one can apply the theory of *sparse resultants*, see (CLO98). For example, the degree of the resultant can be bounded by the Mixed Volume of Newton polytopes defining the supports of two polynomials.[1] For the height

[1] This holds only when the so induced polynomial system contains only *toric* roots, otherwise further corrections are necessary.

of the sparse resultant, see (Som04). However, the computational cost of such bounds is much higher, and therefore they are rarely used in real applications.

Having the resultant estimates, we can present a complete modular approach now. The algorithm comprising two procedures is very similar to the modular GCD algorithm from Section 2.3.2. However, remark that, to obtain the next modulus/evaluation point, we do *not* use lexicographical ordering for multivariate polynomials: instead, polynomials are treated in *recursive* representation. In the first part of the algorithm (RES_INT in Algorithm 2.9) we apply modular homomorphism, and call the procedure RES_MOD in Algorithm 2.10 computing the resultant in a finite field. The latter algorithm proceeds by applying evaluation homomorphisms recursively until it eventually remains to compute the resultant in $\mathbb{Z}_m[x]$ in which case a univariate resultant algorithm is invoked.

Example 2.4.1. To exemplify the modular approach, we outline the computation of the resultant $R := \mathrm{res}_y(f, g)$ of the following polynomials:

$$f = 2xy^2 + (x^2 + 2x - 1)y - x + 1, \;\; g = 2x^2y^2 + (x + 3)y - 3x^2 + 2.$$

Application of the degree bounds (2.5) yields $\deg(R) \le 8$, and hence 9 points suffices for interpolation. The Hadamard's bound (2.6) implies that $|R|_\infty < 198$, hence it suffices to evaluate the resultant for the following three primes: 7, 11 and 13. Taking the first modulus $m = 7$, we find that the respective modular images $f_7, g_7 \in \mathbb{Z}_7[x, y]$ coincide with f and g. Then, evaluating the polynomials at $x = \{1, 2, \ldots, 9\}$ and calculating the resultants in $\mathbb{Z}_7[y]$, we obtain:

$$\begin{aligned}
\mathrm{res}(f_7(1, y), g_7(1, y)) &= \mathrm{res}(2y + 2y^2, -1 - 3y + 2y^2) = -2,\\
\mathrm{res}(f_7(2, y), g_7(2, y)) &= \mathrm{res}(-1 - 3y^2, -3 - 2y + y^2) = 0,\\
\mathrm{res}(f_7(3, y), g_7(3, y)) &= \mathrm{res}(-2 - y^2, 3 - y - 3y^2) = -1,\\
\mathrm{res}(f_7(4, y), g_7(4, y)) &= \mathrm{res}(-3 + 2y + y^2, 3 - 3y^2) = 0,\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
\mathrm{res}(f_7(9, y), g_7(9, y)) &= \mathrm{res}(-1 - 3y^2, -3 - 2y + y^2) = 0.
\end{aligned}$$

Interpolating the polynomial at nine x values gives the first resultant image in $\mathbb{Z}_7[x]$:

$$\mathrm{res}(f, g, y) \bmod 7 = x^8 - 3x^7 + 2x^6 + 3x^5 - 2x^4 + 2x^3 + 2x.$$

Repeating the calculations modulo $m = 11$ yields:

$$\begin{aligned}
\mathrm{res}(f_{11}(1, y), g_{11}(1, y)) &= \mathrm{res}(2y + 2y^2, -1 + 4y + 2y^2) = 1,\\
\mathrm{res}(f_{11}(2, y), g_{11}(2, y)) &= \mathrm{res}(-1 - 4y + 4y^2, 1 + 5y - 3y^2) = 4,\\
\mathrm{res}(f_{11}(3, y), g_{11}(3, y)) &= \mathrm{res}(-2 + 3y - 5y^2, -3 - 5y - 4y^2) = 4,\\
\mathrm{res}(f_{11}(4, y), g_{11}(4, y)) &= \mathrm{res}(-3 + y - 3y^2, -2 - 4y - y^2) = 4,\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
\mathrm{res}(f_{11}(9, y), g_{11}(9, y)) &= \mathrm{res}(3 - y - 4y^2, 1 + y - 3y^2) = -3,
\end{aligned}$$

and the resultant image in $\mathbb{Z}_{11}[x]$ has the following form:

$$\mathrm{res}(f, g, y) \bmod 11 = 5x^8 - 2x^7 + 5x^6 - 5x^5 - 2x^4 + 3x^3 - 3x.$$

Next, applying the coefficient-wise Chinese remaindering, we calculate:

$$\mathrm{res}(f, g, y) \bmod 77 = -6x^8 - 24x^7 + 16x^6 + 17x^5 - 2x^4 - 19x^3 + 30x.$$

Computations for the remaining prime $m = 13$ show that:

$$\mathrm{res}(f, g, y) \bmod 13 = -6x^8 + 2x^7 + 3x^6 + 3x^5 - 2x^4 - 5x^3 + 4x.$$

Finally, by combining the modular images using the CRA, one obtains:

$$\mathrm{res}(f, g, y) \bmod 1001 = -6x^8 - 24x^7 + 16x^6 + 94x^5 - 2x^4 - 96x^3 + 30x,$$

which is the resultant of f and g in $\mathbb{Z}[x]$. ♣

In the next section, we conclude our discussion of modular techniques by deriving the complexity of GCD and resultant algorithms.

2.5 Complexity analysis of modular algorithms

In this section, we analyze the computational complexity of the modular algorithms considered in the previous sections assuming both sequential and parallel models of computation. To facilitate the complexity analysis, we primarily concentrate on the *bivariate* case, that is, where $f, g \in \mathbb{Z}[x, y]$, leaving out a more general case that can be reconstructed by analogy. We begin with the definition of the model of computations, and then recall the complexity of basic operations before discussing the main algorithms. For the complexity of symbolic algorithms, the reader may also consult (Sho09, Section 18), (Ker09, Section 2.4) or (BPR06, Section 8).

2.5.1 Model of computations

There is a number of computational models available to measure the amount of resources needed by an algorithm to execute. These models, for example, include *Turing machines*, *Lambda calculus*, *Boolean circuits*, *Random Access Machines* (RAM), etc., and mainly in the set of allowable operations and their respective costs, see (Yap00). For algebraic manipulation, the most commonly used model is that of *Boolean circuits* where inputs are given by a sequence of bits and all bit operations having at most two input operands (realized by logic gates) have a unit cost. This model is widely adopted in the analysis of symbolic algorithms because the latter ones often need to operate on large integers (or rational numbers) which fall far beyond the range of numbers representable by a single machine word. Naturally, such arithmetic operations cannot be assumed to have a unit cost: instead, we need to bound the number of *bit operations* performed by the algorithm in the worst case. In this context, one speaks about the *bit complexity* of an algorithm, as opposed to *algebraic complexity*, where arithmetic operations on a given algebraic structure (such as the ring of integers $\mathbb{Z}$ or polynomials over some domain $\mathbb{D}$) are assumed to be primitive.

In the Boolean circuit model, it is common to describe each integer quantity $a \in \mathbb{Z}$ involved in computations by a corresponding *bitlength* (or bitsize) which is simply the number of bits needed to store this quantity by the computer. Similarly, for rational number $q \in \mathbb{Q}$, the bitlength can be defined as the maximal bitlength of its numerator and its denominator. The main complexity measures in this model are *circuit depth* and *circuit width*. Translated into the language of symbolic computations, they correspond to the number of arithmetic operations (for instance, over $\mathbb{Z}$) performed by the algorithm in the worst case and the maximal bitlength of numbers involved in these computations. When multiplied together, these measures give a meaningful estimate for the overall complexity of an algorithm. The complexity bounds are usually written in *big-Oh* notation, denoted by $O(x)$.

To analyze the complexity of parallel algorithms we adopt a standard *PRAM* (Parallel Random Access Machine) model of computing. Each component of this model is a random access machine which is an arithmetic processor having its own register file and capable of executing a usual set of arithmetic instructions at unit cost. The PRAM model provides any (problem-dependent) number of processors and assumes that these processors can simultaneously access a sufficiently large block of *shared memory*. It is also supposed that inter-processor communication and synchronization overhead can be neglected since the cost of arithmetic computations carried out by the processors is dominating. We believe that the PRAM model yields a fairly good approximation of the GPU threading model, and thus it can be used to evaluate the computational cost of algorithms running on the GPU. This is justified by the fact that GPU threads residing in the same thread block (see Section 4.1) can communicate through shared memory and synchronize with barriers without any noticeable overhead as long as one keeps high arithmetic intensity of computations which matches the assumptions of the PRAM model. Certainly, the assumption of having unlimited processor resources is not realistic. However, the GPU allows an (almost) arbitrary number of thread blocks to be scheduled for execution (which are then processed concurrently or in a queued fashion). In this case, the real computing times depend on the physical number of processors available on the device. We will denote the parallel complexity by $O_P(x, p)$ meaning that a problem can be solved in $O(x)$ parallel time on p processors.

2.5.2 Complexity of basic operations

First, we recall the complexity of primitive operations on integers and polynomials. We shall write $T_{\mathcal{B}}(a \circ b)$ denoting the bit-complexity of performing an operation $a \circ b$. Let $a, b \in \mathbb{Z}$ be the integers of bitlength $\tau_1, \tau_2 \in \mathbb{N}$, respectively. Here, we assume that a and b might have different bitlengths because, for the analysis of the modular algorithms, it will sometimes be required to deal with *unbalanced* operands. For addition and subtraction, it holds that

$$T_{\mathcal{B}}(a \pm b) = O(\tau_1 + \tau_2), \tag{2.7}$$

while for school-book long multiplication and division:

$$\begin{aligned} T_{\mathcal{B}}(a \cdot b) &= O(\tau_1 \tau_2), && (2.8)\\ T_{\mathcal{B}}(a/b) &= O(\tau_1(\tau_1 - \tau_2)), \text{ where } \tau_1 > \tau_2. && (2.9) \end{aligned}$$

For integer GCD, Collins in (Col74) showed that the complexity of the Euclidean algorithm is bounded by:

$$T_{\mathcal{B}}(\gcd(a,b)) = O(\tau_2(\tau_1 - \tau_3)), \text{ if } \tau_1 > \tau_2, \tag{2.10}$$

and τ_3 is the bitsize of $\gcd(a,b)$. His argument exploits the fact that the number of integer divisions is bounded by τ_1, where, in the edge case, the remainders form a Fibonacci sequence. Then, the bound follows by carefully combining the complexity of single division steps. For Chinese remaindering, suppose we wish to add a residue x modulo m to the product X modulo M, where the bitlengths of M and m are τ_1 and τ_2 ($\tau_1 > \tau_2$), respectively. Analyzing Algorithm 2.3 from Section 2.2.2, we conclude that the complexity of one incremental step is dominated by that of a long integer multiplication, while computing a modular inverse (using the Extended Euclidean algorithm) and the reduction modulo m can also be done within this time. Therefore,

$$T_{\mathcal{B}}(\textsc{cra\_incremental}(\{x,X\},\{m,M\})) = \tau_1\tau_2. \tag{2.11}$$

Summing up the costs of all individual CRA steps, we conclude that, in order to recover an integer X of bitlength τ from k residues, one needs $O(\tau^2)$ bit operations. Using an asymptotically fast “divide-and-conquer” algorithm, this can be done in $O(M_{\mathcal{B}}(\tau)\log\tau)$, where $M_{\mathcal{B}}(\tau)$ stands for the bit complexity of multiplying two τ-bit integers, see (vzGG03, Cor. 10.23).

Let us now consider the polynomial operations. Suppose, $f, g \in \mathbb{Z}[x]$ are polynomials of degree p and q and coefficients with bitlength τ_1 and τ_2, respectively. The complexity of addition and subtraction is:

$$T_{\mathcal{B}}(f \pm g) = O((p\tau_1 + q\tau_2) \cdot \max(p,q)) \tag{2.12}$$

where we consider the two cases separately and then combine the bounds. For classical multiplication, it holds that:

$$T_{\mathcal{B}}(f \cdot g) = O(pq\{\tau_1\tau_2 + \log\min(p,q)\}) \tag{2.13}$$

since it amounts for $O(pq)$ multiplications in $\mathbb{Z}$ followed by $O(pq)$ additions of integers with bitsize at most $\tau_1 + \tau_2 + \log\min(p,q)$.

We finally estimate the amount of work of some relevant operations in $\mathbb{Z}_m[x]$. Let $\hat{f}, \hat{g} \in \mathbb{Z}_m[x]$ be polynomials of degrees p, q, respectively. We shall agree that operations in $\mathbb{Z}_m$ can be performed in a constant time because it is assumed that any modulus m has a *fixed* bitlength, see discussion in Section 2.5.3. For polynomial division using the Euclidean algorithm, we have (see (Sho09, Thm. 17.1)):

$$T_{\mathcal{B}}(\text{quo}(f,g)) = T_{\mathcal{B}}(\text{rem}(f,g)) = O(p(p-q)), \tag{2.14}$$

provided that $p > q$. On a similar note, the complexity of the Euclidean GCD is bounded by:

$$T_{\mathcal{B}}(\gcd(\hat{f},\hat{g})) = O(q(p-r)), \text{ if } p > q, \text{ and } r = \deg(\gcd(\hat{f},\hat{g})), \tag{2.15}$$

where the argument is essentially the same as for integer GCDs, see also (Sho09, Thm. 17.3). To evaluate the polynomial $\hat{f}(x)$ at $\alpha \in \mathbb{Z}_m$ one demands for

$$T_{\mathcal{B}}(\hat{f}(\alpha)) = O(n) \tag{2.16}$$

operations in $\mathbb{Z}_m$ which directly follows from expanding $\hat{f}(\alpha)$ according to Horner's scheme. It remains to show the complexity of polynomial interpolation. From the analysis of Algorithm 2.5 from Section 2.2.3, we immediately conclude that in order to interpolate a polynomial from $n+1$ points, one needs

$$T_{\mathcal{B}}(\text{NEWTON\_INTERP}(\{\alpha_0,\ldots,\alpha_n\},\{y_0,\ldots,y_n\})) = O(n^2) \tag{2.17}$$

operations in $\mathbb{Z}_m$. Indeed, the loop in lines 5–9 has a complexity of $O(n^2)$ finite field operations, and the same amount of work is required to reconstruct a polynomial in lines 11–13.

2.5.3 GCD algorithm

In this section, we are going to derive the complexity of Brown's algorithm applied to bivariate polynomials $f, g \in \mathbb{Z}[x,y]$, see Algorithms 2.6 and 2.7 in Section 2.3.2. We should emphasize, however, that our complexity bounds do not apply to the *worst case* as we agree that unlucky homomorphisms should *never* occur. This is a reasonable assumption since, as noted in Section 2.3.2, the probability of encountering an unlucky homomorphism is inversely proportional to the size of a prime number, and is exceedingly small for machine word-sized primes (31 or 63 bits) commonly used for implementation. Besides, we wish to obtain the bounds which reflect the expected behaviour of the algorithm because, otherwise, an adversary can always construct an example which would cause unlucky homomorphisms to occur for every chosen prime modulus. On the other hand, for the complexity analysis we do *not* enforce the use of asymptotically fast integer arithmetic which makes our bounds more realistic. Further restrictions on the input parameters of the algorithm are listed below:

- prime numbers m have fixed bit-size (usually chosen to be single-word integers) meaning that the operations in $\mathbb{Z}_m$ are assumed to be of unit cost;
- the coefficients of polynomials f and g are of reasonable size, so that the supply of prime numbers cannot be exhausted;
- the degrees of f and g in each variable do not exceed any realistic bounds so that the number of elements in $\mathbb{Z}_m$ for each prime m suffices for interpolation.

Complexity of the serial algorithm. In what follows, suppose $f, g \in \mathbb{Z}[x,y]$ are polynomials of degree at most n in each variable with scalar coefficients bounded by 2^τ, $\tau \in \mathbb{N}$. We first consider Algorithm 2.7 computing a GCD over $\mathbb{Z}_m$ since it appears as a subroutine of Algorithm 2.6. Thus, let $(\hat{f},\hat{g}) = (\phi_m(f),\phi_m(g)) \in \mathbb{Z}_m[x,y]$ be the homomorphic images of f and g. In line 6 of Algorithm 2.7, we compute the contents of $\hat{f}$ and $\hat{g}$ considered as polynomials with coefficients in $\mathbb{Z}_m[y]$, the latter operation reduces to computing the series of GCDs in the coefficient domain. Given that, $\hat{f}$ and $\hat{g}$ have at most $(n+1)$ terms each (considered as polynomials in the variable y), and the cost of a univariate GCD over $\mathbb{Z}_m$ is $O(n^2)$, we conclude that:

$$T_{\mathcal{B}}(\hat{F} \leftarrow \hat{f}/\operatorname{cont}(\hat{f})) = T_{\mathcal{B}}(\hat{G} \leftarrow \hat{g}/\operatorname{cont}(\hat{g})) = O(n^3). \tag{2.18}$$

The time to divide by the contents is again bounded by $O(n^3)$ because there are at most $2(n+1)$ divisions in $\mathbb{Z}_m[x]$ each of which costs $O(n^2)$ operations. The complexity of computing c and q in line 7 is negligible. The number of evaluation points δ needed by the algorithm can be estimated as

$$\delta \leq \max(\deg_y(\hat{f}), \deg_y(\hat{g})) = O(n). \tag{2.19}$$

This is a worst-case bound, and hence we can omit the trial division in line 23. Under the assumption that the degrees of f and g are reasonably bounded, we conclude that there are enough elements in $\mathbb{Z}_m$ to generate $O(n)$ evaluation points. Evaluating $q(\alpha)$ in line 11 amounts for $O(n)$ operations in $\mathbb{Z}_m$. Moreover, as we have agreed that no unlucky homomorphism can occur, the cost of choosing evaluation points in line 11 can be neglected.[1] Application of homomorphisms in line 12, is equivalent to evaluating $(n+1)$ coefficients of $\hat{f}$ and $\hat{g}$ (considered as polynomials with coefficients in $\mathbb{Z}_m[x]$) where each of them has degree at most n, thus:

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_{y-\alpha}(\hat{F})) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_{y-\alpha}(\hat{G})) = O(n^2), \tag{2.20}$$

due to the fact that operations in $\mathbb{Z}_m$ can be performed in $O(1)$ time. In line 13, we invoke the algorithm recursively, which in turn invokes a univariate GCD algorithm in $\mathbb{Z}_m[x]$, therefore:

$$T_{\mathcal{B}}(\tilde{H} \leftarrow \textsc{gcd\_univariate}(\tilde{F}, \tilde{G}, m)) = O(n^2). \tag{2.21}$$

The cost of the remaining operations in lines 13–14 is certainly dominated by $O(n^2)$. In total, the complexity of the main loop (without interpolation) evaluates to $O(n^3)$. Interpolation from $O(n)$ points costs $O(n^2)$ field operations. Since we apply it to each coefficient of $\tilde{H}$ separately, the total cost becomes:

$$T_{\mathcal{B}}(H \leftarrow \textsc{newton\_interp}(\{\alpha\}, \{\tilde{H}\})) = O(n^3). \tag{2.22}$$

Then, the time for computing the primitive part of H in line 22, by essentially the same reasoning as in (2.18), is bounded by

$$T_{\mathcal{B}}(Q \leftarrow \mathrm{pp}(H)) = O(n^3). \tag{2.23}$$

Finally, each multiplication of a coefficient of H by $c \in \mathbb{Z}[y]$ in line 24 can be performed in $O(n^2)$ operations with the total cost again in $O(n^3)$. Summing up these bounds, it follows that the overall running time of Algorithm 2.7 is

$$T_{\mathcal{B}}(\textsc{gcd\_mod}(\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y])) = O(n^3). \tag{2.24}$$

We now turn to the complexity analysis of Algorithm 2.6. Computing the contents together with primitive parts of f and g in line 2 amounts to

$$T_{\mathcal{B}}(F \leftarrow f/\operatorname{cont}(f)) = T_{\mathcal{B}}(G \leftarrow g/\operatorname{cont}(g)) = O(\tau^2 n^2) \tag{2.25}$$

bit operations as it is equivalent to computing at most $(n + 1)^2$ GCDs in $\mathbb{Z}$ and the same number of divisions. The time to calculate a GCD of the contents in line 3 is negligible. To devise the complexity of the main loop of Algorithm 2.6, we need to bound the height

[1] Even if this is not the case and, suppose, we would need up to $O(n)$ iterations to choose each evaluation point (which is a very broad assumption), this cost is not dominating as we shall see below.

of a GCD which gives the number N of primes needed by the algorithm in the worst case. Extended Mignotte bound from (Cor04) implies that

$$|h|_\infty \leq 2^{(n+1)^2}|f|_2, \text{ where } h \in \mathbb{Z}[x,y],\ h \mid f, \tag{2.26}$$

and thus $N = O(n^2 + \tau)$. Again, we use the assumption that the supply of primes numbers cannot be exhausted. Testing for $m \mid q$ in line 8 can be done in $O(\tau)$ bit operations since we divide q by an integer m of a fixed bitlength. Assuming that there are no unlucky primes, the next prime can be found in the first iteration of the loop in line 8. Next, in line 9 we reduce the coefficients of F and G modulo m. Provided that, these polynomials have at most $(n+1)^2$ non-zero terms, the bit complexity is bounded by

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_m(F)) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_m(G)) = O(\tau n^2), \tag{2.27}$$

where we again use the bit complexity of integer division for unbalanced operands. In line 10, we invoke the procedure GCD_MOD which has cost $O(n^3)$. To normalize scalar coefficients of $\tilde{H} = \gcd(\tilde{F}, \tilde{G})$ in line 12, one clearly needs

$$T_{\mathcal{B}}(\tilde{H} \leftarrow \tilde{q} \cdot \mathrm{lcf}(\tilde{H})^{-1} \cdot \tilde{H}) = O(n^2) \tag{2.28}$$

finite field operations. As there are no concerns regarding unlucky primes, the cost of the main loop (not counting CRA) evaluates to $O(n^2(\tau + n)N)$. If we apply CRA incrementally in lines 15–16 to each of $(n+1)^2$ scalar coefficients of $\tilde{H}$, where the product of primes is bounded by N, then the cumulative complexity of Chinese remaindering becomes:

$$T_{\mathcal{B}}(H \leftarrow \textsc{cra\_incremental}(\{m\}, \{\tilde{H}\})) = O(N^2 n^2) \tag{2.29}$$

To extract a primitive part of H in line 20, one needs at most $(n+1)^2$ divisions and GCD computations in the coefficient domain, therefore

$$T_{\mathcal{B}}(Q \leftarrow \mathrm{pp}(H)) = O(\tau^2 n^2). \tag{2.30}$$

Finally, as we use the worst-case bound N for the number of moduli, the trial division in line 21 can be suppressed, and the total cost of the modular algorithm evaluates to

$$T_{\mathcal{B}}(\textsc{gcd\_int}(f, g \in \mathbb{Z}[x,y])) = O(n^2(\tau + n^2)^2). \tag{2.31}$$

Unfortunately, the overall complexity is dominated by that of Chinese remainder algorithm: this is because the bound N on the bitlength of polynomial divisors seems to be from optimal, particularly in the multivariate case. Yet, in the analysis, we have not relied on asymptotically fast methods for integer arithmetic which are quite often used to get better complexity. In our case, the complexity can be slightly improved by employing the asymptotically fast Chinese remaindering, see Section 2.5.2:

$$T_{\mathcal{B}}(\textsc{gcd\_int}(f, g \in \mathbb{Z}[x,y])) = O(n^2\{(\tau + n)(\tau + n^2) + M_{\mathcal{B}}(\tau + n^2)\log n\}).^1 \tag{2.32}$$

Alternatively, we could apply Fast Fourier Transform (FFT) in place of the classical evaluation-interpolation scheme. However, in this situation, our assumption regarding

[1] $M_{\mathcal{B}}(\tau)$ denotes the complexity of multiplying two τ-bit integers.

the absence of unlucky homomorphisms would become rather speculative because, for a finite field FFT, it is no longer possible to choose evaluation points (the roots of unity) at random.

The same bounds are easily obtainable for *univariate* polynomials $f, g \in \mathbb{Z}[x]$. In this case, the number N of moduli is bounded by $O(n + \tau)$ due to Mignotte's bound (Mig74). The cost of applying modular homomorphism becomes $O(n\tau)$, and we also replace the invocation of the algorithm GCD_MOD by the Euclidean GCD working in $\mathbb{Z}_m[x]$ which runs in $O(n^2)$ field operations. The main loop now executes in $O(N \cdot n(n + \tau))$ bit operations. The complexity of Chinese remaindering for $(n + 1)$ coefficients of a GCD polynomial is $O(N^2 n)$. Thus, the overall complexity of a univariate GCD algorithm becomes

$$T_{\mathcal{B}}(\text{GCD\_INT}(f, g \in \mathbb{Z}[x])) = O(n(\tau + n)^2). \tag{2.33}$$

For comparison, calculating a GCD using the subresultant PRS algorithm has a complexity of $O(n^2 M_{\mathcal{B}}\{n(\tau + \log n)\})$ bit operations, see (Ker09, Thm. 2.4.19). The complexity of asymptotically fast GCD algorithm (based on half-GCD approach) evaluates to $\tilde{O}(n^2\tau)$ bit operations,[1] see (Rei97, LR01). Yet, note that, the latter approach also assumes the use of asymptotically fast integer arithmetic while, in our analysis, we do not rely on such assumptions.

Complexity of the parallel algorithm. We briefly analyze the complexity of the parallel algorithm assuming the PRAM model. For Algorithm 2.7, observe that it does not make sense to use more than $\delta = n$ processors unless we can find some way to parallelize a univariate GCD algorithm. Hence, the main loop of Algorithm 2.7 executes in $O_P(n^2, n)$ parallel time, where we assign each processor to one evaluation point. Computing the primitive parts in (2.18) and (2.23) can also be done within this time. Finally, each of $O(n)$ coefficients of a GCD polynomial can be interpolated independently, hence the cost of (2.22) becomes $O_P(n^2, n)$. In summary, we obtain:

$$T_{\mathcal{B}}(\text{GCD\_MOD}(\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y])) = O_P(n^2, n). \tag{2.34}$$

In Algorithm 2.7, we could use N processors, where N is the number of moduli however, then, Chinese remaindering becomes a bottleneck. The easiest way out is to split the problem "coefficient-wisely" between n^2 processors. In that case, the bit complexity of computing primitive parts in (2.25) and (2.30) turns to $O_P(\tau^2, n^2)$ since we process each coefficient independently. On a similar note, the cost of Chinese remaindering in (2.29) reduces to $O_P(N^2, n^2)$. The outer loop of the algorithm now runs in $O_P((\tau + n^2)N, n^2)$ parallel time since we use n^2 processors to apply modular homomorphisms in (2.27) while the procedure GCD_MOD requires n processors to run in parallel. As a result, we have managed to reduce the total complexity of the GCD algorithm to

$$T_{\mathcal{B}}(\text{GCD\_INT}(f, g \in \mathbb{Z}[x, y])) = O_P((\tau + n^2)^2, n^2) \tag{2.35}$$

using parallel processing. In Section 3.3, we shall see that it is possible to obtain even better parallel complexity if we employ the matrix-based approach for the subalgorithms.

[1] $\tilde{O}(\cdot)$ denotes a bit complexity omitting polylogarithmic in n and τ.

2.5.4 Resultant algorithm

We next discuss the computing time of Collins' modular algorithm given by Algorithms 2.9 and 2.10 from Section 2.4.3 which is applied to bivariate polynomials $f, g \in \mathbb{Z}[x, y]$. We shall try to keep the discussion brief because this algorithm essentially repeats the same steps as the GCD algorithm analyzed in the previous section. First, remark that, the assumptions introduced in Section 2.5.3 also apply to the resultant computations. Yet, by unlucky homomorphisms, we now understand only those homomorphisms which cause the leading coefficients of f and g to vanish simultaneously (cf. Theorem 2.4.6 in Section 2.4.3).

Complexity of the serial algorithm. Let $f, g \in \mathbb{Z}[x, y]$ be polynomials of degree at most n in each variable with scalar coefficients bounded by 2^τ, $\tau \in \mathbb{N}$. Our goal is to derive the complexity of computing $\mathrm{res}_y(f, g) \in \mathbb{Z}[x]$, the resultant with respect to the variable y. We again start with Algorithm 2.10 to calculate the resultant of $(\hat{f}, \hat{g}) = (\phi_m(f), \phi_m(g)) \in \mathbb{Z}_m[x, y]$ for each homomorphic image. For the degree bound δ in line 5, it trivially holds by Bezout's theorem that:

$$\delta = \deg(\mathrm{res}_y(\hat{f}, \hat{g})) = O(n^2). \tag{2.36}$$

Evaluating the polynomials at $x = \alpha$ in line 11 costs

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_{x-\alpha}(\hat{f})) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_{x-\alpha}(\hat{g})) = O(n^2), \tag{2.37}$$

operations in $\mathbb{Z}_m$. We suppose that the next evaluation point can be found in the first iteration of the loop as no unlucky homomorphisms can occur. In fact, checking the degree of $\tilde{F}$ and $\tilde{G}$ in line 6 can be done in $O(n)$ field operations, thus we may even assume that at most $O(n)$ degree checks would be required for each evaluation point which does not affect the final complexity. Then, we execute a univariate resultant algorithm having the same complexity as the Euclidean GCD algorithm:

$$T_{\mathcal{B}}(\tilde{R} \leftarrow \textsc{res\_univariate}(\tilde{f}, \tilde{g}, m)) = O(n^2). \tag{2.38}$$

The outer loop of Algorithm 2.10 is then repeated for each of $O(n^2)$ evaluation points, yielding the total complexity of $O(n^4)$ finite field operations. Finally, to interpolate the resultant polynomial of degree $O(n^2)$, we need

$$T_{\mathcal{B}}(R \leftarrow \textsc{newton\_interp}(\{\alpha\}, \{\tilde{R}\})) = O(n^4). \tag{2.39}$$

operations in $\mathbb{Z}_m$. In summary, the complexity of the resultant algorithm in $\mathbb{Z}_m$ is bounded by

$$T_{\mathcal{B}}(\textsc{res\_mod}(\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y])) = O(n^4). \tag{2.40}$$

The resultant in $\mathbb{Z}[x]$ is then calculated by Algorithm 2.9 making essential use of the subalgorithm res_mod via modular homomorphism. We need to estimate the height of the resultant to bound the number N of primes. Hadamard's bound (2.6) in Section 2.4.3 yields

$$N \le |\,\mathrm{res}_y(f, g)|_\infty = O(n(\tau + \log n)). \tag{2.41}$$

To reduce the coefficients of f and g modulo m in line 8, one requires

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_m(f)) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_m(g)) = O(\tau n^2) \tag{2.42}$$

bit operations, see Section 2.5.3. Provided that, in the absence of unlucky homomorphisms, the next valid prime can be found in a single step, we proceed further. Invocation of the algorithm $\textsc{res\_mod}$ to compute $\tilde{R} = \mathrm{res}_y(\tilde{F}, \tilde{G})$ demands for $O(n^4)$ bit operations, and the bit complexity of the main loop (without CRA) evaluates to: $O(Nn^2(n^2+\tau))$. To reconstruct the integer resultant, we apply Chinese remaindering for each of n^2 coefficients of R where the product of primes is bounded by N, hence

$$T_{\mathcal{B}}(R \leftarrow \textsc{cra\_incremental}(\{m\}, \{\tilde{R}\})) = O(N^2 n^2). \tag{2.43}$$

Combining the above estimates, the final complexity of the modular resultant algorithm becomes:

$$T_{\mathcal{B}}(\textsc{res\_int}(f, g \in \mathbb{Z}[x, y])) = O(n^4(\tau + \log n)(n + \tau)), \tag{2.44}$$

whereas the bit complexity of computing the resultants using the subresultant PRS can be bounded as $O(n^6 M_{\mathcal{B}}\{n(\log n + \tau)\})$, see (Ker09, Thm. 2.4.17). Similarly, for the asymptotically fast resultant algorithm, the bit complexity becomes $\tilde{O}(n^4 \tau \log n)$, see (Rei97).

Complexity of the parallel algorithm. By analogy with GCD computations, for Algorithm 2.10 ($\textsc{res\_mod}$) we could use $\delta = n^2$ processors distributing the computations over different evaluation points. Unfortunately, this would not lead to better overall complexity because the latter one is still determined by that of polynomial interpolation. That is why, we leave out the procedure $\textsc{res\_mod}$ and move to Algorithm 2.9. Here, the main bottleneck is again Chinese remaindering. However, we still hope to improve the total complexity by using $N = O(n(\tau + \log n))$ processors, so as to perform the computations modulo each prime in parallel. In this way, the complexity of the main loop of Algorithm 2.10 reduces to $O_P(n^2(n^2 + \tau), N)$ where the two factors contributing to this bound are the complexity of modular reduction (2.42) and that of invoking the subalgorithm $\textsc{res\_mod}$ (2.40). For Chinese remaindering, we use n^2 processors to recover each coefficient of the resultant in parallel, hence the complexity in (2.43) becomes $O_P(N^2, n^2)$. Combining the bounds together, we conclude that the resultant can be computed in

$$\begin{aligned} T_{\mathcal{B}}(\textsc{res\_int}(f, g \in \mathbb{Z}[x, y])) = O_P(n^2(n^2 + \tau), N) \;+\; O_P(N^2, n^2) = \\ O_P(n^2(n^2 + \tau^2), n\{\tau + n\}) \end{aligned} \tag{2.45}$$

parallel time.

To sum up, as we have seen, the estimated parallel performance of the modular GCD and resultant algorithms is largely limited by the fact that CRA and polynomial interpolation in their classical forms are not available for parallel implementation. Likewise, the Euclidean scheme lying in the core of both approaches further restricts the degree of parallelism provided by the modular algorithm. In the next chapter, we discuss matrix-based algorithms which partially allow us to solve the above problems.

In addition, our analysis of modular algorithms suggests that, in order to achieve a high performance, we should consider a parallel platform which has *two* levels of parallelism.

The upper level is “coarse-grained” where only loose communication between processors is needed: for example, for the solution of each modular problem in parallel. While the lower one is “fine-grained” where processors need to work in close cooperation to compute the result: this, for example, is required to realize parallel Chinese remaindering or univariate GCD computations.

3 Matrix algebra and symbolic computation

This chapter is one the main theoretical contributions of this thesis. We review an elegant mathematical theory of shift-structured matrices, and then exploit a deep connection between computations with polynomials and structured matrices to develop algorithms which permit efficient realization on massively-parallel architectures. At first glance, it seems not to make much sense since the modular approach discussed in the previous chapter is readily available for parallelization. However, the graphics hardware imposes additional requirements on the algorithms to be realized. Particularly, such algorithms must exhibit a high homogeneity of computations to enable data-level parallelism that can be usefully exploited on the GPU. The reason for this is because multiple threads executing on the GPU cannot be taken as "full-fledged" processors: conversely, they are optimized to perform *same* computations across *different* data elements.[1] This is where the matrix-based algorithms fit in: indeed, when a problem is expressed in terms of linear algebra, all data dependencies are usually made explicit providing a higher degree of parallelism. Such a level of parallelism is not commonly exploited on the traditional parallel platforms since it implies a close cooperation between threads (e.g., using shared memory) which, in most cases, has a negative impact on performance. In contrast, inter-thread communications on the GPU (within one thread block) are almost negligibly cheap.

This chapter is organized as follows. First, we introduce the theory of shift-structured matrices (or *displacement structure*), and outline the main algorithms operating on such matrices. Although, we shall try to keep the discussion self-contained, it should be noted that there is a large school of thought behind this theory, and therefore we will mostly concentrate only on the results which are relevant to the solution of some concrete matrix problems and skip further details. Afterwards, we will derive the matrix-based analogues for corresponding polynomial algorithms and integrate them to the modular approach. At the end of this chapter, we shall also revisit the resulting parallel complexity of the modular algorithms and compare it with the results obtained in Section 2.5.

3.1 Theory of shift-structured matrices

We begin with examples of structured matrices to give the reader some intuitive feeling which led to the concept of displacement structure. A comprehensive overview of this subject along with numerous applications can be found in (KS95). The interested

[1] This makes the GPU execution model very similar to SIMD, yet with several differences to be identified in Section 4.1.

reader may also find it informative to look in (CS98, SK95, KC94a). For asymptotically fast algorithms on matrices with structure including parallel solutions, we refer to (Pan01, Rei05). Some recent advances in the theory of structured matrices are summarized in (BMO$^+$10). We also remark that, in the present discussion, we do not consider any questions related to the *numerical stability* of such algorithms which is a large topic on its own. This is because our primary goal is to develop algorithms to work over a *finite field*, where these questions are irrelevant. To the best of our knowledge, we were the first to apply structured matrix algorithms in the modular setting while, initially, these algorithms were supposed to be used with inexact numeric computations as needed in many applied fields. In effect, this required some effort to integrate square root and division-free matrix transformations into the original approach.

Throughout this section, we suppose that the reader is familiar with the basics of linear algebra. For arbitrary matrices A and B, $A \oplus B$ denotes the Kronecker sum of them. By I_n we denote an identity matrix of size $n \times n$, and $\mathbf{0}_n$ is a zero matrix. Sometimes we will omit the subscripts simply writing I or $\mathbf{0}$ when the dimensions can be determined from the context. Besides, we shall also agree that matrix elements are indexed starting from $(0, 0)$, to be consistent with indexing of polynomial coefficients. In other words, the first diagonal element of an $n \times n$ matrix M will be denoted by $M_{0,0}$ and the last one by $M_{n-1,n-1}$.

3.1.1 Toeplitz, Hankel and related matrices

First, we consider *Toeplitz* matrices which arise in many theoretical and applied fields: in the solution of certain differential equations, signal and image processing, polynomial computation, Markov chains, etc. By definition, a *Toeplitz* matrix T is an $n \times n$ matrix for which it holds that $T_{i,j} = T_{i-1,j-1}$ $(i, j = 1, \dots, n - 1)$ or, in other words:

$$T = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{1-n} \\ t_1 & t_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-1} \\ t_{n-1} & \dots & t_1 & t_0 \end{bmatrix}. \tag{3.1}$$

Since T can be described by $2n - 1$ instead of n^2 parameters, it is quite natural to assume that operations on T (such as multiplication by a vector, triangular factorization, inversion) can be carried out faster than for general matrices. Next, by Z we denote the so-called *lower shift* matrix of size $n \times n$ which is zeroed everywhere except for ones on its first subdiagonal:

$$Z = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 1 & 0 \end{bmatrix}. \tag{3.2}$$

Multiplying an arbitrary matrix by Z from the left has the effect of shifting the contents of the matrix down by one position (row). Similarly, multiplying by Z^T from the right is the same as shifting the matrix to the right by one column position. It is now straightforward to check that the product ZTZ^T and the difference $R = T - ZTZ^T$ are of the following form:

$$ZTZ^T = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ 0 & t_0 & t_{-1} & \dots & t_{1-n} \\ \vdots & t_1 & t_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & t_{-1} \\ 0 & t_{n-1} & \dots & t_1 & t_0 \end{bmatrix}, \; R = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \dots & t_{1-n} \\ t_1 & 0 & 0 & \dots & 0 \\ t_2 & 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ t_{n-1} & 0 & \dots & 0 & 0 \end{bmatrix}.$$

Clearly, R contains all information to fully describe the matrix T and it has rank 2.[1] Using the fact that an $n \times n$ matrix of rank k can be decomposed as the product of $n \times k$ and $k \times n$ matrices, we can write: $R = GB^T$, where

$$G^T = \begin{bmatrix} t_0/2 & t_1 & t_2 & \dots & t_{n-1} \\ 1 & 0 & \dots & \dots & 0 \end{bmatrix}, \; B^T = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ t_0/2 & t_{-1} & t_{-2} & \dots & t_{1-n} \end{bmatrix}.$$

The matrices G, B will be called the *generators* of T as they offer a compact representation of T via the equation:

$$T - ZTZ^T = GB^T \tag{3.3}$$

known as *displacement equation*. In fact, we can write the explicit representation of T in terms of its generators. Indeed, giving that the lower shift matrix Z is nilpotent, i.e., $Z^n = \mathbf{0}$, the unique solution of (3.3) must satisfy:

$$T = \sum\nolimits_{i=0}^{n-1} Z^i GB^T (Z^T)^i. \tag{3.4}$$

To see why it holds, we multiply the displacement equation by the powers of Z and Z^T from both sides, yielding a sequence of identities:

$$ZTZ^T - Z^2T(Z^T)^2 = ZGB^TZ^T, \; \dots, \; Z^{n-1}T(Z^T)^{n-1} - \underbrace{Z^nT(Z^T)^n}_{0} = Z^{n-1}GB^T(Z^T)^{n-1}.$$

Summing up these equations with the original one gives the desired result. If we write each generator matrix as a pair of column vectors, i.e., $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$, the representation (3.4) is equivalent to:

$$T = L(\mathbf{a})L^T(\mathbf{c}) + L(\mathbf{b})L^T(\mathbf{d}), \tag{3.5}$$

where $L(\mathbf{x})$ denotes a lower-triangular Toeplitz matrix with the first column $\mathbf{x}$. Accordingly, the representation (3.5) implies T satisfies a displacement equation of the form (3.3). Then, a quite striking result from (GS72) states that the inverse of T also admits a similar representation:

$$T^{-1} = L(\tilde{\mathbf{a}})L^T(\tilde{\mathbf{c}}) + L(\tilde{\mathbf{b}})L^T(\tilde{\mathbf{d}}), \tag{3.6}$$

which is known as Gohberg-Semencul formula. As a consequence, T^{-1} must also satisfy (3.3), yet for different G and B. This suggests that T and T^{-1} have similar structure and there should be an "easy" way to transform one matrix to another. This will be the

[1] Every column of R starting from the 3rd one is a multiple of the 2nd column.

central topic of discussion in the next section. Another type of a structured matrix is the so-called *Hankel* matrix H which can be regarded as an "anti-diagonal" Toeplitz matrix:

$$H = \begin{bmatrix} h_0 & h_1 & \dots & h_{n-1} \\ h_1 & h_2 & \iddots & h_n \\ \vdots & \iddots & \iddots & \vdots \\ h_{n-1} & h_n & \dots & h_{2n-2} \end{bmatrix}. \tag{3.7}$$

To obtain a displacement equation for H, we introduce a *ϕ-circulant* matrix Z_ϕ which is a lower-shift matrix Z from (3.2) with an additional non-zero entry ϕ located at the top-right corner $(0, n-1)$. Then, one can write the following equation:

$$Z_1 H - HZ^T = GB^T, \tag{3.8}$$

where G, B are again matrices of size $n \times 2$ containing all necessary information to reconstruct H. This becomes clear if we write (3.8) in explicit form:

$$\underbrace{\begin{bmatrix} h_{n-1} & h_n & \dots & h_{2n-2} \\ h_0 & h_1 & \dots & h_{n-1} \\ \vdots & \iddots & \iddots & \vdots \\ h_{n-2} & h_{n-1} & \dots & h_{2n-3} \end{bmatrix}}_{Z_1 H} - \underbrace{\begin{bmatrix} 0 & h_0 & \dots & h_{n-2} \\ 0 & h_1 & \iddots & h_{n-1} \\ \vdots & \vdots & \iddots & \vdots \\ 0 & h_{n-1} & \dots & h_{2n-3} \end{bmatrix}}_{HZ^T} = \underbrace{\begin{bmatrix} h_{n-1} & h_n - h_0 & \dots & h_{2n-2} - h_{n-2} \\ h_0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ h_{n-2} & 0 & \dots & 0 \end{bmatrix}}_{GB^T}.$$

It is then not surprising that also the inverse of a Hankel matrix has a similar displacement representation. Alternatively, instead of Z_1 we could take a lower shift matrix to arrive at the equation $ZH - HZ^T = GB^T$ having the same displacement rank 2. However, in this case the matrix H cannot be recovered from its generators G and B without additional information.[1]

The idea we would like to illustrate is that there are many shift-structured matrices which are either of standard type (Toeplitz, Hankel, Vandermonde, etc.) or combinations thereof. Whenever a matrix can be identified with some displacement equation, its structure can be readily exploited to simplify operations on it. For example, in our case the Sylvester matrix (see in Section 2.4) is Toeplitz-like because it is formed of two "column-stacked" Toeplitz matrices. In the next section, we give a formal definition of a structured matrix and discuss the algorithms for such matrices.

3.1.2 Displacement structure and generalized Schur algorithms

We should remark that the algorithms for structured matrices, or generalized Schur algorithms, take slightly different forms depending on whether a matrix is *Hermitian* or not.[2] In this section, for expository purposes, we only consider the algorithms for Hermitian (symmetric) matrices, which are easier to derive, leaving the details for the non-Hermitian case to Section 3.1.3. Besides, there are two main types of displacement equations characterized by a *displacement operator*: the first one is of *Sylvester type* as in (3.8):

[1] This is because the matrix H falls into the kernel of a linear operator $\triangledown_{Z,Z}(M) = ZM - MZ^T$.

[2] A Hermitian matrix is a square matrix which is equal to its own conjugate transpose. When all elements of a matrix are real, this simply means that the matrix is symmetric.

$\nabla_{F,A}(M) = FM - MA^T$; and the other one is of *Stein type* as in (3.3): $\triangle_{F,A}(M) = M - FMA^T$, where F and A are arbitrary matrices (usually lower-triangular). Both operators are special cases of a more general displacement operator, see (KS95, § 7.4). In the present discussion, we shall only consider the second one which is relevant to the problems we are going to solve using displacement structure. Furthermore, it will be assumed that matrices under consideration have *real* entries while, in the most general form, displacement structure is defined for arbitrary complex matrices. This will help us avoid unnecessary burden with complex conjugations in the derivation of algorithms.

The algorithms for structured matrices have been evolved as a far-going generalization of the original work of Schur (Sch17) which was concerned with a quite different problem of deciding whether a power series is analytic and bounded in the unit disc; this explains the name – the generalized Schur algorithm. We start with the formal definition of a structured matrix.

Definition 3.1.1. Let $M \in \mathbb{R}^{n \times n}$ be a symmetric matrix with non-negative diagonal entries. M is said to have *displacement structure* if it satisfies the following displacement equation, see (KS95, § 4.3.1):

$$\triangle_{F,F}(M) := M - FMF^T = GJG^T, \tag{3.9}$$

where $F \in \mathbb{R}^{n \times n}$ is lower triangular, $G \in \mathbb{R}^{n \times r}$ is a *generator*, and $J = I_p \oplus -I_q$, with $r = p + q$, is a *signature matrix*. Here, r is called a *displacement rank* of M, and p and q is the number of strictly positive and negative eigenvalues of $\triangle_{F,F}(M)$, respectively. •

For illustration, let us consider a symmetric Toeplitz matrix T as in (3.1) with $t_0 > 0,\ t_1 = t_{-1},\ t_2 = t_{-2}, \ldots, t_{n-1} = t_{1-n}$. This matrix is structured and satisfies the equation:

$$T - ZTZ^T = GJG^T, \text{ where } J = 1 \oplus -1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

while simple computations show that:

$$G^T = \frac{1}{\sqrt{t_0}} \begin{bmatrix} t_0 & t_1 & t_2 & \ldots & t_{n-1} \\ 0 & t_1 & t_2 & \ldots & t_{n-1} \end{bmatrix}.$$

Observe that the generators are not uniquely defined: indeed, if G is a generator matrix of M, then for an arbitrary J-unitary matrix $\Theta \in \mathbb{R}^{r \times r}$, i.e. $\Theta J \Theta^T = J$, $G\Theta$ is also a valid generator for M because $G\Theta J \Theta^T G^T = GJG^T$. We will later see that this property of generators can be utilized to derive fast algorithms for structured matrices. For the non-Hermitian case, displacement structure is defined as follows.

Definition 3.1.2. Let $M \in \mathbb{R}^{n \times n}$ be an arbitrary non-symmetric matrix. M is said to have *displacement structure* if it satisfies the following displacement equation, see (KS95, § 4.3.1):

$$\triangle_{F,A}(M) := M - FMA^T = GB^T, \tag{3.10}$$

where $F, A \in \mathbb{R}^{n \times n}$ are lower triangular, $G, B \in \mathbb{R}^{n \times r}$ are *generator* matrices, and r is a *displacement rank* of M. •

Certainly, one of the most demanding operation in linear algebra is *triangular factorization*, while many other matrix problems, such as computing the inverse, solving a linear system or computing the determinant, can be easily reduced to that of matrix factorization. Here, the central role plays the so-called Schur complement as defined below.

Definition 3.1.3. A *Schur complement* R of a non-singular leading submatrix $M_{0,0}$ of M is defined as:

$$R = M_{1,1} - M_{1,0}M_{0,0}^{-1}M_{0,1}, \text{ where } M = \begin{bmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{bmatrix}. \qquad \bullet$$

Schur complements arise in the course of block Gaussian elimination. This can be exemplified as follows: suppose we have applied a standard Gaussian elimination to zero out the elements in the first column of the following strongly regular matrix $M \in \mathbb{R}^{n\times n}$ (a matrix is called *strongly regular* if its leading principal minors are non-zero):

$$M := \begin{bmatrix} d_0 & \hat{u}_0 \\ \hat{l}_0 & Q \end{bmatrix} \to \begin{bmatrix} d_0 & \hat{u}_0 \\ \mathbf{0} & \tilde{Q} \end{bmatrix}, \text{with } \hat{l}_0 \in \mathbb{R}^{(n-1)\times 1},\ d_0 \in \mathbb{R}, \text{ and } \hat{u}_0 \in \mathbb{R}^{1\times(n-1)}.$$

Then, it holds that $\tilde{Q} = Q - \hat{l}_0 d_0^{-1}\hat{u}_0$. In other words, $\tilde{Q}$ is a Schur complement of the scalar d_0 in matrix M. Proceeding further with Gaussian elimination, we next zero out the elements in the first column of the submatrix $\tilde{Q}$ yielding a Schur complement of the 2×2 leading block of the original matrix M, etc. Thus, denoting $l_i = [d_i \ \ \hat{l}_i]^T$ and $u_i = [d_i \ \ \hat{u}_i]$ in step i ($0 \le i < n$), we obtain a complete $LD^{-1}U$-factorization of M in the form:

$$M = l_0 d_0^{-1} u_0^T + \begin{bmatrix} 0 \\ l_1 \end{bmatrix} d_1^{-1} \begin{bmatrix} 0 \\ u_1 \end{bmatrix}^T + \begin{bmatrix} 0 \\ 0 \\ l_2 \end{bmatrix} d_2^{-1} \begin{bmatrix} 0 \\ 0 \\ u_2 \end{bmatrix}^T + \cdots = LD^{-1}U, \tag{3.11}$$

where L is a lower-triangular, U is an upper-triangular and D is a diagonal matrix. Clearly, if M is symmetric, we get an $LD^{-1}L^T$-factorization instead. Henceforth, we assume that all matrices under consideration are *strongly regular*: this ensures that successive Schur complements do exist or, equivalently, that Gaussian elimination can be carried out without partial pivoting. In fact, the strong-regularity assumption might be dropped if we apply the so-called *look-ahead* Schur algorithm instead, see (SK95), Here, non-strongly regular steps are replaced by *block* Schur complementation steps. We do not consider this algorithm here as it is quite sophisticated, and because, in the context of present work, we will mostly be dealing with strongly-regular matrices. The next theorem is fundamental and shows that Schur complements preserve displacement structure, thus providing us the way how to compute the factorization of a symmetric matrix.

Theorem 3.1.1 (generalized Schur algorithm): (KS95, Lem 7.1, Lem 7.3, Thm. 7.4) Let $M \in \mathbb{R}^{n\times n}$ be a symmetric strongly regular matrix that satisfies a displacement equation:

$$M - FMF^T = GJG^T,$$

where F, G and J are as in Definition 3.1.1. In addition, for the diagonal entries f_i of F it holds that

$$1 - f_i f_j \neq 0 \ \text{ for all } \ i, j.$$

Then, the successive Schur complements M_i of the $i \times i$ leading blocks of M are also structured, satisfying:

$$M_i - F_i M_i F_i^T = G_i J G_i^T, \ (0 \leq i < n) \qquad \diamond$$

where F_i is a submatrix obtained by deleting the first i rows and columns from F. $G_i \in \mathbb{R}^{(n-i)\times r}$ are generator matrices which obey the following recursive relation:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \left\{ G_i + \Big((F_i - f_i I_{n-1})(I_{n-i} - f_i F_i)^{-1} - I_{n-i} \Big) G_i J \frac{g_i^T g_i}{g_i J g_i^T} \right\} \Theta_i, \tag{3.12}$$

where $G_0 = G$, $g_i \in \mathbb{R}^{1\times r}$ is the top row of the matrix G_i, and $\Theta_i \in \mathbb{R}^{r\times r}$ is an arbitrary J-unitary matrix. The triangular factorization, $M = LD^{-1}L^T$, is determined by

$$l_i = (I_{n-i} - f_i F_i)^{-1} G_i J g_i^T, \ d_i = \frac{g_i J g_i^T}{1 - f_i^2}. \tag{3.13}$$

Proof The original proof can be found in (KS95). Here, we rewrite it in a more compact form (without first showing auxiliary results) to make the argument self-contained and (hopefully) easier to understand. It is enough to show the claim for $i = 0$, and the rest follows by induction. Thus, suppose we wish to compute the generator G_1 of the first Schur complement M_1 which by definition satisfies the following relation:

$$\tilde{M} = M - l_0 d_0^{-1} l_0^T = \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{0} & M_1 \end{bmatrix}. \tag{3.14}$$

Observe that one can write the displacement equation for M separately for individual matrix components implying that:

$$l_0 = F l_0 f_0 + G J g_0^T, \ \ d_0(1 - f_0^2) = g_0 J g_0^T. \tag{3.15}$$

Furthermore, since $1 - f_i f_j \neq 0$, we can solve for l_0 and d_0 explicitly:

$$l_0 = (I - f_0 F)^{-1} G J g_0^T, \ \ d_0 = g_0 J g_0^T / (1 - f_0^2), \tag{3.16}$$

which proves (3.13) for $i = 0$. The proof of the generator recursion (3.12) proceeds by "completion-of-squares" argument. First, we substitute $\tilde{M}$ to the original displacement equation and apply (3.15):

$$\begin{aligned} \tilde{M} - F\tilde{M}F^T &= F\frac{l_0 l_0^T}{d_0}F^T - \frac{l_0 l_0^T}{d_0} + GJG^T = F\frac{l_0 l_0^T}{d_0}F^T - \frac{(F l_0 f_0 + G J g_0^T)(g_0 J G_0^T + f_0 l_0^T F^T)}{d_0} + \\ &+ GJG^T = -F l_0 \frac{f_0 g_0}{d_0} J G_0^T - GJ\frac{g_0^T f_0}{d_0} l_0^T F^T + F l_0 \frac{1 - f_0^2}{d_0} l_0^T F^T + GJ\left\{ J - \frac{g_0^T g_0}{d_0} \right\} JG^T, \end{aligned} \tag{3.17}$$

where we use the fact that $GJG^T = GJ^3G^T$ because $J^2 = I_r$. Our goal is to factorize the above expression in the form $\tilde{G}J\tilde{G}^T$. By symmetry, we seek for $k_0, h_0 \in \mathbb{Z}^{r\times r}$ that fulfill the following equations:

$$\frac{f_0 g_0}{d_0} = -h_0 J k_0^T, \ \frac{g_0^T f_0}{d_0} = -k_0 J h_0^T, \ \frac{1 - f_0^2}{d_0} = h_0 J h_0^T, \ J - \frac{g_0^T g_0}{d_0} = k_0 J k_0^T. \tag{3.18}$$

We first show that $J - g_0^T g_0/d_0$ decomposes as $k_0 J k_0^T$. Observe that k_0 should be of the form: $(I_r - QJ)\Theta_0$, where $\Theta_0 \in \mathbb{R}^{r\times r}$ is an arbitrary J-unitary matrix, and $Q \in \mathbb{R}^{r\times r}$ is to be determined. Expanding the product $k_0 J k_0^T$, we conclude that Q must satisfy: $Q + Q^T - QJQ^T = g_0^T g_0/d_0$. To find such Q, we write the following identity using (3.15):

$$\frac{g_0^T g_0}{d_0} = g_0^T g_0 \left(\frac{(1-f_0)^2}{d_0(1-f_0)^2} \right) = g_0^T g_0 \left(\frac{2(1-f_0)}{d_0(1-f_0)^2} - \frac{d_0(1-f_0^2)}{d_0^2(1-f_0)^2} \right) = \frac{2g_0^T g_0}{d_0(1-f_0)} - \frac{g_0^T g_0 J g_0^T g_0}{d_0^2(1-f_0)^2}$$

$$= Q + Q^T - QJQ^T, \text{ and hence } Q = \frac{g_0^T g_0}{d_0(1-f_0)} = \frac{(1+f_0)g_0^T g_0}{g_0 J g_0^T}.$$

Next, using (3.18) and (3.15), one can choose h_0 as follows:

$$\frac{1-f_0^2}{d_0} = \frac{g_0 J g_0^T}{d_0^2} = h_0 J h_0^T, \text{ where } h_0 = \frac{g_0 J}{d_0}\Theta_0 = \frac{(1-f_0^2)g_0 J}{g_0 J g_0^T}\Theta_0. \tag{3.19}$$

It is then straightforward to verify that the remaining equations in (3.18) also hold:

$$h_0 J k_0^T = \frac{g_0 J}{d_0}\Theta_0 J \Theta_0^T \left\{ I_r - \frac{J g_0^T g_0}{d_0(1-f_0)} \right\} = \frac{g_0}{d_0} - \frac{d_0(1-f_0^2)g_0}{d_0^2(1-f_0)} = \frac{(f_0^2 - f_0)g_0}{d_0(1-f_0)} = -\frac{f_0 g_0}{d_0}.$$

Following the proof, a sceptic reader may object that we are somehow able to "foresee" the predefined form of expressions: this is true to some extent as we have tried to make the argument as simple as possible skipping some intermediate steps. Finally, using the obtained k_0 and h_0, we can rewrite (3.17) as

$$(F l_0 h_0 J + G J k_0 J) J (J k_0^T J G^T + J h_0^T l_0^T F^T) = \tilde{G} J \tilde{G}^T.^1 \tag{3.20}$$

Substituting the expressions for h_0 and k_0 to (3.20) together with (3.16) one can write the explicit form of $\tilde{G}$:

$$\tilde{G} = \left\{ F(I - f_0 F)^{-1} G J g_0^T \frac{(1-f_0^2) g_0 J^2}{g_0 J g_0^T} + GJ\left(I_r - \frac{(1+f_0) g_0^T g_0 J}{g_0 J g_0^T} \right) J \right\} \Theta_0 =$$

$$= \left\{ G + \Big((1-f_0^2) F (I - f_0 F)^{-1} - (1+f_0) I \Big) G J \frac{g_0^T g_0}{g_0 J g_0^T} \right\} \Theta_0 =$$

$$= \left\{ G + \Big((F - f_0 I)(I - f_0 F)^{-1} - I \Big) G J \frac{g_0^T g_0}{g_0 J g_0^T} \right\} \Theta_0 = \begin{bmatrix} 0 \\ G_1 \end{bmatrix},$$

where the last equality follows from (3.14). Indeed, since the first row and column of $\tilde{M}$ are zero, the first row of $\tilde{G}$ must be zero too, and the whole argument follows by induction on i. ■

The above theorem allows us to carry out the factorization process without explicitly constructing the Schur complements. Note that, it is not immediate to see that this algorithm actually improves upon the complexity of the matrix factorization. Indeed, the generator recursion (3.12) seems to be quite complicated, especially, as it involves computing matrices of the form $(I_{n-i} - f_i F_i)^{-1}$. However, in the vast majority of cases, the matrix F has

[1] Multiplying each term by J from the right is not necessary but it will help us simplify the expression.

a very simple form (for instance, $F = Z$) such that the matrix inverse can be written with an explicit formula or disappears altogether. In general, provided that the matrix multiplication and inverse in (3.12) can be done in linear time, one demands for $O(nr)$ elementary operations in $\mathbb{R}$ in each step of the algorithm. Here, r is a displacement rank of an $n \times n$ matrix under consideration; it usually holds that: $r \ll n$. The complete algorithm then runs in $O(n^2 r)$ arithmetic operations.

Remark that we have only considered the generalized Schur algorithm for symmetric (Hermitian) matrices: a similar algorithm for the non-Hermitian case can be found in (KS95, § 7.4.1). We do not present it here because, in the next section, we discuss the specialization of the Schur algorithm to “array form”, where the generator recursions are greatly simplified, both for Hermitian and non-Hermitian cases.

3.1.3 Array form of the generalized Schur algorithms

Note that, the term “array form” was introduced in (Kai87) while, in this form, the factorization algorithm consists of a sequence of elementary transformations applied to an array of matrix columns. The idea behind this algorithm is to exploit the fact that we can choose the free parameters Θ_i in Theorem 3.1.1. Before discussing the algorithm, we need to formally define what is meant by a *proper form* generator matrix.

Definition 3.1.4. Let $G \in \mathbb{R}^{n \times r}$ be the generator matrix for a symmetric matrix $M \in \mathbb{R}^{n \times n}$ as in Definition 3.1.1. The generator is said to be in a *proper form* if its top row g contains only a single non-zero element:

$$g = [\, 0 \;\ldots\; 0 \;\delta\; 0 \;\ldots\; 0 \,]. \qquad \bullet$$

The precise position of this non-zero element is yet unspecified but will be clarified later. For non-Hermitian matrices, a proper form generator pair is defined as follows.

Definition 3.1.5. Let $G, B \in \mathbb{R}^{n \times r}$ be a pair of generator matrices for a non-symmetric matrix $M \in \mathbb{R}^{n \times n}$ as in Definition 3.1.2. The generators G, B are said to be in a *proper form* if their top rows, denoted g and b, respectively, contain single non-zero elements at the *same* column position k:

$$g = [\, \underbrace{0 \;\ldots\; 0}_{k} \;\; \delta \;\; \underbrace{0 \;\ldots\; 0}_{r-k-1} \,], \;\; \textit{and} \;\; b = [\, \underbrace{0 \;\ldots\; 0}_{k} \;\; \lambda \;\; \underbrace{0 \;\ldots\; 0}_{r-k-1} \,]. \qquad \bullet$$

Transforming the generators to a proper form can be achieved in many ways: for instance, using elementary Givens/hyperbolic rotations or Householder reflections. We first demonstrate this for Hermitian matrices. Suppose we wish to transform a generator $G \in \mathbb{R}^{n \times 2}$ of a symmetric Toeplitz-like matrix to a proper form. For this, we need to find a matrix $\Theta \in \mathbb{R}^{2 \times 2}$ such that $\Theta J \Theta^T = J$ where $J = 1 \oplus -1$. If $g_0 = [\, a_0 \;\; b_0 \,]$ denotes the first row of G, then Θ is a hyperbolic rotation defined as:

$$\Theta = \begin{bmatrix} c & -s \\ -s & c \end{bmatrix} \begin{array}{l} \text{if } |a_0| > |b_0| \\ (g_0 J g_0^T > 0), \end{array} \quad \Theta = \begin{bmatrix} s & -c \\ -c & s \end{bmatrix} \begin{array}{l} \text{if } |a_0| < |b_0| \\ (g_0 J g_0^T < 0), \end{array} \tag{3.21}$$

where $c = a_0 / \sqrt{|a_0^2 - b_0^2|}$, $s = b / \sqrt{|a_0^2 - b_0^2|}$. It is easy to check that Θ is J-unitary matrix, while $g_0\Theta = [\, \sqrt{a_0^2 - b_0^2} \;\; 0]$ for $|a_0| > |b_0|$, and $g_0\Theta = [0 \;\; \sqrt{b_0^2 - a_0^2}]$ for $|a_0| < |b_0|$. Note that, the case $|a_0| = |b_0|$ is excluded by strong-regularity assumption. If a generator matrix has more than two columns, a combination of several Givens and hyperbolic rotations is necessary to bring the matrix to a proper form, where the profile of a signature matrix J must be taken into account, see (KS95, § 4.4.1). Sometimes rotation matrices which involve square roots and divisions are undesirable: for instance, when these operations are too expensive (such as in a finite field) or they are not supported at all. In this case, the solution is to use the so-called *square-root* and *division-free* transformations. This idea was initially developed for the classical QR-factorization of a matrix to reduce the number of expensive arithmetic operations; see, e.g. (FL94). In application to the generalized Schur algorithm, it works as follows. First, we write a generator matrix $G \in \mathbb{R}^{n\times 2}$ in the following form:

$$G^T = \begin{bmatrix} 1/\sqrt{l_a} & 0 \\ 0 & 1/\sqrt{l_b} \end{bmatrix} \begin{bmatrix} a_0 & a_1 & \dots & a_{n-1} \\ b_0 & b_1 & \dots & b_{n-1} \end{bmatrix}, \tag{3.22}$$

where, initially, $l_a = l_b = 1$. Then, if we compute $\tilde{G} = G\Theta$ using the rotation matrix from (3.21) assuming $|a_0| > |b_0|$, and again factor $\tilde{G}$ in the form (3.22), we can rewrite the matrix rotation without square roots and divisions, that is:

$$\begin{bmatrix} 1/\sqrt{l'_a} & 0 \\ 0 & 1/\sqrt{l'_b} \end{bmatrix}^{-1} \tilde{G}^T = (G\hat{\Theta})^T = \begin{bmatrix} \tilde{a}_0 & \tilde{a}_1 & \dots & \tilde{a}_{n-1} \\ 0 & \tilde{b}_1 & \dots & \tilde{b}_{n-1} \end{bmatrix} \text{ with } \hat{\Theta} = \begin{bmatrix} l_b a_0 & -l_a b_0 \\ -b_0 & a_0 \end{bmatrix},$$

and $l'_a = l_a l_b (l_b a_0^2 - l_a b_0^2) = l_a l_b \tilde{a}_0$, $l'_b = (l_b a_0^2 - l_a b_0^2) = \tilde{a}_0$. It is useful to consider a small example here.

Example 3.1.1. Let $G \in \mathbb{Z}^{5\times 2}$ be a generator for some “Toeplitz-like” matrix:

$$G^T = \begin{bmatrix} 3 & 9 & -1 & 11 & 2 \\ -7 & 4 & 7 & 15 & -3 \end{bmatrix}.$$

According to (3.21), we choose an appropriate hyperbolic rotation matrix Θ to transform G to a proper form:

$$\Theta = \frac{1}{2\sqrt{10}} \begin{bmatrix} -7 & -3 \\ -3 & -7 \end{bmatrix}, \text{ and } (G\Theta)^T = \frac{1}{2\sqrt{10}} \begin{bmatrix} 0 & -75 & -14 & -122 & -5 \\ 40 & -55 & -46 & -138 & 15 \end{bmatrix}.$$

Note that, $G\Theta$ has a zero element at $(0,0)$ since $|3| < |-7|$ in the first row of G. Now, using square-root and division-free form, we obtain the following matrices:

$$\tilde{\Theta} = \begin{bmatrix} -7 & -3 \\ -3 & -7 \end{bmatrix}, \text{ and } (G\tilde{\Theta})^T = \begin{bmatrix} 0 & -75 & -14 & -122 & -5 \\ 40 & -55 & -46 & -138 & 15 \end{bmatrix},$$

where $l'_a = (-7)^2 - 3^2 = 40$ and $l'_b = l'_a$. In other words, we have simply factored out a common denominator and stored it implicitly in the form of l'_a and l'_b. ♣

Suppose now $G, B \in \mathbb{R}^{n\times 2}$ is a generator pair of a non-Hermitian matrix, and let $g_0 = [\,a \;\; b\,]$, $b_0 = [\,c \;\; d\,]$ be the respective first rows of these matrices. We seek for a pair of “rotation” matrices $\Theta, \Gamma \in \mathbb{R}^{2\times 2}$ satisfying

$$[\,a \;\; b\,]\Theta = [\,\delta \;\; 0\,], \;\; [\,c \;\; d\,]\Gamma = [\,\lambda \;\; 0], \text{ and } \Theta\Gamma^T = I_2, \tag{3.23}$$

where the latter condition ensures that $(G\Theta, B\Gamma)$ is a valid generator pair. Using the above conditions, we find that

$$\Theta = \begin{bmatrix} c & b \\ d & -a \end{bmatrix}, \Gamma = \frac{1}{D}\begin{bmatrix} a & d \\ b & -c \end{bmatrix}, \text{and } D = ac + bd, \tag{3.24}$$

whereas $D \neq 0$ is guaranteed by *strong-regularity* assumption introduced in Section 3.1.2. Apparently, this is not the only way to define the rotation matrices, see (KS95, § 4.4.3) for an in-depth discussion. When the displacement rank of a matrix under consideration is greater than two, it becomes increasingly more difficult to find the right matrices as the number of parameters to choose grows quadratically. In this case, it is recommended to use Householder reflections, see (KS95, § 4.4.4). Having discussed the matrix transformations, we are ready now to derive a generalized Schur algorithm in array form. As usual we begin with a symmetric case.

Theorem 3.1.2 (array form of the generalized Schur algorithm): (KS95, Thm. 4.2) Let $M \in \mathbb{R}^{n\times n}$ be a symmetric strongly regular matrix that satisfies a displacement equation:

$$M - FMF^T = GJG^T,$$

where $F \in \mathbb{R}^{n\times n}$, $J \in \mathbb{R}^{r\times r}$ and $G \in \mathbb{R}^{n\times r}$ are as in Theorem 3.1.1 and Definition 3.1.1, $r = p + q$ is the sum of p strictly positive and q strictly negative eigenvalues of GJG^T. The successive Schur complements M_i of M with respect to its leading $i \times i$ blocks are also structured, satisfying:

$$M_i - F_i M_i F_i^T = G_i J G_i^T. \quad (0 \leq i < n)$$

The generators $G_i \in \mathbb{R}^{(n-i)\times r}$ can be computed using the following recurrence:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i G_i \Theta_i \begin{bmatrix} \mathbf{0}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}_{r-k-1} \end{bmatrix} + G_i \Theta_i \begin{bmatrix} I_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{r-k-1} \end{bmatrix}, \tag{3.25}$$

which should be understood as follows: "take the k-th column of the matrix $G_i\Theta_i$ and multiply it from the left by Φ_i keeping the rest columns of $G_i\Theta_i$ intact, the resulting matrix contains the next generator G_{i+1}." Here $G_0 = G$, $\Phi_i = (F_i - f_i I_{n-i})(I_{n-i} - f_i F_i)^{-1}$, and $\Theta_i \in \mathbb{R}^{r\times r}$ are J-unitary matrices chosen to annihilate all except one entry of the top row g_i of the G_i, that is:

$$g_i\Theta_i = [\,\underbrace{0 \;\ldots\; 0}_{k} \;\; \delta_i \;\; \underbrace{0 \;\ldots\; 0}_{r-k-1}\,].$$

◇

This entry has to be in the first p positions if $g_i J g_i^T > 0$ (i.e. $0 \leq k < p$), and in the last q positions if $g_i J g_i^T < 0$ (i.e. $k \geq p$), by strong-regularity $g_i J g_i^T \neq 0$. The triangular factorization, $M = LD^{-1}L^T$, is determined by

$$l_i = \delta_i (I_{n-i} - f_i F_i)^{-1} G_i \Theta_i J \begin{bmatrix} \mathbf{0} \\ 1 \\ \mathbf{0} \end{bmatrix}, \; d_i = \frac{J_{kk}\delta_i^2}{1 - f_i^2}, \tag{3.26}$$

Proof It should be noted that the conditions on the location of a non-zero element δ_i inside g_i are needed to guarantee that the proper form generator does actually exist. This can be proven using the hyperbolic singular value decomposition, see (KS95, Lem. 4.3). We omit these details here to keep the argument simple.

Although, this theorem admits an independent (and quite elegant) proof, we would like to show that it can be easily derived as a consequence of Theorem 3.1.1 to outline a deep connection between the two results. Hence, let $\tilde{G}_i = G_i\Theta_i$ and $\tilde{g}_i$ be its first row. Since $\tilde{G}_i$ is a valid generator, we can apply the recursions (3.12) to compute G_{i+1}. Owing to the special form of $\tilde{g}_i$, one can directly see that $\tilde{g}_i J\tilde{g}_i^T = J_{kk}\delta_i^2$, and thus

$$J\frac{\tilde{g}_i^T\tilde{g}_i}{\tilde{g}_i J\tilde{g}_i^T} = \begin{bmatrix} \mathbf{0}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}_{r-k-1} \end{bmatrix} := \Omega.$$

Then, straightforward manipulations show that (3.12) transforms to:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \tilde{G}_i + \Phi_i\tilde{G}_i\Omega - \tilde{G}_i\Omega,$$

from where (3.25) follows immediately. To arrive at (3.26), we again expand the corresponding formulas for triangular factors (3.13) and use the special structure of $\tilde{G}_i$. ■

As before, each step of the generator recursion (3.25) demands for $O(nr)$ operations in $\mathbb{R}$, if we assume that matrices Φ_i can be computed in linear time. Then, the total complexity of the algorithm is bounded by $O(n^2r)$ arithmetic operations.

Example 3.1.2. To exemplify how the algorithm works, we consider the factorization of a symmetric Toeplitz matrix:

$$T = \begin{bmatrix} 9 & -4 & 2 & 5 & 13 \\ -4 & 9 & -4 & 2 & 5 \\ 2 & -4 & 9 & -4 & 2 \\ 5 & 2 & -4 & 9 & -4 \\ 13 & 5 & 2 & -4 & 9 \end{bmatrix} \quad \text{with} \quad \begin{array}{l} T - ZTZ^T = GJG^T,\ J = 1 \oplus -1, \\ G^T = 1/3\begin{bmatrix} 9 & -4 & 2 & 5 & 13 \\ 0 & -4 & 2 & 5 & 13 \end{bmatrix}. \end{array}$$

In this case, the recursion (3.25) simplifies to:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = Z_iG_i\Theta_i\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + G_i\Theta_i\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$

where Z_i denotes a matrix obtained by deleting the first i rows and columns from a lower shift matrix Z, see (3.2). Note that, $G_0 := G$ is already in a proper form, hence we can directly obtain the next generator matrix and extract the first triangular factor (d_0, l_0) according to (3.26):

$$G_1^T = \frac{1}{3}\begin{bmatrix} 9 & -4 & 2 & 5 \\ -4 & 2 & 5 & 13 \end{bmatrix},\ l_0^T = G_{0,0}\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}G^T = \begin{bmatrix} 9 & -4 & 2 & 5 & 13 \end{bmatrix},\ d_0 = 9.$$

Next, we apply a hyperbolic rotation (3.21) to G_1 computing $\tilde{G}_1 = G_1\Theta_1$ and G_2:

$$\tilde{G}_1^T = \frac{1}{3\sqrt{65}}\begin{bmatrix} 65 & -28 & 38 & 97 \\ 0 & 2 & 53 & 137 \end{bmatrix},\ G_2^T = \frac{1}{3\sqrt{65}}\begin{bmatrix} 65 & -28 & 38 \\ 2 & 53 & 137 \end{bmatrix},$$

while the second triangular factor taken from the first column of $\tilde{G}_1$ is:

$$l_1^T = \tfrac{1}{9}\begin{bmatrix} 65 & -28 & 38 & 97 \end{bmatrix},\ d_1 = \tfrac{65}{9}.$$

Continuing this process in the same manner, one obtains:

$$\tilde{G}_2^T = \frac{1}{\sqrt{30485}}\begin{bmatrix} 469 & -214 & 244 \\ 0 & 389 & 981 \end{bmatrix},\ G_3^T = \frac{1}{\sqrt{30485}}\begin{bmatrix} 469 & -214 \\ 389 & 981 \end{bmatrix},$$
$$l_2^T = \tfrac{1}{65}\begin{bmatrix} 469 & -214 & 244 \end{bmatrix},\ d_2 = \tfrac{469}{65},$$

and finally:

$$\tilde{G}_3^T = \frac{1}{4\sqrt{30954}}\begin{bmatrix} 1056 & -7415 \\ 0 & 8359 \end{bmatrix},\ G_4^T = \frac{1}{4\sqrt{30954}}\begin{bmatrix} 1056 \\ 8359 \end{bmatrix},\ \tilde{G}_4^T = \begin{bmatrix} 0 \\ \sqrt{\frac{146605}{1056}} \end{bmatrix},\ .$$
$$l_3^T = \tfrac{1}{469}\begin{bmatrix} 1056 & -7415 \end{bmatrix},\ d_3 = \tfrac{1056}{469},\ d_4 = -\tfrac{146605}{1056}.$$

Observe that the last generator $\tilde{G}_4$ has a non-zero entry at the location $(0, 1)$. The latter is due to the fact that for G_4 it holds that $g_4 J g_4^T < 0$ (or $1056 < 8359$). Now, to visualize the factorization, we can merge the diagonal elements d_i into one of the factors, that is, $LD^{-1}L^T \rightarrow LU^T$:

$$L = \begin{bmatrix} 9 & 0 & 0 & 0 & 0 \\ -4 & \frac{65}{9} & 0 & 0 & 0 \\ 2 & \frac{-28}{9} & \frac{469}{65} & 0 & 0 \\ 5 & \frac{38}{9} & \frac{-214}{65} & \frac{1056}{469} & 0 \\ 13 & \frac{97}{9} & \frac{244}{65} & \frac{-7415}{469} & \frac{-146605}{1056} \end{bmatrix},\ U^T = \begin{bmatrix} 1 & \frac{-4}{9} & \frac{2}{9} & \frac{5}{9} & \frac{13}{9} \\ 0 & 1 & \frac{-28}{65} & \frac{38}{65} & \frac{97}{65} \\ 0 & 0 & 1 & \frac{-214}{469} & \frac{244}{469} \\ 0 & 0 & 0 & 1 & \frac{-7415}{1056} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad \clubsuit$$

Theorem 3.1.3 (array form of the generalized Schur algorithm 2): (KS95, Cor. 7.16) Let $M \in \mathbb{R}^{n \times n}$ be a non-symmetric strongly regular matrix that satisfies a displacement equation:

$$M - FMA^T = GB^T,$$

where $F, A \in \mathbb{R}^{n \times n}$, $J \in \mathbb{R}^{r \times r}$ and $G \in \mathbb{R}^{n \times r}$ are as in Definition 3.1.2. In addition, for the diagonal entries f_i, a_i of F and A it holds that

$$1 - f_i a_j \neq 0 \ \text{ for all } \ i, j.$$

Then, the successive Schur complements M_i of M with respect to its leading $i \times i$ blocks are also structured, satisfying:

$$M_i - F_i M_i A_i^T = G_i B_i^T,\ \ (0 \leq i < n)$$

where F_i and A_i are obtained from F and A, respectively, by deleting the first i rows and columns. The generators $G_i, B_i \in \mathbb{R}^{(n-i) \times r}$ can be computed using the following recurrences:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i G_i \Theta_i \begin{bmatrix} \mathbf{0}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}_{r-k-1} \end{bmatrix} + G_i \Theta_i \begin{bmatrix} I_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{r-k-1} \end{bmatrix}, \qquad (3.27)$$

$$\begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i B_i \Gamma_i \begin{bmatrix} \mathbf{0}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}_{r-k-1} \end{bmatrix} + B_i \Gamma_i \begin{bmatrix} I_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{r-k-1} \end{bmatrix}, \tag{3.28}$$

where $(G_0, B_0) = (G, B)$, $\Phi_i = (F_i - f_i I_{n-i})(I_{n-i} - a_i F_i)^{-1}$, and $\Psi_i = (A_i - a_i I_{n-i})(I_{n-i} - f_i A_i)^{-1}$. $\Theta_i, \Gamma_i \in \mathbb{R}^{r \times r}$ are matrices satisfying $\Theta_i \Gamma_i^T = I_r$ chosen to annihilate all except one entry of the top rows g_i and b_i of G_i and B_i, respectively:

$$g_i \Theta_i = [\, \underbrace{0 \ \ldots \ 0}_{k} \ \ \delta_i \ \ \underbrace{0 \ \ldots \ 0}_{r-k-1} \,], \ \text{ and } \ b_i \Gamma_i = [\, \underbrace{0 \ \ldots \ 0}_{k} \ \ \lambda_i \ \ \underbrace{0 \ \ldots \ 0}_{r-k-1} \,].$$

◇

The triangular factors of $M = LD^{-1}U$ are given by: $d_i = \delta_i \lambda_i / (1 - f_i a_i)$,

$$l_i = \lambda_i (I_{n-i} - a_i F_i)^{-1} G_i \Theta_i \begin{bmatrix} \mathbf{0} \\ 1 \\ \mathbf{0} \end{bmatrix}, \ u_i^T = \delta_i (I_{n-i} - f_i A_i)^{-1} B_i \Gamma_i \begin{bmatrix} \mathbf{0} \\ 1 \\ \mathbf{0} \end{bmatrix}. \tag{3.29}$$

Proof As before, we prove the theorem by induction on i. For the first Schur complement M_1 of M we can write the following equation:

$$\tilde{M} = M - l_0 d_0^{-1} u_0 = \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{0} & M_1 \end{bmatrix}. \tag{3.30}$$

Using the displacement equation for M and the condition $1 - f_i a_j \neq 0$, one can write the triangular factors in explicit form:

$$l_0 = (I - a_0 F)^{-1} G b_0^T, \ \ u_0 = g_0 B^T (I - f_0 A^T)^{-1}, \ \ d_0 = g_0 b_0^T / (1 - f_0 a_0). \tag{3.31}$$

Then, we multiply the generators by rotation matrices to obtain: $\tilde{G} = G\Theta_0$ and $\tilde{B} = B\Gamma_0$ with respective top rows $\tilde{g}_0$ and $\tilde{b}_0$. Given that $\tilde{G}$ and $\tilde{B}$ are in the proper form, we can express the product GB^T in the following way:

$$GB^T = \tilde{G}\tilde{B}^T = \sum_{i=1, i \neq k}^{r} \begin{bmatrix} 0 \\ \mathbf{w}_i \end{bmatrix} \begin{bmatrix} 0 \\ \mathbf{v}_i \end{bmatrix}^T + \mathbf{w}_k \mathbf{v}_k^T = \Lambda + \mathbf{w}_k \mathbf{v}_k^T, \tag{3.32}$$

where $\mathbf{w}_i$ and $\mathbf{v}_i$ are the columns of $\tilde{G}$ and $\tilde{B}$, respectively. Next, from (3.31) using (3.32) one obtains:

$$l_0 = \lambda_0 (I - a_0 F)^{-1} \mathbf{w}_k, \ \ u_0 = \delta_0 \mathbf{v}_k^T (I - f_0 A^T)^{-1}, \ \ d_0 = \delta_0 \lambda_0 / (1 - f_0 a_0), \tag{3.33}$$

which shows (3.29) for $i = 0$. We attempt to put a displacement equation for $\tilde{M}$ into "perfect square" form. Using (3.30) together with (3.32) and (3.33) yields

$$\begin{aligned} \tilde{M} - F\tilde{M}A^T = GB^T - \frac{l_0 u_0}{d_0} + F\frac{l_0 u_0}{d_0} A^T = \underbrace{GB^T}_{\Lambda + \mathbf{w}_k \mathbf{v}_k^T} - (1 - f_0 a_0) \underbrace{(I - a_0 F)^{-1} \mathbf{w}_k \mathbf{v}_k^T (I - f_0 A^T)^{-1}}_{\Delta} \\ + (1 - f_0 a_0) F \underbrace{(I - a_0 F)^{-1} \mathbf{w}_k \mathbf{v}_k^T (I - f_0 A^T)^{-1}}_{\Delta} A^T = \Lambda + (I - a_0 F) \Delta (I - f_0 A^T) \\ - (1 - f_0 a_0) \Delta + (1 - f_0 a_0) F \Delta A^T = \Lambda + (F - f_0 I) \Delta (A - a_0 I)^T = \Lambda + \Phi_0 \mathbf{w}_k \mathbf{v}_k^T \Psi_0^T \end{aligned} \tag{3.34}$$

From the last equation, it is clear that $\tilde{M} - F\tilde{M}A^T$ can be factored in the form $\begin{bmatrix} 0 \\ G_1 \end{bmatrix} \begin{bmatrix} 0 \\ B_1 \end{bmatrix}^T$ as defined in (3.27) and (3.28) since

$$\Lambda = G\Theta_0 \begin{bmatrix} I_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{r-k-1} \end{bmatrix} \Gamma_0^T B^T, \quad \mathbf{w}_k \mathbf{v}_k^T = G\Theta_0 \begin{bmatrix} \mathbf{0}_k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}_{r-k-1} \end{bmatrix} \Gamma_0^T B^T,$$

and the first row and column of $\tilde{M}$ are zeroed everywhere. The complete argument follows by induction on i. ■

Apparently, the complexity analysis used for Theorems 3.1.1 and 3.1.2 does also apply in the asymmetric (non-Hermitian) case which implies that the complexity of the factorization algorithm is bounded by $O(n^2 r)$ arithmetic operations in $\mathbb{R}$.

Example 3.1.3. We can now modify the previous example by taking an asymmetric Toeplitz matrix instead:

$$T = \begin{bmatrix} 13 & 5 & 2 & -4 & 9 \\ 3 & 13 & 5 & 2 & -4 \\ -1 & 3 & 13 & 5 & 2 \\ -2 & -1 & 3 & 13 & 5 \\ 8 & -2 & -1 & 3 & 13 \end{bmatrix} \quad \text{with} \quad \begin{aligned} & T - ZTZ^T = GB^T, \\ & G^T = \begin{bmatrix} 13 & 3 & -1 & -2 & 8 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \\ & B^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 5 & 2 & -4 & 9 \end{bmatrix}. \end{aligned}$$

The recursions (3.27) and (3.28) can be written as:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = Z_i G_i \Theta_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + G_i \Theta_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = Z_i B_i \Gamma_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + B_i \Gamma_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

For reasons of space, we shall only display the proper form generators in each step of the algorithm. Setting $(G_0, B_0) := (G, B)$ and applying the rotation formulas in (3.24) to compute $(\tilde{G}_0, \tilde{B}_0) = (G_0\Theta_0, B_0\Gamma_0)$, one obtains:

$$\tilde{G}_0^T = \begin{bmatrix} 13 & 3 & -1 & -2 & 8 \\ 0 & 3 & -1 & -2 & 8 \end{bmatrix}, \quad \tilde{B}_0^T = \frac{1}{13} \begin{bmatrix} 13 & 5 & 2 & -4 & 9 \\ 0 & -5 & -2 & 4 & -9 \end{bmatrix}.$$

Then, we extract the first triangular factors (l_0, u_0, d_0) using (3.29):

$$l_0^T = \begin{bmatrix} 13 & 3 & -1 & -2 & 8 \end{bmatrix}, \quad u_0 = \begin{bmatrix} 13 & 5 & 2 & -4 & 9 \end{bmatrix}, \quad d_0 = 13.$$

The next generator pair (G_1, B_1) can be obtained simply by shifting the first columns of the matrices $\tilde{G}_0$ and $\tilde{B}_0$, down by one row position, respectively. Thus, the proper generators in the second step along with triangular factors are written as follows:

$$\begin{aligned} & \tilde{G}_1^T = \tfrac{1}{7} \begin{bmatrix} 7 & 2 & -3 & -3 \\ 0 & 13 & 299 & -65 \end{bmatrix}, \quad \tilde{B}_1^T = \tfrac{1}{169} \begin{bmatrix} 2002 & 767 & 494 & -1027 \\ 0 & 1 & -62 & 137 \end{bmatrix} \\ & l_1^T = \tfrac{1}{13} \begin{bmatrix} 154 & 44 & -3 & -66 \end{bmatrix}, \quad u_1 = \tfrac{1}{7} \begin{bmatrix} 154 & 59 & 38 & -79 \end{bmatrix}, \quad d_1 = \tfrac{154}{13}. \end{aligned}$$

In the third step of the Schur algorithm, we have:

$$\begin{aligned} & \tilde{G}_2^T = \tfrac{1}{1078} \begin{bmatrix} 12782 & 3661 & -308 \\ 0 & -1521 & 9971 \end{bmatrix}, \quad \tilde{B}_2^T = \tfrac{1}{14027} \begin{bmatrix} 14027 & 4563 & 5239 \\ 0 & 5173 & -11340 \end{bmatrix} \\ & l_2^T = \tfrac{1}{154} \begin{bmatrix} 1826 & -523 & -44 \end{bmatrix}, \quad u_2 = \tfrac{1}{7} \begin{bmatrix} 83 & 27 & 31 \end{bmatrix}, \quad d_2 = \tfrac{83}{7}. \end{aligned}$$

Carrying out this process further, we compute the remaining triangular factors (we do not show the generator expressions as they become quite complicated):

$$l_3^T = \tfrac{1}{1826}\begin{bmatrix} 20701 & 12430 \end{bmatrix},\ u_3 = \tfrac{1}{1826}\begin{bmatrix} 20701 & 9126 \end{bmatrix},\ d_3 = \tfrac{20701}{1826},\ d_4 = \tfrac{40634}{20701}.$$

Again, merging the diagonal elements with a lower-triangular part, i.e., $LD^{-1}U^T \to L_1U^T$, yields:

$$L_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{3}{13} & 1 & 0 & 0 & 0 \\ \frac{-1}{13} & \frac{2}{7} & 1 & 0 & 0 \\ \frac{-2}{13} & \frac{-3}{154} & \frac{523}{1826} & 1 & 0 \\ \frac{8}{13} & \frac{-3}{7} & \frac{-2}{83} & \frac{12430}{20701} & 1 \end{bmatrix},\ U^T = \begin{bmatrix} 13 & 5 & 2 & -4 & 9 \\ 0 & \frac{154}{13} & \frac{59}{13} & \frac{38}{13} & \frac{-79}{13} \\ 0 & 0 & \frac{83}{7} & \frac{27}{7} & \frac{31}{7} \\ 0 & 0 & 0 & \frac{20701}{1826} & \frac{4563}{913} \\ 0 & 0 & 0 & 0 & \frac{40634}{20701} \end{bmatrix}. \quad \clubsuit$$

Now, we have all necessary background to derive the matrix-based analogues for the required polynomial algorithms. But before doing this, we would like to introduce some useful methods illustrating how to apply the algorithms given in this and the previous section to the solution of various structured matrix problems. This should also help the reader gain a deeper insight into the nature of matrix computations.

3.1.4 Developing algorithms for structured matrices

In this section, we tried to collect a number of techniques showing that the family of Schur algorithms is indeed a very powerful and flexible tool which can provide a general solution to the diversity of matrix problems. Many of these and other techniques are highlighted in (KS95, Pan01). The key idea behind them is that each step of the factorization algorithm can be written in a matrix notation using Schur complements such that we can always form a "composite" matrix whose Schur complement, after a certain number of steps, gives precisely the result we wish to compute. In what follows, our central object of manipulations will be a non-Hermitian strongly-regular matrix $M \in \mathbb{R}^{n\times n}$ with a displacement equation: $M - FMA^T = GB^T$ with $G, B \in \mathbb{R}^{n\times r}$. However, the considerations below apply on an equal basis to symmetric matrices by changing the displacement equation accordingly.

Simultaneous factorization of a matrix and its inverse. Suppose, our goal is to compute the triangular factors of M and M^{-1}. To achieve this, we construct the following embedding $W \in \mathbb{R}^{2n\times 2n}$ of M:

$$W = \begin{bmatrix} -M & I_n \\ I_n & \mathbf{0} \end{bmatrix},\ \text{such that}\ W - \hat{F}W\hat{A}^T = \hat{G}\hat{B}^T,$$

where $\hat{F} = F \oplus \mathbb{Z}_n$ and $\hat{A} = A \oplus \mathbb{Z}_n$. We remark that, W can have a slightly higher displacement rank than M but it is still much lower than the full rank of M. For instance, it is known that if M is a Toeplitz-like matrix, then the displacement rank of W cannot exceed 4, see (KC94b). After running n steps of the Schur recursion, we obtain the partial triangularization of the form:

$$\begin{bmatrix} -M & I_n \\ I_n & \mathbf{0} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} D^{-1} \begin{bmatrix} U_1 & U_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & M^{-1} \end{bmatrix},$$

where the second (matrix) term on the right-hand side is the Schur complement of submatrix $-M$ in W. Equating the terms on both sides, implies that

$$-M = L_1 D^{-1} U_1, \; I = L_2 D^{-1} U_1, \; I = L_1 D^{-1} U_2,$$

and therefore $-M^{-1} = L_2 D^{-1} U_2$ where L_2 and U_2^T are necessarily upper triangular because W is banded. Note that the triangular factors L_2 and U_2 are given implicitly in the form of matrix generators $(\hat{G}, \hat{B})$.

Solving a linear system of equations. Now, assume that we aim at solving the following linear system: $M\mathbf{x} = \mathbf{b}$, where $\mathbf{b} \in \mathbb{R}^{n \times 1}$. To accomplish this, we could, of course, compute the triangular factorization of M, and then use back-substitution. Yet, we can do better by exploiting the properties of Schur complements. Similarly to the previous case, we construct an auxiliary matrix W which is now of size $2n \times (n+1)$:

$$W = \begin{bmatrix} M & -\mathbf{b} \\ I_n & \mathbf{0} \end{bmatrix}, \text{ and } W - \hat{F} W \hat{A}^T = \hat{G} \hat{B}^T,$$

with $\hat{F} = F \oplus Z_n$ and $\hat{A} = A \oplus 0$. Then, n steps of the generalized Schur algorithm yields the Schur complement R of the submatrix M of W which equals precisely:

$$R = \mathbf{0} - I_n(-M)^{-1}\mathbf{b} = M^{-1}\mathbf{b},$$

the solution of our linear system. In fact, R is simply computed as a product of generators $\hat{G}_n \hat{B}_n^T$ which in step n are of size $n \times r$ and $1 \times r$, respectively.

Orthogonal factorization. As a next example, consider the task of computing an orthogonal or QR-factorization of M, where R is upper-triangular and Q is a unitary matrix. For that, we form a *symmetric* matrix $W \in \mathbb{R}^{2n \times 2n}$ in the following way:

$$W = \begin{bmatrix} M^T M & M^T \\ M & \mathbf{0} \end{bmatrix}, \text{ and } W - \hat{F} W \hat{F}^T = \hat{G} J \hat{G}^T,$$

where $\hat{F} = Z_n \oplus Z_n$. Note that, it is usually not necessary to calculate the matrix product $M^T M$ explicitly because the generator $\hat{G}$ can be expressed in terms of the coefficients of M. After n steps of the factorization algorithm, we shall have the partial triangularization:

$$\begin{bmatrix} M^T M & M^T \\ M & \mathbf{0} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} D^{-1} \begin{bmatrix} L_1^T & L_2^T \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -I \end{bmatrix}.$$

From the above equality, we conclude that:

$$M^T M = L_1 D^{-1} L_1^T, \; L_2 D^{-1} L_1^T = M, \; L_2 D^{-1} L_2^T = I,$$

which, in particular, shows that $L_2 D^{-1/2}$ is a unitary matrix, and the QR-factorization of M is given by:

$$Q = L_2 D^{-1/2}, \; R = (L_1 D^{-1/2})^T,$$

where D is a diagonal matrix with positive elements (because $M^T M$ is positive-definite) and by $D^{1/2}$ we denote a diagonal matrix whose entries are the square roots of the entries of D.

3.2 Polynomial algorithms in matrix algebra setting

Recall that, at the end of Section 2.5 we came to conclusion that the theoretical parallel performance of the modular GCD and resultant algorithm is restricted by the fact that the basic subalgorithms (such as computing a GCD in $\mathbb{Z}_m[x]$ or interpolation) in their original forms are not readily available for parallelization. In this section, we reconsider these subalgorithms using the displacement structure approach to enable parallel processing.

3.2.1 Resultant by factorization of Sylvester's matrix

We begin with the derivation of the univariate resultant algorithm. This algorithm originally appeared in (Eme10c), and was later improved in (Eme10b).

Let $f, g \in \mathbb{Z}[x]$ be polynomials of degrees p and q, respectively; and $S \in \mathbb{Z}^{n\times n}$ $(n = p + q)$ be the associated Sylvester's matrix. Using the theory from the previous sections, we can write a displacement equation for S whose displacement rank is 2:

$$S^T - FS^TA^T = GB^T,^1 \text{ where } F = Z_n, A = Z_q \oplus Z_p, \tag{3.35}$$

and by Z_n we denote a lower shift matrix of size $n \times n$. The corresponding generators $G, B \in \mathbb{Z}^{n\times 2}$ can be easily expressed in terms of the coefficients of f and g:

$$G^T = \underbrace{\begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0 \dots 0 \\ g_q & q_{q-1} \dots & g_0 & 0 & \dots \quad 0 \end{bmatrix}}_{n=p+q}, \quad \begin{matrix} B \equiv 0, \text{ except for} \\ B_{0,0} = B_{q,1} = 1. \end{matrix} \tag{3.36}$$

Note that, it is not the only possible way to write a displacement equation for S: we could also use a classical "Toeplitz" displacement operator Δ_{Z,Z^T} (see Section 3.1.2) to obtain generators of the same size. However, the equation (3.35) leads to simpler generator expressions, and hence to a slightly more efficient final algorithm. In general, there is no universal procedure to find an optimal displacement equation for a concrete matrix problem at hand. This process is largely based on the "intuitive" reasoning and the ability to "foresee" the final generator recursions which lead to many forms of generalized Schur algorithms. A good exposition for this will be given in Section 3.2.4 where we develop an algorithm to compute the resultant cofactors (see Section 2.4.1 for definition).

For now, let us get back to the resultant computation. Our goal is to obtain the triangular factorization of S. Then, by elementary properties of matrix determinants, the resultant equals to the product of diagonal elements. For the time being, we assume that S is *strongly regular*, which is of course not always the case. The techniques how to deal with non-strong regularity will be elaborated upon in Section 4.4.2. For matrices F and A in (3.35) the condition $1-f_ia_j \neq 0$ of Theorem 3.1.3 is trivially satisfied since the diagonal entries are identically zero. Thus, for $(\tilde{G}_i, \tilde{B}_i) = (G_i\Theta_i, B_i\Gamma_i)$ the generator recursion can be written in the following form $(0 \le i < n)$:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = F_i\tilde{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = A_i\tilde{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \tag{3.37}$$

[1] There is no conceptual difference in considering S or S^T, both cases lead to the same result. However, we use S^T here to keep our equations consistent with the definition of Sylvester's matrix in Section 2.4.

Algorithm 3.1 Resultants by factoring Sylvester's matrix

```
procedure RESULTANT_SYLVESTER(f : Polynomial, g : Polynomial)
    $p = \deg(f),\ q = \deg(g),\ n = p + q$
    $f \leftarrow f/f_p$                                  ▷ convert f to monic polynomial
    let $G = (\mathbf{a}, \mathbf{b}),\ B = (\mathbf{c}, \mathbf{d})$     ▷ set up the generator pair as in (3.36)
    for $j = 0$ to $q - 1$ do        ▷ "lite" iterations of the algorithm: only the column b is updated
        for $i = j + 1$ to $p + j$ do                      ▷ multiply by rotation matrix
            $\mathbf{b}_i = \mathbf{b}_i - \mathbf{a}_i\mathbf{b}_j$
        od
        $\mathbf{c}_{2q-j} = \mathbf{b}_j$                        ▷ update a single entry of c
        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i$   for $\forall i = j \dots n - 2$              ▷ shift down the first column
    od
    $l_a = 1,\ l_c = 1,\ res = 1,\ l_{res} = 1$        ▷ initialize the common denominators and resultant
    for $j = q$ to $n - 1$ do     ▷ the remaining "full" iterations: all generator columns participate
        for $i = j$ to $n - 1$ do                          ▷ multiply with the rotation matrix
            $\mathbf{a}_i = l_a(\mathbf{a}_i\mathbf{c}_j + \mathbf{b}_i\mathbf{d}_j),\ \mathbf{b}_i = l_c(\mathbf{a}_i\mathbf{b}_j - \mathbf{b}_i\mathbf{a}_j)$
            $\mathbf{c}_i = l_c(\mathbf{c}_i\mathbf{a}_j + \mathbf{d}_i\mathbf{b}_j),\ \mathbf{d}_i = l_a(\mathbf{c}_i\mathbf{d}_j - \mathbf{d}_i\mathbf{c}_j)$
        od
        $l_c = l_a l_c^2,\ l_a = \mathbf{a}_j,\ res = res \cdot \mathbf{c}_j,\ l_{res} = l_{res} \cdot l_c$   ▷ update the denominators and the resultant
        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i, \mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$   for $\forall i = j \dots n - 2$    ▷ shift down the first columns of G and B
    od
    return $res \cdot (f_p)^q / l_{res}$                   ▷ compensate for monic polynomial
end procedure
```

where G_i, B_i are matrices of size $(n - i) \times 2$, and F_i (or A_i) is obtained from F (or A) by deleting the first i columns and rows. The matrices Θ_i and Γ_i are chosen to transform the top rows g_i and b_i of G_i and B_i, respectively, to the form:

$$g_i\Theta_i = [\ \delta_i \ \ 0\ ], \quad \text{and} \quad b_i\Gamma_i = [\ \lambda_i \ \ 0\ ].$$

The relation (3.37) should be read as follows: "*multiply the first column of $\tilde{G}_i$ (or $\tilde{B}_i$) from the left by a corresponding matrix F_i (or A_i), leaving the second column of $\tilde{G}_i$ (or $\tilde{B}_i$) intact.*" As noted earlier, the effect of multiplying by a lower shift matrix is the same as shifting the contents of a matrix down by one row position, hence the multiplication by F_i or A_i does not involve any arithmetic operations.[1] Moreover, by Theorem 3.1.3 the diagonal entries d_i in the factorization of S can be calculated as follows: $d_i = \delta_i\lambda_i$.

Expanding the generator recursions (3.37) we can directly arrive at the resultant algorithm. Yet, recall that, the main purpose of this algorithm is to compute the resultants in $\mathbb{Z}_m[x]$ which constitutes the core of the modular approach. Unfortunately, the rotation formulas (3.24) (from Section 3.1.3) used to transform the generators to a proper form involve divisions which is highly undesirable in a finite field. To avoid this, we exploit an idea similar to that of division-free Givens rotations described in (FL94). Namely, we can use "external" denominators for each generator column to collect the division factors. In other words, if the generators are expressed in the following form:

$$G_i^T = \begin{bmatrix} 1/l_a & 0 \\ 0 & 1/l_b \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & \dots \\ b_0 & b_1 & b_2 & \dots \end{bmatrix}, \ B_i^T = \begin{bmatrix} 1/l_c & 0 \\ 0 & 1/l_d \end{bmatrix} \begin{bmatrix} c_0 & c_1 & c_2 & \dots \\ d_0 & d_1 & d_2 & \dots \end{bmatrix},$$

[1] The multiplication by $A = Z_q \oplus Z_p$ is equivalent to the multiplication by two lower-shift matrices separately.

then we can rewrite the rotation formulas (3.24) to compute $(\tilde{G}_i, \tilde{B}_i) = (G_i\Theta_i, B_i\Gamma_i)$ without divisions:

$$(\tilde{G}_i)_j = \begin{bmatrix} l_a(a_jc_0 + b_jd_0) \\ l_b(a_jb_0 - b_ja_0) \end{bmatrix}^T, \ (\tilde{B}_i)_j = \begin{bmatrix} l_c(c_ja_0 + d_jb_0) \\ l_d(c_jd_0 - d_jc_0) \end{bmatrix}^T, \tag{3.38}$$

where $(\tilde{G}_i)_j$ and $(\tilde{B}_i)_j$ denote the j-th rows of respective matrices. Straightforward computations show that the column denominators l_a, l_b, l_c and l_d are *pairwise* equal, and thus we can only keep two of them. These denominators can be updated in the following way:

$$\tilde{l}_a = \tilde{l}_d = l_a(a_0c_0 + b_0d_0), \text{ and } \tilde{l}_c = \tilde{l}_b = l_al_c^2. \tag{3.39}$$

Now, the resultant algorithm follows by unwinding the recursions (3.37) and using the relations (3.38) and (3.39). The pseudocode is given by Algorithm 3.1. Some comments are due here to gain better understanding of the algorithm. For convenience, we write the generator matrices as a pair of column vectors: $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$. In each iteration we update the generators according to (3.38) and collect one factor of the resultant. After n iterations $(n = p + q)$ the generators vanish completely, and the product of collected factors yields the resultant.

The algorithm is split in two parts: lines 5–11, where only a single column $\mathbf{b}$ of G is updated; and lines 13–20 where all four columns participate in the update. In what follows, we will refer to these parts of the algorithm as "lite" and "full" iterations, respectively. This improvement is possible because, according to (3.36), the matrix B has only two non-zero entries: $c_0 = d_q = 1$. Next, if we ensure that f is monic (that is, $a_0 = f_p \equiv 1$), we can see from (3.24) that: $D = a_0c_0 + b_0d_0 = a_0 \equiv 1$. Hence, it follows that the denominators equal *identically* to 1 throughout the first q steps of the Schur algorithm (or "lite" iterations). Substituting $a_0 = c_0 = 1$ and $d_0 = 0$ into (3.38) leads to largely simplified generator recursions. By the same token, the resultant factors a_0c_0 are *unit* during "lite" iterations, and hence not need to be collected. At the end, in line 21 we multiply the resultant by $(f_p)^q$ to compensate for monic f. Lastly, remark that, strong-regularity assumption introduced at the beginning of this section guarantees that denominators l_a and l_c do not vanish in throughout the algorithm.

3.2.2 Vandermonde system and polynomial interpolation

For the task of interpolation, suppose we would like to find a polynomial $f \in \mathbb{Z}[x]$ of degree less then n satisfying the set of equations: $f(x_i) = y_i$, for $0 \le i < n$. As noted in Section 2.2.3, the coefficients $\{a_i\}$ of f are the solutions of the following Vandermonde system:

$$V\mathbf{a} = \mathbf{y}, \text{ with } V_{ij} = x_i^j, \ (0 \le i, j < n - 1), \tag{3.40}$$

where $V \in \mathbb{Z}^{n \times n}$ is a Vandermonde matrix, and $\mathbf{y}$ is a column vector of the values $\{y_i\}$. To solve this linear system, we apply the techniques from Section 3.1.4 to form a matrix $W \in \mathbb{Z}^{2n \times (n+1)}$ which contains V and the vector $\mathbf{y}$ as submatrices:

$$W = \begin{bmatrix} V & -\mathbf{y} \\ I_n & \mathbf{0} \end{bmatrix}.$$

After n steps, we obtain a Schur complement R of V which is: $R = \mathbf{0} - I_nV^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$, i.e., equals precisely the solution of system (3.40). The matrix W has a displacement rank

2 and satisfies the equation:

$$W - FWA^T = GB^T, \tag{3.41}$$

where $A = Z_n \oplus 0 \in \mathbb{Z}^{(n+1)\times(n+1)}$ and $F = \mathrm{diag}(x_0 \dots x_{n-1}) \oplus Z_n \in \mathbb{Z}^{2n\times 2n}$. Remark that the matrix F is a composition of a diagonal and lower-shift matrix. This is due to the special structure of a Vandermonde matrix, see (KS95, § 7.2.1) for details. The generator matrices $G \in \mathbb{Z}^{2n\times 2}$ and $B \in \mathbb{Z}^{(n+1)\times 2}$ have the following form:

$$G^T = \begin{bmatrix} 1 & \dots & 1 & 1 & 0 \dots 0 \\ y_0 & \dots & y_{n-1} & 0 & 0 \dots 0 \end{bmatrix}, \quad \begin{array}{l} B \equiv 0, \ \text{ except for} \\ B_{0,0} = 1, \ B_{n,1} = -1. \end{array} \tag{3.42}$$

Again, we apply Theorem 3.1.3 to derive generator recursions. The condition $1 - f_i a_j \neq 0$ is satisfied because A has zero diagonal entries. Thus, the next generator pair (G_{i+1}, B_{i+1}) can be found from the proper form generators $(\tilde{G}_i, \tilde{B}_i) = (G_i\Theta_i, B_i\Gamma_i)$ as follows $(0 \le i < n)$:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i \tilde{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i \tilde{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \tag{3.43}$$

which should be read as: "take the first column of $\tilde{G}_i$ (or $\tilde{B}_i$) and multiply it with Φ_i (or Ψ_i), keeping the second column unchanged." Here, $\Phi_i = F_i - f_i I_i$, $\Psi_i = A_i(I_i - f_i A_i)^{-1}$, where F_i and A_i are obtained by deleting the first i rows and columns from F and A, respectively. After n steps of the algorithm, the product GB^T yields the solution of system (3.40). Although, the equations look complicated at first glance, shortly we will see that, in essence, it suffices to work with a single generator G because, informally speaking, B does not carry any useful information.

Let $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ be the generators as defined in (3.42), where we again associate the matrices with column vectors. We will show that it suffices to work with a *single* generator G which leads to an efficient interpolation algorithm. The key to understanding is that, in contrast to the resultant algorithm, we are not interested in *intermediate* results of the factorization, instead our goal is to obtain the final Schur complement R which is given by the product GB^T after n steps of the Schur algorithm.[1] Observe that, at the beginning $B := B_0$ has only two non-zero entries ($c_0 = 1$ and $d_n = -1$) and is already in a proper form (since $d_0 = 0$). Using (3.43), straightforward manipulations show that:

$$B_1^T = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} & 0 \\ 0 & 0 & & \dots & 0 & 1 \end{bmatrix}, \quad B_2^T = \begin{bmatrix} 1 & x_1 + x_0 & x_1^2 + x_0x_1 + x_0^2 & * & 0 \\ 0 & 0 & \dots & 0 & * \end{bmatrix},$$

where $B_1 \in \mathbb{Z}^{n\times 2}$, $B_2 \in \mathbb{Z}^{(n-1)\times 2}$, and asterisk (*) denotes unimportant entries. A crucial observation is that the leading rows of matrices B_i $(0 \le i < n)$ equal identically to $[\,1 \ \ 0\,]$ which can be verified by expanding the respective formulas. Next, provided that only the leading rows of G and B are needed to setup the rotation matrices in (3.24), we conclude that B does *not* affect the update of the matrix G. After n steps of the Schur algorithm, it follows that the last generator B_n has the form: $B_n = [\,0 \ \ 1\,]$,[2] which implies that the desired product $G_n B_n^T$ is simply given by the second column of G_n.

[1] Here n denotes the number of interpolation points.

[2] To be precise, $B_n = [\,0 \ \ 1/K\,]$, for some $K \in \mathbb{Z}$, but in a division-free algorithm K is part of a common denominator.

Algorithm 3.2 Polynomial interpolation

```
 1: procedure VANDERMONDE_INTERP($\mathbf{x}$ : Vector, $\mathbf{y}$ : Vector, $n$ : Integer)
 2:                                  ▷ returns the coefficients of $f(x)$, s.t., $f(x_i) = y_i$, $0 \le i < n$
 3:     let $G = (\mathbf{a}, \mathbf{b})$, $l_{int} = 1$          ▷ set up the generator matrix as in (3.42)
 4:     for $j = 0$ to $n - 1$ do
 5:         for $i = j + 1$ to $j + n - 1$ do                 ▷ multiply b by rotation matrix
 6:             $\mathbf{b}_i = \mathbf{b}_i\mathbf{a}_j - \mathbf{a}_i\mathbf{b}_j$
 7:         od
 8:         $l_{int} = l_{int} \cdot \mathbf{a}_j$, $s = 0$, $t = 0$                    ▷ update the denominator
 9:         for $i = j + 1$ to $j + n$ do      ▷ multiply the column a with $\Phi_j$ as in (3.43) from the left
10:             if $(i < n)$ then $s = \mathbf{a}_i$, $t = \mathbf{x}_i$                ▷ consider different cases
11:             elif $(i > n$ and $i \le j + n)$ then $s = \mathbf{a}_{i-1}$, $t = 1$ fi
12:             $\mathbf{a}_i = s \cdot t - \mathbf{a}_i \cdot \mathbf{x}_j$
13:         od
14:         $\mathbf{b}_{j+n} = -\mathbf{b}_j$, $\mathbf{a}_{n+j+1} = 1$       ▷ update the last non-zero entries of a and b
15:     od
16:     $\mathbf{b}_i \leftarrow -\mathbf{b}_i/l_{int}$ for $\forall i = n \ldots 2n - 1$   ▷ divide the coefficients by the denominator
17:     return $(b_n \ldots b_{2n-1})$                          ▷ return the coefficients of $f$
18: end procedure
```

Now, we try to simplify the recursion for the generator G. By the above observations, we have $(c_0, d_0) = (1, 0)$ throughout the whole algorithm. Then, (3.38) implies that: $\tilde{a}_j = l_a(a_jc_0 + b_jd_0) \equiv l_aa_j$. Hence, only the column vector $\mathbf{b}$ needs to be multiplied by the rotation matrix, and we can omit the denominator l_a for the column $\mathbf{a}$. Besides, observe that only n entries of the generator G are *non-zero* at a time. Thus, we can use a sort of "sliding window" approach where only n relevant entries of G get updated in each iteration of the Schur algorithm.

The pseudocode is given in Algorithm 3.2. Here, lines 10–12 are the effect of multiplying Φ_j from (3.43) by the column vector $\mathbf{a}$. Note that, we use auxiliary variables s and t to write the update of $\mathbf{a}$ in a "uniform" way. This is done with the intension to later avoid excessive branching in the GPU code.[1] Written in a usual form, the elements of $\mathbf{a}$ are updated in iteration j according to the following rules:

$$\mathbf{a}_i = \begin{cases} \mathbf{a}_i(\mathbf{x}_i - \mathbf{x}_j), & i = j + 1, \ldots, n - 1 \\ \mathbf{a}_i\mathbf{x}_j, & i = n \\ \mathbf{a}_{i-1} - \mathbf{a}_i\mathbf{x}_j. & i = n + 1, \ldots, j + n \end{cases}$$

As a last remark, observe that, the Vandermonde matrix is strongly regular as long as interpolation points x_i are pairwise distinct, hence the algorithm is guaranteed to succeed without any additional assumptions.

3.2.3 GCD computation

To compute a GCD of univariate polynomials, we first recall a well-known result relating a GCD with the triangularization of Sylvester's matrix. In addition, we refer to (Eme11) where the original algorithm was derived.

[1] Short conditional statements are likely to be replaced by predicated instructions to avoid branching in the GPU code, see Section 4.1.

Theorem 3.2.1: (Lai69) Let S be Sylvester's matrix for polynomials $f, g \in \mathbb{F}[x]$ with coefficients over some field $\mathbb{F}$. If S is put in echelon form, using row transformations only, then the last non-zero row gives the coefficients of $\gcd(f, g) \in \mathbb{F}[x]$.

Proof Note that, a matrix is said to be in *row echelon form* if all non-zero rows are situated above any rows having all zeros, and the first non-zero coefficient of a non-zero row is always strictly to the right of the one from the row above. We reproduce the proof here as this result plays the central role in the correctness of our GCD approach. Suppose, f and g are polynomials of degrees p and q, respectively. From the application of Extended Euclidean Algorithm over $\mathbb{F}$ (see Section 2.1.3), it follows that:

$$\gcd(f, g) = f(x)s(x) + g(x)t(x), \quad \text{with} \quad \deg(s) < q \text{ and } \deg(t) < p. \tag{3.44}$$

Let $\tilde{S}$ denote the matrix S in a row echelon form. By Theorem 2.4.2 from Section 2.4, if $\deg(\gcd(f, g)) > 0$, then Sylvester's matrix S is singular and hence $\tilde{S}$ must necessarily contain zero rows. We associate with each row i of S a polynomial $u_i(x)$, and with a corresponding row i of $\tilde{S}$ a polynomial $e_i(x)$:

$$S \cdot \begin{bmatrix} x^{p+q-1} \\ \vdots \\ x \\ 1 \end{bmatrix} = \begin{bmatrix} u_{p+q-1}(x) \\ \vdots \\ u_1(x) \\ u_0(x) \end{bmatrix}, \quad \tilde{S} \cdot \begin{bmatrix} x^{p+q-1} \\ \vdots \\ x \\ 1 \end{bmatrix} = \begin{bmatrix} e_{p+q-1}(x) \\ \vdots \\ e_d(x) \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Because $\tilde{S}$ is in row echelon form, $\deg(e_i) > \deg(e_{i-1})$ for $i = d+1, \ldots, p+q-1$. Let $k = \deg(e_d)$ be the degree of the last non-zero polynomial. We first show that any non-zero $u(x) \in \mathbb{F}[x]$ which is a linear combination of u_i's, that is, $u(x) = \sum_i^{p+q-1} \alpha_i u_i(x)$ for some $\alpha_i \in \mathbb{F}$, has degree not less that k. This is easy to see, if we observe that $u(x)$ also admits the following representation: $u(x) = \sum_i^d \beta_i e_i(x), \beta_i \in \mathbb{F}$. It holds because each row of $\tilde{S}$ is a linear combination of the rows of S. By construction, k is minimal among the degrees of $\{e_i(x)\}$ and $\deg(e_i) > \deg(e_{i-1})$, thus all leading terms in the sum $\sum_i^d \beta_i e_i(x)$ cannot be cancelled out simultaneously, and we conclude that $\deg(u) \geq k$.

Now, by construction of Sylvester's matrix, we have: $u(x) = f(x)s(x)+g(x)t(x)$, where $\deg(s) < q$ and $\deg(t) < p$. This can be easily verified if we expand the polynomials $u_i(x)$ according to definition. Apparently, $e_d(x)$ can also be written in that form which by (3.44) implies that $e_d(x)$ is divisible by a GCD. However, as we have shown above, there are no non-zero polynomials $u(x)$ of degree less than k expressible in the form $f(x)s(x)+g(x)t(x)$, and thus $e_d(x)$ is a GCD. ■

The above theorem asserts that, if we triangularize Sylvester's matrix, for instance, using Gaussian elimination, we obtain a GCD in the last nonzero row of the triangular factor. Assume, $f, g \in \mathbb{Z}[x]$ are polynomials of degrees p and q ($p \geq q$), respectively, and $S \in \mathbb{Z}^{n \times n}$ ($n = p+q$) is the associated Sylvester's matrix. To compute the factorization of S, we could apply the generalized Schur algorithm directly. Unfortunately, it is not possible to handle non-strong regularity as in the case of resultants (which is discussed Section 4.4.2). The main reason for this is because, if polynomials are not relatively prime, Sylvester's matrix S is singular by definition, and hence it is not possible to tell in which step of the factorization algorithm a GCD is actually computed.

To get around this difficulty, we can triangularize the *symmetric* matrix $W = S^T S$ instead and arrive at orthogonal factorization of S as we have studied in Section 3.1.4. Indeed, if W is factored in the form: $W = R^T R$, then R is an upper-triangular factor in the QR-factorization of S:

$$W = S^T S = (QR)^T QR = R^T Q^T QR = R^T R, \tag{3.45}$$

where: $Q^T Q = I$ since Q is orthogonal. We should remark that W is *not* necessarily positive-definite because S may be singular, as noted above. Yet, we can guarantee that the Schur algorithm applied to W does not break down during the first $n-k$ steps, where $k = \deg(\gcd(f, g))$, since $n-k$ leading submatrices of W are *positive-definite*. The matrix W is structured satisfying the following equation:

$$W - Z_n W Z_n^T = GJG^T \text{ with } G \in \mathbb{Z}^{n\times 4},\ J = I_2 \oplus -I_2, \tag{3.46}$$

As mentioned in Section 3.1.4, it is often not necessary to compute the entries of W explicitly since the generator G can be expressed in terms of the elements of the original matrix which is true in our case:

$$G^T = \underbrace{\begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0 & \dots & 0 \\ g_q & q_{q-1} & \dots & g_0 & 0 & \dots & 0 \\ 0 & \dots & 0 & f_p & f_{p-1} & \dots & f_1 \\ 0 & \dots & 0 & g_q & g_{q-1} & \dots & g_1 \end{bmatrix}}_{n=p+q}. \tag{3.47}$$

To carry out the generator recursion for G, we could apply the generalized Schur algorithm in array form as given by Theorem 3.1.2 in Section 3.1.3. However, keeping in mind that our final goal is to compute a GCD in $\mathbb{Z}_m[x]$, some complications may arise: namely, computing proper form generators requires the construction of rotation matrices which is not easy in a finite field. To deal with this problem, in Section 3.2.1 we have developed division-free transformations for an asymmetric generator pair. Yet, in the current "symmetric" case, such transformations become increasingly more expensive since not only divisions but also square-roots need to be eliminated. We could apply square-root and division-free Givens rotations as exemplified in Section 3.1.3 but, due to the large displacement rank of G in (3.46), it appears that using the Schur recursion in its original form is computationally more attractive in our case. Indeed, directly using Theorem 3.1.1 from Section 3.1.2, we obtain the following generator recursion:

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \left\{ G_i - (I_{n-i} - Z_i) G_i \frac{J g_i^T g_i}{g_i J g_i^T} \right\} \Theta_i, \tag{3.48}$$

where we trivially set $\Theta_i = I$. To derive the actual GCD algorithm based on this recursion, we proceed by defining $L^i = G_i J g_i^T$ in step i which, according to (3.13), is precisely the $(i+1)$-th row of the triangular factor of W. Hence, the goal of our algorithm is to compute

Algorithm 3.3 Matrix-based univariate GCD algorithm

```
 1: procedure GCD_SYLVESTER(f : Polynomial, g : Polynomial)
 2:    $p = degree(f),\ q = degree(g),\ n = p + q,\ det = 1$
 3:    let $G = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$          ▷ setup the matrix generator as in (3.47)
 4:    for $i = 0$ to $q - 1$ do                                 ▷ initial q iterations are “simplified”
 5:       for $j = i$ to $n - 1$ do                              ▷ the G’s top row: $g_i = \{\mathbf{a}_i, \mathbf{b}_i, 0, 0\}$
 6:          $L^i_j = \mathbf{a}_i\mathbf{a}_j + \mathbf{b}_i\mathbf{b}_j$          ▷ triangular factor: $L^i = G_iJg^T_i$
 7:       od
 8:       for $j = i + 1$ to $n - 1$ do                          ▷ compute the next gen.: $G_{i+1}$
 9:          $F^i_j = L^i_j - L^i_{j-1}$                          ▷ matrix prod.: $F^i = (I_{n-i} - Z_i) \cdot L^i$
10:          $\mathbf{a}_j = \mathbf{a}_j \cdot L^i_i - \mathbf{a}_i \cdot F^i_j,\ \mathbf{b}_j = \mathbf{b}_j \cdot L^i_i - \mathbf{b}_i \cdot F^i_j$
11:       od
12:       $det = det \cdot L^i_i$                                ▷ collect the denominators
13:    od
14:    for $j = q$ to $n - 1$ do                                 ▷ bring to the common denom.
15:       $\mathbf{c}_j = \mathbf{c}_j \cdot det, \mathbf{d}_j = \mathbf{d}_j \cdot det$
16:    od                                                        ▷ the remaining p “full” iterations:
17:    for $i = q$ to $n - 1$ do                                 ▷ the G’s top row: $g_i = \{\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \mathbf{d}_i\}$
18:       for $j = i$ to $n - 1$ do                              ▷ triangular factor: $L^i = G_iJg^T_i$
19:          $L^i_j = \mathbf{a}_i\mathbf{a}_j + \mathbf{b}_i\mathbf{b}_j - \mathbf{c}_i\mathbf{c}_j - \mathbf{d}_i\mathbf{d}_j$
20:       od
21:       for $j = i + 1$ to $n - 1$ do      ▷ generator recursion: $G_{i+1} = G_i \cdot L^i_i - (I_{n-i} - Z_i) \cdot L^i \cdot g_i$
22:          $F^i_j = L^i_j - L^i_{j-1}$                          ▷ matrix prod.: $F^i = (I_{n-i} - Z_i) \cdot L^i$
23:          $\mathbf{a}_j = \mathbf{a}_j \cdot L^i_i - \mathbf{a}_i \cdot F^i_j,\ \mathbf{b}_j = \mathbf{b}_j \cdot L^i_i - \mathbf{b}_i \cdot F^i_j$
24:          $\mathbf{c}_j = \mathbf{c}_j \cdot L^i_i - \mathbf{c}_i \cdot F^i_j,\ \mathbf{d}_j = \mathbf{d}_j \cdot L^i_i - \mathbf{d}_i \cdot F^i_j$
25:       od
26:       if $(\mathbf{a}_{i+1} = \mathbf{c}_{i+1}$ and $\mathbf{b}_{i+1} = \mathbf{d}_{i+1})$ then   ▷ check if the columns are not linearly independent
27:          return $(L^i_i, L^i_{i+1}, \ldots, L^i_{n-1})/L^i_i$              ▷ return monic gcd
28:       fi
29:    od
30:    return 1                                                  ▷ polynomials are coprime
31: end procedure
```

the last nonzero L^i. In what follows, we will adopt the notation writing L^i_j to denote the j-th element of the i-th column of the triangular factor. To minimize the number of finite field divisions, we can collect all divisors $g_iJg^T_i$ in (3.48) in a common denominator. Moreover, it follows that: $L^i_i = g_iJg^T_i$, hence we do not need to compute the divisors separately.

Unwinding the recursion (3.48), we obtain a GCD algorithm with pseudocode given by Algorithm 3.3. Here, G is represented implicitly by four columns: $G = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) \in \mathbb{Z}^{n\times 4}$. We split the algorithm in two parts: lines 4–13 where only two generator columns ($\mathbf{a}$ and $\mathbf{b}$) are updated; and lines 17–29 where all four columns participate. Henceforth, we will refer to these parts of the algorithm as “lite” and “full” iterations, respectively. The reason for this partitioning lies in the particular structure of G in (3.47) having a block of zeros in the first q rows ($p \geq q$). As a result, we can skip updating the columns $\mathbf{c}$ and $\mathbf{d}$ throughout the first q steps of the algorithm. The purpose of the loop in lines 14–16 is to bring the columns $\mathbf{c}$ and $\mathbf{d}$ to the common denominator with $\mathbf{a}$ and $\mathbf{b}$ before running the “full” iterations. Finally, to detect a GCD in lines 26–31, we check if the denominator L^{i+1}_{i+1} for the next step of the algorithm vanishes indicating that L^i is the last non-zero row of the triangular factor. Note that, there is no need for this test in the first q iterations by obvious reason because the degree of $\gcd(f, g)$ is not larger than the degree of either polynomial.

3.2.4 Computing resultant cofactors

Finally, we consider the problem of computing resultant cofactors (see Section 2.4.1). Although, strictly speaking, this seems to be unrelated to the main contribution of this thesis, we believe that the algorithm described below may be of independent interest and serves as a good exposition of the power and elegance of the structured matrix theory. As a starting motivation, observe that, there is no direct approach (for instance, such as the PRS algorithm) to computing the resultant cofactors except for the evaluation of the whole subresultant sequence, see (BPR06, § 8.3.6). The latter operation is an order of magnitude more difficult then the initial problem appears to be. We begin with describing the problem in matrix terms (see also Theorem 2.4.1).

Let $f, g \in \mathbb{Z}[x]$ be polynomials of degrees p and q, respectively; and $S \in \mathbb{Z}^{n\times n}$ ($n = p + q$) be the corresponding Sylvester's matrix. By definition, the cofactors are polynomials $s(x)$ and $t(x)$ of degree at most $q-1$ and $p-1$, respectively, which satisfy the following equation:

$$s \cdot f + t \cdot g = \operatorname{res}(f, g). \tag{3.49}$$

By equating the coefficients of the same power of x on both sides, we can express (3.49) in matrix form:

$$\begin{bmatrix} f_p & \dots & 0 & g_q & \dots & 0 \\ f_{p-1} & \ddots & \vdots & g_{q-1} & \ddots & \vdots \\ \vdots & \ddots & 0 & \vdots & \ddots & 0 \\ f_0 & & f_p & g_0 & & g_q \\ 0 & \ddots & f_{p-1} & 0 & \ddots & g_{q-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & f_0 & 0 & \dots & g_0 \end{bmatrix} \begin{bmatrix} s_{q-1} \\ \vdots \\ s_1 \\ s_0 \\ t_{p-1} \\ \vdots \\ t_1 \\ t_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \operatorname{res}(f, g) \end{bmatrix},$$

which is equivalent to writing

$$S^T \cdot \mathbf{v}^T = [\, 0 \ \dots \ 0 \ \ \operatorname{res}(f, g) \,]^T,$$

where $\mathbf{v}$ is the column vector of coefficients of $s(x)$ and $t(x)$. From the last equation, it is easy to see that the cofactors can be obtained from the last column of S^{-T} multiplied by the resultant $\operatorname{res}(f, g)$. To compute S^{-T}, we consult Section 3.1.4 for the techniques on computing a matrix inverse. We proceed by constructing a matrix $W \in \mathbb{Z}^{2n\times(n+1)}$ as follows:

$$W = \begin{bmatrix} S^T & -\mathbf{b} \\ I_n & \mathbf{0} \end{bmatrix}, \ \text{ such that } \ W - FWA^T = GB^T,$$

with $F = Z_n \oplus Z_n$, $A = Z_q \oplus Z_p \oplus 0$, and $\mathbf{b} = [\, 0 \ \dots \ 0 \ \ 1 \,]$. Using W, the last column of S^{-T} can be computed in n steps of the generalized Schur algorithm. Indeed, the nth Schur complement of W equals: $\mathbf{0} - I_n S^{-T}(-\mathbf{b})$.

Next, remark that we do *not* know $\operatorname{res}(f, g)$ in advance (needed to scale the last column of S^{-T}) but it can be computed *simultaneously* during the triangularization of W. The main disadvantage of this approach, however, is that W has displacement rank 3 which is by 1 higher than that of Sylvester's matrix. Carrying out the Schur recursion for asymmetric rank-3 generators is significantly more difficult since, in each step, we need to operate on 6 (!) matrix columns instead of 4. The intuition suggests that there should be an easier solution: after all, we only need to compute the *single* column of S^{-T}. Remarkably, we can slightly modify the displacement equation to lower the displacement rank. For that, we introduce a matrix $\Sigma = I_n \oplus 0$ to zero out a single position in W, so that W has now displacement rank 2 with respect to the following equation:

$$W\Sigma - FWA^T = GB^T, \tag{3.50}$$

while the generators $G \in \mathbb{Z}^{2n\times 2}$ and $B \in \mathbb{Z}^{(n+1)\times 2}$ are easily expressible in terms of the coefficients of f and g:

$$G^T = \left[\begin{array}{cccccc} f_p & f_{p-1} & \dots & f_0 \overbrace{0 \dots 0}^{q-1} & 1 & \overbrace{0 \dots 0}^{n-1} \\ g_q & q_{q-1} & \dots & g_0 \underbrace{0 \;\dots\; 0}_{n-1} & 1 & \underbrace{0 \dots 0}_{p-1} \end{array}\right] \quad \begin{array}{c} B \equiv 0 \text{ except} \\ B_{0,0} = B_{q,1} = 1. \end{array} \tag{3.51}$$

Observe that, in (3.50) we use a displacement operator of a more general form but, in fact, this makes the generator recursions only slightly more complicated, see (KS95, § 7.4.2). Denoting by $(\tilde{G}_i, \tilde{B}_i) = (G_i\Theta_i, B_i\Gamma_i)$ the proper form generators in step i of the algorithm $(0 \le i < n)$, the next generator pair (G_{i+1}, B_{i+1}) fulfills the following recurrence (cf. Theorem 3.1.3):

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i\tilde{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i\tilde{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \tilde{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \tag{3.52}$$

where $\Phi_i = F_i$, $\Psi_i = A_i\Sigma_i^{-1}$ and F_i, A_i, Σ_i are obtained from the corresponding matrices by deleting the first i rows and columns. However, one can immediately see the flaw in the above relations: namely, the matrix Σ_i is *singular*, and thus Ψ_i cannot be computed. In terms of displacements, this means that it is not possible to recover the matrix W from its generators completely. To be precise, we cannot solve uniquely for the upper-triangular factors u_i which, by (3.50), must satisfy:

$$u_i\Sigma_i - f_iu_iA_i^T = g_iB_i^T, \tag{3.53}$$

where g_i is the leading row of G_i in step i of the algorithm. Since $f_i \equiv 0$ and $\Sigma_i = I_{n-i} \oplus 0$, it appears that the last entry of each u_i cannot be determined from this relation. However, one of remarkable features of the generalized Schur algorithm is that it enables us to carry out a generator recursion even in such "deficient" cases provided that we can somehow "guess" the missing information. We next show how to do this. Running n steps of the factorization algorithm yields the following matrix identity:

$$\begin{bmatrix} S^T & -\mathbf{b} \\ I & \mathbf{0} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} D^{-1} \begin{bmatrix} U_1 & U_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & S^{-T}\mathbf{b} \end{bmatrix},$$

Algorithm 3.4 Computing resultant cofactors from Sylvester's matrix

```
procedure RESULTANT_COFACTORS(f : Polynomial, g : Polynomial)
    $p = \deg(f),\ q = \deg(g),\ n = p + q$
    $f \leftarrow f/f_p$                                   ▷ convert f to monic polynomial
    let $G = (\mathbf{a}, \mathbf{b}),\ B = (\mathbf{c}, \mathbf{d})$          ▷ set up the generators as in (3.51)
    for $j = 0$ to $q - 1$ do                     ▷ “lite” iterations: only the column b is updated
        for $i = j + 1$ to $p + j$ do                      ▷ multiply by rotation matrix
            $\mathbf{b}_i = \mathbf{b}_i - \mathbf{a}_i\mathbf{b}_j$
        od
        $\mathbf{b}_{n+j} = -\mathbf{b}_j,\ \mathbf{c}_q = \mathbf{b}_j$           ▷ update the single elements of b and c
        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i$ for $\forall i = j \ldots p + j$   ▷ multiply a by $F = Z_n \oplus Z_n$ from the left
        $\mathbf{a}_n = 0$
        $\mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$ for $\forall i = j \ldots q + j$   ▷ multiply c by $A = Z_q \oplus Z_p \oplus 0$ from the left
        $\mathbf{c}_q = 0, \mathbf{c}_n = 0$
    od
    $\mathbf{a}_{q+n} = 1,\ res = 1$              ▷ initialize the remaining entry of a and the resultant
    for $j = q$ to $n - 1$ do          ▷ “full” iterations: both generator matrices transformed
        $det = \mathbf{a}_j\mathbf{c}_j + \mathbf{b}_j\mathbf{d}_j$               ▷ calculate the denominator from (3.24)
        for $i = j$ to $2n - 1$ do                         ▷ transform G to a proper form:
            $\mathbf{a}_i = \mathbf{a}_i\mathbf{c}_j + \mathbf{b}_i\mathbf{d}_j,\ \mathbf{b}_i = \mathbf{a}_i\mathbf{b}_j - \mathbf{b}_i\mathbf{a}_j$
            if ($j \bmod 2 = 0$) then $\mathbf{a}_i = \mathbf{a}_i/det,\ \mathbf{b}_i = \mathbf{b}_i/det$ fi   ▷ normalize the coefficients of G
        od
        for $i = j$ to $n - 1$ do                          ▷ transform B to a proper form:
            $\mathbf{c}_i = \mathbf{c}_i\mathbf{a}_j + \mathbf{d}_i\mathbf{b}_j,\ \mathbf{d}_i = \mathbf{c}_i\mathbf{d}_j - \mathbf{d}_i\mathbf{c}_j$
            if ($j \bmod 2 = 1$) then $\mathbf{c}_i = \mathbf{c}_i/det,\ \mathbf{d}_i = \mathbf{d}_i/det$ fi   ▷ normalize the coefficients of B
        od
        $res = res \cdot \mathbf{a}_j \cdot \mathbf{c}_j$                                ▷ update the resultant
        if ($j = n - 1$) then
            $res = res \cdot (f_p)^q$                          ▷ compute the actual resultant
            $s \leftarrow \{\mathbf{a}_n, \ldots, \mathbf{a}_{n+q-1}\} \cdot res/(\mathbf{a}_{n-1} \cdot f_p)$   ▷ scale coefficients of the cofactor s
            $t \leftarrow \{\mathbf{a}_{n+q}, \ldots, \mathbf{a}_{2n-1}\} \cdot res/\mathbf{a}_{n-1}$        ▷ scale coefficients of the cofactor t
            return $(s, t)$                                        ▷ return the cofactors
        fi
        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i$ for $\forall i = j \ldots n + j$   ▷ multiply a by $F = Z_n \oplus Z_n$ from the left
        $\mathbf{a}_n = 0$
        $\mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$ for $\forall i = j \ldots n - 2$   ▷ multiply c by $A = Z_q \oplus Z_p \oplus 0$ from the left
        $\mathbf{c}_q = 0, \mathbf{c}_n = 0$
    od
end procedure
```

where the matrices L_1, L_2 and U_1 are of size $n \times n$ while U_2 is $n \times 1$. Observe that, U_2 contains precisely those searched-for last elements of each u_i. By equating the matrices on both sides, we conclude that:

$$-\mathbf{b} = L_1 D^{-1} U_2.$$

Here the matrix $L_1 D^{-1}$ has unit diagonal elements which follows directly from the factorization formula (3.11) in Section 3.1.2. If we expand the above equation, it can be shown by induction that $U_2 = -\mathbf{b}$, and hence we can recover u_i in (3.53) completely without the need for inverting Σ_i. As a result, we can simply take $\Psi_i = A_i$ in (3.52) to compute the next generator B_{i+1}. The reason for this is because the matrix Ψ_i essentially comes from the upper-triangular factor u_i, if we recall the proof of Theorem 3.1.3; see also (KS95, Cor. 7.16).

For expository purposes, in the pseudocode we shall use the classical rotation formulas (3.24) from Section 3.1.3 to update the matrix generators leaving the reader an opportunity to work out the details of the division-free version (this can be done in exactly the same manner as for resultants). Another motivation for choosing the classical formulas stems from the fact that the cofactors are usually *very large* expressions and it is often not needed to compute them exactly. In this sense, the rotation formulas (3.24) are more preferable as they can be used with multi-precision arithmetic (such as BigFloat) without overflow concerns while division-free rotations are only applicable in a finite field. The algorithm's pseudocode is given by Algorithm 3.4. It has a lot in common with the resultant algorithm previously discussed: the main difference is that the generator matrices $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$ are now of size $2n \times 2$ and $(n + 1) \times 2$, respectively. That is why, we skip the detailed description of the algorithm referring to the comments in Section 3.2.1, and concentrate only on the important parts.

The algorithm is again split in two parts: lines 5–14, where only a single column of G is updated, and lines 16–37, where both generator matrices are modified in each step. Remark that, in lines 18–25 we use the rotation formulas involving divisions. In effect, there are many ways to update the generator matrices: here we divide the columns of G or B *in turns* (line 20 for even j's and line 24 for odd j's) by the denominator to keep the magnitudes of the generator columns comparable. Notice also that, multiplication of the columns $\mathbf{a}$ and $\mathbf{c}$ by the matrices A and F in lines 10–13 and 33–34 takes a slightly different form since each of them is now composed of a pair of lower-shift matrices. The cofactors are extracted during the last iteration in lines 27–32. By the displacement equation, the last column of S^{-T} is expressed as $G_n B_n^T$ – the product of the generators in step n of the algorithm. However, since the matrix B_n is of the form $[\ 1\ \ 0\ ]$, we simply take the first column $\mathbf{a}$ of G and scale it by the resultant to obtain the cofactors. The cofactor s is also divided by the leading coefficient f_p in line 29 to compensate for the monic form of f.

At this point, we conclude our discussion of matrix algorithms. To sum up, we have seen that many computer algebra problems can be solved in a very elegant way using matrix algebra methods. Besides, at the price of a small computational overhead (which does not change the overall complexity) we obtain neatly structured algorithms readily available for parallelization. In the next section, we briefly review a parallel complexity of the modular GCD and resultant algorithms when their relevant parts are replaced with the matrix-based analogues.

3.3 Complexity of modular algorithms revisited

Complexity bounds derived in this section shall provide us with a good measure for parallel performance that can potentially be achieved on the graphics card. Eventually, in Sections 4.4.4 and 4.5.5 we will see that the performance measured through benchmarking agrees with asymptotic behaviour of the algorithms. This should also justify the correctness our implementation.

Before going to the main topic of discussion, we should remark that the complexity of matrix subalgorithms presented in Section 3.2 is bounded by $O(n^2)$ arithmetic operations. It is trivial to see since all four algorithms considered above (Algorithms 3.1, 3.2, 3.3 and 3.4) contain two nested loops where the number of iterations of each loop is at most n.

Moreover, because of a highly structured way of computations, we can easily "vectorize" the inner loops of these algorithms. Indeed, each row of a generator matrix (G or B) can be updated independently in each iteration leading to $O_P(n, n)$ parallel complexity. What concerns Chinese remaindering, observe that, we can adapt the Mixed-radix (MR) conversion algorithm (Algorithm 2.4 from Section 2.2.2). From its pseudocode, it is clear that $O(n)$ MR digits can be computed in a linear parallel time using n processors as well.[1] It is then remains to recover the actual integer value from MR digits by evaluating the Horner's scheme. We discuss the complexity of this step in Section 3.3.1. Certainly, to implement such parallel algorithms, a target platform should support a very fine-grained parallelism because threads need to work in a close cooperation to compute the result. Traditional parallel platforms are usually not considered here because inter-thread communications would have a significant impact on the performance. Yet, assuming the PRAM model, which provides a good approximation of the GPU (see Section 2.5.1), we can neglect the communication overhead, and, keeping that in mind, improve the parallel complexity of the modular algorithms.

3.3.1 GCD algorithm

First, we revisit the complexity analysis of the GCD algorithm from Section 2.3.2. We shall try to keep the discussion concise because the main details have been already worked out in Section 2.5.3. Again, we concentrate on a bivariate case only. Suppose $f, g \in \mathbb{Z}[x, y]$ are polynomials of degree at most n in each variable with scalar coefficients bounded by 2^τ, $\tau \in \mathbb{N}$; and let $\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y]$ be their homomorphic images modulo m, respectively.

For Algorithm 2.7 computing a GCD of $\hat{f}$ and $\hat{g}$, we now use $O(n^2)$ processors because the univariate GCD algorithm (Algorithm 3.3) can benefit from parallel processing. Extracting the primitive parts of $\hat{f}$ and $\hat{g}$ as in (2.18) requires

$$T_{\mathcal{B}}(\hat{F} \leftarrow \hat{f}/\operatorname{cont}(\hat{f})) = T_{\mathcal{B}}(\hat{G} \leftarrow \hat{g}/\operatorname{cont}(\hat{g})) = O_P(n, n^2) \tag{3.54}$$

parallel times as it is equivalent to computing $O(n)$ univariate GCDs in $\mathbb{Z}_m$ each of which can be computed in parallel. Next, applying the evaluation homomorphism, cf. (2.20), costs

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_{y-\alpha}(\hat{F})) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_{y-\alpha}(\hat{G})) = O_P(n, n) \tag{3.55}$$

because we can evaluate each coefficient of $\tilde{f}$ and $\tilde{g}$ (considered as polynomials in $\mathbb{Z}_m[y]$) in linear time. The parallel complexity of the univariate GCD algorithm also becomes

$$T_{\mathcal{B}}(\tilde{H} \leftarrow \textsc{gcd\_sylvester}(\tilde{F}, \tilde{G}, m)) = O_P(n, n). \tag{3.56}$$

Thus, the inner loop of the algorithm $\textsc{gcd\_mod}$ takes $O_P(n, n^2)$ since $O(n)$ evaluation points can be processed in parallel. Finally, interpolation from $O(n)$ values using Algorithm 3.2 (for each coefficient in parallel) does not worsen the overall complexity:

$$T_{\mathcal{B}}(H \leftarrow \textsc{vandermonde\_interp}(\{\alpha\}, \{\tilde{H}\})) = O_P(n, n^2). \tag{3.57}$$

[1] We assume that the modular inverses c_i in Algorithm 2.4 can be precomputed in advance.

Certainly, the complexity of the remaining operations also falls within this bound, and hence

$$T_{\mathcal{B}}(\textsc{gcd\_mod}(\hat{f},\hat{g} \in \mathbb{Z}_m[x,y])) = O_P(n, n^2). \tag{3.58}$$

We next turn to the complexity of Algorithm 2.6. In this case, we can use $N \times n^2$ processors, where $N = O(n^2 + \tau)$, the number of primes used by the algorithm, cf. (2.26). So that, we are able to process each homomorphic image in parallel. Primitive parts of f and g, cf. (2.25), can be computed in

$$T_{\mathcal{B}}(F \leftarrow f/\operatorname{cont}(f)) = T_{\mathcal{B}}(G \leftarrow g/\operatorname{cont}(g)) = O_P(\tau^2, n^2) \tag{3.59}$$

parallel time because it amount to computing n^2 integer GCDs (at most) and the same number of divisions. Modular reduction for each prime (2.27) costs

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_m(F)) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_m(G)) = O_P(\tau, n^2), \tag{3.60}$$

arithmetic operations since we reduce each coefficient independently. Invocation of GCD_MOD requires $O_P(n, n^2)$ parallel time, and the cost of the remaining operations in the inner loop is negligible. In total, by running the algorithm for each modulus in parallel, one can achieve the complexity of $O_P(n + \tau, Nn^2)$ bit operations. What concerns Chinese remaindering, we can recover the MR digits $\{\gamma\}$ for each coefficient of a GCD $\tilde{H}$ using

$$T_{\mathcal{B}}(\{\gamma\} \leftarrow \textsc{mrc\_algorithm}(\{m\}, \{\tilde{H}\})) = O_P(N, Nn^2) \tag{3.61}$$

field operations. Next, it remains to evaluate a Horner's scheme of MR digits to obtain the actual large integers (see Section 2.2.2). This can be done in $\log N$ steps using N processors if we perform the computations in a "tree-like" fashion. To obtain the bit complexity, observe that, in step i ($i = 1, \ldots, \lceil \log N/2 \rceil$) each processor has to multiply two numbers of bit-size 2^i, while the number of working processors halves in each step. Therefore, we can use *spare* processors to speed-up the multiplication. Indeed, a pair τ-bit numbers can be multiplied in $O_P(\tau, \tau)$ time if we parallelize school-book multiplication method in a straightforward way. That is why, step i has the bit complexity of $O_P(2^i, N)$. Summing up the complexity of all steps yields

$$\sum_{i=1}^{\log N} 2^i = O(N)$$

bit operations per processor. Altogether, the cost of Chinese remaindering evaluates to $O_P(N, Nn^2)$ bit operations since we compute a Horner's scheme for each of $O(n^2)$ GCD coefficients in parallel. Computing a primitive part of a GCD as in (2.30) demands for

$$T_{\mathcal{B}}(Q \leftarrow \operatorname{pp}(H)) = O_P(\tau^2, n^2) \tag{3.62}$$

bit operations. The total bit complexity of the modular GCD then becomes

$$T_{\mathcal{B}}(\textsc{gcd\_int}(f, g \in \mathbb{Z}[x,y])) = O_P(N, Nn^2) + O_P(\tau^2, n^2) = O_P(n^2 + \tau^2, n^2(n^2 + \tau)). \tag{3.63}$$

Comparing this bound to (2.35) in Section 2.5.3, we see a significant improvement in terms of bit operations per processor. Important observation is that the resulting complexity is *linear* in the number of primes N which, certainly, also holds in the univariate case and agrees with the experimental results as we observe in Section 4.5.5.

3.3.2 Resultant algorithm

For the complexity of the serial algorithm, we refer to Section 2.5.4. Again, we shall emphasize only on the important points, skipping many secondary details. We begin with the subalgorithm RES_MOD (Algorithm 2.10) computing the resultant of polynomials $\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y]$ in a finite field. Evaluating the polynomials at $x = \alpha$ has a cost of

$$T_{\mathcal{B}}(\tilde{F} \leftarrow \phi_{x-\alpha}(\hat{f})) = T_{\mathcal{B}}(\tilde{G} \leftarrow \phi_{x-\alpha}(\hat{g})) = O_P(n, n), \tag{3.64}$$

operations in $\mathbb{Z}_m$ since we can evaluate each coefficient of $\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y]$ in parallel. Next, we apply Algorithm 3.1 to compute the resultant over $\mathbb{Z}_m$ having the same parallel complexity:

$$T_{\mathcal{B}}(\tilde{R} \leftarrow \text{RESULTANT\_SYLVESTER}(\tilde{f}, \tilde{g}, m)) = O_P(n, n). \tag{3.65}$$

Note that, the number of evaluation points is bounded by $O(n^2)$, cf. (2.36). Therefore, if we were to use n^3 processors, the main loop of RES_MOD would run in a linear parallel time (this is what is done in the actual realization). Unfortunately, the final complexity of the algorithm would still be determined by the cost of interpolation, and hence we suggest to use n^2 processors for the analysis. Indeed, to recover the resultant polynomial from $O(n^2)$ values using Algorithm 3.2, one needs

$$T_{\mathcal{B}}(H \leftarrow \text{VANDERMONDE\_INTERP}(\{\alpha\}, \{\tilde{H}\})) = O_P(n^2, n^2). \tag{3.66}$$

operations in $\mathbb{Z}_m$. Hence, the parallel complexity of RES_MOD is bounded by

$$T_{\mathcal{B}}(\text{RES\_MOD}(\hat{f}, \hat{g} \in \mathbb{Z}_m[x, y])) = O_P(n^2, n^2). \tag{3.67}$$

For Algorithm 2.10, computing the resultant of $f, g \in \mathbb{Z}[x, y]$, we use $N \times n^2$ processors where $N = O(n(\tau + \log n))$, see (2.41). The modular reduction (2.42) and invocation of the procedure RES_MOD (2.40) then takes $O_P(n^2 + \tau, Nn^2)$ parallel time since we perform the computations modulo each prime in parallel. To recover integer coefficients of the resultant, we again use the MRC algorithm computing a set of digits $\{\gamma\}$ which demands for

$$T_{\mathcal{B}}(\{\gamma\} \leftarrow \text{MRC\_ALGORITHM}(\{m\}, \{\tilde{R}\})) = O_P(N, Nn^2) \tag{3.68}$$

operations. Referring to the discussion in Section 3.3.1, the actual integers can also be computed in $O_P(N, Nn^2)$ parallel time. Thus, the resulting parallel complexity of the algorithm RES_6INT is

$$T_{\mathcal{B}}(\text{RES\_INT}(f, g \in \mathbb{Z}[x, y])) = O_P(n(n + \tau), n^3(\tau + \log n)). \tag{3.69}$$

As one can see, this bound is substantially better than the one given in (2.45). In addition, observe that, the attained complexity is *quadratic* in the degree and *linear* in the coefficient bitlength which is well observed in the experiments, see Section 4.4.4.

In this chapter, we have discussed the theory of displacement structure and derived the matrix-based algorithms providing a necessary background for efficient GPU realization. Theoretical considerations shows that these algorithms enable us to improve the parallel complexity of the resultant and GCD computations within the context of the PRAM model. This, in essence, is achieved by exploiting data-level (or fine-grained) parallelism inherent to the matrix computations. In the next chapter, we discuss the main realization details of GPU algorithms to show the feasibility of our approach in practice.

4 Realization and experiments

In the past two chapters, we have considered the algorithms for symbolic computations from the theoretical perspective. However, no matter how elegant a mathematical theory is, there is always a certain amount of disparity between theory and practice when it comes to the realization of the algorithms on a concrete platform. This is, in particular, true if the specifics of the target platform do not allow us to easily estimate how good an algorithm would perform in practice. As a result, there is always a trial-and-error component in the actual development process. For example, there are many parameters governing the performance of an algorithm running on the GPU which are not possible to account for during the initial design phase. This, in effect, may sometimes lead to controversial results.

This chapter is devoted to the main aspects of the realization of the symbolic algorithms on the graphics card. We begin with an introduction to the architecture of graphics processing units and CUDA framework. Then, we go through a number of code examples to illustrate some practical ways of writing an efficient GPU code. The readers, not familiar with the graphics accelerators, may also find this helpful to gain a better understanding of the principles of GPU programming. We next turn to the implementation of the modular resultant and GCD algorithms. Although, we shall try to keep the discussion relatively high-level, it will sometimes be necessary to consider the actual thread execution level: particularly, when taking about the time-critical subalgorithms. In conclusion, we run the set of benchmarks to compare our algorithms with CPU-based analogues to demonstrate the efficiency of parallel processing.

4.1 General purpose computing on GPUs

In this overview, we describe the principles of general purpose computing on graphics processors or, in other words, how the modern GPUs can be utilized for the tasks not necessarily related to computer graphics. Despite the fact that, we primarily focus on the graphics cards supporting CUDA framework (CUD10) promoted by NVIDIA, the main ideas and principles formulated are common to other GPU architectures as well. Besides, with the release of a new standard for heterogeneous programming OpenCL (Mun08), there is a tendency among the hardware manufacturers in many-core field to unify their architectures providing better support for OpenCL, while, in essence, CUDA bears a lot in common with OpenCL. We start by looking at the main hardware features of the GPU.

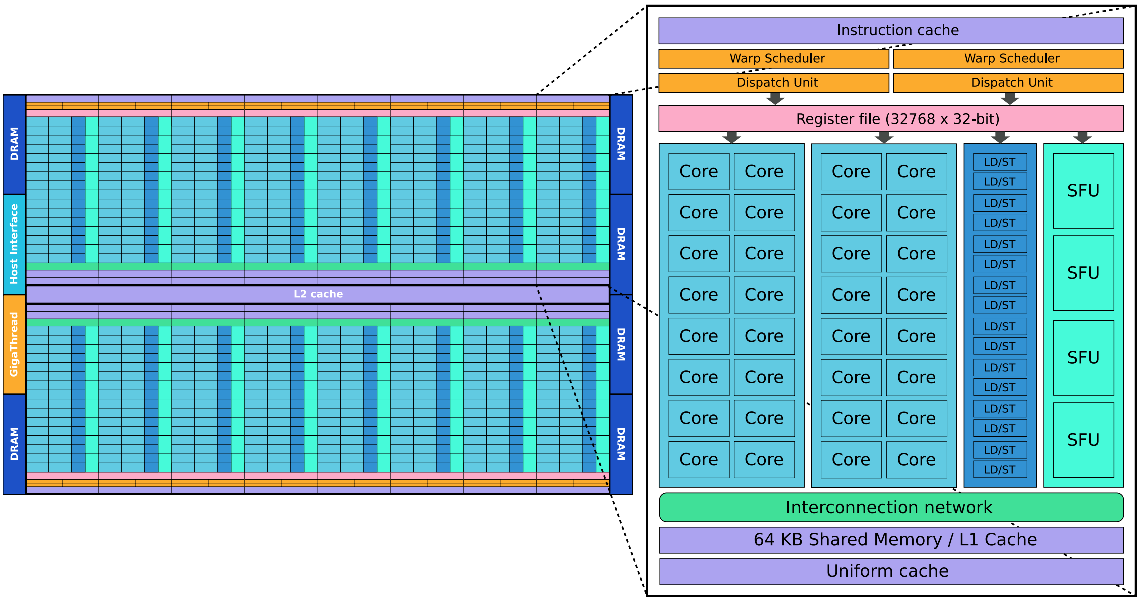

Figure 4.1: Architecture of a Fermi based GPU comprising 16 MPs (*left*); Streaming Multiprocessor (MP) with 32 CUDA cores (*right*)

4.1.1 GPU architecture

Commodity graphics hardware has evolved tremendously over the past decades starting from plain accelerators for basic polygon rendering to fully-programmable processors with outstanding computational power. With the emergence of shader programming, used to customize a (previously) fixed-function graphics pipeline, the developers have realized how to utilize the GPU to solve many complex problems in a more efficient way than on a CPU. However, the shader programming model was far from optimal since problems needed to be carefully translated into unnatural graphics-oriented environment. The situation has changed with the release of NVIDIA Tesla GPUs (LNOM08) supporting CUDA framework. Tesla design featured a “unified shader architecture” where the vertex and fragment processors are unified in the so-called *Streaming Multiprocessors* or MPs capable of running shader programs as well as general purpose parallel programs. In its turn, CUDA programming model provided a sufficient level of abstraction from the graphics hardware which simplified the design and implementation of parallel algorithms. The next generation GPU architecture, Fermi (Fer10), essentially followed the same unified approach as Tesla did but improved upon a number of important points. Below, we consider the latter architecture in greater detail.

The architecture of a Fermi based GPU with 16 Multiprocessors arranged around a block of common L2 cache is shown in Figure 4.1. As one can see, the most of the die area of the GPU goes into the actual data processing rather than a sophisticated flow control or caching. This illustrates a fundamental difference between CPUs and graphics processors where the latter ones are oriented to computationally intensive tasks. Each Multiprocessor includes 32 CUDA cores (scalar in-order processors), 16 load/store units (LD/ST) to handle memory operations, 4 special-function units (SFUs) for transcendental math, a 32K-word register file and 64 Kb of shared memory configurable as L1 cache. On the GPU, threads are scheduled for execution in groups of 32 threads called *warps*.

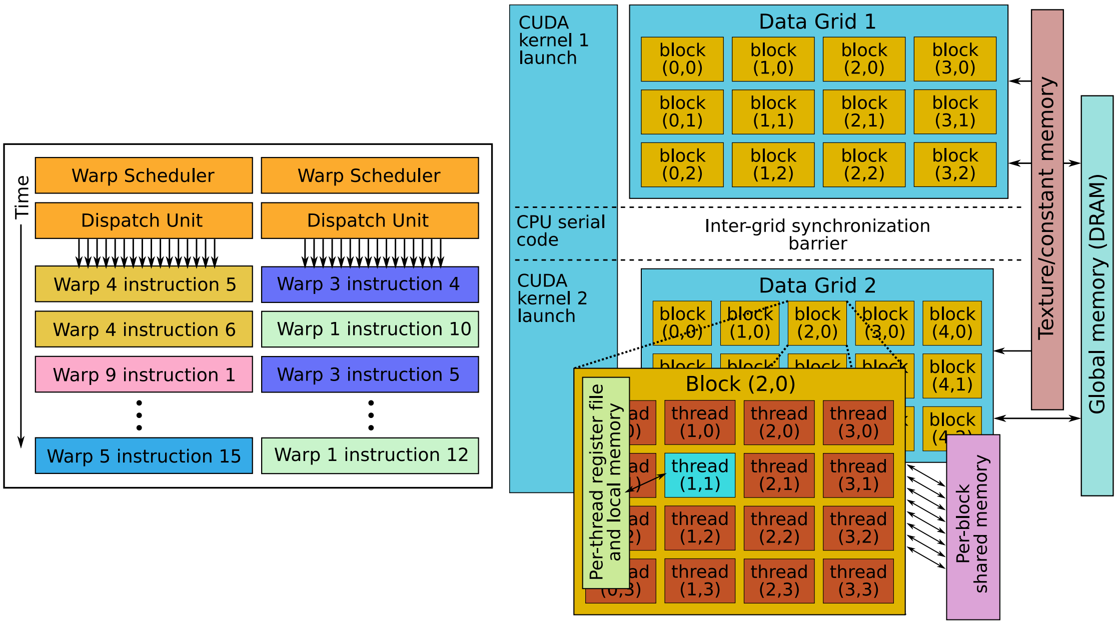

Figure 4.2: Scheduling warps for execution on the Multiprocessor (*left*); CUDA programming model, thread and memory hierarchy (*right*)

Threads of a warp always execute *synchronously*: in other words the same instruction is issued for the whole warp at a time. In this sense, the GPU can be regarded as 32-lane SIMD vector processor. Usual arithmetic operations, including addition/subtraction, bit operations, as well as integer and single-precision floating-point multiply and fused multiply-add (FMA), can be scheduled for execution on CUDA cores in *every* clock cycle.[1] Double-precision arithmetic operations can be scheduled for execution in every two clock periods which is 8x faster than on Tesla GPUs. Each MP has two warp schedules and instruction dispatch units (see Figure 4.1) allowing two warps to be executed *simultaneously* on an MP. Altogether, it takes two clock cycles to dispatch and execute an instruction for a warp since each warp is processed by an *execution block* consisting of 16 cores. The warp schedulers issue instructions for a pair of active warps that are ready to execute (no register dependencies, not waiting on synchronization point or memory access). That is why, any memory latencies or read-after-write hazards can be effectively hidden as long as there are enough active warps on an MP (in other words, by exploiting instruction level parallelism). Dual warp scheduling is shown in Figure 4.2 (left).

Since threads of a warp cannot physically execute different instructions at a time, when the warp encounters a data-dependent branch condition, all taken branch paths have to be processed *serially* until threads "converge" back to a single execution path. This problem, known as *thread divergence*, can bring a noticeable overhead in computations when a branch condition occurs in a time-critical section of the code. Different warps can execute disjoint paths without penalties. Such a model of execution is called SIMT (Single Instruction Multiple Thread), and it allows programmers to write thread-level parallel

[1] It should be understood that the actual *execution* of an instruction can take about 22 clock cycles depending on the complexity. Yet, this latency is completely hidden as long as an MP has enough warps to execute.

code without regard to the underlying vector organization. At the same time, whenever performance becomes a critical factor, grouping threads into warps can be taken into account to attain the full efficiency. Besides, to reduce the impact of thread divergence, all instructions on the GPU support *hardware predication* which enables short conditional statements to be executed without branching. Namely, if a predicate evaluates to ‘false’ for some threads of a warp, then the results of executing an instruction will not be written for these threads: i.e., this instruction will be treated as ‘no operation’. For threads with ‘true’ predicate, execution proceeds in a normal way.

4.1.2 CUDA framework

CUDA (CUD10) is a heterogeneous serial-parallel programming model, meaning that a serial execution on the host machine (CPU) is interleaved with a parallel execution on the device (GPU). A program running on the GPU across a large number of parallel threads is referred to as *kernel* in CUDA terminology. Kernels are written in the C language extended with additional keywords to express parallelism. At the highest level, multiple threads running on the GPU are grouped in a *grid* of *thread blocks* which is launched on a single CUDA kernel. Each block can contain up to 1024 threads.[1] Block and grid configuration for each kernel call are set by the user, see Figure 4.2 (right). As noted earlier, a minimal scheduling entity on the GPU is a warp. All warps of a thread block are assigned for execution to a single Multiprocessor. Threads in a block can communicate via MP’s *shared memory* and synchronize with barriers. Different thread blocks execute independent from each other and cannot exchange data during a kernel launch. It is only possible to pass data from one block to another between the kernel calls using off-chip global memory. Thread block independence is one of the cornerstones of the GPU architecture which enables a binary program to run unchanged on the devices with *any* number of physical Multiprocessors (and, thereby, achieving transparent scalability).

With regard to memory organization, CUDA introduces 6 memory spaces illustrated in Figure 4.2 (right). *Register* and *local* memory constitute a thread’s private memory storage. Registers of an MP are *statically* allocated to threads of a block at the beginning of a kernel call.[2] Note that, registers is a scarce resource and should be used cleverly to prevent register spilling. Local memory resides in an external DRAM and is typically used by the compiler for per-thread large temporary data and register spills. *Shared memory* is an on-chip memory which enables inter-thread communication within a block and has the same lifetime as the block. Shared memory is organized in 32 banks (or 16 banks on Tesla GPUs) to facilitate concurrent access. Consecutive addresses are mapped to different banks. If threads of a warp (or threads of a half-warp on Tesla) access memory from *different* banks at the same time, the corresponding requests can be serviced simultaneously. Otherwise, if two or more addresses fall into the same bank, a *bank conflict* occurs and the memory requests are *serialized* resulting in as many conflict-free requests as necessary. Each MP has a total of 64 Kb shared memory that is partly used as L1 cache (depending on the user configuration). The remaining three memory spaces, including *global*, *texture* and *constant* memory, are the parts of GPU’s external memory (same as

[1] On previous generation Tesla GPUs, the maximum block size was limited to 512 threads.

[2] Due to static register allocation, the context switching between warps induces no overhead.

local memory). They are visible to the entire grid of thread blocks and have the life-time of an application.

Global memory is the only read-write memory of the GPU which is accessible by all thread blocks. This off-chip memory has a much higher latency than shared memory, and thus it is recommended to access it in such a way that memory accesses by individual threads can be *coalesced* in a single wide memory access. However, the coalescing rules are much less restrictive on Fermi GPUs since global memory is cached. Read-only texture memory is optimized for spatial locality: this, for instance, might be useful when the data being accessed resides in a 2D array. Constant memory, as the name implies, is typically used to store program constants: when all threads of a warp read from the same memory location, this results in a single memory request to the read-only constant cache (in case of a cache hit) or device memory otherwise.

4.1.3 GPU optimization strategies

We finally outline some common optimization strategies to yield the best performance on the GPU. The first quite obvious yet very important strategy is to arrange the computations in a way to expose as much parallelism as possible, and thereby maximize the hardware utilization. Hardware utilization is characterized by the *occupancy* metric which is one of the major performance indicators on the GPU. Occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps on a Multiprocessor. For instance, on Fermi the maximal number of warps per MP is 48 which corresponds to 1536 threads, while the previous generation Tesla GPUs can have up to 32 resident warps on an MP (1024 threads).

The second strategy is to maximize memory throughput which mainly means optimizing the global memory data transfers. Global memory access, unless used properly, can substantially limit the performance making a whole algorithm *memory bound*: for example, reading data from global memory that is not yet in cache can cost about 400–800 clock cycles. As mentioned before, it is highly recommended to use coalescing access optimization to prevent wasting global memory bandwidth. Another common practice is to preload the data from device memory all at once, then use shared memory for subsequent computations, and, at the end, write the results back to global memory. In addition, one should pay attention to arithmetic intensity of computations to keep the warp schedulers busy during memory access periods.

Finally, register and shared memory allocation must be kept under control which has a direct impact on occupancy. Recall that, the MP has a limited-size register file (32K-word registers) and 64Kb of on-chip shared memory. When a grid is launched for a GPU kernel, the amount of registers and shared memory used by the kernel determines the maximal number of resident blocks per Multiprocessor (with maximum 8 blocks per MP). It is therefore preferable to use smaller thread blocks (containing 64–128 threads) instead of large ones.[1] Besides, starting from CUDA 3.0, the programmer can specify *launch bounds* for each kernel informing the compiler about the desired number of threads per block and blocks per Multiprocessor. The compiler, in its turn, might decide to allocate some

[1] Upon a grid launch, the physical registers of an MP are split evenly between threads of a block.

registers in local memory to meet the user requirements. Another technique to optimize register usage is to declare frequently used local variables with volatile keyword. This instructs the compiler to keep the respective variables in physical registers (instead of substituting the corresponding expressions) which often has a positive effect on reducing the register usage.

So far we have made the first exposure to the GPU architecture and CUDA framework. In the following section, we will go through a number of code examples to help the reader acquire some basic "look-and-feel" of GPU programming.

4.2 GPU programming: case study

In the course of our case study, we shall emphasize more on the practical aspects rather then theoretical background since, in the first place, our aim is to introduce the reader to some common programming practices on the GPU. Large background on data-parallel algorithms can be found in the classical literature (HS86, Roo99, J9́2). We would also recommend to visit the web-site http://gpgpu.org and have a look at the CUDPP library[1] implementing data-parallel algorithms which constitute the "basic building blocks" for many GPU applications.

4.2.1 Introduction

We assume that the reader has some basic knowledge of parallel programming, and recommend to consult a CUDA programming guide (CUD10) for details. Below, we only highlight the basic concepts. The first thing to understand about a GPU program is that any statement is to be executed by *each* participating thread independently. Threads of a CUDA block are referenced by a predefined variable which we denote by thid. For instance, the statement 'x = x + thid * 2' entails that each thread adds to the contents of its register (or local) variable x the assigned thread index. That is, the 0th thread has its value x unchanged, the 1st thread computes x = x + 2, the 2nd thread: x = x + 4, etc. The same applies to accessing any memory location: that is, the statement 'x = A[thid * 4 + 1]' entails that threads read the data from the array A with stride 4. In other words, the 0th thread reads A[1], the 1st thread – A[5], and so on. As a remark, if A is a shared memory array, such an access will result in 4-way bank conflicts since every 4th thread of a warp reads from the same memory bank. Particular attention must be paid to conditional statements which allows us to split the program execution into several code paths. The key observation here is that threads of the same SIMD-group (warp) cannot physically follow different code paths, and therefore all taken branches will be executed in a way as if they were the pieces of one serial program. Thread synchronization can be achieved by calling _syncthreads() library function which forces threads to wait on a barrier until all of them reach this synchronization point.

[1] http://code.google.com/p/cudpp

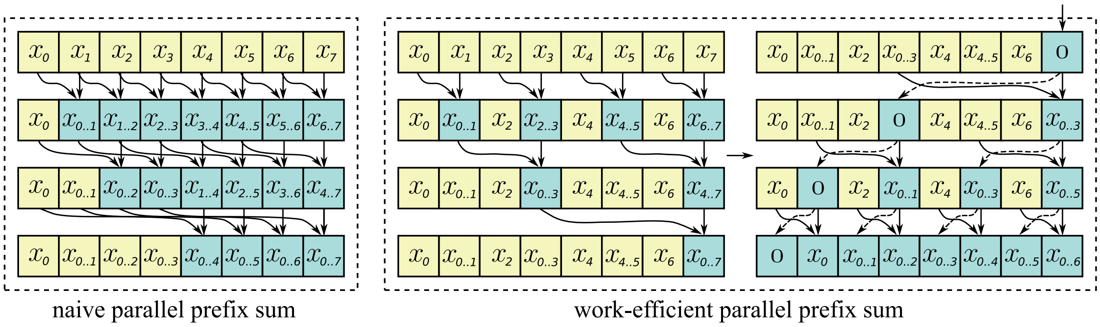

Figure 4.3: Naive parallel reduction (*left*); and work-efficient parallel reduction algorithms (*right*)

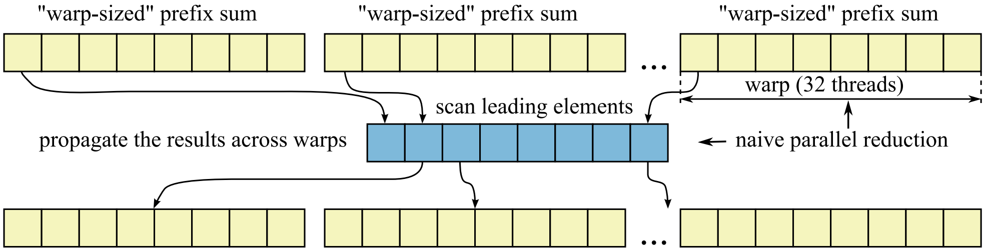

Figure 4.4: "Warp-sized" parallel reduction: the naive approach is used for each warp separately, afterwards the results are combined in a final reduction step

4.2.2 Parallel prefix sum

We begin with a commonly used operation of computing a prefix sum. Using the terminology of data-parallel algorithms, it is also known as *scan* or *parallel reduction*. More precisely, what we need is the operation computing *all-prefix-sums* as defined below.

Definition 4.2.1. For a binary associative operation $\oplus$ and an ordered set of elements $\{a_0, a_1, \ldots, a_{n-1}\}$, the *all-prefix-sums* operation returns the ordered set

$$\{a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})\}. \qquad \bullet$$

The all-prefix-sums operation is a fundamental block of many parallel algorithms including quicksort, string comparison, histograms, large integer addition, stream compaction, polynomial evaluation, lexical analysis, etc. Clearly, the problem of computing all-prefix-sums can be solved in $O(n)$ time using a simple sequential algorithm.

For the parallel solution on the GPU, in (Har07) two algorithms are described, both executing in $O(\log n)$ parallel time. These approaches can be best illustrated graphically as shown in Figure 4.3, where $x_{i..j}$ is a shortcut for $x_i \oplus \cdots \oplus x_j$. In the right diagram, $\mathbf{0}$ denotes an *identity* symbol, that is, an element satisfying $x_i \oplus \mathbf{0} = x_i$ for all x_i. Typically, the size of data is chosen in a way that all manipulations can be performed in GPU's shared memory to save on global memory bandwidth. Unfortunately, neither of these algorithms is efficient enough on the GPU. The first (naive) approach has to do a lot of extra work since the results from the previous steps are not effectively reused. The second (work-efficient) algorithm is supposed to overcome this problem by collecting the intermediate results in the first up-sweep phase, and then updating the remaining elements "from the root to leaves" in the second down-sweep phase. The efficiency of this

Listing 4.1 “warp-sized” parallel reduction with 256 threads

```
 1: template < class OP >  type32                                  ▷ OP: prefix operation
 2: WARP_SCAN(type32 x, int thid) {                 ▷ computes all-prefix-sums of 256 values x
 3:     extern _shared_ type32 data[];                     ▷ dynamic array of shared memory
 4:     const int WS = 32, HF = WS/2;          ▷ initialize warp-size and half-warp constants
 5:     volatile type32 *scan = data + HF +      ▷ shared memory space for parallel reduction
 6:         (thid%WS) + (thid/WS) * (WS + HF + 1);
 7:     scan[−16] = Ident;           ▷ fill the gap with identity elements: OP(Ident, x) = x
 8:     type32 t; scan[0] = x;                                 ▷ load data to shared memory
 9:     t = OP(t, scan[−1]), scan[0] = t;               ▷ compute prefix sums for each warp
10:     t = OP(t, scan[−2]), scan[0] = t;                  ▷ using naive reduction algorithm
11:     t = OP(t, scan[−4]), scan[0] = t;
12:     t = OP(t, scan[−8]), scan[0] = t;
13:     t = OP(t, scan[−16]), scan[0] = t;
14:     volatile type32 *postscan = data + HF +     ▷ “post-scan” leading elements of each warp
15:         (256/WS) * (WS + HF + 1);
16:     _syncthreads();                                           ▷ synchronization barrier
17:     if (thid < 8) {                                      ▷ post-scan 8 leading elements
18:         volatile type32 *scan2 = postscan + thid;
19:         scan2[−16] = Ident;                                  ▷ fill with identity elements
20:         t = data[HF + WS − 1 + thid * (WS + HF + 1)];   ▷ read in the prefix sums of each warp
21:         scan2[0] = t, t = OP(t, scan2[−1]);
22:         scan2[0] = t, t = OP(t, scan2[−2]);
23:         scan2[0] = t, t = OP(t, scan2[−4]);
24:         scan2[0] = t;
25:     }
26:     _syncthreads();                                           ▷ synchronization barrier
27:     t = OP(scan[0], postscan[thid/WS − 1]);   ▷ update warp prefix sums, postscan[−1] = Ident
28:     return t;                                           ▷ return prefix sums in registers
29: }
```

approach is attained at the price of increased number of parallel steps leading to worse occupancy. Furthermore, the data needs to be accessed exclusively with power-of-two strides causing many shared memory bank conflicts (unless special techniques to eliminate bank conflicts are employed). Lastly, both approaches suffer from synchronization overhead since threads need to be synchronized after each step to make sure the contents of shared memory are updated correctly.

The better way here is to exploit warp-level parallelism: the corresponding algorithm, called “warp-sized” parallel reduction, is illustrated in Figure 4.4. It was originally introduced in CUDPP library. In the following discussion, we consider a slightly more optimized version of this algorithm. The idea of the algorithm is to partition a block of data into warp-sized chunks and process them independently using the naive parallel reduction algorithm. Since warps execute instructions in a SIMD fashion, no synchronization points are necessary. In the second run, we scan the “warp sums” (the leading elements of each warp), and finally update the results after the first run using the scanned warp sums. The pseudocode of the algorithm computing all-prefix-sums of 256 values is given by Listing 4.1. Some comments are due here. In the code, `type32` defines any 32-bit data type, floating-point or integer, `OP` denotes a prefix operation we apply to the data elements, `thid` is a thread identifier, and `Ident` is an identity symbol. In line 5, we allocate 49 words (WS + HF + 1) of shared memory per each warp. The data is loaded to shared memory in line 8 while in line 7 additional 16 words are filled with identity elements to allow negative offsets (and, thus, save on guards for memory access in the reduction algorithm). The actual prefix sums are computed in lines 9–13 across each warp. The algorithm works according to the graphical description in Figure 4.3 (left). Note that, the array `scan` is declared using `volatile` keyword to enforce the writes to shared memory instead of "caching" the intermediate results in registers. In the second pass (lines 17–25), we collect and scan the warp prefix sums. Since the total number of warps is 8 ($32 \times 8 = 256$), it suffices to use 8 threads for that. In the end, each thread uses a corresponding warp prefix sum to update the result from the first pass (see line 27).

In summary, observe that the entire algorithm requires only *two* synchronization points (lines 16 and 26) irrespective of the actual data size. Besides, all threads are occupied during the first run, while in the second run only threads from a single warp participate, and thus there are no occupancy penalties. As a last remark, observe that the algorithm admits further optimizations: for instance, one might decide to process several elements per thread to reduce the block size and improve the arithmetic intensity of computations, etc.

4.2.3 Vectorization of a serial algorithm

As a next example, we shall consider how to port the actual serial algorithm to the GPU. Interestingly enough, the classical literature on parallel programming allocates much space to describe sophisticated data structures or typical algorithms, such as computing prefix sums or parallel matrix multiplication, while the question of translating a concrete algorithm to a parallel environment is not answered often enough. We shall try to fill this gap in a present discussion.

As a reference, we take the interpolation algorithm (Algorithm 3.2) from Section 3.2.2. The algorithm entails two nested loops and, as noted in Section 3.3, can be executed in $O_P(n, n)$ parallel time. For simplicity, we shall only exploit *thread-level* parallelism assuming that the algorithm can be run entirely by one CUDA block. Some ideas how to distribute the computations across numerous thread blocks will be outlined in Section 4.5.3. Also, for the time being, we shall not concern ourselves with the realization of modular arithmetic which is a separate topic in our discussion, see Section 4.3. From the pseudocode, we see that it suffices to use n threads, with n being the number of evaluation points, since only n relevant entries of the generator matrix are updated at a time. Listing 4.2 provides a quite straightforward vectorization of Algorithm 3.2. We allocate three shared memory arrays `A`, `B` and `X` of size `block_sz` for the generator columns $G = (a, b)$ and evaluation points, respectively. Here, `block_sz` denotes the actual number of threads used per block which is equal to n aligned by 32 (warp size). We dedicate one thread to process one row of the generator matrix G. Thus, the inner loop in Algorithm 3.2 disappears. Threads are enumerated by `thid`, while `last_thid` is the ID of the last working thread. In the code, SUB_MUL_MOD(x, y, z, w) denotes an operation computing $(x \cdot y - z \cdot w)$ over a prime field. Its concrete realization will be considered in Section 4.3.

In lines 24–30, we express the update of the column a using the temporary variables `s` and `t` to prevent excessive branching and, thus, thread divergence. In addition, we incorporate the update of the denominator `det[0]` to the main computations which is also done to reduce branching overhead (see lines 17 and 22). Here, the key observation is that having one thread to do some additional work is equivalent to having a *whole* warp

Listing 4.2 Polynomial interpolation algorithm

```
 1: int                                                   ▷ returns the coefficients of $f(x)$, s.t., $f(x_i) = y_i,\ 0 \le i < n$
 2: VANDERMONDE_INTERP(int *g_X, int *g_Y, const int n, const int block_sz, int thid) {
 3:     extern _shared_ int data[];                       ▷ dynamic array of shared memory
 4:     const int last_thid = n − 1;                      ▷ ID of the last working thread
 5:     int *B = data + thid, *A = B + block_sz;          ▷ init shared memory arrays
 6:     int *X = data + block_sz * 2, *det = X + block_sz, a, b, x;
 7:     if (thid <= last_thid) {
 8:         B[0] = g_Y[thid], X[thid] = g_X[thid];        ▷ load $y$-values g_Y and eval. points g_X
 9:         A[0] = 1;                                     ▷ fill the column a with 1's
10:     }
11:     if (thid == 0) { det[0] = 1; }                    ▷ initially, the denominator is set to 1
12:     _syncthreads();                                   ▷ synchronization barrier
13:     for(int j = 0; j < n; j++) {
14:         if (thid < last_thid) {
15:             a = A[1], b = B[1];                       ▷ read in the generator columns G = (a, b)
16:         } else if (thid == last_thid) {
17:             a = 0, b = det[0];                        ▷ last thread updates the denominator
18:         }
19:         int b0 = r[0], a0 = r[block_sz];              ▷ read in a leading generator row $(a_0, b_0)$
20:         b = SUB_MUL_MOD(b, a0, a, b0);                ▷ update the column vector b
21:         if (thid == last_thid) {
22:             a = 1, det[0] = b, b = −b0;               ▷ save the denominator, set b[j + n] = −b0
23:         }
24:         s = 0, t = 0, k = thid + j + 1;               ▷ k is used as a case selector
25:         if (k < n) {
26:             s = a, t = X[k];
27:         } else if (k > n) {
28:             s = A[0], t = 1;                          ▷ A[0] is a preceding element of a
29:         }
30:         a = SUB_MUL_MOD(s, t, a, X[j]);               ▷ update the column vector a
31:         _syncthreads();                               ▷ synchronization barrier
32:         A[0] = a, B[0] = b;                           ▷ shift down the generator columns
33:         _syncthreads();                               ▷ synchronization barrier
34:     }
35:     return (b/det[0]);                                ▷ divide the coefficients by the denominator
36: }
```

to do this work. In addition, this would cause threads from other warps to wait longer at synchronization points. Therefore, in a time-critical code, it is always desirable to arrange the computations in a "uniform" way.

At the end of each iteration, the generator columns are shifted down in shared memory (line 32). Remark that, the shared memory arrays `A` and `B` are accessed in a quite unusual way using just the indices 0 and 1: this is because we have added a thread identifier `thid` to the address of each variable upon initialization, see line 5.

Unfortunately, the main weakness of the above algorithm is its *low* arithmetic complexity. Indeed, there are only two actual "arithmetic" statements in the loop (lines 20 and 30), while the remaining code is needed for case distinction and memory operations. One way to improve this is to exploit *instruction-level parallelism* by unrolling the inner loop. At the same time, this would enable us to process higher-degree polynomials using thread blocks of the same size. However, we need to be careful here since loop

Listing 4.3 Polynomial interpolation with unrolled inner loop

```
template < int Mod4 >  int4                    ▷ returns the coefficients of f(x), s.t., $f(x_i) = y_i$, $0 \le i < n$
VANDERMONDE_QUAD_INTERP(int4 *g_X, int4 *g_Y, const int n, const int block_sz, int thid) {
    extern _shared_ int data[];                                    ▷ dynamic array of shared memory
    const int last_thid = (n + 3)/4 − 1;                            ▷ ID of the last working thread
    int *B = data + thid, *A = B + block_sz, *X = A + block_sz;     ▷ init shared memory arrays
    int *det = data + block_sz * 3, a_prev = 0;
    int4 a, b, w;                          ▷ SIMD-vectors to store the columns of a generator matrix
    if (thid <= last_thid) {
        b = g_Y[thid], w = g_X[thid];                     ▷ load y-values g_Y and eval. points g_X
        a.x = a.y = a.z = a.w = 1;                              ▷ fill the column vector a with 1's
    }
    X[0] = w.x, B[0] = b.x, A[0] = 1;
    if (thid == 0) { det[0] = 1; }                         ▷ initially, the denominator is set to 1
    _syncthreads();                                                     ▷ synchronization barrier
    for(int j = n − 1; j >= 0; j−−) {
        b.x = b.y, b.y = b.z, b.z = b.w;                       ▷ shift down the generator columns
        a.x = a.y, a.y = a.z, a.z = a.w;                              ▷ and the array eval. points
        w.x = w.y, w.y = w.z, w.z = w.w;
        if (thid < last_thid) {
            b.w = B[1], w.w = X[1];
        } else if (thid == last_thid) {
            a.w = 0, b.w = det[0];                          ▷ last thread updates the denominator
        }
        int b0 = r[0], a0 = data[block_sz];                    ▷ read in a leading generator row
        b.x = SUB_MUL_MOD(b.x, a0, a.x, b0);                         ▷ update the column vector b
        b.y = SUB_MUL_MOD(b.y, a0, a.y, b0);
        b.z = SUB_MUL_MOD(b.z, a0, a.z, b0);
        b.w = SUB_MUL_MOD(b.w, a0, a.w, b0);
        if (thid == last_thid) {
            det[0] = b;                                                   ▷ save the denominator
            if (Mod4 == 1) { a.x = 1, b.x = −b0; }      ▷ set b[j + n] = −b0; compile-time decision
            else if (Mod4 == 2) { a.y = 1, b.y = −b0; }
            else if (Mod4 == 3) { a.z = 1, b.z = −b0; }
            else { a.w = 1, b.w = −b0; }
        }                                                        ▷ read in the top eval. point x[j]:
        x0 = r[block_sz * 2], k = j − 4 * thid − 1;                          ▷ k is a case selector
        if (thid == j/4) { a_prev = 0; }
        _syncthreads();                                                 ▷ synchronization barrier
        int t1 = 0, s1 = x0, tmp;
        if (k >= 0) { s1 = SUB_MOD(s1, w.x); }                          ▷ computes s1 = s1 − w.x
        else { t1 = a_prev; }                                ▷ a_prev is an element preceding a.x
        tmp = a.x, a.x = SUB_MUL_MOD(t1, s1, a.x);                  ▷ computes a.x = t1 − s1 * a.x
        .......
        .......                                ▷ updating the vector elements a.x, a.y, a.z, a.w
        .......
        B[0] = b.x; A[0] = a.x, a_prev = a.x, X[0] = w.x;       ▷ write the results to shared memory
        _syncthreads();                                                 ▷ synchronization barrier
    }                                                  ▷ divide the coefficients by the denominator:
    return (b.x/det[0], b.y/det[0], b.z/det[0], b.w/det[0]);
}
```

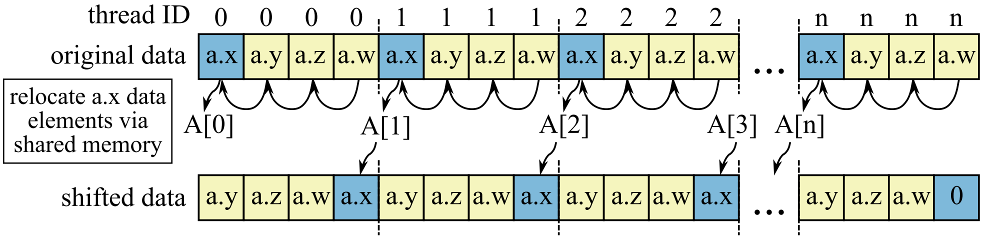
thread ID
0 0 0 0 1 1 1 1 2 2 2 2 n n n n
original data
a.x a.y a.z a.w a.x a.y a.z a.w a.x a.y a.z a.w ... a.x a.y a.z a.w
relocate a.x data elements via shared memory
A[0]
A[1]
A[2]
A[3]
A[n]
shifted data
a.y a.z a.w a.x a.y a.z a.w a.x a.y a.z a.w a.x ... a.y a.z a.w 0

Figure 4.5: Left-shift of a row vector stored implicitly in threads' register space. Each thread keeps 4 consecutive data elements in variables `a.x`, `a.y`, `a.z` and `a.w`.

unrolling increases register pressure, and therefore can have a negative effect on the occupancy. We have decided to unroll the loop by the factor of 4: in this situation we still have enough registers per thread. Accordingly, the number of working threads decreases to $\lceil n/4 \rceil$, where n is the number of evaluation points. Thread local variables `a` and `b` representing the column vectors of the matrix $G = (a, b)$ are now replaced by SIMD-vectors of type `int4` storing 4 consecutive data elements. The implementation is given in Listing 4.3. Despite the seeming complexity, the algorithm follows the same line of thoughts as the one from Listing 4.2. We highlight the main features of the algorithm. Each thread now processes 4 rows of the generator matrix at a time which is reflected in lines 25–28 and 39–46. Note that, we have also bypassed some details on updating the column vector `a` for reasons of space.

Perhaps one of the tricky parts is how the shifting of the column vectors `a` and `b` and the array of evaluation points `w` is realized (see lines 16–20 and 46–47). Here, the idea is to shift the vector contents in register space and use shared memory only to relocate the "corner" elements (`a.x`, `b.x` or `w.x`) as illustrated in Figure 4.5. In that way, we can significantly reduce the usage of shared memory. Another remarkable feature of the algorithm is the use of *template* parameter `Mod4` denoting the "data parity" $n \bmod 4$. The motivation for this is that the last element of a column vector may fall into different components of length-4 SIMD-vector, depending on the actual number of points n. Hence, to avoid lengthy conditional statements selecting vector components, we can parameterize the algorithm by '$n \bmod 4$', letting the compiler do register selection at compile time (see lines 31–34). The last thing that worth attention is that we no longer keep the evaluation points in shared memory but, instead, allocate a SIMD-vector `w` for that, to be consistent with our data storage pattern.

In conclusion, the algorithm in Listing 4.3 has a much higher arithmetic intensity. Furthermore, we have significantly reduced the usage of shared memory by moving the relevant data to register space. Provided that, the maximal number of threads per block is limited by 1024 (on Fermi GPUs), we can now interpolate polynomials of degree up to 4096.

4.2.4 FFT algorithm

In the sequel of our case study, we consider a slightly more advanced topic of computing the Fast Fourier Transform (FFT) on the GPU. For expository purposes, we shall focus only on 512-point and 1024-point transforms. Due to the fact that the FFT has applications in almost every field of modern science, the interest to efficient GPU implementations

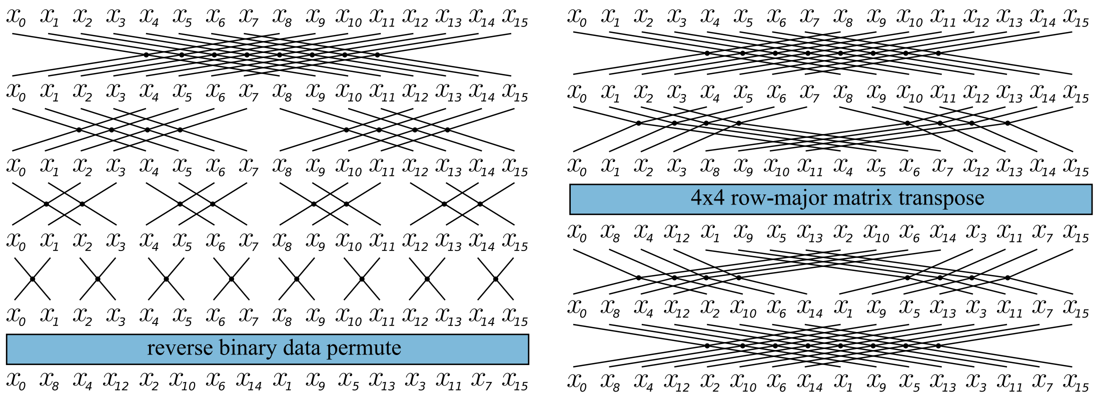
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
reverse binary data permute
x_0 x_8 x_4 x_{12} x_2 x_{10} x_6 x_{14} x_1 x_9 x_5 x_{13} x_3 x_{11} x_7 x_{15}
x_0 x_1 x_2 x_3 x_8 x_9 x_{10} x_{11} x_4 x_5 x_6 x_7 x_{12} x_{13} x_{14} x_{15}
4x4 row-major matrix transpose
x_0 x_8 x_4 x_{12} x_1 x_9 x_5 x_{13} x_2 x_{10} x_6 x_{14} x_3 x_{11} x_7 x_{15}

Figure 4.6: Butterfly networks for decimation-in-frequency Cooley-Tukey (*left*) and Bailey's variation of Stockham (*right*) FFT algorithms

has risen tremendously with the release of CUDA framework. At the time of writing, several GPU libraries have been developed which are specialized on computing floating-point (or complex-arithmetic) Fourier transforms, for example: (CUF10, GPU, NUF), see also (GLD⁺08). To date, there are also two algorithms dealing with finite field transforms on the GPU (Eme09, MP10). We discuss some details of our work here. We assume that the reader has some basic knowledge about the FFT algorithms. A very good practical study with examples and source code can be found in (JÏ1, Chapter 21). To recall the basic facts, let $\mathbb{F}$ be a field with some properties to be identified shortly.[1] Usually, this is a field of complex numbers $\mathbb{C}$ or a prime field. Then, for a vector $a \subset \mathbb{F}^n$, the n-point Discrete Fourier Transform (DFT) is a linear map $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^n$ defined as $c := \mathcal{F}[a]$, where $c_k = \sum_{k=0}^{n-1} a_k w_n^k$ for $0 \le k < n$. Here, w_n is an n-th primitive root of unity having the following properties: $w_n^n = 1$ and $w_n^k \ne 1$ $(0 < k < n)$. The inverse DFT $\mathcal{F}^{-1}$ is defined in a similar way: $a := \mathcal{F}^{-1}[c]$ with $a_j = \frac{1}{n}\sum_{j=0}^{n-1} c_j w_n^{-j}$ for $0 \le j < n$.

Naturally, the DFT exists if there is an n-th primitive root of unity (an element of order n) in a field $\mathbb{F}$, and n can be inverted in $\mathbb{F}$. In $\mathbb{C}$, we simply take $w_n = \exp(\pm 2\pi i/n)$, while in a finite field specific conditions must be met for w_n to exist, see (ER83). In a matrix form, the DFT can be formulated as a multiplication by the so-called *Fourier* matrix $V_{\mathcal{F}}(w_n) := [w_n^{kl}] \in \mathbb{C}^{n\times n}$, for $0 \le k, l < n$, which belongs to the class of Vandermonde matrices due to the properties of the roots of unity.

All the variety of FFT algorithms originate from different factorizations of the Fourier matrix. In particular, when it comes to the realization on a parallel platform, usually the method of choice is a Stockham self-sorting FFT algorithm. Here, the reason is that a classical Cooley-Tukey FFT accesses data with a *power-of-two* strides which can have a significant impact on the throughput of GPU's shared memory (due to bank conflicts), let alone the wasted external memory bandwidth (see Section 4.1.1). Moreover, the Cooley-Tukey algorithm has an index permutation phase which results in almost randomized memory access. In contrast, the Stockham FFT incorporates index permutations directly into the algorithm, and accesses all data exclusively with *unit* strides. However, the price for that is that the data processing cannot be done "in-place" anymore. On the GPU, we

[1] Strictly speaking, it is only required for $\mathbb{F}$ being a ring with a particular cyclic group for the Fourier transform to exist. Yet, we shall not consider these subtleties here.

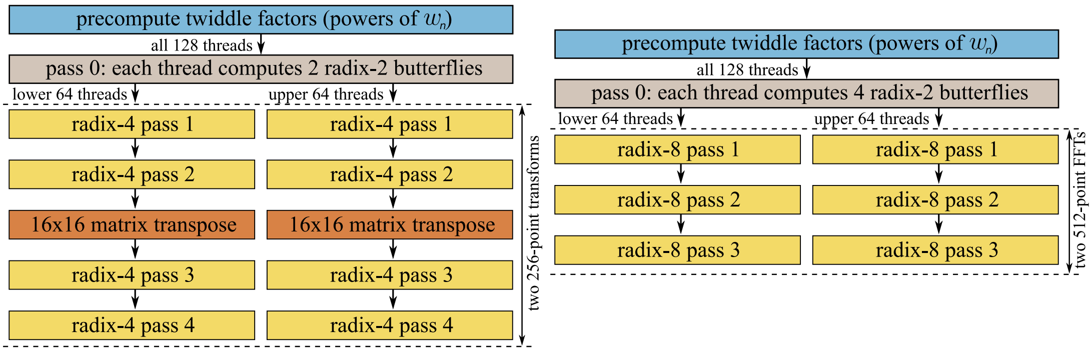

Figure 4.7: Mapping 512-point FFT to a CUDA block of 128 threads (*left*); 1024-point FFT with 128 threads per block (*right*)

use the Bailey's formulation (Bai88) of this algorithm. Without excessive matrix notation, the differences between two algorithms are illustrated in Figure 4.6. One can see that, on the right diagram, both operands of each radix-2 "butterfly" operation are accessed in a contiguous manner. In fact, Bailey's algorithm consists of two algorithm variants separated by a matrix transpose in the matrix data is taken in a row-major order. For the first variant, the size of contiguous blocks of data fetched from the memory *halves* in each step, while the output data pattern is always the same, see Figure 4.6 (right). For the second variant, everything is vice a versa: that is, the data is always fetched using the same pattern while the results after a "butterfly" operation are saved back in contiguous blocks of *increasing* size. The crossover point between the two variants can be chosen arbitrary depending on the transform size and the SIMD-vector length on a target architecture. Certainly, the same algorithm's outline can be used for the FFTs of higher radices.

We now exemplify how to compute 512 and 1024-point FFTs on the GPU. The reason why we have taken these particular transforms is because they fit nicely in a single CUDA block, and can be used as "basic building blocks" to construct larger transforms. The main design question is how to distribute the work between threads: here, we are free to choose the FFT-radix (which is tied to the number of threads per block) as well as the crossover between two algorithm variants (see above). For 512-point FFT, the naive solution would be to use radix-2 transform, and therefore 256 threads per block since each thread is then assigned to computing one FFT-butterfly. However, this way, the number of arithmetic operations per thread would be far too low, while shared memory access would be quite intensive. The better solution is to use radix-4 transform with 128 threads per block instead, and factor 512-point FFT as 2×4^4. Schematic view of the FFT algorithm is given in Figure 4.7 (left). We start by precomputing the so-called twiddle factors (the powers of w_n), so that they can be accessed in a contiguous manner throughout the algorithm from shared memory. Next, we perform a radix-2 step, and partition threads in two halves of 64 threads each to compute two 256-point transforms separately. Each 256-point FFT is realized in four radix-4 steps ($256 = 64 \times 4$), where, after each step, the data is reordered in shared memory. Recall that, a radix-k "butterfly" operation is formally defined as:

$$[y_0, \dots, y_{k-1}]^T = V_{\mathcal{F}}(w_k)\,\mathrm{diag}(1, \alpha^j, \dots, \alpha^{(k-1)j})[x_0, \dots, x_{k-1}]^T,$$

where α^j is a twiddle factor and $V_{\mathcal{F}}(w_k)$ is an $k \times k$ Fourier matrix. For instance, one radix-4 step of the algorithm can be realized as follows, see also (Eme09, Section 5.4):

```
procedure RADIX4_STEP($x_0$, $x_1$, $x_2$, $x_3$, $\alpha$, $w_8$)
    $(d_0, d_1) \leftarrow$ BFY_RADIX2($c_2$, $c_0$, $\alpha^2$), $(d_2, d_3) \leftarrow$ BFY_RADIX2($c_3$, $c_1$, $\alpha^2$)
    $(y_0, y_2) \leftarrow$ BFY_RADIX2($d_2$, $d_0$, $\alpha$), $(y_1, y_3) \leftarrow$ BFY_RADIX2($d_3$, $d_1$, $\alpha \cdot w_8$)
    return [$y_0$, $y_1$, $y_2$, $y_3$]
end procedure
```

In the above pseudocode, w_8 denotes the 8-th root of unity, α is a suitable twiddle factor, and BFY_RADIX2(x_0, x_1, w) is a primitive radix-2 "butterfly" operation computing: $(y_0, y_1) \leftarrow x_0 \pm x_1 \cdot w$. After two radix-4 steps, we perform the matrix transposition (in shared memory) and switch to the second variant of Bailey's approach. The reason why we do this is because it becomes increasingly less efficient to fetch the input data with each step for reasons explained above.

To realize 1024-point transform, we again take 128 threads per block, but use different factorization of the Fourier matrix. Namely, we split the transform as 2×8^3 since $1024 = 128 \times 8$. The procedure is outlined in Figure 4.7 (right). The algorithm starts by performing one radix-2 step, then the transform is split in two 512-point FFTs computed by each group of 64 threads separately in 3 steps, so that each thread performs a radix-8 "butterfly" operation in each step ($512 = 64 \times 8$). Note that, in this case, we only use the *first* algorithm variant in Bailey's approach since the matrix transposition would require too large memory space (32×16 words). Furthermore, during the last radix-8 step, the contiguous blocks of the input data have size 8, hence shared memory access does not cause much performance penalty (in comparison to radix-4 steps). Naturally, the same FFT-layout can be utilized for 512-point transform, considered previously, where we would use 64 threads per block instead. Generally, the transforms of higher radices are more preferable since they improve the arithmetic intensity of the computations while reducing the amount of memory transactions. However, higher radices also have a negative effect on register usage. That is why, in real applications, choosing one or another factorization of the transform will mostly be determined by the complexity of the underlying arithmetic: for example, whether it is floating-point, arbitrary modular, or special modular in case of Fermat/Mersenne transforms, etc. According to the benchmarks in (Eme09, Section 6), with this approach, we have been able to achieve up to 462 GFlop/s[1] for 512-point finite-field FFTs on the GeForce GTX 280 graphics processor, thereby utilizing about 50 % of its peak theoretical performance. The latter one is estimated as 933 GFlop/s (single-precision floating-point peak performance). Though, these results seem to be somewhat outdated at the time of writing, they still show the efficiency of the proposed approach.

Finally, to realize arbitrary-size transforms on the GPU, one usually applies a hierarchical approach where a larger transform is decomposed into multiple smaller ones that fit in GPU's shared memory, and therefore can be computed by one thread block: for instance, using the algorithms given above. Then, the transformed sequences are combined together by multiplying them with appropriate twiddle factors.

[1] GFlop/s stands for "10^9 floating-point operations per second."

Listing 4.4 24-bit modular arithmetic for Tesla GPUs

```
int
MUL_MOD(int a, int b, int m, float invm) {          ▷ computes $a \cdot b \bmod m$
    float hf = _uint2float_rz(_umul24hi(a, b)),      ▷ compute 32 MSB of the product $a \cdot b$
          prodf = _fmul_rn(hf, invm);                ▷ $invm = (float)(1 \ll 16)/m$
    int l = _float2uint_rz(prodf),                   ▷ truncate towards zero
        r = _umul24(a, b) − _umul24(l, m);           ▷ $r \in [-2m + \varepsilon; m + \varepsilon]$
    if (r < 0) {                                     ▷ adjust the result if negative sign
        r = r + _umul24(m, 0x1000002);               ▷ $r = r + m \cdot 2$
    }
    return umin(r, r − m);                           ▷ subtract m if $r \geq m$
}
```

4.3 Modular arithmetic

As we have seen in Section 2.2, the homomorphism approach have many advantages over the classical symbolic algorithms which also has a positive effect on the attained asymptotic complexity (see Section 2.5). However, to observe these benefits in practice, one needs to pay attention to some technical aspects since, in the end, everything boils down to the efficiency of the underlying arithmetic. In this section, we discuss the subroutines providing modular arithmetic support for the main algorithms discussed in the following sections.

Realization of the fast modular arithmetic on the GPU is not an easy task to accomplish since the graphics hardware was heavily optimized for floating-point performance, while, for example, integer division and modulo ('%') operations are particularly slow on the GPU and should not be used in a time-critical code. More than that, GPUs with Tesla architecture support only 24-bit integer multiplication natively while 32-bit multiplication is demoted in more primitive operations. Fortunately, the next generation Fermi GPUs support 32-bit integer arithmetic fully in hardware. To keep our algorithms backward-compatible, we shall consider the realization of relevant modular operations on *both* architectures.

4.3.1 Primitive operations

Tesla architecture. Certainly, the most demanding operation is the modular multiplication whose realization we consider in detail. On Tesla GPUs, we restrict ourselves to 24-bit modular arithmetic which reflects the native hardware capabilities. Another reason for that is because a 24-bit residue fits in the mantissa of a single-precision floating-point number, and thus we can replace expensive integer division by floating-point operations. Integer multiplication is realized in two instructions: mul24.lo and mul24.hi which compute 32 least and most significant bits (LSB and MSB) of the product of 24-bit integer operands, respectively. These instructions can be accessed directly from CUDA using inline assembly (PTX10).

Note that, in some earlier works, the authors were not aware of the inline PTX assembly and had to use numerous tricks to get the modular arithmetic working. For example, in (MPS07) it was suggested to use composite moduli consisting of 2 primes whose product fits in 24 bits. Hence, unfolding the CRT (Chinese Remainder Theorem) over these

Listing 4.5 24-bit modular arithmetic for Tesla GPUs

```
int                                                          ▷ computes $(x_1y_1 - x_2y_2) \bmod m$
SUB_MUL_MOD(int x1, int y1, int x2, int y2, int m, float inv1, float inv2) {
    float h1 = _uint2float_rz(_umul24hi(x1, y1)),            ▷ two inlined MUL_MOD operations
          h2 = _uint2float_rz(_umul24hi(x2, y2));
    int l1 = _float2uint_rz(_fmul_rn(h1, inv1)),             ▷ inv1 = 65536.0f/m
        l2 = _float2uint_rz(_fmul_rn(h2, inv1));             ▷ multiply and truncate
    int r = mc + _umul24(x1, y1) − _umul24(l1, m) −          ▷ $mc = m \cdot 100$
          _umul24(x2, y2) + umul24(l2, m)                    ▷ compute difference of two MUL_MOD's
    float rf = _uint2float_rn(r) ∗ inv2 + e23;   ▷ $rf = \lfloor r/m \rfloor$, inv2 = 1.0f/m, e23 = (float)(1 ≪ 23)
    r = r − _umul24(_float_as_int(rf), m);                   ▷ compute: $r = r - \lfloor r/m \rfloor \cdot m$
    return (r < 0 ? r + m : r);
}
```

1	imul.hi.u24.u24 r9, r0, r1; (r0 = x1, r1 = y1)	9	**imad**.u24 r4 (c2), r7, r8, r11; (r7 = m)
2	i2f.f32.u32.trunc r10, r9;	10	f2i.u32.f32.trunc r0, r0;
3	imul.hi.u24.u24 r9, r2, r3; (r2 = x2, r3 = y2)	11	**imad**.u24 r4 (c2), r2, r3, r4;
4	fmul.trunc r10, r10, r4; (r4 = inv1)	12	**imad**.u24 r0, r0, r7, r4;
5	**imad**.u24 r11, r0, r1, r8;	13	i2f.f32.u32 r4, r0;
6	i2f.f32.u32.trunc r0, r9;	14	**fmad** r4, r4, r5, c[0x1][0x3]; (r5 = inv2)
7	f2i.u32.f32.trunc r8, r10;	15	**imad**.u24.c0 r0 (c2), r4, r7, r0;
8	fmul.trunc r0, r0, r4;	16	iadd r0 (c0.sign), r7, r0;

Table 4.1: Disassembly of '$(x_1y_1 - x_2y_2) \bmod m$' operation on Tesla architecture.

two primes, the modular multiplication can proceed without intermediate values that exceed 24 bits. We find that this method requires too many arithmetic operations. The authors of (HW09) proposed to use 12-bit residues since the reduction after multiplication can proceed in floating-point without overflow concerns. In the other paper (BCC$^+$09), 280-bit residues were partitioned in 10-bit limbs to facilitate multiplication. As a result, neither of these techniques can exploit the GPU capabilities at full. Another alternative would be to use an algorithm based on Montgomery multiplication but, after much trial-and-error, we have found out that a simpler approach that uses floating-point arithmetic works best in practice.

The algorithm computing $a \cdot b \bmod m$ for 24-bit residues a and b is given by the procedure MUL_MOD in Listing 4.4. A description of the CUDA-specific functions used in the code is provided below:

- _umul24/_umul24hi: return 32 least and most significant bits of the product of 24-bit unsigned integer operands, respectively;
- _uint2float_rz: convert an unsigned integer to single-precision floating-point number using "round towards zero" rounding mode;
- _float2uint_rz/_float2uint_rn: convert a single-precision floating-point number to unsigned integer using "round towards zero" and "round to nearest even" rounding modes, respectively;
- _fmul_rn: return the product of two single-precision floating-point numbers using "round to nearest even" rounding mode;
- _float_as_int: reinterpret a single-precision floating-point number as an integer without conversion;

Listing 4.6 31-bit modular multiplication on Fermi cards

```
 1: int
 2: MUL_MOD(int a, int b, int m, double inv) {                        ▷ computes $a \cdot b \bmod m$
 3:     int hi = _umulhi(a * 2, b * 2);                               ▷ compute 32 MSB of the product
 4:                               ▷ multiply and truncate, $inv = (double)(1 \ll 30)/m$:
 5:     double rf = _uint2double_rn(hi) * inv + (double)(3 ≪ 51);     ▷ $rf = \lfloor a \cdot b \cdot 2^{30}/m \rfloor$
 6:     int r = a * b − _double2loint(rf) * m;                        ▷ compute partial residue
 7:     return (r < 0 ? r + m : r);                                   ▷ adjust by m if negative sign
 8: }
 9: int                                                               ▷ computes $(x_1y_1 - x_2y_2) \bmod m$
10: SUB_MUL_MOD(int x1, int y1, int x2, int y2, int m, double inv) {
11:     int r1 = MUL_MOD(x1, y1, m, inv),                             ▷ compute $x_1 \cdot y_1 \bmod m$
12:         r2 = MUL_MOD(x2, y2, m, inv);                             ▷ compute $x_2 \cdot x_2 \bmod m$
13:     r1 = r1 − r2;                                                 ▷ subtract the residues
14:     return (r1 < 0 ? r1 + m : r1);
15: }
```

- umin: return the minimal of two unsigned integers.

The idea behind the algorithm can be explained as follows. First, we partition the product $a \cdot b$ in 32- and 16-bit parts, and then apply the following congruence:

$$a \cdot b = 2^{16}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = (2^{16}hi - m \cdot l) + lo = a \cdot b - m \cdot l,$$

where $0 \le \lambda < m$. Denoting $r := a \cdot b - m \cdot l$, it can be shown that $r \in [-2m + \varepsilon; m + \varepsilon]$ for $0 \le \varepsilon < m$. Since r fits in 32 bits, it suffices to consider only 32 least significant bits of both products $a \cdot b$ and $m \cdot l$ in order to compute it (see line 6). At the end, in lines 7–10, we further reduce r to bring it to the valid range $[0; m - 1]$. The procedure, SUB_MUL_MOD, in Listing 4.5 evaluates an expression: $(x_1y_1 - x_2y_2) \bmod m$, which is frequently used by division-free matrix algorithms, see Section 3.2. In essence, the algorithm entails two inlined MUL_MOD operations with the exception that we compute the difference of the partial residues in lines 7–8 before the final reduction step. The advantage is that the compiler can merge the subsequent *multiply* and *add* instructions producing more efficient code. The remaining lines 9–11 are needed to bring r to the valid residue range. In particular, in line 10 we also use a *mantissa trick* (Hec96) to multiply by $1/m$ and truncate the result using one multiply-add instruction. We have studied the efficiency of our approach using cuobjdump disassembler shipped as part of CUDA toolkit. Table 4.1 shows the produced machine code for SUB_MUL_MOD operation. We see that it maps to 16 native GPU instructions where 6 of them, shown in bold face, are fused multiply-adds (FMAs). Instruction semantics can be found in the description of cuobjdump tool.[1]

Fermi architecture. On Fermi cards, 32-bit integer multiplication is no longer a problem, and we can enjoy 31-bit modular arithmetic. Additionally, we can use double-precision arithmetic which is now only two times slower than single-precision. One method to multiply two residues, which takes advantage of floating-point, is described in (MP10). Yet, this approach was essentially borrowed from the CPU code and, thus, is not optimal on Fermi GPUs. To achieve better performance, we can utilize _umulhi intrinsic available

[1] www.dahlsys.com/upload/cuobjdump.pdf

1	shl r8, r0, 0x1; (r0 = x1)	10	**dfma** r8, r4, r8, r6; ($\{r4, r5\}$ = inv, $\{r6, r7\} = 3 \ll 51$)
2	shl r9, r1, 0x1; (r1 = y1)	11	imul.u32.u32 r9, r3, r2;
3	shl r10, r2, 0x1; (r2 = x2)	12	**dfma** r4, r4, r10, r6;
4	shl r11, r3, 0x1; (r3 = y2)	13	**imad**.u32.u32 r0, -r13, r8, r0; (r13 = m)
5	imul.u32.u32.hi r8, r8, r9;	14	**imad**.u32.u32 r4, -r13, r4, r9;
6	imul.u32.u32 r0, r1, r0;	15	**vadd**.ud.u32.u32.min r0, r0, r13, r0;
7	imul.u32.u32.hi r10, r10, r11;	16	**vadd**.ud.u32.u32.min r4, r4, r13, r4;
8	i2f.f64.u32 r8, r8;	17	iadd r0, r0, -r4;
9	i2f.f64.u32 r10, r10;	18	**vadd**.ud.u32.u32.min r0, r0, -r13, r0;

Table 4.2: Disassembly of '$(x_1 y_1 - x_2 y_2) \bmod m$' operation on Fermi architecture.

in CUDA which returns 32 most significant bits of a 64-bit integer product. The procedure MUL_MOD realizing this idea is given by Listing 4.6 and CUDA-specific functions are listed below:

- _umulhi: return 32 most significant bits of the product of 32-bit unsigned integer operands;
- _uint2double_rn: convert an unsigned integer to double-precision floating-point number using "round towards nearest even" rounding mode;
- _double2loint: extract a lower 32-bit word of a double-precision floating-point number and reinterpret it as an integer without conversion.

The algorithm is very similar to its 24-bit counterpart. To compute $a \cdot b \bmod m$ for 31-bit residues a and b, we partition the product $a \cdot b$ in 32- and 30-bit parts (hi and lo), and use the following congruence:

$$a \cdot b = 2^{30} hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = (2^{30} hi - m \cdot l) + lo = a \cdot b - m \cdot l,$$

where $0 \le \lambda < m$. Again denoting $r := a \cdot b - m \cdot l$, we can show that $r \in [-m + \varepsilon; \varepsilon]$ for $0 \le \varepsilon < m$. Hence, it only remains to adjust r by m in case of negative sign. In line 5, we use a "magic number" 3^{51} in order to truncate a double-precision number to the nearest integer after multiplication, see (Hec96). This trick is frequently used to avoid explicit conversion from floating-point to integers. What concerns the operation SUB_MUL_MOD, there is no performance gain in inlining the code (as it used to be for 24-bit modular arithmetic), hence we simply invoke the procedure MUL_MOD twice to compute the result. Note that, in the listing, '$(r1 < 0\ ?\ r1 + m : r1)$' is equivalent to '$\text{umin}(r1, r1 + m)$' which can be mapped to a single Fermi's instruction from Video ISA if we use appropriate assembler intrinsics, see (PTX10). We again examine the quality of the produced machine code for the operation SUB_MUL_MOD using cuobjdump tool, see Table 4.2. This time SUB_MUL_MOD maps to 18 native GPU instructions: in the disassembly FMA and Video instructions are highlighted in bold face.

4.3.2 Modular inverse

We next consider the operation of computing a modular inverse which is used at several places in the matrix-based algorithms (see Section 3.2). Note that, its realization is very similar on both architectures, therefore we next discuss the approach optimized for Fermi cards. To compute a modular inverse, we could apply the Extended Euclidean Algorithm

Algorithm 4.1 Kaliski Montgomery modular inverse algorithm

```
procedure KALISKI_MONTGOMERY_INVERSE(x, m)
    $v \leftarrow x,\ u \leftarrow m,\ s \leftarrow 1,\ r \leftarrow 0,\ k \leftarrow 0;$
    while $v > 0$ do
        if $u \bmod 2 = 0$ then $u \leftarrow u/2,\ s \leftarrow 2s$
        else if $v \bmod 2 = 0$ then $v \leftarrow v/2,\ r \leftarrow 2r$
        else if $u > v$ then $u \leftarrow (u - v)/2,\ r \leftarrow r + s,\ s \leftarrow 2s$
        else $v \leftarrow (v - u)/2,\ s \leftarrow s + r,\ r \leftarrow 2r$ fi
        $k \leftarrow k + 1$
    od
    if $r \geq m$ then $r \leftarrow r - m$ fi
    return $\{r \leftarrow m - r, k\}$                ▷ returns $r = x^{-1}2^k \bmod m$
end procedure
```

(see Section 2.1.3). However, the EEA extensively uses divisions, while the graphics hardware does not have native support for integer division operation. That is why, we have decided to use Montgomery's algorithm instead. As a starting point, we have taken the algorithm from (dDBQ04) designed for FPGA implementation which uses only primitive arithmetic operations. The latter algorithm, in its turn, was based on a binary GCD algorithm, originally proposed by Kaliski (Kal95), which computes a Montgomery modular inverse. In other words, for 31-bit residue x modulo m, the algorithm by Kaliski computes $r := x^{-1}2^k \bmod m$ iteratively where $31 \leq k \leq 62$. To be able to track the origins of our approach, in Algorithm 4.1 we give a pseudocode of Kaliski's algorithm. Here the number k of iterations is bounded by the moduli bit-length. In (dDBQ04), the above algorithm was essentially rewritten to facilitate realization on the hardware platform. We further modify it to yield less number of arithmetic operations. Note that, the original approaches from (Kal95, dDBQ04) use the second iterative phase to remove the unnecessary factor 2^k of r. Instead, here we follow a different approach, proposed (SK00), which is based on the *Montgomery multiplication*. We next discuss its details. Recall that, for two residues a and b modulo m, the Montgomery multiplication computes:

$$\text{MonPro}(a, b, m) := a \cdot b \cdot 2^{-s} \bmod m, \quad \text{with} \quad s = \lceil \log_2 m \rceil.$$

Montgomery's algorithm can be realized in just few lines of code as follows, see (BZ10, Algorithm 2.7):

$$c \leftarrow a \cdot b,\ q \leftarrow \mu \cdot c \bmod \beta,\ \text{MonPro}(a, b, m) \leftarrow (c + q \cdot m)/\beta,$$

where, in our case, $\beta = 2^{32}$ and $\mu = -m^{-1} \bmod \beta$ can be precomputed in advance. Thus, in order to divide out the factor 2^k from r, the trick is perform two Montgomery multiplications with special factors (powers of two). First, whenever $k \geq 32$, we compute MonPro$(r, 1, m)$, such that:

$$r \leftarrow \text{MonPro}(r, 1, m) = x^{-1}2^{k-32} \bmod m, \quad k \leftarrow k - 32.$$

Finally, the second Montgomery multiplication by 2^{32-k}, removes the remaining factor 2^k for $0 < k < 32$:

$$r \leftarrow \text{MonPro}(r, 2^{32-k}, m) = (x^{-1})(2^k)(2^{k-32})(2^{32}) \bmod m = x^{-1} \bmod m.$$

Listing 4.7 31-bit Montgomery modular inverse

```
 1: int
 2: MONTGOMERY_INVERSE(int x, int m, int mu) {                 ▷ computes $x^{-1}$ mod m
 3:     int v = x, u = m, s = 1, r = 0, k = 0;
 4:     while (v! = 0) {                                        ▷ first stage: compute $r = x^{-1}2^{k}$ mod m
 5:         rs = r;
 6:         if (v & 1) {
 7:             uv = v;
 8:             if ((v xor u) < 0) { v = v + u; }
 9:             else { v = v − u; }
10:             if ((v xor uv) < 0) { u = uv, rs = s; }
11:             s = s + r;
12:         }
13:         v = v/2, r = rs ∗ 2, k = k + 1;
14:     }
15:     r = m − r;                                              ▷ second stage: get rid of $2^{k}$ factor
16:     if (r < 0) { r = r + m; }                               ▷ $r = x^{-1}2^{k}$ mod m, $31 \le k \le 62$
17:     if (k >= 32) {        ▷ first Montgomery multiplication: $r = x^{-1}2^{k-m}$ (mod m)
18:         s = r ∗ mu;                                         ▷ $mu = -m^{-1}$ mod $2^{32}$
19:         u = s ∗ m, v = _umulhi(s, m);                       ▷ $(v, u) = s \cdot m$ (63 bits)
20:         u = u + r, r = v + (u < r), k = k − 32;             ▷ $r = ((v, u) + r)/2^{32}$
21:     }
22:     if (k == 0) return r;                                   ▷ second Montgomery multiplication:
23:     s = r ≪ (32 − k), d = s ∗ mu;                           ▷ $mu = -m^{-1}$ mod $2^{32}$
24:     u = d ∗ m, v = _umulhi(d, m);                           ▷ $(v, u) = d \cdot m$ (63 bits)
25:     d = r ≫ k, u = u + s;
26:     r = v + d + (u < s);                                    ▷ $r = ((v, u) + s)/2^{32} + d$
27:     return r;
28: }
```

The modular inverse algorithm implementing these ideas is given in Listing 4.7. In the first stage, lines 3–14, we compute iteratively $x^{-1}2^{k}$ mod m. Then, the additional factor 2^{k} is removed in lines 15–26 by means of two Montgomery multiplications. Comments in the pseudocode should help further understanding the algorithm.

4.3.3 Modular reduction

The purpose of modular reduction is to compute a set of residues of a large integer modulo a set of primes (m_1, $m_2, \ldots,$ m_n). This corresponds to the first stage of a modular algorithm, see Section 2.2. According to some empirical evidence, the cost of this operation might become very high, if not dominating, when the modular algorithm is applied to polynomials with large coefficients but moderate degrees. Therefore, it is highly desirable to perform this operation directly on the GPU, also because the realization is plain easy.

We dedicate one thread to compute a residue modulo some prime m_i using a simple iterative algorithm, known as single-word division, which can be found in many textbooks: see, for instance, (BZ10). Thread workload is perfectly balanced since each thread does the same job. There are only two complications: the first one is how to access the input data in an optimal way since integers being reduced might be too large for allocating them entirely in shared memory; while the second one relates to the fact that integer division, as noted earlier, is highly inefficient on the GPU. To deal with the first problem, we can "stream" the data through the kernel: that is, partition and process the data in chunks of size just enough to fit in shared memory. As we only need a single pass through the data, no further difficulties can occur. Moreover, we can use a "double-buffering" technique to preload the next chunk in registers while the computations are performed on the current chunk residing in shared memory. For the second problem, luckily there are methods to avoid integer division using some precomputation. We have adopted the algorithm given in (MG11), which also seems to be the method of choice implemented in Gmp library.[1] The algorithm is based on precomputing a reciprocal:

$$v = \lfloor (2^{64} - 1)/(2m) \rfloor - 2^{32},$$

where m is a 31-bit modulus, and $2^{31} \le v < 2^{32}$. Then, a single-word division of a large integer X (consisting of n machine words) by m can be carried out as follows:

```
procedure DIV_BY_LIMB(X: array, n, m, v)
    $r \leftarrow 0,\ d \leftarrow m \cdot 2$
    for $i = n - 1$ downto 0 do                         ▷ divide a large integer X by d
        $(u_1, u_0) \leftarrow (r, X[i])$                ▷ load the next limb of X
        $(q_1, q_0) \leftarrow v \cdot u_1 + (u_1, u_0)$
        $q_1 \leftarrow (q_1 + 1) \bmod 2^{32}$
        $r \leftarrow (u_0 - q_1 \cdot d) \bmod 2^{32}$
        if $r > q_0$ then $r \leftarrow (r + d) \bmod 2^{32}$ fi
        if $r \ge d$ then $r \leftarrow r - d$ fi
    od
    if $r \ge m$ then $r \leftarrow r - m$ fi
    return r                                             ▷ return a residue X mod m
end procedure
```

A similar algorithm for 24-bit modular arithmetic can be derived by analogy.

4.3.4 Mixed radix conversion

Lastly, we turn to the realization of the MRC algorithm. To recapitulate, for a set of residues $(x_1, x_2, \ldots, x_n)$ and the corresponding relatively prime moduli $(m_1, m_2, \ldots, m_n)$, the MRC algorithm searches for a large integer X in the mixed-radix form:

$$X = \gamma_1 M_1 + \gamma_2 M_2 + \cdots + \gamma_n M_n,$$

where $M_1 = 1$, $M_i = m_1 m_2 \ldots m_{i-1}$ $(i = 2, \ldots, n)$. A key property of this algorithm is that the MR digits $\{\gamma_i\}$ can be computed *without* resorting to large integer arithmetic. This fact has a decisive effect for selecting this algorithm for the realization on the GPU. As mentioned before, the MR digits can be computed in $O_P(n, n)$ parallel time. Indeed, by looking at formulas (2.4) in Section 2.2.2, we see that this algorithm fits very well to our model of computation. A simple parallel algorithm computing the MR digits is shown in Figure 4.8 (left). Here, in step i $(i = 1, \ldots, n - 1)$, we use a previously computed digit γ_i to update all the remaining digits γ_{i+1} through γ_n. Beside the assigned digit γ_j, in step i, each thread also evaluates $M_{i+1} = M_i m_i \bmod m_j$. To further simplify the computations, we can sort the moduli set in *increasing order*, such that:

$$m_1 < m_2 < \cdots < m_n.$$

[1] http://gmplib.org

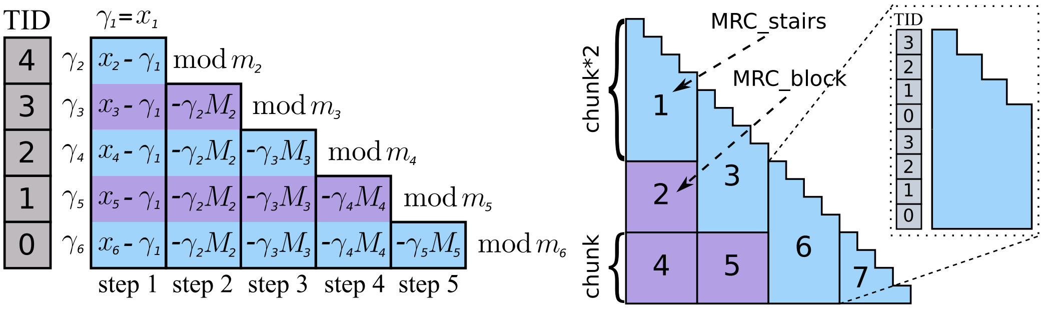

Figure 4.8: Basic MRC algorithm working with 5 threads indexed by **TID** (*left*). Improved MRC algorithm running with m_chunk := 4 threads; the numbers show the order in which the subalgorithms are applied (*right*).

Then, the expressions of the form $\gamma_j M_j c_i \bmod m_i$ for $j < i$, with c_i's are defined in (2.4), can be evaluated without preceding modular reduction of γ_j since $\gamma_j < m_i$. The same argument applies to updating M_i's.

Sadly, this solution does not work when the "capacities" of a CUDA thread block are exceeded because threads need to work cooperatively. As a consequence, we cannot process more than 1024 31-bit residues in the edge case (limited by the maximum block size on Fermi GPUs). To deal with this limitation, we could take advantage of instruction-level parallelism by letting each thread process several residues at a time, as we did it in Section 4.2.3 for the interpolation algorithm. However, this solution would show very bad occupancy due to the fact that, now, the number of working threads *decreases* in each iteration. To figure out an optimal solution, we observe that, in most extreme practical cases, the involved integer quantities do not exceed 90–150 thousand bits in length, which corresponds to 3–5 thousand 31-bit moduli. Hence, we propose to split the inputs in *chunks* (of size m_chunk to be defined later) and compute the MR digits in a loop by *one* thread block. This solution is a good compromise between a more general approach based on block-level parallelism (where the computation is distributed over several thread blocks) since we save on global memory transfers, and a simple parallel algorithm whose drawbacks are outlined above.

Figure 4.8 (right) sketches the idea of our approach whose geometric interpretation is to "cover a triangle" using two shapes (subroutines): MRC_stairs and MRC_block. The algorithm requires m_chunk threads per block to be run on the GPU; the pseudocode is given by Algorithm 4.2. The first subroutine (MRC_stairs) has essentially the same outline as the naive approach shown in Figure 4.8 (left). The main difference is that, currently, we process *twice* as many digits per thread which also explains the shape of MRC_stairs in the figure. By assigning threads as shown in Figure 4.8 (right), we ensure that all threads are *occupied* in each step (except the very last one). The purpose of the procedure MRC_block is to (optionally) preload and update a set of m_chunk MR digits using the digits computed in the preceding MRC_stairs call(s).

Since the number of processed chunks needed for MRC_block calls increases in the course of the algorithm, the main challenge here is where to store the constantly growing amounts of data. Indeed, if we simply keep on allocating the data in registers/shared memory space, eventually all multiprocessor resources will be exhausted and the kernel launch will abort with the failure. On the other hand, we do not want to completely give up

Algorithm 4.2 MRC block algorithm (GPU part)

```
procedure MRC_BLOCK_ALG( chunk[0 . . . n_chunks − 1] )
    MRC_load(chunk[0], chunk[1])                        ▷ load the first two chunks
    for i = 0 to n_chunks − 1 do
        MRC_stairs(chunk[i], chunk[i + 1])
        if (i < n_chunks − 1) then
            MRC_load(chunk[i + 1])                      ▷ preload the next chunk
        fi
        for j = 0 to i − 1 do             ▷ update a new chunk ‘i + 1’ using the previous ones
            MRC_block(chunk[i + 1], chunk[j])
        od
    od
    MRC_stair_last(chunk[i + 1])
end procedure
```

on the advantages of using fast register/shared memory while, in many cases, only a few chunks of data would be needed. We solve this problem by *parameterizing* the kernel with the number of chunks while leaving the parameter m_chunk flexible. When the number of chunks is small, all data is stored in register space and shared memory. By reaching a certain threshold, the data is to be placed in GPU's local memory.[1] The selection of a concrete kernel specialization depends on the actual number of moduli, and is based on heuristics favoring small thread blocks to large ones by adjusting the parameter m_chunk.

In summary, with our “chunk” approach we are no longer constrained by the hardware limitations on the size of a thread block, and can potentially process any number of residues. Besides, this approach provides us a greater control over the occupancy since, compared to the naive algorithm, the work is now distributed more evenly between threads and the chunk size can be chosen appropriately in each particular case.

4.4 Resultants of bivariate polynomials

This section covers the main realization details of the modular resultant algorithm on the GPU. We begin with a high-level structure of the algorithm, and clear up the question how to deal with non-strongly regular Sylvester's matrices raised in Section 3.2.1. Then, we go through the realization of each GPU kernel separately. For the original works on the bivariate resultant computation, we refer to (Eme10c, Eme10b), see also (Eme10a).

4.4.1 High-level structure

At the highest level, our approach is based on Collins' modular algorithm as given in Section 2.4.3. We recall steps of this algorithm applied to bivariate polynomials $f, g \in \mathbb{Z}[x, y]$:

(a) apply modular homomorphism reducing the coefficients of f and g modulo sufficiently many primes: $\mathbb{Z}[x, y] \to \mathbb{Z}_m[x, y]$;

(b) for each modular image, choose a set of points $\alpha_m^{(i)} \in \mathbb{Z}_m$ and evaluate the polynomials at $x = \alpha_m^{(i)}$ (evaluation homomorphism): $\mathbb{Z}_m[x, y] \to \mathbb{Z}_m[x, y]/(x - \alpha_m^{(i)})$;

[1] Observe that, on Fermi all accesses to local memory are cached.

(c) compute a set of univariate resultants in $\mathbb{Z}_m[x]$ using the matrix-based approach (Section 3.2.1): $\text{res}_y(f,g)|_{\alpha_m^{(i)}} : \mathbb{Z}_m[x,y]/(x-\alpha_m^{(i)}) \to \mathbb{Z}_m[x]/(x-\alpha_m^{(i)})$;

(d) interpolate resultant polynomial for each prime m using the matrix-based approach (Section 3.2.2): $\mathbb{Z}_m[x]/(x-\alpha_m^{(i)}) \to \mathbb{Z}_m[x]$;

(e) lift the resultant coefficients by means of Chinese remaindering (Section 4.3.4): $\mathbb{Z}_m[x] \to \mathbb{Z}[x]$.

Steps (a)–(d) and partly (e) are implemented on the graphics processor, thereby minimizing the amount of work on the host machine. In essence, what remains to be done on the CPU is to convert the resultant coefficients from the mixed-radix representation to positional number system.

The number of primes and evaluation points needed by the algorithm can be estimated using Hadamard's bound on resultant's height and degree, see Section 2.4.3. We also refer to (Mon05) where some practical methods are outlined: for instance, one can compute the bounds over columns and rows of Sylvester's matrix, and then pick up a minimal one. As noted in Section 2.4.3, another possibility to obtain good bounds is to use the theory of sparse resultants as it can be shown that Sylvester's resultant of two polynomials is a special case of a more general mixed sparse resultant. In vague terms, the idea behind this method is to consider the geometry of the Newton polytopes of original polynomials to obtain sharp estimates. Yet, this topic is largely beyond the scope of this work. We refer to (CLO98, Chapter 7) for some basic facts and to (Som04) for the height of a mixed sparse resultant.

To recover the resultant from its homomorphic images, we have to deal with "unlucky homomorphisms". Although, Theorem 2.4.6 in Section 2.4.3 provides some mild requirements on selecting the homomorphic images, in the realization we use only those homomorphisms that do not cause the leading coefficients of *either* polynomial to vanish because this simplifies the recovery process. Dealing with "bad" primes is easy: we can discard them right away during the initial modular reduction of polynomials. To account for "bad" evaluation points, we propose to run the algorithm with an *excess* amount of points (typically 1–2% more than required). Thus, if the algorithm breaks down for some $\alpha_m^{(i)} \in \mathbb{Z}_m$, we simply *ignore* the result and take another evaluation point. This situation is easily detectable since "bad" evaluation points produce a *zero* denominator l_{res} (see Algorithm 3.1 in Section 3.2.1). In a very unlucky case, when we cannot reconstruct the resultant due to the lack of points, we restart the algorithm to compute the extra information. Note that, the algorithm may also fail due to non-strong regularity of Sylvester's matrix. This situation is considered next.

4.4.2 Dealing with non-strong regularity

Here, we outline some practical ways how to prevent the algorithm's failure. First, remark that, non-strong regularity indicates a presence of some non-trivial relation between polynomial coefficients which occurs quite rarely in practice. Moreover, the majority of such situations can be handled in exact same way as "bad" evaluation points.[1] Indeed, if the

[1] In fact, *both* non-strong regularity of Sylvester's matrix and "bad" evaluation points yield a zero denominator l_{res} in Algorithm 3.1, and therefore are not distinguishable from the algorithm's perspective.

computation fails for some point $\alpha_m^{(i)}$, there is, in general, a large set of other evaluation points to select from. Consider the following example:

$$
\begin{aligned}
f &= y^8 + y^6 - 3y^4 - 3y^3 + (x+6)y^2 + 2y - 5x,\\
g &= (2x^3 - 13)y^6 + 5y^4 - 4y^2 - 9y + 10x + 1.
\end{aligned}
$$

Then, for $\alpha = \{0, \ldots, 100000\}$, Sylvester's matrix of $f(\alpha, y)$ and $g(\alpha, y)$ is non-strongly regular only for a single point $\alpha = 2$. However, the problem occurs if, for polynomials $f, g \in \mathbb{Z}[x, y]$, some minors of Sylvester's matrix $S^{(y)}$ (defined in Section 2.4.1) vanish *identically*. According to large empirical evidence, it mostly happens when one of the polynomials has a *zero* trailing coefficient. This can be exemplified as follows. Let

$$
f = y^8 + (4x^2 - 12)y^6 + (12x^3 + 2)y^4 + (20x^4 - 28x^2 + 12)y^2 - 18x^4 - 3,
$$

and $g = f'_y$ (the first derivative of f w.r.t. the variable y). Hence, we have: $g_0 \equiv 0$ and every second principal minor of $S^{(y)}$ (skipping the first seven) vanishes identically. One simple way to fix this is to consider the resultant of *swapped* polynomials, i.e., take $\text{res}_y(f, g) = -\text{res}_y(g, f)$. Indeed, in the example above, the matrix $S^{(y)}$ for g and f (swapped) is strongly-regular. Although, sometimes this does not work. A more general approach is to divide out a factor y of g (which causes the algorithm to break down), compute the "reduced" resultant, and then compensate for the factor y. In other words, suppose $f_0 \neq 0$ and $g = h \cdot y^k$ $(k > 0)$, then it holds that:[1]

$$
\text{res}_y(f, g) = \text{res}_y(f, h) \cdot f_0^k.
$$

Our long-term practical experiences show that the above two cases cover 99% of all "bad" situations. However, in "extremely pathological" cases, when neither of the above works, we can further exploit the properties of the resultants in the attempt to "randomize" Sylvester's matrix. For example, consider the following:

$$
\begin{aligned}
&\text{res}_y(f, g) = \text{res}_y(h, g) \ \text{ if } \ h = f + g \cdot q, \ \text{ and } \ \deg_y h = \deg_y f,\\
&\text{res}_y(f, g)^2 = \text{res}_y(f^2, g) = \text{res}_y(f, g^2).
\end{aligned}
$$

In the latter case, we also need to compute a polynomial square root to extract the actual resultant which can be achieved by a linear-time algorithm.

4.4.3 Realization

Schematic view of the algorithm running on the GPU is depicted in Figure 4.9. The computations start on the host machine, where we search and remove "bad" primes from the moduli set, that is, the primes dividing the leading coefficients of input polynomials. Then, the control is transferred to the GPU executing six CUDA kernels corresponding to the steps of Collins' modular algorithm. The kernels are listed with the respective grid and block configurations in the figure; we next proceed with an in-depth discussion of each of them.

[1] Here we assume that the polynomials are coprime in $\mathbb{Z}[x, y]$ which implies that $f_0 \neq 0$.

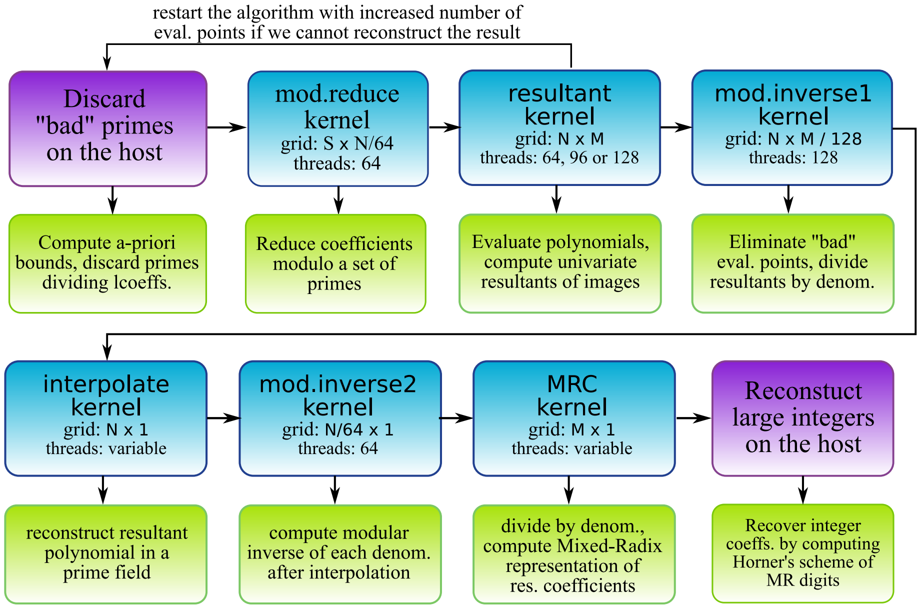
restart the algorithm with increased number of eval. points if we cannot reconstruct the result
Discard "bad" primes on the host
mod.reduce kernel grid: S x N/64 threads: 64
resultant kernel grid: N x M threads: 64, 96 or 128
mod.inverse1 kernel grid: N x M / 128 threads: 128
Compute a-priori bounds, discard primes dividing lcoeffs.
Reduce coefficients modulo a set of primes
Evaluate polynomials, compute univariate resultants of images
Eliminate "bad" eval. points, divide resultants by denom.
interpolate kernel grid: N x 1 threads: variable
mod.inverse2 kernel grid: N/64 x 1 threads: 64
MRC kernel grid: M x 1 threads: variable
Reconstuct large integers on the host
reconstruct resultant polynomial in a prime field
compute modular inverse of each denom. after interpolation
divide by denom., compute Mixed-Radix representation of res. coefficients
Recover integer coeffs. by computing Horner's scheme of MR digits

Figure 4.9: Structure of the modular algorithm running on the GPU. **Abbrev.:** S: total number of scalar coefficients of both polynomials; N: number of moduli; M: number of evaluation points.

As a general remark, in our implementation we have used a number of standard optimization techniques: some of them were already mentioned in Section 4.1.3. These techniques, for example, include constant propagation via templates, loop unrolling, exploiting warp-level parallelism, declaring frequently used local variables with `volatile` keyword, giving the preference to small thread blocks instead of the large ones, etc. Besides, we have also specified *launch bounds* for each GPU kernel. The latter technique alone gave us about 30% additional speed-up. Lastly, we keep the moduli set and corresponding reciprocals (`inv`) needed by the modular arithmetic (see Section 4.3) in *constant* memory space. The reason for this is because, each GPU kernel in Figure 4.9 (except the MRC kernel) uses *one* modulus per thread block for all computations. As a result, all threads of a block read from the same address which generates a single memory request which goes through constant memory cache (see also Section 4.1.2). Besides, a direct access from constant memory has a positive effect on reducing register usage.

Modular reduce kernel. The first 'mod.reduce' kernel performs modular reduction of polynomial coefficients. The grid configuration is chosen to be $S \times N/64$, where S is a total number of scalar coefficients of both polynomials and N is the size of moduli set. In other words, we partition the moduli set in chunks of size 64 primes each and assign one CUDA block to compute 64 residues of some polynomial coefficient. Further details on its realization can be found in Section 4.3.3.

Resultant kernel. Resultant kernel constitutes the core of the algorithm. We designate one thread block to compute a univariate resultant modulo a prime m_i for some evaluation point $\alpha_j \in \mathbb{Z}_{m_i}$. Accordingly, a grid configuration for this kernel is chosen to be $N \times M$, see Figure 4.9. We provide four kernel specializations for 32×2, 64, 96 and 128 threads

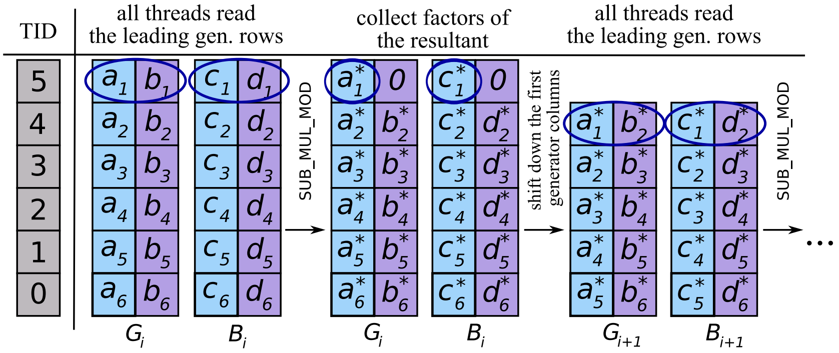
TID
all threads read the leading gen. rows
collect factors of the resultant
all threads read the leading gen. rows
5
4
3
2
1
0
a_1 b_1 c_1 d_1
a_2 b_2 c_2 d_2
a_3 b_3 c_3 d_3
a_4 b_4 c_4 d_4
a_5 b_5 c_5 d_5
a_6 b_6 c_6 d_6
SUB_MUL_MOD
a_1^* 0 c_1^* 0
a_2^* b_2^* c_2^* d_2^*
a_3^* b_3^* c_3^* d_3^*
a_4^* b_4^* c_4^* d_4^*
a_5^* b_5^* c_5^* d_5^*
a_6^* b_6^* c_6^* d_6^*
shift down the first generator columns
a_1^* b_2^* c_1^* d_2^*
a_2^* b_3^* c_2^* d_3^*
a_3^* b_4^* c_3^* d_4^*
a_4^* b_5^* c_4^* d_5^*
a_5^* b_6^* c_5^* d_6^*
SUB_MUL_MOD
...
G_i
B_i
G_i
B_i
G_{i+1}
B_{i+1}

Figure 4.10: The workflow of the univariate resultant algorithm running with 6 threads indexed by **TID**

per per block. The number of threads depends on the degree of input polynomials.[1] A kernel with P threads handles polynomials with the degree range $[P/2, \ldots, P-1]$.

First, the input polynomials are evaluated at $x = \alpha_j$. We use simple Horner's scheme where each thread iteratively computes one coefficient of $f(\alpha_j, y)$ and $g(\alpha_j, y)$. Next, we run the resultant algorithm (Algorithm 3.1) derived in Section 3.2.1. The algorithm maps quite straightforwardly to the GPU: the outer loop is split up in "lite" and "full" iterations while the inner loop is vectorized. In other words, we associate one thread with four data elements: $(\mathbf{a}_i, \mathbf{b}_i)$ and $(\mathbf{c}_i, \mathbf{d}_i)$ which correspond to one row of each of the matrices $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$. In each iteration, the current top generator rows are shared between all threads. Then, each thread applies the rotation formulas from Section 3.1.3, implemented as a SUB_MUL_MOD operation (see Section 4.3), to its data elements. At the end, the first columns $\mathbf{a}$ and $\mathbf{c}$ are "shifted down" preparing for the next iteration, and resultant factors are saved in shared memory. One step of the algorithm during "full" iterations is depicted in Figure 4.10.

For "lite" iterations, we also unroll the outer loop by the factor of two for higher arithmetic intensity, so that one thread now processes two rows of each of matrices G and B. In this way, we double the maximal degree of polynomials that can be handled, and ensure that all threads are occupied. Besides, at the beginning of "full" iterations, we can guarantee that at least half of threads are busy in the corner case ($d = P/2$). Observe that, the number of working threads decreases with the length of the generators in the outer loop. Therefore, we use load balancing strategy to improve thread occupancy: when at least half of threads enter the idle state, we switch to another subroutine where the computations are structured in such a way that threads only do a half of a job. Finally, once the length of the generators descends below the warp boundary (32), the remaining algorithm steps are performed *without* synchronization points since warp, as a minimal scheduling entity, does not need to be synchronized. Finally, the product of all resultant factors (as well as the product of the denominators $\mathrm{l}_{\mathrm{res}}$) is computed using the "warp-sized" parallel reduction algorithm discussed in Section 4.2.2.

Modular inverse 1 and stream compaction kernel. The purpose of this kernel is to divide the resultants computed in the previous step by respective denominators $\mathrm{l}_{\mathrm{res}}$ (see Algo-

[1] The first kernel with 32×2 threads computes two resultants at a time.

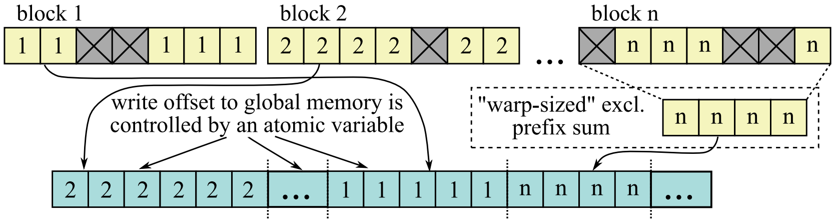
block 1
block 2
block n
1 1 1 1 1
2 2 2 2 2 2
n n n n
...
write offset to global memory is controlled by an atomic variable
"warp-sized" excl. prefix sum
n n n n
2 2 2 2 2 2 ...
1 1 1 1 1
n n n n ...

Figure 4.11: Stream compaction across several thread blocks, 'x' marks elements being eliminated. Results for each block are written back to global memory in unspecified order.

rithm 3.1), and eliminate "bad" evaluation points.[1] For each modulus m_i, we partition the set of resultants in 128-element chunks and assign one chunk to a block having 128 threads. The kernel is launched on a grid of size $N \times M/128$, see 'mod.inverse1' in Figure 4.9. Division is performed using Montgomery modular inverse algorithm, see Listing 4.7.

"Bad" evaluation points are discarded using a *stream compaction algorithm*. Our realization is based on the following observations. **1.** The number of evaluation points M can be quite large (generally on the order of 1–3 thousand), hence it is inefficient/infeasible to process all of them by one CUDA block. **2.** Running hierarchical stream compaction in global memory (several kernel calls) does not pay off because "bad" evaluation points occur quite rarely on the average.[2] **3.** The actual order of evaluation points does not matter for interpolation. Keeping that in mind, we have found the following solution optimal: Each block runs the stream compaction algorithm on its 128-element chunk using warp-sized prefix sum in shared memory (see Section 4.2.2). This algorithm computes exclusive all-prefix-sums of a sequence of 0's and 1's where 0's correspond to elements being eliminated. The resulting sequence is then used as a "relocation map" to move all valid elements to new locations in shared memory, see (BOA09). Note also that, on Fermi we can use ballot voting primitive and popc intrinsic to compute the prefix sum of 0's and 1's more efficiently, see also (Eme10a).

Finally, the "scanned" sequence of evaluation points is written back to global memory. The current writing position is controlled by a global variable which gets updated (atomically) each time a block outputs its results to global memory, see Figure 4.11.

Interpolation kernel. Interpolation kernel implements Algorithm 3.2 from Section 3.2.2. Its realization has already been discussed in Section 4.2.3 as part of our case study. Therefore, we only highlight the main features of the algorithm. Here, one CUDA block is dedicated to interpolating a polynomial modulo some prime m_i. Similar to the resultant kernel, the inner loop of the algorithm is vectorized. Since the interpolation algorithm is simpler than that of resultants, one thread is now assigned to process *two* or *four* rows (depending on the number of evaluation points M) of the generator matrix $G = (\mathbf{a}, \mathbf{b})$. In each iteration we update M relevant entries of G in a "sliding window" fashion.

The number of threads per block is chosen to depend on the number of evaluation

[1] Those are the points for which the algorithm returns a zero denominator l_{res} either due to "unlucky" homomorphism or because of non-strongly regular Sylvester's matrix, see Section 4.4.2.

[2] According to extensive experiments, every 3–4 evaluation points out of 10–50 thousand are typically "bad" for random polynomials.

points M. As a consequence, the maximal degree of the resultant polynomial is limited to 2048 on Tesla and 4096 on Fermi GPUs which corresponds to the maximal number of threads per block on the corresponding GPUs. According to our experiences, these limitations satisfy the demands of most practical applications. For reasons of efficiency, we have also parameterized the kernel by the “data parity”: in other words, by $M \bmod 2$ or $M \bmod 4$ depending on the loop unrolling factor, instead of the data size itself. By parameterizing the kernel in such a way, we can substantially reduce branching and register usage inside the algorithm which has a positive effect on performance, see Section 4.2.3.

Modular inverse 2 kernel. In this kernel, ‘mod.inverse2’ in Figure 4.9, we precompute the modular inverses for subsequent division of polynomial coefficients by respective denominators l_{int} computed by the interpolation algorithm. The realization is rather straightforward and closely resembles the ‘mod.inverse1’ kernel considered above (with the exception that the stream compaction is no longer required). We, therefore, skip further discussions of this kernel.

MRC kernel. The remaining MRC kernel reconstructs the integer coefficients of a resultant polynomial in mixed-radix representation. Before computing the MR digits, we also multiply each resultant coefficient by the modular inverse of some denominator l_{int} computed by preceding ‘mod.inverse2’ kernel. The MRC kernel is launched on a grid of size $M \times 1$ such that all resultant coefficients are processed in parallel. For the algorithm description, we refer to Section 4.3.4. Finally, the computed MR digits are uploaded back to the host machine where the actual large integer coefficients are reconstructed by evaluating the Horner’s scheme.

In the next section, we examine the performance of the GPU algorithm by comparing it with a CPU-based approach.

4.4.4 Experiments

To run the experiments, we have used a desktop with 2.8GHz 8-Core Intel Xeon W3530 (8 MB L2 cache) CPU and GeForce GTX580 graphics card. To recover large integer coefficients from the mixed-radix representation, we have employed GMP 5.0.1 library.[1] As a contestant we have chosen a host-based resultant algorithm from 64-bit compilation of Maple 14.[2] The reason why we have chosen this algorithm is because Maple’s implementation of Collins’ approach is known to be one of the most efficient among CPU-based algorithms. Furthermore, we are not aware of any matured GPU-based resultant algorithm available to date. In should be noted that for polynomials with integer coefficients Maple has a *built-in* implementation of the resultant algorithm. In other words, this algorithm is not written in a high-level Maple’s language but is available in a precompiled binary form. To be precise, Maple implements *probabilistic* approach due to (Mon05) which, in many cases, can significantly reduce the computation times since it does not rely on the theoretical upper bounds for the number of homomorphic images. Finally, remark that this version of Maple can take advantage of multiple CPU cores available on the host

[1] `http://gmplib.org`

[2] kernelopts(wordsize) returns 64 which verifies 64-bit Maple.

#	Configuration	degree	GPU	Maple	# blocks
1-2.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{7}, \mathbf{20}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{11}, \mathbf{16}\}$ (sparse) bits : **32** / **300**	548 / 548	0.016 / 0.174	1.16 / 11.9	353×628
3-4.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{40}, \mathbf{19}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{39}, \mathbf{17}\}$ (dense) bits : **32** / **100**	1664 / 1664	0.066 / 0.189	6.9 / 16.6	122×1519
5-6.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{12}, \mathbf{62}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{10}, \mathbf{40}\}$ bits : **24** (sparse / dense)	1107 / 2777	0.146 / 0.498	11.9 / 45.7	96×2902
7-8.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{40}, \mathbf{31}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{30}, \mathbf{20}\}$ bits : **100** (sparse / dense)	1689 / 2432	0.294 / 0.486	28.3 / 57.5	174×2491
9-10.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{10}, \mathbf{80}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{10}, \mathbf{90}\}$ bits : **32** (sparse / dense)	1736 / 3210	0.854 / 1.94	78 / 189	187×1784
11-12.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{20}, \mathbf{60}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{23}, \mathbf{75}\}$ (sparse) bits : **32** / **200**	2981 / 2981	1.11 / 7.0	114 / 663	888×3032
13-14.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{42}, \mathbf{31}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{33}, \mathbf{23}\}$ (dense) bits : **24** / **400**	2027 / 2027	0.096 / 1.93	12.7 / 201	705×2109
15-16.	$\mathrm{deg}_{x/y}(f) : \{\mathbf{20}, \mathbf{41}\}$ $\mathrm{deg}_{x/y}(g) : \{\mathbf{11}, \mathbf{29}\}$ bits : **900** (sparse / dense)	1011 / 2910	3.9 / 14.3	247 / ? (> 15 min)	2045×2854

Table 4.3: Computing the resultant of f and g w.r.t. the variable y (in seconds). *1st col.:* instance number; *2nd col.:* $\mathrm{deg}_{x/y}$: degree in variables x and y, resp.; bits: coefficient bitlength; sparse/dense: varying density of polynomials; *3rd col.:* resultant degree; *last col.:* grid configuration of the resultant kernel $(\mathrm{N} \times \mathrm{M})$

machine which can be verified using 'kernelopts(multithreaded)' command. In our case, the number of CPU cores in use is set to 8 by default.

For the benchmarks, we have varied several parameters of input polynomials including x- and y-degree, coefficient bitlength and density (the number of non-zero coefficients). The results are listed in Table 4.3. All timings are given in seconds. In the table, each configuration (2nd column) corresponds to the *pair* of experiments where the varying parameter, which is either the bitlength or polynomial density, is written with a slash '/'. Accordingly, the running times for each experiment separately are also written with a slash. For the GPU timing (4th column), we have accounted for all stages of the algorithm including the modular reduction and recovering the integer coefficients on the host machine. The last column '# blocks' specifies a grid configuration used by the resultant kernel, cf. Section 4.4.3. Note that, the GPU algorithm has been configured to use 31-bit modular arithmetic optimized for Fermi processors (Section 4.3.1). Benchmarks using 24-bit modular arithmetic can be found in (Eme10b).

From Table 4.3, we see that the parallel algorithm achieves about 50–100x speed-up

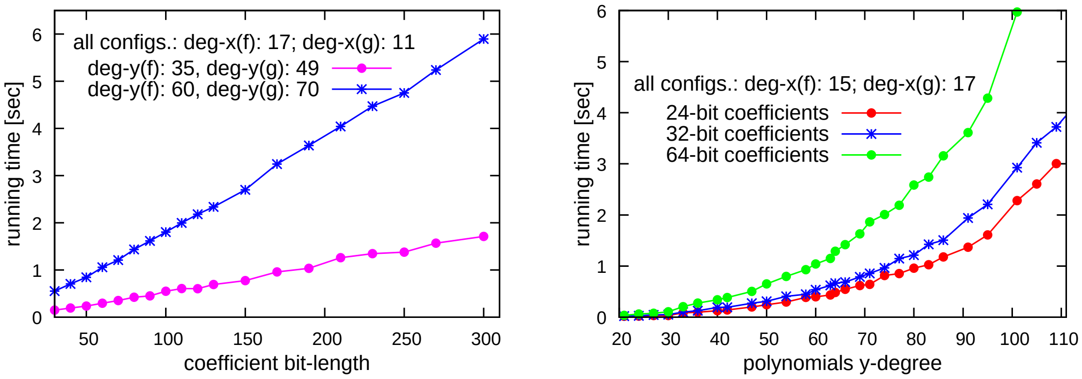

Figure 4.12: Running time versus coefficient bitlength (*left*); and polynomial y-degree (*right*)

over the host-based approach. Looking at the timings more carefully, one can see that Maple's algorithm performs better for sparse polynomials which implicitly indicates that it uses the PRS algorithm (Section 2.4.2) in its core while our matrix-based approach is insensitive to the density parameter. On the other hand, our algorithm is faster for polynomials of high x-degree. This is expected because, with the x-degree, the number of thread blocks increases leading to better hardware utilization, while the size of Sylvester's matrix stays the same (since we compute the resultant w.r.t. the variable y). On the contrary, increasing the y-degree penalizes the performance as it causes the *number of threads* per block to increase. Similarly, for larger coefficient bitlengths, the attained performance is typically better, again, due to the increased degree of parallelism. Two graphs in Figure 4.12 examine the running time as a function of the coefficients bitlength (left) and polynomials y-degree (right) while keeping other parameters fixed. Increasing the bitlength only affects the number of primes used by the algorithm while the number of evaluation points remains the same. Accordingly, we have a clear *linear* dependency of the running time on the coefficient bitlength. A different situation is observed when we change the y-degree of polynomials. This has an impact on both the number of primes and evaluation points, therefore performance scales *quadratically* in the right diagram. Also, notice a jump of the running time between the y-degrees 90 and 100: this is caused by the fact that algorithm switches to another instantiation of the resultant kernel (the one with 128 threads per block) when all thread resources are exhausted, see Section 4.4.3.

A histogram in Figure 4.13 depicts a relative contribution of different stages of the algorithm to the overall time. The numbers along the x-axis correspond to the configurations in Table 4.3. Apparently, the time for the resultant kernel, 'resultant' in the figure, is dominating since this kernel has the largest grid size among others (see Figure 4.9). The second largest time is either for reconstructing the large integers on the host 'MR recover' (in case of large coefficient bitlength) or polynomial interpolation (for high polynomial degrees). The fact that 'MR recover' stage can occupy almost the half of the total running time is a bit surprising considering the relative simplicity of the involved computations. Therefore, we think about moving this stage to the graphics processor as well.

As a general remark, we have observed that even for polynomials with moderate degree, our algorithm reaches hardware saturation very fast since the complexity of computing resultants increases rapidly with the input parameters. A clear indication for this

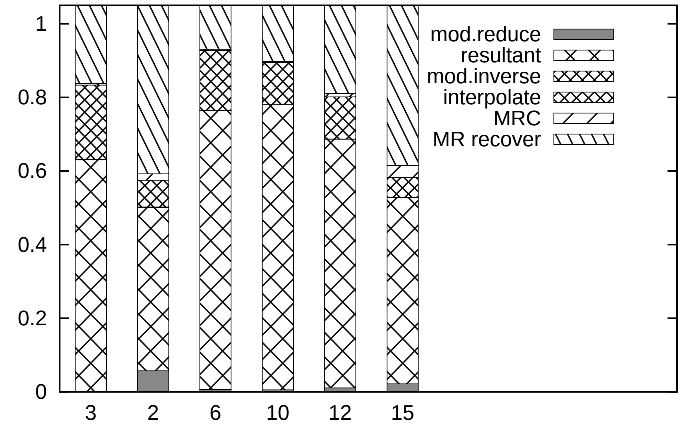

Figure 4.13: Relative contribution of different stages to the overall time for configurations taken from Table 4.3

is the number of thread blocks used by the resultant kernel which is given in the last column of Table 4.3. Besides that, due to high arithmetic intensity of computations, the time for GPU–host memory transfer was *negligibly* small for all instances we have tried.[1] In total, this confirms that our implementation is not memory-bound, and thus has a great scalability potential on future generation GPUs having more physical processors.

Still, there is a number of things we wish to improve about the algorithm. First, as mentioned above, there is a large performance benefit in further reducing the amount of work to be done on the host side, even if this is related to performing some "heavy" computations, such as large integer arithmetic, on the graphics processor. Second, the current implementation limits the degree of polynomials in the main variable[2] to 128, while the degree of the resultant polynomial is limited to 4096. Although, such limits are wide enough for a broad range of applications, this might not be sufficient in a long term perspective. These constraints are due to the fact that, at the current stage of development, our algorithm does not exploit block-level parallelism on the graphics card. The final solution should follow the same outline as a block GCD algorithm considered in Section 4.5.2. Lastly, the graphics hardware evolves very dynamically over the past decades, and new programming features are released every six months. As a result, it is always a challenge to keep the implementation up-to-date with respect to the modern development trends. For example, it appears to be very promising to use *concurrent kernel execution* supported by Fermi processors. In essence, this feature enables several CUDA kernels to be active at the same time on the GPU. Thus, it is possible to further reduce the running times by launching CUDA kernels *asynchronously* while the previous GPU calls are not finished yet and, thereby, achieve a partial *overlap* in computations (provided that, we can arrange the computations in such a way that there is no data dependency between them).

[1] We allocate *page-locked* host memory to speed up GPU–host memory transfers.

[2] The one with respect to which the resultant is computed.

4.5 Univariate GCD computation

Our realization of the modular GCD algorithm was primarily oriented to computing a GCD of univariate polynomials where the emphasis is put on supporting arbitrary large polynomial degrees and very large coefficient bitlength. The main motivation for this was that the typical applications, such as the solution of systems of polynomial equations, are heavily based on univariate GCD computation, while computing a GCD of multivariate polynomials can, in many cases, be avoided using various filter techniques. The secondary reason for this was the "proof of concept", in the sense that, we wished to verify whether block-level parallelism (supported by the GPU) can be integrated in the matrix factorization algorithm (see Section 3.1.2).

We start with a general description of the algorithm. Next, we outline some practical ways to estimate the number of homomorphic images required by the modular approach. Finally, we discuss the GPU part of the algorithm in detail. For the conference paper describing the original work, we refer to (Eme11).

4.5.1 Algorithm design and improved GCD bounds

From a high-level perspective, our algorithm uses the ideas of Brown's modular algorithm introduced in Section 2.3. For univariate polynomials $f, g \in \mathbb{Z}[x]$, this algorithm comprises three steps:

(a) apply modular homomorphism reducing the coefficients of f and g modulo sufficiently many primes: $\mathbb{Z}[x] \to \mathbb{Z}_m[x]$;

(b) compute a set of univariate GCDs in $\mathbb{Z}_m[x]$ using the matrix-based approach (Section 3.2.3): $\gcd(f, g) \bmod m : \mathbb{Z}_m[x] \to \mathbb{Z}_m[x]$;

(c) lift the coefficients of a GCD using Chinese remaindering (Section 4.3.4): $\mathbb{Z}_m[x] \to \mathbb{Z}[x]$.

Similar to the resultant computation, we outsource the steps **a**, **b** and partly **c** to the graphics processor leaving the final conversion of GCD coefficients from the Mixed-radix to conventional representation to the host machine since the latter step involves computations with large integers.

One of the challenges we have been faced with during the initial algorithm's design was estimating the number of primes required to reconstruct a GCD polynomial. Remember that the idea of Brown's algorithm is to use "intuitive bounds" (instead of the worst-case bounds) to compute the number of homomorphic images, provided that the validity of the results can later be verified using *trial division* (see Section 2.3.2). This trick, however, requires processing the homomorphic images incrementally. While, in the parallel setting, we certainly do not want to abandon a great deal of parallelism for the sake of incremental processing. More important, it would not be possible to perform a trial division directly on the GPU (unless we come up with another modular algorithm), and hence the division check might become a major performance bottleneck. Altogether, this prompted us to search for alternative bounds on the height of a GCD. Luckily, in the univariate case, there is a number of methods to obtain good estimates in practice (which, however, do not improve the asymptotic bound). We discuss some of these methods here. An interested reader may also consult a recent survey paper (Abb09).

In what follows, let $f \in \mathbb{Z}[x]$ be a polynomial of degree n and $h \in \mathbb{Z}[x]$ be its divisor of degree $\delta \leq n$. If δ is known, we can apply *degree-aware* bounds which, sometimes, are much better than the usual estimates. Among them, the most interesting one is the *Binomial bound.* The bound is (for $0 \leq i \leq \delta$):

$$|h_i| \leq \hat{h}_i, \quad \text{where} \quad \hat{h}(x) = |\operatorname{lcf}(h)|(x+\rho)^{\delta}, \tag{4.1}$$

and ρ is the upper bound for the magnitude of any complex root of f. Observe that, $\operatorname{lcf}(h)$ is easily obtainable, if we know that h is a GCD. The next one is the refinement of Mignotte's bound (Mig74), saying that:

$$|h|_i \leq \binom{\delta}{i}\mathcal{M}(f) \leq \binom{\delta}{i}|f|_2, \tag{4.2}$$

where $\mathcal{M}(f)$ denotes the Mahler measure of f. The third one is the Knuth-Cohen bound published in (Knu97):

$$|h|_i \leq \binom{\delta-1}{i}|f|_2 + \binom{\delta-1}{i-1}|\operatorname{lcf}(f)|. \tag{4.3}$$

As a rule of thumb, neither of these bounds is favorable in different situations, and hence a combination thereof works best in practice. Also, in (Abb09), the *reversal trick* is proposed which is based on the property that polynomial multiplication and reversal commute. In other words,

$$\text{if} \ \ f = g_1 g_2, \ \ \text{then} \ \ \overline{f} = \overline{g}_1\overline{g}_2,$$

where $\overline{f}$ is a polynomial whose coefficients order is reversed. Meaning that, a bound for the i-th coefficient of $\overline{h}$ is also valid as a bound for the $(\delta - i)$-th coefficient of h and vice a versa. Hence, the idea is to simply compute the two sets of bounds for individual coefficients, and then pick up the minimal ones. This trick, in particular, can greatly improve the accuracy of the Binomial bound (4.1) when the trailing coefficient of a GCD is small. It can also be used in combination with (4.3), while Mignotte's bound (4.2) is invariant under reversal.

4.5.2 Realization: overview

In the light of the previous discussion, we suggest the following outline of the GPU algorithm: start with the number of primes given by the height of the original polynomials (this is enough in the vast majority of situations). Then, once the modular GCDs are computed, apply *degree-aware* bounds and enlarge the moduli set if needed. To deal with "unlucky" homomorphisms, we first scan the set of primes on the host machine eliminating those ones that divide the leading coefficients of either polynomial. Next, once the set of modular GCDs is available, we perform the degree anomaly check (cf. Theorem 2.3.1) using stream compaction algorithm to discard those modular images which have a degree higher than that of the others. Note that, in the real implementation, we also employ a *modular filter* to quickly detect coprime polynomials. The idea is rather simple: at the beginning, we invoke the whole algorithm for a single prime to test if the degree of the resulting GCD image is zero. If so, we conclude that the polynomials are coprime and the algorithm can be terminated prematurely. Otherwise, the computations are repeated with a full set of moduli.

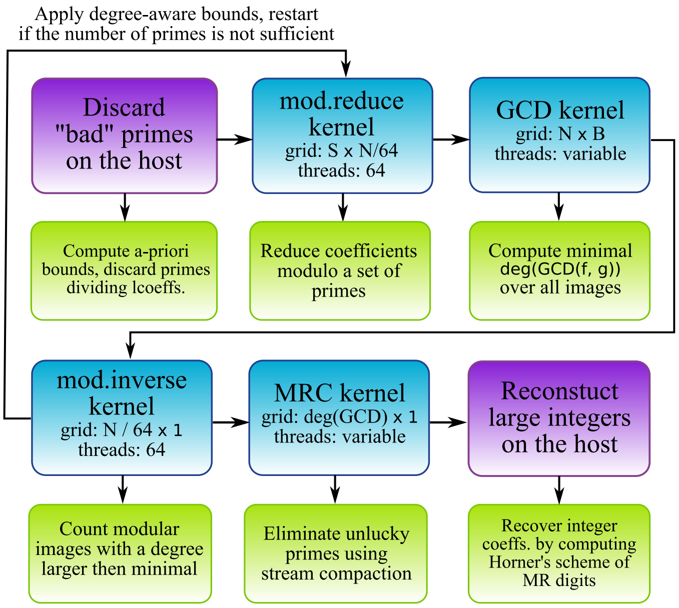
Apply degree-aware bounds, restart
if the number of primes is not sufficient
Discard "bad" primes on the host
mod.reduce kernel
grid: S x N/64
threads: 64
GCD kernel
grid: N x B
threads: variable
Compute a-priori bounds, discard primes dividing lcoeffs.
Reduce coefficients modulo a set of primes
Compute minimal deg(GCD(f, g)) over all images
mod.inverse kernel
grid: N / 64 x 1
threads: 64
MRC kernel
grid: deg(GCD) x 1
threads: variable
Reconstuct large integers on the host
Count modular images with a degree larger then minimal
Eliminate unlucky primes using stream compaction
Recover integer coeffs. by computing Horner's scheme of MR digits

Figure 4.14: High-level view of a GCD algorithm. **Abbrev.:** S: total number of scalar coefficients of both polynomials; N: number of moduli; B: number of thread blocks per modular GCD, see Section 4.5.4.

A block-diagram of the algorithm is shown in Figure 4.14. Each GPU kernel in the diagram is specified with corresponding grid and block configurations. The first kernel, 'mod.reduce', performs modular reduction of polynomial coefficients using the algorithm in Section 4.3.3. The set of primes is partitioned in chunks of 64 primes each, so that a single CUDA block is assigned to computing 64 residues of one polynomial coefficient. Here, threads do not need to cooperate with each other, and therefore, in terms of performance, it is better to keep the block size small. The second GCD kernel, computing the image GCDs, constitutes the core of the algorithm. We consider its realization separately in the next section. Additionally, in this kernel we accumulate a minimal GCD degree over all modular images using CUDA atomic primitives in preparation for the degree check.

The next 'mod.inverse' kernel is responsible for calculating the modular inverses of GCD's leading coefficients for each modular image (see line 27 in Algorithm 3.3). The implementation uses Montgomery modular inverse algorithm as given in Section 4.3.2. The input is processed in chunks of size 64, thus a grid configuration is set to $N/64 \times 1$ with N being the number of moduli. Besides, in 'mod.inverse' kernel we count the number of "unlucky" primes by comparing the degree of GCD images with the minimal degree computed in the previous step. If the degree-aware bounds (calculated on the CPU) suggest that the number of primes (not counting "unlucky" ones) still suffices to recover the result, we proceed to the next step. Otherwise, the algorithm is restarted with enlarged moduli set.

The remaining 'MRC kernel', discussed in Section 4.3.4, computes the mixed-radix representation of GCD coefficients. In this kernel, we also eliminate the previously detected "unlucky" primes using a stream compaction algorithm. Its realization is very similar that of 'mod.inverse 1' kernel, given as part of the resultant algorithm (Section 4.4.3). Thus, we skip further remarks here. In the last step, the mixed-radix digits are evalu-

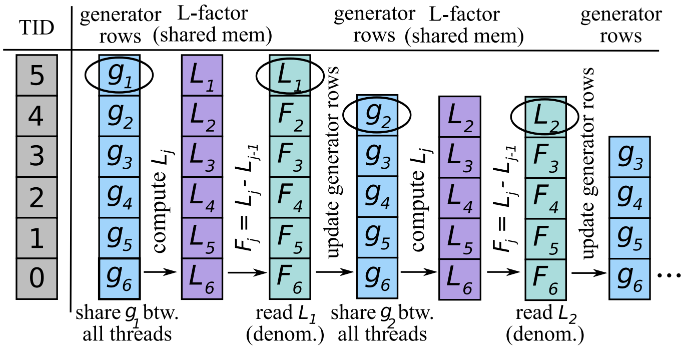
TID
generator rows
L-factor (shared mem)
generator rows
L-factor (shared mem)
generator rows
5
4
3
2
1
0
g_1 g_2 g_3 g_4 g_5 g_6
compute L_j
L_1 L_2 L_3 L_4 L_5 L_6
$F_j = L_j - L_{j-1}$
L_1 F_2 F_3 F_4 F_5 F_6
update generator rows
g_2 g_3 g_4 g_5 g_6
compute L_j
L_2 L_3 L_4 L_5 L_6
$F_j = L_j - L_{j-1}$
L_2 F_3 F_4 F_5 F_6
update generator rows
g_3 g_4 g_5 g_6 ...
share g_1 btw. all threads
read L_1 (denom.)
share g_2 btw. all threads
read L_2 (denom.)

Figure 4.15: Schematic view of a simple GCD algorithm running with 6 threads indexed by **TID**.

ated using Horner's rule on the CPU to obtain the actual large integers. The rest of the discussion is devoted to the GCD kernel playing a central role in the algorithm.

4.5.3 Modular GCD: details of realization

Computing a GCD in $\mathbb{Z}_m[x]$ is done using our matrix-based approach discussed in Section 3.2.3. To recall the notation, let G be the generator matrix of size $n \times 4$ as defined in (3.47), where n is the sum of degrees of the input polynomials. Provided that, the rows of the generator matrix can be updated independently (due to data level parallelism), one can easily conceive of a simple parallel algorithm running in $O_P(n, n)$ time (see Section 2.5.1 for asymptotic notation). In what follows, we shall write L^i_j denoting the j-th column of the triangular factor in step i of the algorithm, which is defined as:

$$L^i_j = a_i a_j + b_i b_j - c_i c_j - d_i d_j \quad \text{and} \quad F^i_j = L^i_j - L^i_{j-1},$$

where $g_j = (a_j, b_j, c_j, d_j)$ and $g_i = (a_i, b_i, c_i, d_i)$ are the two rows of G. We designate one thread to process a single row g_j, such that the inner loop of Algorithm 3.3 disappears. Two steps of the basic parallel algorithm are illustrated in Figure 4.15. Here, the number of occupied threads decreases by one with the size of the generator G in each iteration of the algorithm. An iteration begins by loading the current top (leading) row of G in shared memory to make it visible to all threads of a block. Then, each thread evaluates L^i_j and writes the results back to shared memory. Next, we shift down the first column of the generator matrix by computing F^i_j (lines 9 and 22 of Algorithm 3.3). Finally, each participating thread updates a respective generator row which completes the iteration. Note that, in the realization we also distinguish between "lite" and "full" algorithm steps, as defined in Section 3.2.3, but we forget these details for the time being.

Unfortunately, with the basic approach outlined above, we again run into the problem of limited CUDA block size since threads need to communicate in order to compute the result, and hence they must belong to a single thread block. This problem we have already encountered in the design of the MRC algorithm (Section 4.3.4), where the solution was to process the data in chunks by one CUDA block. In present situation, however, this solution is not applicable since we do not have preliminary estimates on the degree of the input polynomials. Thus, the only adequate solution is to exploit block-level parallelism: that is, to distribute the computations over numerous thread blocks. To achieve

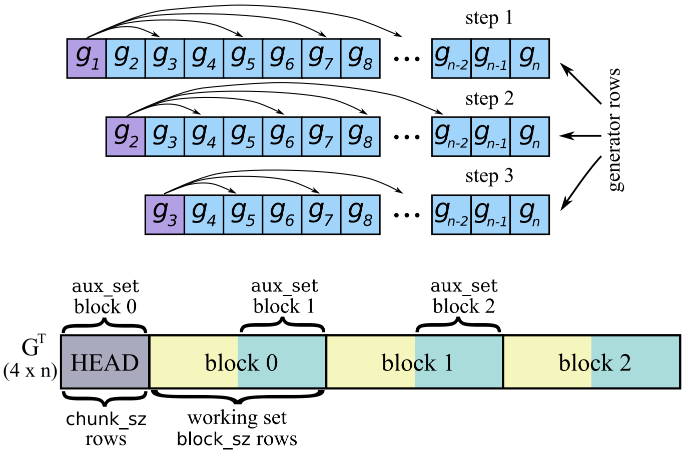
step 1
g_1 g_2 g_3 g_4 g_5 g_6 g_7 g_8 ... g_{n-2} g_{n-1} g_n
step 2
g_2 g_3 g_4 g_5 g_6 g_7 g_8 ... g_{n-2} g_{n-1} g_n
step 3
g_3 g_4 g_5 g_6 g_7 g_8 ... g_{n-2} g_{n-1} g_n
generator rows
aux_set block 0
aux_set block 1
aux_set block 2
G^T (4 x n)
HEAD
block 0
block 1
block 2
chunk_sz rows
working set block_sz rows

Figure 4.16: In each step of the algorithm, the leading generator row g_i is used to update all the remaining rows $g_{i+1}, \ldots, g_n$ (*top*); partitioning of the generator matrix $G^T \in \mathbb{Z}^{4 \times n}$; aux_set: additional matrix rows loaded by a thread block (*bottom*)

this, we need to abstract away from the details and try to understand the specifics of data movements performed by the algorithm. This can best be described graphically as in Figure 4.16 (top). Looking at the diagram, one can see that, in order to acquire (partial) independence in computations, we need to introduce some *data redundancy*, so that different thread blocks can perform several steps of the algorithm without the need for communication.

For this purpose, we partition the rows of the matrix G in chunks of size chunk_sz[1] as in Figure 4.16 (bottom). The first chunk_sz rows of G will be denoted by HEAD. We select thread block to be of size block_sz := chunk_sz · 2 threads, and assign the rows of G to thread blocks as follows. **1.** All blocks share the current HEAD. **2.** Each block has its *working set* which consists of block_sz rows (after skipping HEAD). **3.** Another chunk_sz rows before the working set are assigned to each block which we denote by aux_set. For 0th block, aux_set is identical to HEAD. The aim of this data partitioning is to run chunk_sz iterations of the algorithm without any data exchange. First, in iteration i, we need the top row g_i of G_i to compute the elements of the triangular factor L^i: this is why each block requires chunk_sz leading rows of G (HEAD). Second, during the generator update, one computes $F^i_j = L^i_j - L^i_{j-1}$, hence we need additional chunk_sz rows before a block's working set in order to have access to the "preceding" elements L^i_{j-1} of L^i_j's in each step.

The core of the algorithm comprises two phases: **gen_run** where we prepare the "updating sequence" using the rows of HEAD and aux_set; and **update_run** where this sequence is applied to the working set. Sample workflow of the algorithm is shown in Figure 4.17 (left). As in the basic algorithm, we assign one thread to work with a single generator row g_i. In the first iteration, we take the current leading row g_1 and use it together with aux_set in order to compute the columns of the triangular factors: $L^1_{k+1}, \ldots, L^1_{k+4}$. Then, we update the subsequent rows $g_2, \ldots, g_4$ of HEAD, as well as the rows $g_{k+2}, \ldots, g_{k+4}$ of aux_set. The row g_{k+1} is left *unchanged* since it requires a *previous* element L^1_k for the update which is not available (for this reason, g_{k+1} is marked with a yellow box in the fig-

[1] Chunk size is chosen to be divisible by 32 (warp size) for reasons of efficiency.

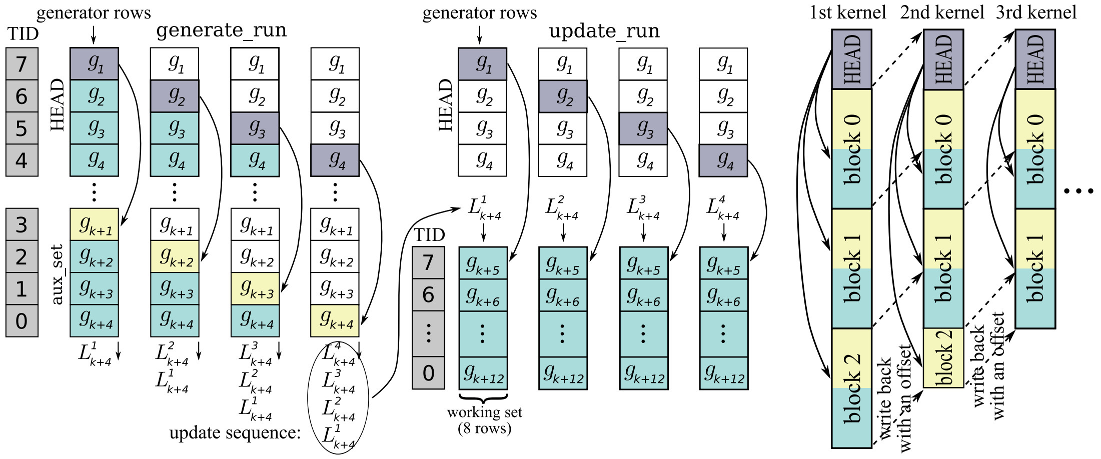

Figure 4.17: Sample workflow of the block algorithm with block_sz = 8: first, a sequence $\{L^i_{k+4}\}$ is generated, and then applied to the working set; elements in the white boxes are not modified by the algorithm (*left*). Block GCD algorithm running with several thread blocks. Each kernel call corresponds to chunk_sz steps of the serial algorithm. Dashed arrows mark the regions in global memory where each block writes back its results (*right*).

ure). We also keep L^1_{k+4} which is required to updating the first row g_{k+5} of the working set. Analogously, in the second step, we use g_2 to compute $L^2_{k+2}, \ldots, L^2_{k+4}$, and update the rows g_3, g_4 and g_{k+3}, g_{k+4}. This time, g_{k+2} is only used to compute the triangular factor, and an element L^2_{k+4} is saved. As one can see from Figure 4.17 (left), the rows of G are processed in a "stair-like" fashion. At the end of **gen_run**, we have a full sequence $L^4_{k+4}, \ldots, L^1_{k+4}$.

In the next stage, **update_run**, the updating sequence is applied to the working set. Note that, we do not need to process the rows of HEAD once again because the results can be effectively reused from the previous stage. In each step, we take the leading row of HEAD and extract the last element of the updating sequence to process the working set. It can be seen as though this sequence is "pushed" onto the working set from the top: that is, in step 1 we take L^1_{k+4} to compute F^1_{k+5}, in step 2 we take L^2_{k+4} to compute F^2_{k+5}, and so on. Remark that, in each step *all rows* of the working set ($g_{k+5}, \ldots, g_{k+12}$) get updated, hence we achieve the full thread occupancy here.

4.5.4 Modular GCD: overall approach

Having the basic routine at hand, we can now present the overall approach working with arbitrary number of thread blocks. From the host perspective, the algorithm consists of several kernel calls invoking repeatedly the procedure outlined in Figure 4.17 (left), where each kernel call is equivalent to chunk_sz steps of Algorithm 3.3. At the end of each call, participating thread blocks save their working sets back to global memory with an *offset* specified by the parameter chunk_sz. In this way, the first half of the 0th block's working set becomes HEAD for the next kernel call, and so on, see Figure 4.17 (right). To choose the parameter chunk_sz, we have performed a number of experiments confirming that the value chunk_sz = 128 works best in practice.

The host part of the algorithm is given in Algorithm 4.3. Here, we provide a template argument to the CUDA kernel (lines 8 and 17) to distinguish between "lite" and "full" iterations as defined in Section 3.2.3. The number of "lite" iterations is controlled by

Algorithm 4.3 Host part of the block GCD algorithm

```
procedure GCD_HOST_PART(Polynomial f, Polynomial g)
    p = f.degree(), q = g.degree(), n = p + q                ▷ assume: $p \geq q$
    start_i = 0                                              ▷ setup iteration counter
    B = (p + chunk_sz)/(2 ∗ chunk_sz)
    dim3 thids(block_sz)                                     ▷ # of threads per block
    dim3 grid(N, B)                                          ▷ grid of thread blocks
    while (1) {                                              ▷ first run “lite” iterations
        gcd_block_kernel<chunk_sz, false>⋘ grid, thids ⋙
              (p_out, p_in, start_i, 0)                      ▷ kernel launch
        start_i += chunk_sz                                  ▷ increase iteration counter
        swap(p_in, p_out)                                    ▷ “ping-pong” memory access
        if (start_i >= q) break                              ▷ finished “lite” iterations
    }
    ofs = (q − start_i + chunk_sz)                           ▷ compute mem. ofs
    while (1) {                                              ▷ run “full” iterations
        dim3 grid(N, B)                                      ▷ # of blocks B decreases
        gcd_block_kernel<chunk_sz, true>⋘ grid, thids ⋙
              (p_out, p_in, start_i, ofs)                    ▷ kernel launch
        start_i += chunk_sz                                  ▷ increase iteration counter
        swap(p_in, p_out)                                    ▷ “ping-pong” memory access
        sz = n − start_i                                     ▷ the current size of generator matrix
        if (sz <= 3 ∗ chunk_sz) break                        ▷ break if sz is small
        B = (sz + chunk_sz − 1)/(2 ∗ chunk_sz)
    }
                                                             ▷ simple kernel runs the remaining iterations
    gcd_simple_kernel⋘ grid, thids ⋙(p_in, p_out)
end procedure
```

the counter ‘start_i’ advanced by chunk_sz in each step. We also use double-buffering – ‘p_in’ and ‘p_out’ – to prevent data corruption due to simultaneous memory access. The kernel is launched on a 2D grid of size $N \times B$, where N is the number of moduli and B is the number of blocks per modular GCD. For p and q being the degrees of the original polynomials, at the beginning of the algorithm we set:

$$\mathsf{B} = (\max(\mathsf{p}, \mathsf{q}) + \mathsf{chunk\_sz})/(\mathsf{chunk\_sz} \cdot 2).$$

This number of blocks suffices because, by looking at the matrix G in (3.47), we see that the first two columns relevant during “lite” iterations of the algorithm have no more than $\max(p, q) + 1$ *nonzero* entries each. Note that, during “full” iterations, the parameter B decreases every two steps of the algorithm. Here, we also have to check for the vanishing denominator which indicates that a GCD is computed (see lines 26–28 in Algorithm 3.3). Certainly, we do not want to waste the GPU cycles here: therefore, as soon as this happens, a leading CUDA block sets up a global flag which forces all subsequent kernel calls to quit immediately.

4.5.5 Performance evaluation

Our platform to run the experiments was pretty much the same as the one used for benchmarking resultants: that is, a desktop machine with 2.8GHz 8-Core Intel Xeon W3530 (8 MB L2 cache) CPU and NVIDIA GTX580 graphics card running under 64-bit Linux

configuration	deg(GCD)	GPU	Maple
deg(f/g): **923/412**, bits(f/g): **300/200** (sparse)	100	1.3 ms	56 ms
deg(f/g): **1000/400**, bits(f/g): **300/200** (dense)	100	1.5 ms	104 ms
deg(f/g): **744/1126**, bits(f/g): **5000/5000** (sparse)	652	20 ms	156.0 s
deg(f/g): **1599/989**, bits(f/g): **140/2100** (sparse)	330	9 ms	1.9 s
deg(f/g): **2000/1500**, bits(f/g): **149/2109** (dense)	500	16 ms	? (timeout)
deg(f/g): **2300/2100**, bits(f/g): **35/1015** (dense)	1400	15 ms	33.4 s
deg(f/g): **3669/3957**, bits(f/g): **3000/2000** (-)	3257	70 ms	3.0 s
deg(f/g): **4900/4900**, bits(f/g): **46/46** (dense)	2500	43 ms	0.55 s
deg(f/g): **10000/10000**, bits(f/g): **162/165** (-)	5000	0.24 s	81.7 s
deg(f/g): **10000/10000**, bits(f/g): **3733/768** (-)	5000	1.03 s	82.0 s

Table 4.4: Benchmarks for single GCDs. deg(f / g) and bits(f / g): degrees and coefficient bitlength of the input polynomials f and g, respectively.

platform. Large integer arithmetic has been provided by GMP 5.0.1 library.[1] Again, our main contestant was the modular GCD algorithm from 64-bit compilation of Maple 14.[2] Maple's implementation of a GCD algorithm is built-in for integer polynomials, and relies on several algorithms selected according to the size of the inputs and other heuristics. These algorithms, for example, include heuristic GCD, EZGCD and sparse modular GCD algorithms, see (LF95) for comparison. For very large polynomials, Maple also employs an asymptotically fast Half-GCD algorithm (TY90). As noted before, this version of Maple can take advantage of multiple CPU cores present on the host machine which can be verified using 'kernelopts(multithreaded)' command. In our case, the number of logical CPUs in use has been set to 8 by default.

The timings for computing a GCD of two polynomials are listed in Table 4.4. Among the input parameters, we have varied polynomial degrees, coefficient bitlength and the density of polynomials (the number of non-zero terms). In the experiments, we have not considered coprime polynomials since Maple as well as our algorithm provide special means to quickly check for a trivial GCD. In general, one can see that the timings for our algorithm are now significantly better than those published in the original work (Eme11). The reason for this are numerous small improvements taking place throughout the whole algorithm and, among others, because of the fact that the modular reduction has been moved to the graphics processor (see Section 4.3.3). It turned out that the latter operation could occupy more than a half of the total running time of the algorithm in extreme cases. Going back to the benchmarks, we see that Maple's GCD performs better for sparse polynomials while our (matrix-based) approach is largely insensitive to polynomial density. On the other hand, it appears that Maple has a serious trouble dealing with *unbalanced* operands: that is, when two input polynomials substantially differ in the degrees or the size of coefficients. A practical study confirms that, these situations occur quite often: for instance, when comparing two algebraic numbers or computing the polynomial factoriza-

[1] `http://gmplib.org`

[2] kernelopts(wordsize) returns 64 which verifies 64-bit Maple.

configuration	N	deg(GCD)	GPU	Maple
deg(f/g) : **500/600**, bits(f/g) : **70/70** (dense)	100	50 (avg.)	7.3 ms	1.4 s
deg(f/g) : **500/600**, bits(f/g) : **70/70** (dense)	200	50 (avg.)	10.7 ms	2.7 s
deg(f/g) : **500/600**, bits(f/g) : **70/70** (dense)	400	50 (avg.)	24 ms	5.4 s
deg(f/g) : **900/800**, bits(f/g) : **200/200** (sparse)	50	100 (avg.)	19 ms	2.9 s
deg(f/g) : **900/800**, bits(f/g) : **200/200** (sparse)	100	100 (avg.)	35 ms	6.1 s
deg(f/g) : **900/800**, bits(f/g) : **200/200** (sparse)	200	100 (avg.)	60 ms	14.0 s
deg(f/g) : **1800/1800**, bits(f/g) : **250/30** (sparse)	50	900 (avg.)	70 ms	25.7 s

Table 4.5: The running times for computing a batch of **N** GCDs of random polynomials. Abbreviations are as in Table 4.4.

tion. We anticipate that, in this case, Maple cannot correctly decide which algorithm's version to use which results in a huge slowdown. It is also worth noting that the performance of Maple's algorithm deteriorates greatly for very large polynomial degrees: here, it could be the case that the size of a CPU cache already becomes a bottleneck.

Looking at the GPU timings, however, we can argue that the graphics hardware stays underutilized for the most examples in Table 4.4 (though it may not seem obvious at first glance). This stems from the fact that, in contrast to bivariate resultants, a univariate GCD computation alone does not provide us with a sufficient amount of parallelism to keep the GPU circuits busy all the time, and thereby cannot mitigate the negative effects of external memory latencies. An implicit indication for this is that the running times do not change much when going from one configuration to another. That is why, to test the hardware at full occupancy, we have also run the experiments where a *batch* of GCDs of low degree random polynomials is computed. Such "batched" GCD computations can, for instance, be used in the solution of multivariate GCD problems with the help of the modular approach (see Section 2.3). Altogether, the speed-up attained for a batch of GCDs in Table 4.5 is clearly more impressive. Indeed, on the average, the GPU algorithm requires less then a millisecond per one GCD. Here, the batch size (the number GCDs computed) is given by the parameter **N**. From the table, we also see that the hardware saturation is reached for **N** somewhere between 50 and 100 or between 100 and 200 depending on the configuration, since the GPU running times do not scale linearly within this range.

In Figure 4.18, we examine the performance of the algorithm versus the polynomial's degree (left) and coefficient bitlength (right). In the left diagram, the dependency on the degree is most likely to be *sub-quadratic* since the parallel complexity of the GPU algorithm is linear by design; however, with increasing the degree, more thread blocks need to contribute to one modular GCD. As a result, the amount of redundant work increases, see Section 4.5.3, making the final complexity sub-quadratic. In contrast, increasing the coefficient bitlength only causes the number of moduli to increase (or the number of *independent* thread blocks) which is why the performance only scales linearly in the right figure. Figure 4.19 displays a CUDA profiler output for the GPU algorithm executed for the last configuration in Table 4.4. In the figure, the large blue and green shapes correspond to "lite" and "full" iterations of the matrix-based GCD algorithm, respectively

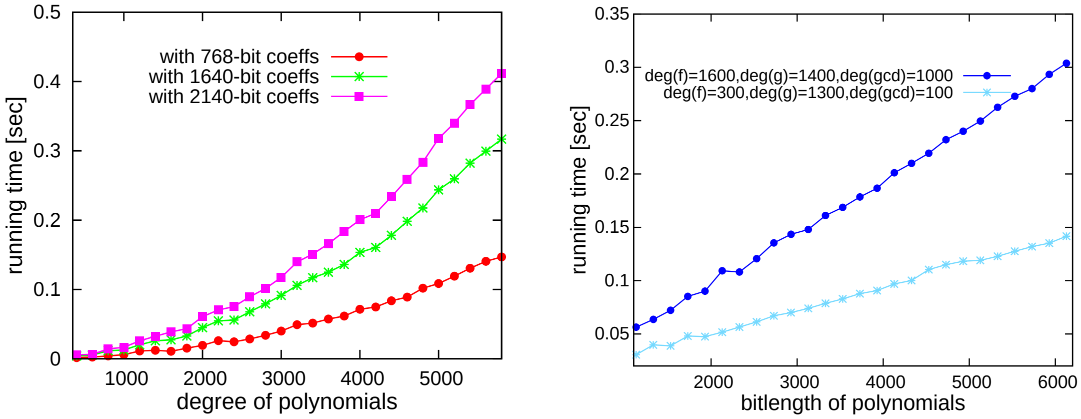

Figure 4.18: Execution time vs. polynomial degree (*left*) and coefficient bitlength (*right*)

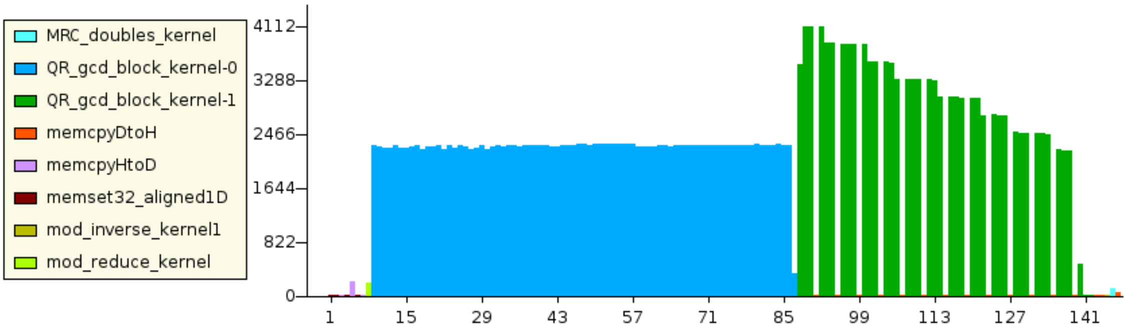

Figure 4.19: CUDA kernel statistics for testing the GCD algorithm on the last configuration in Table 4.4

(see Section 3.2.3). Here, the height plot for "full" iterations has a typical trapezoid form because the number of working thread blocks declines in each step of the algorithm. Also, notice that, the contribution of other stages of the modular algorithm is negligibly small compared to the cost of the univariate GCD algorithm alone. This is a natural thing to expect for polynomials of very high degree. On a similar note, the overhead of GPU–host memory transfer, 'memcpyDtoH' and 'memcpyHtoD' in the profiler diagram, has no impact on the algorithm performance indicating that the arithmetic intensity of involved computations is sufficiently high.[1]

As for concluding remarks, most of the comments made at the end of Section 4.4.4 do also apply to the GCD algorithm. Specifically, we would like to mention that our successful attempt to integrate block-level parallelism to the GCD algorithm can be used to further improve the resultant and interpolation algorithms as well (see Sections 3.2.1 and 3.2.2), owing to the fact that they are just the particular cases of the same generalized approach. Besides, we believe that this result has a value on its own since the factorization of structured matrices has a wide range of significant applications which go far beyond the polynomial algebra. Another promising task would be to extend the GCD algorithm to *multivariate domain*, especially, because all the required components of the modular approach, including evaluation/interpolation, have already been tested on the resultants.

In the next and final chapter of this thesis, we discuss two important applications of the

[1] Here, we we have also used *page-locked* host memory to speed-up data transfers.

developed algorithm to see how they behave on a real data because synthetic benchmarks might not always reveal the real strengths and weaknesses.

5 Applications

Among possible applications of the GPU algorithms developed in this thesis, we foremost consider the problem of computing real solutions of a system of bivariate polynomial equations. Polynomial systems of equations arise naturally in many fields of science and engineering and have a fundamental importance in non-linear computational geometry or Computer-Aided Geometric Design. For this problem, we utilize a novel approach (which we shall refer to as BISOLVE) proposed quite lately at the time of writing. It demonstrates a significant performance improvement over other state-of-the-art algorithms available to date. One of the major strengths of BISOLVE relates to the fact that it restricts a required set of symbolic operations to that of computing resultant and GCD, and thereby can benefit to the full degree from the developed GPU algorithms. In the course of our discussion, we shall also analyze the complexity of BISOLVE to persuade the reader that the demonstrated high performance of the algorithm also finds a confirmation in theory.

As a second application of parallel algorithms, we address the problem of geometrically correct rasterization of algebraic curves defined in $\mathbb{R}^2$ by implicit equations.[1] Implicitly defined algebraic curves have proven very useful in the solution of many geometric and model-based problems because of their ability to provide a compact representation of complex geometric objects. It is also known that the accurate visualization of such curves is not always feasible (or, otherwise, can be very inefficient) unless some topological information of a curve is available. We show how to exploit the power of GPUs to dramatically speed-up the visualization of algebraic curves.

5.1 Solution of a bivariate polynomial system

This section is devoted to the algorithm BISOLVE proposed in (BES11) to computing the real solutions of a system induced by two bivariate polynomials. Such non-linear systems of equations, for instance, arise in the topological study of algebraic curves and surfaces, where they are used to identify the "events points" (tangents and singularities). We give a detailed review of the algorithm and also discuss its further development to compute the topology of algebraic curves (BEKS11a, BEKS11b). Finally, we evaluate the performance of the algorithm on a number of challenging benchmarks with and without the GPU acceleration. The complexity analysis of BISOLVE will be given separately in Section 5.2.

[1] What is precisely understood by a geometrically correct rasterization will be made clear in Section 5.3.

5.1.1 Problem definition and related work

To state the problem in a mathematically concise way, let $f, g \in \mathbb{Z}[x, y]$ be polynomials of total degrees m and n, respectively. We further assume that f and g share no common non-trivial factor in $\mathbb{Z}[x, y]$. This is equivalent to saying that the following polynomial system

$$f(x,y) = \sum_{i,j \in \mathbb{N}: i+j \leq m} f_{ij} x^i y^j = 0, \quad g(x,y) = \sum_{i,j \in \mathbb{N}: i+j \leq n} g_{ij} x^i y^j = 0 \tag{5.1}$$

is *zero-dimensional* having at most $n \times m$ solutions in $\mathbb{C}^2$ by Bezout's theorem. The algorithm Bisolve computes a set of disjoint boxes $B_k \subset \mathbb{R}^2$ isolating all real solutions of (5.1). In other words, the union of all B_k contains

$$V_{\mathbb{R}} := \{(x, y) \in \mathbb{R}^2 | f(x, y) = g(x, y) = 0\}, \tag{5.2}$$

the set of real solutions of (5.1).

For the related work, we can distinguish between two classes of algorithms. The first class includes numeric algorithms which approximate the solution of a problem to a certain precision, and thus cannot be regarded as "certified" and "complete" methods. The main representatives of this family are homotopy (SW05b) and subdivision approaches (AMW08, MP09). Naturally, their major strength lies in the use of approximate computations, such as provided by the software libraries IntBits, ALIAS, IntLab or MPFI, to process many instances in a very efficient way, albeit with no guarantees on the computed results. Though, subdivision methods can be made certifying and complete by considering worst case separation bounds for the solutions, this approach has not been proven effective in practice so far. The second class comprises the so-called *elimination* methods based on *(sparse) resultants*, *rational univariate representation*, *Groebner bases* or *eigenvalue* computations; see, for instance, (Pet99, Stu02). Our approach belongs to this family as well. In broad terms, the idea of these methods is to project the solutions onto several different axes (projection step), and then match the possible candidates using some "validation" procedure (lifting step). Recent exact and complete implementations for computing the topology of algebraic curves and surfaces (BKS10, CLP$^+$09b, EKW07, SW05a, BEKS11a) also make use of such elimination techniques. However, recalling the example from the introduction, we see that the cost of symbolic computations can quickly become dominating which restricts the use of such elimination methods. To make matters worse, a system under consideration might be in *non-generic* position (when there are two or more covertical solutions along some projecting direction) which significantly complicates the lifting step. In the latter situation, the existing approaches perform a coordinate transformation (or project in generic direction) which eventually increases the complexity of the input polynomials. In this respect, Bisolve constitutes a notable exception since, unlike the previous algorithms, its validation (or lifting) step does not involve any symbolic computations, neither it requires any coordinate change if the given system is in non-generic position. We next introduce the notation used throughout the algorithm description and complexity analysis.

For an interval $I = (a, b) \subset \mathbb{R}$, we define $w_I := b - a$ to be the *width*, $m_I := (a + b)/2$ the *center* and $r_I := (b - a)/2$ the *radius* of I. A disc in $\mathbb{C}$ is denoted by $\Delta = \Delta_r(m)$, where

$m \in \mathbb{C}$ defines the center of Δ and $r \in \mathbb{R}^+$ its radius. For a polynomial $F(x) = \sum_{i=0}^k F_i x^i \in \mathbb{R}[x]$, which is not necessarily square-free, with roots $z_1 \dots z_k \in \mathbb{C}$, we have:

- the *separation* $\operatorname{sep}(z_i, F)$ of z_i is the minimal distance from z_i to any root $z_j \neq z_i$;
- the root separation $\operatorname{sep}(F)$ of F defined as the minimum of all $\operatorname{sep}(z_i, F)$;
- $\Sigma(F) = \sum_{i=1}^k \log \operatorname{sep}(z_i, F)^{-1}$;
- the *Mahler measure* defined as $\mathcal{M}(F) := |\operatorname{lcf}(F)| \prod_{i=0}^k \max\{1, |z_i|\}$;
- $\Gamma(F) := \max_i |z_i|$ is the maximal magnitude of any root of F.

In the algorithm description and complexity analysis, we shall also use the recursive representation of f and g:

$$f(x,y) = \sum_{i=0}^{m_x} f_i^{(x)}(y)x^i = \sum_{i=0}^{m_y} f_i^{(y)}(x)y^i, \quad g(x,y) = \sum_{i=0}^{n_x} g_i^{(x)}(y)x^i = \sum_{i=0}^{n_y} g_i^{(y)}(x)y^i,$$

where $f_i^{(y)}$, $g_i^{(y)} \in \mathbb{Z}[x]$ and $f_i^{(x)}$, $g_i^{(x)} \in \mathbb{Z}[y]$. By $R^{(y)} = \operatorname{res}_x(f,g) \in \mathbb{Z}[x]$ and $R^{(x)} = \operatorname{res}_y(f,g) \in \mathbb{Z}[y]$ we denote the resultants of f and g with respect to variables x and y, respectively (see Section 2.4.1). Sometimes we shall omit the variable index, simply writing R for $R^{(x)}$ or $R^{(y)}$, if the two polynomials are interchangeable within a context. On a similar note, a square-free part of either polynomial will be denoted by R^*.

5.1.2 Algorithm review

We next recall the main steps of the algorithm. At the highest level, Bisolve comprises three subroutines which we consider in the order of their appearance in the algorithm.

Project : We begin with projecting the complex solutions of (5.1) onto the x- and y-axes and consider the real ones. In other words, we take the two sets:

$$V_{\mathbb{C}}^{(x)} := \{x \in \mathbb{C} | \exists y \in \mathbb{C} \wedge f(x,y) = g(x,y) = 0\}, \quad V_{\mathbb{C}}^{(y)} := \{y \in \mathbb{C} | \exists x \in \mathbb{C} \wedge f(x,y) = g(x,y) = 0\},$$

and compute their restrictions: $V_{\mathbb{R}}^{(x)} := V_{\mathbb{C}}^{(x)} \cap \mathbb{R}$ and $V_{\mathbb{R}}^{(y)} := V_{\mathbb{C}}^{(y)} \cap \mathbb{R}$ to the real values. The latter operation can be achieved by computing the resultants $R^{(y)}$ and $R^{(x)}$, respectively; and extracting the square-free parts R^* of both polynomials. Finally, we isolate the real roots α_i of R^* using the Descartes method, see (CA76, RZ04). It is then clear that the real solutions $V_{\mathbb{R}}$ of (5.1) are contained in the product

$$C := V_{\mathbb{R}}^{(x)} \times V_{\mathbb{R}}^{(y)} \subset \mathbb{R}^2, \tag{5.3}$$

which we call a set of *candidate solutions* of (5.1). This completes the first step of the algorithm. For the complexity analysis, we use a novel approach for real root isolation, proposed in (Sag11), which we refer to as Newdsc. Newdsc is a subdivision algorithm based on the combination of Descartes' Rule of Signs and Newton iteration. It achieves quadratic convergence for most iterations, and, with respect to the bit complexity, achieves the best bound known for this problem from the works of V. Pan or A. Schönhage (see also (Pan97) for an overview). Yet, in contrast to the above mentioned asymptotically fast algorithms, Newdsc concentrates on the real roots only and is much easier to access and to implement. The complexity of Newdsc is summarized in the following theorem:

Theorem 5.1.1: Given a square-free polynomial $F \in \mathbb{Z}[x]$ of magnitude (N, μ) and an integer $L \in \mathbb{N}$, we can compute isolating intervals (for all real roots) of width 2^{-L} or less using no more than

$$\tilde{O}(N^3\mu + N^2L)$$

bit operations. For proof, see (Sag11, Theorem 10). ◇

Note also that, in the actual implementation, we first compute a *square-free factorization* (or even a full factorization) of R, instead of simply taking R^*, to facilitate the real root isolation. Namely, we determine square-free and relatively prime factors $r_i \in \mathbb{Z}[x]$, $i = 1, \ldots, \deg(R)$, such that $R(x) = \prod_{i=1}^{\deg(R)} (r_i(x))^i$, where some factors $r_i(x)$ can be equal to 1. The factorization can be computed using Yun's algorithm (vzGG03, Alg. 14.21) which iteratively computes the greatest common divisors of R and its higher derivatives. This helps us reduce the costs of the subsequent real root isolation and further manipulations on polynomial roots. Yet, for the simplicity of the algorithm's description and the complexity analysis, it suffices to use the square free part R^* which does not alter the algorithm in any significant way.

Separate : In the second step, we further separate the real roots of R from the complex ones which will be needed for the validation of candidate solutions. For each root α, we refine an isolating interval $I := I(\alpha)$ computed by the Descartes algorithm until the $\Delta_{8r_I}(m_I)$ does not contain any root of R except α. In the implementation, we use the quadratic interval refinement (QIR for short); see (Abb06, KS11a). This method demonstrates high efficiency in practice due to its relative simplicity and the fact that the number of significant bits is *doubled* in each refinement step as opposed to the classical bisection method. The termination criterion of the refinement is based on the following test:

$$|(R^*)'(m_I)| - \frac{3}{2}\sum\nolimits_{k\geq 2}\left|\frac{(R^*)^{(k)}(m_I)}{k!}\right|(8r_I)^k > 0, \tag{5.4}$$

which guarantees that $\Delta_{8r_I}(m_I)$ isolates a root α from all other roots of R^*, see (BES11, Thm. 3.2) for a proof. In its turn, this enables us to compute a (non-zero) lower bound

$$LB(\alpha) := 2^{-2\deg(R)}|R(m_I - 2r_I)|, \tag{5.5}$$

for $|R(x)|$ on the boundary of $\Delta(\alpha) := \Delta_{2r_I}(m_I)$, that is: $|R(x)| > LB(\alpha)$ for all $x \in \partial\Delta(\alpha)$. For proof, see (BES11, Lem. 3.3).

In summary, at the end of Separate, we have a set of isolating intervals $I(\alpha)$ and $I(\beta)$ as well as isolating discs $\Delta(\alpha) := \Delta_{2r_{I(\alpha)}}(m_{I(\alpha)})$ and $\Delta(\beta)$ for all real roots α and β of $R^{(y)}$ and $R^{(x)}$, respectively. Besides, we have also computed the lower bounds $LB(\alpha)$ and $LB(\beta)$ for the values of $|R^{(y)}|$ and $|R^{(x)}|$ on the boundary of $\Delta(\alpha)$ and $\Delta(\beta)$. Finally, each real solution of the system (5.1) is contained in some polydisc $\Delta(\alpha,\beta) := \Delta(\alpha) \times \Delta(\beta) \subset \mathbb{C}^2$, and each of these polydiscs contains at most one solution.

Validate : In this last step, the candidates of C are either discarded or certified to be a solution of (5.1). For certification, we use a novel *inclusion predicate* to be introduced below. Note that, in the actual implementation, the inclusion predicate is used in combination with *bitstream Descartes algorithm* (EKK⁺05), as described in (BES11, Section 4.2), to early exclude many of the candidates from C. Such a hybrid approach works very well in practice. However, to keep matters simple, we shall assume that the candidate exclusion is entirely based on the interval arithmetic tests which does not affect the algorithm's complexity.

To establish the inclusion predicate, suppose we have a polydisc $\Delta(\alpha,\beta)$ and respective lower bounds $LB(\alpha)$ and $LB(\beta)$ as computed in Separate. First, we expand the resultants in terms of the cofactors (see Section 2.4.1):

$$R^{(y)} = u^{(y)} \cdot f + v^{(y)} \cdot g, \quad R^{(x)} = u^{(x)} \cdot f + v^{(x)} \cdot g, \tag{5.6}$$

where $u^{(y)}, v^{(y)} \in \mathbb{Z}[x,y]$ are determinants of "Sylvester-like" matrices $U^{(y)}$ and $V^{(y)}$ of size $(n_y + m_y)$:

$$U^{(y)} = \begin{vmatrix} f^{(y)}_{m_y} & f^{(y)}_{m_y-1,y} & \dots & f^{(y)}_0 & 0 \dots & y^{n_y-1} \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & f^{(y)}_{m_y} & f^{(y)}_{m_y-1} & \dots & 1 \\ g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \dots & g^{(y)}_0 & 0 \dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \dots & 0 \end{vmatrix}, V^{(y)} = \begin{vmatrix} f^{(y)}_{m_y} & f^{(y)}_{m_y-1} & \dots & f^{(y)}_0 & 0 \dots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & f^{(y)}_{m_y} & f^{(y)}_{m_y-1} & \dots & 0 \\ g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \dots & g^{(y)}_0 & 0 \dots & y^{m_y-1} \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \dots 0 & g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \dots & 1 \end{vmatrix},$$

and $u^{(x)}$ and $v^{(x)}$ are defined in a similar way (cf. Theorem 2.4.1). We next proceed by evaluating the upper bounds $UB(\alpha,\beta,u^{(y)})$ and $UB(\alpha,\beta,v^{(y)})$ for $|u^{(y)}|$ and $|v^{(y)}|$ on $\Delta(\alpha,\beta)$, respectively. Apparently, we wish to avoid computing the cofactors explicitly which are usually very large expressions: instead, we can use the fact that $u^{(y)}$ and $v^{(y)}$ are given in the determinantal form. In other words, we proceed by upper bounding the absolute values of the entries of $U^{(y)}$ and $V^{(y)}$, and then apply Hadamard's inequality to $U^{(y)}$ and $V^{(y)}$ to arrive at the upper bounds for $|u^{(y)}|$ and $|u^{(y)}|$ on $\Delta(\alpha,\beta)$. The upper bounds for $|u^{(x)}|$ and $|u^{(x)}|$ are derived in analogous manner. To formulate the inclusion test we begin with the auxiliary theorem.

Theorem 5.1.2: Let α and β be arbitrary real roots of $R^{(y)}$ and $R^{(x)}$, respectively. Then,

1. the polydisc $\Delta(\alpha,\beta) := \Delta(\alpha) \times \Delta(\beta) \subset \mathbb{C}^2$ contains at most one (complex) solution of (5.1). If $\Delta(\alpha,\beta)$ contains a solution of (5.1), then this solution is real valued and equals (α,β).
2. For an arbitrary point $(z_1,z_2) \in \mathbb{C}^2$ on the boundary of $\Delta(\alpha,\beta)$, it holds that

$$|R^{(y)}(z_1)| > LB(\alpha) \text{ if } z_1 \in \partial\Delta(\alpha), \text{ and } |R^{(x)}(z_2)| > LB(\beta) \text{ if } z_2 \in \partial\Delta(\beta).$$

For proof, see (BES11, Thm. 3.3). ◇

The actual inclusion predicate is based on the following result.

Theorem 5.1.3: If there exists an $(x_0,y_0) \in \Delta(\alpha,\beta)$ with

$$UB(\alpha,\beta,u^{(y)}) \cdot |f(x_0,y_0)| + UB(\alpha,\beta,v^{(y)}) \cdot |g(x_0,y_0)| < LB(\alpha), \tag{5.7}$$

$$UB(\alpha,\beta,u^{(x)}) \cdot |f(x_0,y_0)| + UB(\alpha,\beta,v^{(x)}) \cdot |g(x_0,y_0)| < LB(\beta), \tag{5.8}$$

then $\Delta(\alpha,\beta)$ contains a solution of (5.1) and, thus, $f(\alpha,\beta) = 0$. ◇

Proof Below, we reproduce the original proof from (BES11, Thm. 3.4) as it plays a central role in the correctness of the whole approach. The main idea is to use a homotopy argument. Namely, we consider the parameterized system

$$f(x,y) - (1-t)\cdot f(x_0,y_0) = g(x,y) - (1-t)\cdot g(x_0,y_0) \ \ = 0, \tag{5.9}$$

where t is an arbitrary real value in $[0,1]$. For $t = 1$, (5.9) is equivalent to our initial system (5.1). For $t = 0$, (5.9) has a solution in $\Delta(\alpha,\beta)$, namely, (x_0, y_0). The complex solutions of (5.9) continuously depend on the parameter t. Hence, there exists a "solution path" $\Gamma : [0,1] \mapsto \mathbb{C}^2$ which connects $\Gamma(0) = (x_0, y_0)$ with a solution $\Gamma(1) \in \mathbb{C}^2$ of (5.1). We show that $\Gamma(t)$ does not leave the polydisc $\Delta(\alpha,\beta)$ and, thus, (5.1) has a solution in $\Delta(\alpha,\beta)$: Assume that the path $\Gamma(t)$ leaves the polydisc, then there exists a $t' \in [0,1]$ with $(x', y') = \Gamma(t') \in \partial\Delta(\alpha,\beta)$. We assume that $x' \in \partial\Delta(\alpha)$ (the case $y' \in \partial\Delta(\beta)$ is treated in analogous manner). Since (x', y') is a solution of (5.9) for $t = t'$, we must have $|f(x', y')| \leq |f(x_0, y_0)|$ and $|g(x', y')| \leq |g(x_0, y_0)|$. Hence, it follows that

$$\begin{aligned}|R^{(y)}(x')| &= |u^{(y)}(x', y')f(x', y') + v^{(y)}(x', y')g(x', y')|\\ &\leq |u^{(y)}(x', y')| \cdot |f(x', y')| + |v^{(y)}(x', y')| \cdot |g(x', y')|\\ &\leq UB(\alpha,\beta, u^{(y)}) \cdot |f(x_0, y_0)| + UB(\alpha,\beta, v^{(y)}) \cdot |g(x_0, y_0)| < LB(\alpha).\end{aligned}$$

This contradicts the fact that $|R^{(y)}(x')|$ is lower bounded by $LB(\alpha)$. It follows that $\Delta(\alpha,\beta)$ contains a solution of (5.1) and, by Theorem 5.1.2, this solution must be (α,β). ■

The inclusion predicate works as follows. Let $B(\alpha,\beta) = I(\alpha) \times I(\beta) \subset \mathbb{R}^2$ by a *candidate box* containing some candidate solution $(\alpha,\beta) \in C$. Each $B(\alpha,\beta)$ is refined using the QIR method until one of the following happens:

- we can ensure that $f(\alpha,\beta) \neq 0$ or $g(\alpha,\beta) \neq 0$ by evaluating the box functions $\Box f(B(\alpha,\beta))$ and $\Box g(B(\alpha,\beta))$ using interval arithmetic on $B(\alpha,\beta)$, and then checking the resulting intervals for zero inclusion. If (α,β) is not a solution, the exclusion test must eventually succeed provided that $B(\alpha,\beta)$ is small enough. The details on the polynomial evaluation using interval arithmetic will be given in Section 5.2.2;
- for an arbitrary point $(x_0, y_0) \in B(\alpha,\beta)$, both conditions (5.7) and (5.8) are satisfied. In this case, Theorem 5.1.3 guarantees that (α,β) is a solution of the system (5.1).

5.1.3 Extending Bisolve to curve analysis

We next discuss how Bisolve can be extended to determining the topology of an algebraic curve. We shall refer to this approach as BiCurveAnalysis. Recall that, a plane real algebraic curve is defined implicitly as a zero set of a bivariate polynomial:

$$C = \{(x, y) \in \mathbb{R}^2 : f(x, y) = 0\}, \ \text{ where } \ f \in \mathbb{Z}[x, y].$$

We will also study such curves in greater detail in Section 5.3 in the context of accurate curve visualization. For now, it is only worth noting that computing the topology of algebraic curves belongs to the set of fundamental problems in real algebraic geometry with many applications in computational geometry, computer graphics and modeling.

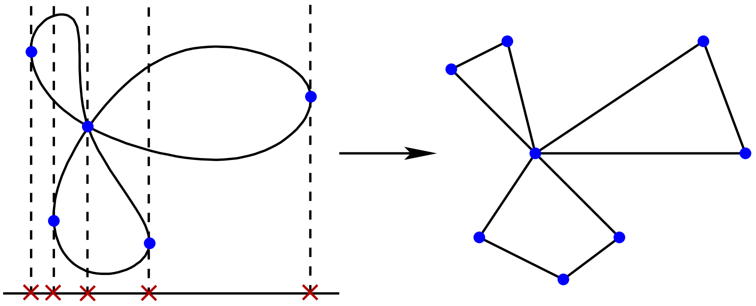

Figure 5.1: Algebraic curve with projected "event points" (*left*) and its topology graph (*right*).

The topology of a curve C is given by a planar graph $\mathcal{G}_C$ embedded in $\mathbb{R}^2$ that is isotopic to C, see Figure 5.1. We additionally require all vertices of $\mathcal{G}_C$ to lie on the curve C. At the highest level, the algorithm computing a curve topology follows a classical CAD (Cylindrical Algebraic Decomposition) approach (Col75). As before, we shall restrict ourselves to a very concise outline because a full algorithm description, together with filtering techniques and a framework for arrangement computation, falls beyond the scope of this thesis. For the complete approach we refer to (BEKS11a). We next describe the three main steps of the algorithm.

Projection phase. In the first step, we project all *x-critical* or *event* points (α,β) of a curve C onto the x-axis. For such points, it holds that $f(\alpha,\beta) = f_y(\alpha,\beta) = 0$. Geometrically, they are exactly the points where C has a vertical tangent or is singular (see also Section 5.3.1). Altogether, the projection phase is equivalent to the first step of the algorithm Bisolve applied to polynomials f and f_y, see Section 5.2.1 (Project), except that we do not need to compute the projected solutions in both directions. Thus, adopting the notation from Bisolve, at the end of this phase, we have a set $I := I(\alpha)$ of isolating intervals for the real roots α of R^*, a square-free part of $R^{(y)} := \mathrm{res}_x(f, f_y)$.

Lifting phase. Here, we first isolate the real roots of *intermediate* (square-free) polynomials $f(q_I, y) \in \mathbb{Q}[y]$, where q_I is a fixed rational value chosen arbitrarily from each interval $I = (\alpha, \alpha')$, where α and α' is a pair of consecutive roots of R^*. Since polynomials $f(q_I, y)$ have rational coefficients and are square-free, the Descartes algorithm applies directly yielding the numbers m_I which correspond to the number of real roots of $f(q_I, y)$ or, equivalently, the number of arcs of C above I.

Once this is done, for each x-critical value α we determine the real roots $y_{\alpha,1}, \ldots, y_{\alpha,m_\alpha}$ of a (non square-free) *fiber* polynomial $f(\alpha, y) \in \mathbb{R}[y]$, where m_α is the multiplicity of α as a root of $R^{(y)}$. Note that, a polynomial $f(\alpha, y)$ always has multiple roots and, generally, algebraic (non-rational) coefficients. That is why, we cannot directly use the Descartes method to isolate its roots. Instead, we use the method based on the iterative solution of bivariate polynomial systems induced by the set of higher derivatives of f. This method, called Lift in the original paper, is complete, i.e., it is guaranteed to succeed in all degenerate situations. In fact, Lift is used in a combination with FastLift – a fast method for fiber computations – acting as a filter. The latter approach relies on a numerical solver to compute (arbitrary good) approximations of complex roots of $f(\alpha, y)$ as well as on an exact certification step to prove the existence of roots within the computed complex discs. Altogether, FastLift shows impressive performance as reported in (BEKS11a, BEKS11b). However, to keep the discussion concise, in what follows we only consider the Lift method. Lift starts by computing the solutions $p_i = (\alpha,\beta_i)$, $i = 1,\ldots,l$ of the system $f = f_y = 0$ for each x-coordinate α using the algorithm Bisolve. Then, for each of these points p_i, we compute

$$k_i := \min\{k : \frac{\partial^k f}{\partial y^k}(\alpha,\beta_i) \neq 0\} \geq 2. \tag{5.10}$$

We further adopt the notation writing f_{y^k} to denote $\partial^k f/\partial y^k$, $k \geq 1$. The above computation can be done by iteratively invoking Bisolve for systems $f_y = f_{y^2} = 0$, $f_{y^2} = f_{y^3} = 0$, etc., and sorting the solutions along the vertical line $x = \alpha$. We eventually end up with disjoint intervals $I_1,\ldots,I_l$ and corresponding multiplicities $k_1,\ldots,k_l$ such that $I_j := I(\beta_j)$ contains a k_j-fold root β_j of $f(\alpha,y)$. Note that, the intervals I_j are already isolating for multiple roots of $f(\alpha,y)$, however, they might still contain the ordinary roots of $f(\alpha,y)$. Therefore, we further refine each I_j until we can guarantee via interval arithmetic that $\partial^{k_j} f/\partial y^{k_j}(\alpha,y)$ does not vanish on I_j. Then, by the mean value theorem, I_j cannot contain any other root of $f(\alpha,y)$ except β_j.

Finally, it remains to determine the ordinary roots of $f(\alpha,y)$. In the actual algorithm, we use the *Bitstream Descartes* method (EKK$^+$05) which we already know from the Bisolve routine. However, for the clarity of presentation, we may assume that the Bitstream Descartes is replaced by interval arithmetic tests. Then, the ordinary roots of $f(\alpha,y)$ can be determined as follows. We take the corresponding intervals I returned by the first invocation of Bisolve (i.e., for the system $f = f_y = 0$), and refine each of them until we can show that I does not intersect with any interval I_j for a multiple root of $f(\alpha,y)$, or is completely contained in one of them. In the latter case, such an interval I cannot contain an ordinary root of $f(\alpha,y)$, and thus can be safely discarded. While in the former case, I is stored as isolating for some ordinary root of $f(\alpha,y)$.

Connection phase. As a result of previous computations, we obtain the set of vertices V of $\mathcal{G}_C$ given by the union of all intermediate points $(q_I, y_{I,i})$ and points $(\alpha, y_{\alpha,i})$ with an x-critical value α. In this last phase, we determine which of these vertices are connected by an arc of C. To achieve this, for each value α, one has to distinguish between two cases: namely, when there is exactly one x-critical point above α (*generic case*) or there are several covertical x-critical points (*non-generic case*). Then, the algorithm uses some combinatorial reasoning to decide how the intermediate and "special" points need to be connected. For each connected pair of vertices, we add a corresponding edge to $\mathcal{G}_C$. It is then rather straightforward to prove that the resulting topology graph $\mathcal{G}_C$ is isotopic to C.

In the following section, we consider the benchmarks only for Bisolve since our primary goal was to challenge the GPU parts of the algorithm, while both approaches, Bisolve and BiCurveAnalysis, essentially use the same set of symbolic operations. A complete set benchmarks for the algorithm BiCurveAnalysis can be found in (BEKS11b).

5.1.4 Performance evaluation

For experiments we have used our default configuration which consists of a desktop machine with 2.8GHz 8-Core Intel Xeon W3530 having 8 MB of L2 cache and GeForce

Instance	Description
L4_circles	four circles w.r.t. L4-norm, clustered solutions
curve_issac	a curve appeared in (CGL09)
tryme	covertical solutions, many candidates to check
large_curves	large number of solutions
degree_6_surf	silhouette of an algebraic surface, covertical solutions in both directions
challenge_12_2*	many candidate solutions to be checked
SA_4_4_eps*	singular points with high tangencies, displaced
FTT_5_4_4*	many non-rational singularities
dfold_10_6*	a curve with many half-branches
cov_sol_20	covertical solutions
mignotte_xy	a product of x/y- Mignotte polynomials, displaced; many clustered solutions
spider	degenerate curve, many clustered solutions
hard_one	vertical lines as component of one curve, many candidates to test
grid_deg_10	large coefficients, curve in generic position
huge_cusp	large coefficients, high-curvature points
cusps_and_flexes	high-curvature points
L6_circles	four circles w.r.t. L6-norm, clustered solutions
ten_circles	set of 10 random circles multiplied together, rational solutions
curve24	curvature of degree 8 curve, many singularities
compact_surf	silhouette of an algebraic surface, many singularities, isolated solutions
13_sings_9	large coefficients, high-curvature points
swinnerston_dyer	covertical solutions in both directions
challenge_12_1*	many candidate solutions to be checked
SA_2_4_eps*	singular points with high tangencies, displaced
spiral29_24	Taylor expansion of a spiral intersecting a curve with many branches, many candidates to check

Table 5.1: Equations of singular algebraic curves. Curves marked with (*) are taken from (Lab10).

GTX580 graphics processor under 64-bit Linux platform. The exact number types have been provided by Gmp 5.0.1 library. In addition, we have also used Ntl 5.5 library[1] to compare the performance of our GCD algorithm against the best CPU-based implementation available to date. The algorithm Bisolve has been integrated in Cgal (Computational Geometry Algorithms Library, www.cgal.org) as a prototypical package. We remark that the design of Cgal follows generic programming paradigm which greatly facilitates benchmarking as it enables us to exchange relevant parts of the implementation. In particular, we have been able to easily replace the default resultant and GCD implementations in Cgal with the GPU-based algorithms using template specialization.

As reference implementations, we have taken the algorithms Isolate developed by Fabrice Rouillier and Lgp by Xiao-Shan Gao et al.[2] Both algorithms are interfaced through 64-bit Maple 14. Note that that this version of Maple can benefit from multiple CPUs available on the host machine. This parameter, set by 'kernelopts(multithreaded)' command, defaults to 8 on our desktop. We further remark that all three implementations (including ours) make the essential use of RealSolving (Rs) package[3] for real root isolation which is known to be one of the best univariate solvers available to date. Fi-

[1] Gmp: http://gmplib.org, Ntl: http://www.shoup.net/ntl

[2] http://www.mmrc.iss.ac.cn/~xgao/software.html

[3] http://www.loria.fr/equipes/vegas/rs

Instance	y-degree	bits	# sols	Bs	Bs+gres	Bs+gres +ggcd	Lgp	Isolate
L4_circles	16	29	17	2.47	1.57	1.55	7.6	**1.3**
curve_issac	15	16	18	3.68	**2.77**	2.8	3.3	29.8
tryme	24, 34	117, 24	20	66.2	22.7	**22.0**	107.8	397.4
large_curves	24, 19	25, 103	137	84.4	73.8	**73.46**	98.1	311.6
degree_6_surf	42	47	13	95.7	14.1	**13.6**	131.2	?
challenge_12_2	40	43	99	69.9	14.0	**13.8**	277.7	351.6
SA_4_4_eps	33	228	2	94.5	2.39	**1.76**	54.5	158.6
FTT_5_4_4	40	39	62	42.8	10.3	**9.8**	195.6	256.4
dfold_10_6	32	17	21	21.06	4.43	4.45	**3.76**	**3.8**
cov_sol_20	20	128	8	18.93	8.67	**8.35**	171.6	532.4
mignotte_xy	32	84	30	314.2	253.2	**249.4**	?	?
spider	28	250	38	215.0	51.6	**46.7**	?	?
hard_one	27, 6	94, 49	46	7.74	7.0	**6.8**	17.5	64.5
grid_deg_10	10	505	20	2.75	**1.32**	**1.32**	2.64	111.2
huge_cusp	8	3044	24	14.8	9.4	**7.58**	116.7	?
cusps_and_flexes	9	386	20	1.62	1.02	**0.93**	2.43	381.5
L6_circles	24	58	18	16.9	4.31	**4.11**	51.6	21.4
ten_circles	20	22	45	8.7	6.55	6.44	**4.9**	5.8
curve24	24	26	28	31.3	**14.3**	14.7	37.9	86.0
compact_surf	18	296	57	12.1	4.41	**4.18**	12.0	871.9
13_sings_9	9	383	35	2.33	1.8	**1.64**	2.8	341.9
swinnerston_dyer	40	24	63	37.9	19.3	**19.1**	27.9	73.8
challenge_12_1	30	32	99	17.63	**6.65**	6.8	37.1	44.0
SA_2_4_eps	17	220	6	4.39	0.36	**0.31**	4.7	3.31
spiral29_24	29, 24	37, 25	51	59.5	**30.4**	30.9	76.5	215.3

Table 5.2: Execution time in seconds for singular curves given in Table 5.1. '*y-degree*': the degree of input polynomials in y-variable; '*bits*': coefficient bitlength; '*# sols*': the number of solutions in $\mathbb{R}^2$; question mark '?' indicates that algorithm was aborted by error or timeout ($>$ 1500 sec).

nally, we have run Bisolve with all filters, including *combinatorial*, *bidirectional* as well as *bitstream Descartes* switched on since our main goal was to challenge the GPU part of the algorithm. For a full description of the filtering techniques used in Bisolve, we refer to (BES11, Section 4.2).

In the benchmarks, we distinguish between polynomial systems which correspond to singular algebraic curves[1] and those induced by a pair of random polynomials with increasing coefficient bitlength. The former systems, whose description is provided in Table 5.1, can have clustered (bad separated) and/or covertical solutions or require many candidates being checked. In contrast, the curves specified by random polynomials are unlikely to have any complicated topology: here the intention was to see how different algorithms can handle polynomials with large coefficients.

The timings for singular curves are listed in Table 5.2. Whenever only a single number is specified in the second column of the table ('y-degree'), then the competing approaches compute the solutions of a system induced by some $f \in \mathbb{Z}[x, y]$ and its first derivative f'_y. The columns 'Bs', 'Bs+gres' and 'Bs+gres+ggcd' display the running times for our algorithm without GPU support, with GPU resultants and with GPU resultants and GCDs,

[1] For the definition of an algebraic curve see Section 5.3.1.

Configuration	# sols	Bs	Bs+gres	Bs+gres +ggcd	Lgp	Isolate
y-degree: **15/13**, bits: **64**	10	5.59	3.46	**3.15**	28.9	?
y-degree: **14/12**, bits: **128**	12	5.16	3.14	**2.46**	26.3	652.2
y-degree: **13/13**, bits: **256**	10	7.15	3.22	**3.02**	33.5	?
y-degree: **13/12**, bits: **384**	6	9.11	4.42	**2.32**	?	?
y-degree: **11/11**, bits: **512**	2	4.3	1.02	**0.59**	17.2	?
y-degree: **11/10**, bits: **768**	15	17.2	12.8	**6.35**	46.9	?
y-degree: **10/9**, bits: **1024**	12	13.4	10.0	**4.87**	18.9	?
y-degree: **10/9**, bits: **1568**	4	16.44	9.94	**3.49**	30.9	?
y-degree: **9/9**, bits: **2048**	9	14.13	7.11	**5.25**	134.3	?
y-degree: **8/7**, bits: **4096**	4	18.94	12.3	**11.3**	42.7	?
y-degree: **5/5**, bits: **6000**	2	2.28	**1.2**	**1.2**	19.8	?

Table 5.3: Execution time in seconds for random polynomials. '*y-degree*': the degree of input polynomials in y-variable, '*bits*': bitlength of scalar coefficients, '*# sols*': the number of solutions in $\mathbb{R}^2$; question mark '?' indicates that algorithm has been aborted by error or timeout (> 1500 sec)

respectively. In the former two cases, we use a GCD implementation from Ntl library. For the CPU-based implementation of the resultant algorithm, we use the one available in Cgal by default. As one can see from the table, our algorithm is, in general, superior to Lgp and Isolate. Indeed, even with the disabled GPU support Bisolve shows a noticeable performance improvement, especially in the tough cases when the competing approaches time out. However, for three instances, 'L4_circles', 'dfold_10_6' and 'ten_circles', our approach is slightly slower than the competitors. Upon detailed examination of the running times, we found that this slowdown was mainly caused by the subtle nature of the Bitstream descartes algorithm used as a filter in our approach. We are, nevertheless, convinced that this minor problem can be eliminated by revisiting our implementation. It is also worth mentioning that the default resultant algorithm available within Cgal is not mature enough and clearly looses against the Maple's algorithm (used by Lgp and Isolate). This justifies somewhat larger-than-expected running times for our approach when the GPU support is disabled.

Further analyzing the results in Table 5.2, we observe that Isolate performs particularly bad for polynomials with large coefficient bitlength while our approach and Lgp are less sensitive to this parameter. For Bisolve, we also see that the GPU-based resultant algorithm sometimes can give a huge performance boost, especially for high-degree polynomials. From other perspective, this confirms that the resultant is the only time-consuming symbolic operation used by our approach. The running times in the columns 'Bs+gres' and 'Bs+gres+ggcd' indicate that the GPU-based GCD algorithm only gives any noticeable speed-up when polynomials have large coefficients. The reason for this is because computing a GCD of moderate-degree univariate polynomials does not provide a sufficient amount of parallelism to keep the GPU circuits busy: the behaviour that we have also observed in the experiments in Section 4.5.5. Furthermore, the competing Ntl's GCD approach is already optimal for polynomials in this range. On the average, the GCD computation alone takes about 3–10 % of the running time, hence there is no much room for improvement.

Let us now turn to the second part of experiments dealing with random polynomials. Here, the solution of an induced bivariate system does not present any challenge to our approach since, in many cases, the candidate solutions can be selected based on a simple counting argument, see *combinatorial filter* in (BES11), and thus our inclusion predicate is not supposed to be used. Hence, the only expected contributors to the running time are the resultant, GCD and univariate root isolation. We note, however, that for other competing approaches this is not necessarily the case because the idea of the combinatorial filter is based on a clever combination of the bitstream Descartes root isolator (EKK+05) and simultaneous root refinement. Table 5.3 shows the running times. Unfortunately, we did not manage to run Isolate on these instances as the algorithm was constantly reporting 'segment violation' error. Nevertheless, we can see that Bisolve clearly outperforms Lgp for all examples, even if no GPU support is available (see the column 'Bs' in the table). Using the GPU for resultant and GCD computations yields an additional 2–4x speedup. Besides, observe that the GPU-based GCD algorithm performs significantly better than in the first part of experiments, giving alone about 2x speed-up over Ntl's algorithm. The reason for this is quite natural because polynomials with large coefficients need more homomorphic images for the GCD computation which increases the level of parallelism.

In summary, we see that the GPU algorithms perform well not only in synthetic benchmarks from the previous chapter but also on real data. As a result, we can state that the symbolic operations no longer constitute a global bottleneck allowing us to solve more complicated problems which were previously beyond the reach of traditional software.

5.2 Complexity analysis of Bisolve

To derive the bit complexity of the algorithm, we assume that f and g have total degree at most n and the scalar coefficients bounded by 2^τ in absolute value, $\tau \in \mathbb{N}$. For convenience, we shall also write that the polynomials have *magnitude* (n, τ). Throughout the complexity analysis, we assume that the multiplication of two integers can always be done in *asymptotically* fast way. That is, the bit complexity of multiplying two k-bit integers will be bounded by: $\mathcal{M}_{\mathcal{B}}(k) = O(k \log k \log\log k)$. Besides, in our derivations, we shall not take into account polylogarithmic factors in n or τ and write $\tilde{O}$ to denote such a complexity bound.

The whole argument will follow a similar outline as the one used for presenting the algorithm itself in the previous section: we first derive the complexity of three stages separately, and then combine the bounds afterwards.

5.2.1 Project and Separate phases

Project : The algorithm starts by computing the resultants $R^{(x)}$ and $R^{(y)}$. For this task, we utilize an asymptotically fast subresultant algorithm based on Half-GCD computation whose complexity is stated by the following theorem.

Theorem 5.2.1: (Rei97) Let $F, G \in \mathbb{Z}[x, y]$ be polynomials with scalar coefficients bounded by 2^μ, and $\deg_y(F) = p$, $\deg_y(G) = q$, $p \geq q$ and $\deg_x(F) \leq d$, $\deg_x(G) \leq d$. Then, computing the coefficients of $\mathrm{res}_y(F, G) \in \mathbb{Z}[x]$ requires

$$O(p \log p \cdot \mathcal{M}_{\mathcal{B}}\{\mu(p + q)^2 d\})$$

bit operations. ◊

Thus, computing $R^{(x)}$ and $R^{(y)}$ demands for $\tilde{O}(n^4\tau\log n)$ bit operations, and the resulting polynomials have magnitude

$$(n^2, O(n(\log n + \tau))).$$

Next, we compute $R/\gcd(R, R')$ to extract a square-free part R^* of R. This operation has a bit complexity $\tilde{O}(n^5(\tau + \log n))$, see (Rei97, LR01), R^* is of magnitude

$$(n^2, O(n(n + \tau))). \tag{5.11}$$

Finally, we isolate the real roots α_i of R^* using the algorithm Newdsc already mentioned in Section 5.1.2. By Theorem 5.1.1 the cost of real root isolation for R^* is bounded by

$$\tilde{O}(n^8 + n^7\tau) \tag{5.12}$$

which determines the complexity of the first step. We further remark that the same complexity bound can be achieved by using an asymptotically fast numerical solver, e.g., see (Pan97), to approximate all complex roots of R^*.

Separate : Before proceeding with the complexity analysis of Separate, we prove an auxiliary result to upper bound $\Sigma(F) = \sum_z \log\operatorname{sep}(z, F)^{-1}$ for a (not necessarily square-free) polynomial F with magnitude (N, μ), where the sum is taken over all roots of F counted with multiplicity. This result might be of independent interest as we are not aware of any similar bounds which apply to polynomials with multiple roots.

Theorem 5.2.2: Let $F \in \mathbb{Z}[x]$ be a polynomial of magnitude (N, μ). We denote $z_1, \ldots, z_d$ the distinct complex roots of F and $s_i := \operatorname{mult}(z_i, F)$ the multiplicity of z_i. Then, for arbitrary non-negative integers m_i, with $m_i \le s_i$, we have

$$\sum_{i=1}^{d} m_i \log\operatorname{sep}(z_i, F)^{-1} = \tilde{O}(N^2 + N\mu).$$

◇

Proof We consider the factorization of F (over $\mathbb{Z}$) into square-free and pair-wise coprime factors:

$$F(x) = \prod\nolimits_{i=1}^{k} Q_i(x)^{s_i}, \quad d_i := \deg(Q_i) \ge 1,$$

so that $Q_i(x)$ and $F(x)/Q_i(x)^{s_i}$ are coprime, and $\sum_{i=1}^{k} d_i s_i = N$. We further denote F^* the square-free part of F and $d := \deg(F^*) = \sum_{i=1}^{k} d_i$ its degree. Then, for two arbitrary roots α and β of F^*, it holds that

$$\begin{aligned}|(F^*)'(\alpha)| &= |\operatorname{lcf}(F^*)| \cdot |\alpha - \beta| \prod_{\gamma \ne \alpha,\beta} |\gamma - \alpha| \le |\operatorname{lcf}(F^*)| \cdot |\alpha - \beta| \prod_{\gamma \ne \alpha,\beta} 2\max\{1, |\alpha|, |\gamma|\} \\ &\le 2^{d-2}|\alpha - \beta| \max\{1, |\alpha|\}^{d-3}\mathcal{M}(F^*)\end{aligned}$$

since $\mathcal{M}(F^*) = |\operatorname{lcf}(F^*)| \cdot \prod_{z:F^*(z)=0} \max\{1, |z|\}$. Suppose, w.l.o.g., that α is a root of Q_i and β is a root of F^* closest to α. Then, according to the above inequality, we have

$$\operatorname{sep}(\alpha, F) = |\alpha - \beta| \ge \frac{|(F^*)'(\alpha)|}{2^{d-2}\max\{1, |\alpha|\}^{d-3}\mathcal{M}(F^*)}$$

We now apply this inequality to the product over all $\operatorname{sep}(\alpha_j, F)$, $j = 1, \dots, d_i$, where $\alpha_1, \dots, \alpha_{d_i}$ denote the roots of Q_i:

$$\begin{aligned}\prod_{j=1}^{d_i} \operatorname{sep}(\alpha_j, F) &\geq 2^{(2-d)d_i} \mathcal{M}(Q_i)^{3-d} \mathcal{M}(F^*)^{-d_i} \prod_{j=1}^{d_i} |(F^*)'(\alpha_j)| \\ &= 2^{(2-d)d_i} \mathcal{M}(Q_i)^{3-d} \mathcal{M}(F^*)^{-d_i} \prod_{j=1}^{d_i} |(Q_i)'(\alpha_j) \cdot \frac{F^*}{Q_i}(\alpha_j)|\end{aligned} \tag{5.13}$$

since $(F^*)'(\alpha_j) = \underbrace{Q_i(\alpha_j)}_{=0} \cdot \left(\frac{F^*}{Q_i}\right)' (\alpha_j) + (Q_i)'(\alpha_j) \cdot \frac{F^*}{Q_i}(\alpha_j)$. In addition, we have

$$\prod_{j=1}^{d_i} |Q_i'(\alpha_j)| = |\operatorname{lcf}(Q_i)^{2-d_i} \operatorname{Disc}(Q_i)| \geq |\operatorname{lcf}(Q_i)^{2-d_i}|, \text{ and}$$

$$\prod_{j=1}^{d_i} |\frac{F^*}{Q_i}(\alpha_j)| = |\operatorname{lcf}(Q_i)^{d_i-d} \operatorname{res}(Q_i, \frac{F^*}{Q_i})| \geq |\operatorname{lcf}(Q_i)^{d_i-d}|$$

because $\operatorname{Disc}(Q_i)$ and $\operatorname{res}(Q_i, \frac{F^*}{Q_i})$ are non-zero integers. Applying the latter two inequalities to (5.13) now yields:

$$\prod_{j=1}^{d_i} \operatorname{sep}(\alpha_j, F) \geq 2^{(2-d)d_i} \mathcal{M}(Q_i)^{3-d} \mathcal{M}(F^*)^{-d_i} |\operatorname{lcf}(Q_i)^{2-d}|.$$

Finally, we consider the product of the separations of all roots to the respective powers s_i:

$$\begin{aligned}\prod_{i=1}^{k} \prod_{j=1}^{d_i} \operatorname{sep}(\alpha_j, F)^{s_i} &\geq \prod_{i=1}^{k} 2^{(2-d)d_i s_i} \mathcal{M}(Q_i)^{(3-d)s_i} \cdot \mathcal{M}(F^*)^{-d_i s_i} \cdot \prod_{i=1}^{k} |\operatorname{lcf}(Q_i)|^{-s_i} \\ &= 2^{(2-d)N} \mathcal{M}(F)^{3-d} \mathcal{M}(F^*)^{-N} |\operatorname{lcf}(F)|^{-1} = 2^{-\tilde{O}(N^2 + N\mu)}\end{aligned}$$

since $\prod_{i=1}^{k} \mathcal{M}(Q_i)^{s_i} = \mathcal{M}(F)$ by the multiplicativity of the Mahler measure, and $\mathcal{M}(F^*) \leq \mathcal{M}(F) = 2^{\tilde{O}(\mu)}$. Hence, for the case $m_i = s_i$ for all $i = 1, \dots, d$, the claim eventually follows by taking the logarithm on both sides. Since for each root z of F, $\operatorname{sep}(z, F)$ is upper bounded by two times the maximal absolute value of all roots of F, we have $\operatorname{sep}(z, F) < 2^{\mu+2}$ according to the Cauchy root bound, see e.g. (Yap00). Thus, the claim also follows for arbitrary integers m_i with $0 \leq m_i \leq s_i$. ■

Let us now return to the analysis of Separate. In the projection step, we have already determined intervals $I := I(\alpha)$ which isolate the real roots α of R^*. Now, each I has to be refined until the inequality (5.4) holds. This ensures that $\Delta_{8r_I}(m_I)$ isolates $\alpha \in I$ from all other roots of R^*, and thus the value $LB(\alpha)$ as defined in (5.5) constitutes a lower bound for $|R(\alpha)|$ on the boundary of $\Delta(\alpha) = \Delta_{2r_I}(m_I)$. In each iteration, we approximate α to a certain number L of bits after the binary point. Then, we check whether the inequality (5.4) holds. If the latter inequality does not hold, we double L and proceed. According to (SY09, Lemma 2), the inequality

$$|(R^*)'(z) - \frac{3}{2} \sum\nolimits_{k \geq 2} \left| \frac{(R^*)^{(k)}(m_I)}{k!} \right| r^k > 0$$

succeeds for all $r < \operatorname{sep}(z_i, R^*)/(4n^4) \le \operatorname{sep}(z_i, R^*)/(4\deg(R^*)^2)$.[1] It follows that (5.4) is guaranteed to succeed for

$$r_I < \operatorname{sep}(z_i, R^*)/(32n^4) = \operatorname{sep}(z_i, R)/(32n^4).$$

Hence we have to approximate α to at most

$$2\log(32n^4/\operatorname{sep}(\alpha, R)) = O(\log(\operatorname{sep}(\alpha, R)^{-1} + \log n)$$

many bits after the binary point. According to Theorem 5.2.2, $\log\operatorname{sep}(\alpha, R)^{-1}$ is bounded by $O(n^4 + n^3\tau)$. Therefore, each real root α has to be refined to at most $\tilde{O}(n^4 + n^3\tau)$ many bits. Such a computation requires

$$\tilde{O}(n^4(n^4 + n^3\tau)) = \tilde{O}(n^8 + n^7\tau) \tag{5.14}$$

bit operations (for all real roots) due to (Sag11, Theorem 10) or alternatively, using (Pan97). It remains to estimate the cost for checking whether (5.4) holds. In order to do so, we first compute $(R^*)'(x + m_I)$, the Taylor expansion of $(R^*)'$ at $x = m_I$. Since m_I is a dyadic number that is representable by $O(n^2 + n\tau + \log\operatorname{sep}(\alpha, R)^{-1})$ many bits, the cost for this computation is bounded by

$$\tilde{O}(\deg(R^*)^2(n^2 + n\tau + \log\operatorname{sep}(\alpha, R)^{-1})) = \tilde{O}(n^4(n^2 + n\tau + \log\operatorname{sep}(\alpha, R)^{-1})),$$

where we use the asymptotically fast Taylor shifts (vzGG03). Then, we replace x by $8r_I$ yielding $(R^*)'(m_I + 8r_I x)$. This is equivalent to shifting the k-th (dyadic) coefficient of $f(m_I + x)$ by $k\log(8r_I)$ many bits. The resulting polynomial has dyadic coefficients of bitsize

$$O(n^2 + n\tau + n^2\log\operatorname{sep}(\alpha, R)^{-1}),$$

and the final evaluation demands for $O(n^2(n^2 + n\tau + n^2\log\operatorname{sep}(\alpha, R)^{-1}))$ many bit operations. Summing up over all real roots α of R thus yields the bound

$$\sum_{\alpha}\tilde{O}(n^4(n^2 + n\tau + \log\operatorname{sep}(\alpha, R)^{-1})) = \tilde{O}(n^8 + n^7\tau) \tag{5.15}$$

for the overall cost since there at most n^2 many real roots and $\Sigma(R^*) = \tilde{O}(n^4 + n^3\tau)$. Finally, by comparing (5.14) and (5.15) with (5.12), we conclude that the complexity of Separate is not worse than that of Project phase.

5.2.2 Validate phase

In the final stage, Validate, we have a set of candidate solutions C and corresponding disjoint polydiscs $\Delta(\alpha,\beta) := \Delta(\alpha) \times \Delta(\beta) \subset \mathbb{C}^2$. Each of the polydiscs contains at most one solution of (5.1), namely, (α,β). The actual solutions of the system are chosen from C based on the inclusion test from Theorem 5.1.3, while the other candidates are excluded using interval arithmetic. We split the complexity analysis of Validate in two parts: first, we derive a lower bound $LB(\alpha)$ for $|R|$ on the boundary of $\Delta(\alpha)$ as well as an upper bound

[1] In (SY09, Lemma 2), a constant is $\sqrt{2}$ is given instead of 3/2. However, the same proof also applies to the “3/2-case”.

for the values of $|u^{(y)}|$ and $|v^{(y)}|$ on $\Delta(\alpha,\beta)$ as needed by the inclusion predicate. In the second part, we estimate how good each candidate (α,β) must be approximated in order to certify it as a solution or to discard it.

Estimating the lower bounds. We first compute lower and upper bounds for $LB(\alpha) = 2^{-2\deg R}|R(m_I - 2r_I)|$ which, in turn, constitutes a lower bound for the values of $|R(z)|$ on the boundary of the disc $\Delta(\alpha) := \Delta_{2r_I}(m_I)$, where $I := I(\alpha)$ is the isolating interval for α obtained in Separate; then, the similar bounds also apply to $LB(\beta)$, the lower bound for $|R^{(x)}|$ on the boundary of $\Delta(\beta)$, see Section 5.1.2 (Separate).

In the analysis of Separate, we have already argued that approximating α to an error of $\text{sep}(\alpha,R)/(32n^4)$ or less guarantees that the inequality (5.4) holds, and thus the disc $\Delta_{8r_I}(m_I)$ isolates α. In each iteration of the refinement, we double the number of bits to which α is approximated and check whether (5.4) holds. Hence, it follows that the so-obtained interval $I(\alpha)$ has width $w_I > (\text{sep}(\alpha,R)/(32n^4))^2$. In addition, since the disc $\Delta_{8r_I}(m_I)$ isolates α, we have $w_I < \text{sep}(\alpha,R)/7$. We fix these bounds for w_I:

$$\frac{\text{sep}(\alpha,R)^2}{1024n^8} < w_I \le \frac{\text{sep}(\alpha,R)}{7}. \tag{5.16}$$

Let us now consider the factorization of R into linear factors, that is, $R(z) = \text{lcf}(R)\cdot\prod_{i=1}^{d}(z-z_i)^{s_i}$, where $z_1,\dots,z_d$ denote the distinct complex roots of R and s_i the corresponding multiplicities. Then, with $\alpha = z_j$, we have

$$\frac{\text{sep}(z_j,R)}{4} > |(m_I - 2r_I) - z_j| > \frac{\text{sep}(z_j,R)^2}{2048n^8}, \text{ and } 2|z_j - z_i| > |(m_I - 2r_I) - z_i| > \frac{|z_j - z_i|}{2}$$

for all $i \neq j$. Hence, it follows that

$$\begin{aligned} LB(\alpha) &= LB(z_j) = 2^{-2\deg R}\cdot|R(m_I - 2r_I)| \\ &= 2^{-2\deg R}|\,\text{lcf}(R)|\cdot|(m_I - 2r_I) - z_j|^{s_j}\prod_{i\neq j}|(m_I - 2r_I) - z_i|^{s_i} \\ &< 2^{-2\deg R}|\,\text{lcf}(R)|\cdot(\text{sep}(z_j,R)/4)^{s_j}\prod_{i\neq j}|2(z_j - z_i)|^{s_i} \\ &< \text{sep}(z_j,R)^{s_j}\cdot|\,\text{lcf}(R)|\cdot\prod_{i\neq j}|z_j - z_i|^{s_i} < \text{sep}(z_j,R)^{s_j}\frac{|R^{(s_j)}(z_j)|}{s_j!} \\ &= 2^{O(n^2+n\tau)}\max\{1,|z_j|\}^{n^2}\,\text{sep}(z_j,R)^{s_j} = 2^{O(s_j(n^2+n\tau))}\max\{1,|z_j|\}^{n^2} \end{aligned} \tag{5.17}$$

since $R^{(s_j)}/(s_j!) \in \mathbb{Z}[x]$ has magnitude $(n^2, n(n+\tau))$, and $\text{sep}(z_j,R) < 2\max_i|z_i| = 2^{O(n(n+\tau))}$ according to Cauchy's Bound. We can also compute a lower bound for $LB(\alpha)$:

$$\begin{aligned} LB(\alpha) &> 2^{-2\deg R}|\,\text{lcf}(R)|\cdot\left(\frac{\text{sep}(z_j,R)^2}{2048n^8}\right)^{s_i}\prod_{i\neq j}\left(\frac{|z_j - z_i|}{2}\right)^{s_i} \\ &> \frac{2^{-3\deg R}}{(2048n^8)^{s_i}}\cdot|\,\text{lcf}(R)|\,\text{sep}(z_j,R)^{2s_j}\prod_{i\neq j}|z_j - z_i|^{s_i}. \end{aligned} \tag{5.18}$$

Note that, we are mainly interested in a bound for the product of all $LB(\alpha)$, thus we first consider the product

$$\Pi := \prod_{j=1}^{d} \left(\frac{2^{-3\deg R}}{(2048n^8)^{s_i}} \cdot |\operatorname{lcf}(R)| \operatorname{sep}(z_j, R)^{2s_j} \prod_{i \neq j} |z_j - z_i|^{s_i} \right)$$

of the bound in (5.18) over all $j = 1, \ldots, d$. Since $\sum_i s_i = d \leq \deg R \leq n^2$, it follows that

$$\prod_{j=1}^{d} \frac{2^{-3\deg R}}{(2048n^8)^{s_i}} = 2^{-O(n^4)}.$$

For the product of the remaining factors, we first write $R = \prod_{s=1}^{s_0} Q_s^s$ with square-free, pairwise coprime $Q_s \in \mathbb{Z}[x]$. Since $R^{(s)}/s!$ has integer coefficients, we have

$$1 \leq |\operatorname{res}(Q_s, \frac{R^{(s)}}{s!})| = |\operatorname{lcf}(Q_s)|^{\deg(R)-s} \prod_{z:Q_s(z)=0} R^{(s)}(z),$$

and thus

$$\begin{aligned}
&\prod_{j=1}^{d} \left(|\operatorname{lcf}(R)| \operatorname{sep}(z_j, R)^{2s_j} \prod_{i \neq j} |z_j - z_i|^{s_i} \right) \\
&> |\operatorname{lcf}(R)|^d 2^{-2\Sigma(R)} \prod_j \prod_{i \neq j} |z_i - z_j|^{s_j} = 2^{-2\Sigma(R)} \prod_j \frac{|R^{(s_j)}(z_i)|}{s_j!} \\
&= 2^{-2\Sigma(R)} \prod_{s=1}^{s_0} |\operatorname{lcf}(Q_s)|^{s-\deg(R)} |\operatorname{res}(Q_s, \frac{R^{(s)}}{s!})| \\
&> 2^{-2\Sigma(R)} |\operatorname{lcf}(R)| \cdot |\operatorname{lcf}(R^*)|^{-\deg(R)} = 2^{-\tilde{O}(n^4+n^3\tau)},
\end{aligned}$$

where we used that $|\operatorname{lcf}(R)| \leq 2^{O(n(\log n+\tau))}$, $\deg R \leq n^2$, and $\Sigma(R) = \tilde{O}(n^4 + n^3\tau)$. Hence, Π is lower bounded by $2^{-\tilde{O}(n^4+n^3\tau)}$. Similar to the computation in (5.17), we can also determine an upper bound for the j-th factor in Π. Namely, we have $2^{-3\deg R}(2048n^8)^{-s_j} < 1$, $\operatorname{sep}(z_j, R)^{s_j} = 2^{O(s_j n(\log n+\tau))}$ and

$$\operatorname{lcf}(R) \prod_{i \neq j} |z_j - z_i|^{s_i} = \frac{|R^{(s_j)}(z_j)|}{s_j!} < 2^{O(n(\log n+\tau))} \max\{1, |z_j|\}^{n^2}.$$

Thus, for an arbitrary subset $J \subset \{1, \ldots, d\}$, the partial product

$$\Pi' := \prod_{j \in J} \left(\frac{2^{-3\deg R}}{(2048n^8)^{s_i}} \cdot |\operatorname{lcf}(R)| \operatorname{sep}(z_j, R)^{2s_j} \prod_{i \neq j} |z_j - z_i|^{s_i} \right)$$

is smaller than

$$2^{O(n^4+n^3\tau)} \prod_{j \in J} \max\{1, |z_j|\}^n = 2^{O(n^4+n^3\tau)}$$

since $\prod_{j \in J} \max\{1, |z_j|\}^n \leq \mathcal{M}(R) = 2^{O(n(\log n+\tau))}$. Finally, since the product over all $LB(\alpha)$ is lower bounded by a partial product of Π, it follows that $\prod_\alpha LB(\alpha) = 2^{-\tilde{O}(n^4+n^3\tau)}$. The same argument further shows that each $LB(\alpha)$ is lower bounded by $2^{-\tilde{O}(n^4+n^3\tau)}$ as well.

Estimating the upper bounds. In order to compute the upper bounds $UB(\alpha,\beta,u^{(y)})$ and $UB(\alpha,\beta,v^{(y)})$ for $|u^{(y)}|$ and $|v^{(y)}|$ on $\Delta(\alpha,\beta)$ we apply Hadamard's inequality to the matrices $U^{(y)}$ and $V^{(y)}$, see Section 5.1.2 (Validate). By analogy, these estimates then also extend to the upper bounds $UB(\alpha,\beta,u^{(x)})$ and $UB(\alpha,\beta,v^{(x)})$ for $|u^{(x)}|$ and $|v^{(x)}|$ on $\Delta(\alpha,\beta)$.

In the actual realization, we use interval arithmetic for a box in $\mathbb{C}^2$ which contains $\Delta(\alpha,\beta)$ in order to estimate the absolute values of the respective matrix entries U_{ij} and V_{ij}, and then apply Hadamard's bound. For the complexity analysis, we follow a slightly different but even simpler approach: From the construction of $\Delta(\alpha,\beta)$, the disc $\Delta(\alpha)$ has radius less than $\operatorname{sep}(\alpha,R^{(y)})/4$, and $\Delta(\beta)$ has radius less than $\operatorname{sep}(\beta,R^{(x)})/4$ according to (5.16). Hence, the latter two radii are upper bounded by $2\max\{1,|\alpha|\}$ and $2\max\{1,|\beta|\}$, respectively. Recall that the matrix $U^{(y)}$ is of the form:

$$U^{(y)} = \begin{vmatrix} f^{(y)}_{m_y} & f^{(y)}_{m_y-1} & \cdots & f^{(y)}_0 & 0 \;\ldots & y^{n_y-1} \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \ldots\; 0 & f^{(y)}_{m_y} & f^{(y)}_{m_y-1} & \cdots & 1 \\ g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \cdots & g^{(y)}_0 & 0 \;\ldots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \vdots \\ 0 & \ldots\; 0 & g^{(y)}_{n_y} & g^{(y)}_{n_y-1} & \cdots & 0 \end{vmatrix},$$

where the polynomials $f_i^{(y)}(x)$ and $g_i^{(y)}(x)$ are of magnitude (n,τ) (see Section 5.1.1). Thus, for each point $(\hat{x},\hat{y}) \in \Delta(\alpha,\beta)$, the following inequality holds:

$$|f_i^{(y)}(\hat{x})| \leq (n+1) \cdot 2^{\tau}(2\max\{1,|\alpha|\})^n,$$

and a similar bound applies to $|g_i^{(y)}(\hat{x})|$ as well. For the last column of $U^{(y)}$, we have: $(\hat{y})^{n_y-1} \leq (2\max\{1,|\beta|\})^n$, and thus $|U_{ij}^{(y)}(\hat{x},\hat{y})| \leq (n{+}1)\cdot 2^{\tau+n}\max\{1,|\alpha|,|\beta|\}^n$. By Hadamard's inequality, $|u^{(y)}| = |\det(U^{(y)})| < \prod_i |U_i^{(y)}|_2$ where $|U_i^{(y)}|_2$ is the 2-norm of the i-th row vector of $U^{(y)}$. Hence, when using the latter bounds for the entries of $U^{(y)}$, we obtain an upper bound $UB(\alpha,\beta,u^{(y)})$ for $|u^{(y)}|$ on the polydisc $\Delta(\alpha,\beta)$, such that $UB(\alpha,\beta,u^{(y)}) \geq 1$ and

$$\begin{aligned} \log|UB(\alpha,\beta,u^{(y)})| &= O(n(\tau+n) + n^2\log\max\{1,|\alpha|,|\beta|\}) \\ &= O(n^4 + n^3\tau). \end{aligned} \tag{5.19}$$

Again, we are looking for amortization effects: Taking the product of the latter bounds over all candidates (α,β) yields:

$$\begin{aligned} \sum_{\alpha,\beta} \log UB(\alpha,\beta,u^{(y)}) &= \sum_{\alpha,\beta} O(n^2+n\tau) + \sum_{\alpha,\beta} n^2 \log\max\{1,|\alpha|,|\beta|\} \\ &\leq O(n^6+n^5\tau) + n^2\sum_{\beta}\sum_{\alpha}\log\max\{1,|\alpha|\} + n^2\sum_{\alpha}\sum_{\beta}\log\max\{1,|\beta|\}) \\ &\leq O(n^6+n^5\tau) + n^2\log\mathcal{M}(R^{(y)}) + n^2\log\mathcal{M}(R^{(x)}) = \tilde{O}(n^6+n^5\tau) \end{aligned} \tag{5.20}$$

because there are at most n^2 many roots α and β. A completely similar argument shows that the bounds in (5.19) and (5.20) are also valid for $UB(\alpha,\beta,v^{(y)})$, $UB(\alpha,\beta,u^{(x)})$ and

$UB(\alpha,\beta,v^{(x)})$.

The inclusion test. For a given candidate $(\alpha,\beta) \in C$ and $\mathcal{B} := B(\alpha,\beta) = I(\alpha) \times I(\beta) \subset \mathbb{R}^2$ the corresponding candidate box, we define

$$\delta(\mathcal{B}) := \frac{\min(LB(\alpha), LB(\beta))}{\max_{w\in\{u^{(x)},u^{(y)},v^{(x)},v^{(y)}\}} UB(\alpha,\beta,w)}.$$

From the bounds that we have computed in the previous section, we conclude that $\log \delta(\mathcal{B})^{-1} = \tilde{O}(n^4 + n^3\tau)$. According to Theorem 5.1.3, $\mathcal{B}$ is isolating for a solution of (5.1) if and only if there exists an $(x_0, y_0) \in \mathcal{B}$ with

$$|f(x_0, y_0)| + |g(x_0, y_0)| < \delta(\mathcal{B}). \tag{5.21}$$

Hence, by contraposition, we must have

$$|f(x_0, y_0)| + |g(x_0, y_0)| \geq \delta(\mathcal{B}) \tag{5.22}$$

for all $(x_0, y_0) \in \mathcal{B}$ if $\mathcal{B}$ contains no solution. In order to certify or discard (α,β) as a solution of the system, we evaluate f and g on $\mathcal{B}$ using *interval arithmetic* with precision $\rho := \rho(\mathcal{B}) = \lceil -\log s \rceil$, where $s := \max\{w_{I(\alpha)}, w_{I(\beta)}\}$ is the size of $\mathcal{B}$. As a result of this evaluation, we obtain intervals $\mathfrak{B}(f(\alpha,\beta),\rho)$ and $\mathfrak{B}(g(\alpha,\beta),\rho)$ which contain $f(\mathcal{B})$ and $g(\mathcal{B})$, respectively. The above consideration shows that it suffices to use a precision ρ such that both intervals $\mathfrak{B}(f(\alpha,\beta),\rho)$ and $\mathfrak{B}(g(\alpha,\beta),\rho)$ have width less than $\delta(\mathcal{B})/2$. Namely, if this happens, then either one of the intervals does not contain zero or we must have $|f(x_0, y_0)| + |g(x_0, y_0)| < \delta(\mathcal{B})$ for all $(x_0, y_0) \in \mathcal{B}$. In the first case, we can discard (α,β), whereas, in the second case, we can guarantee that (α,β) is a solution.

The width of $\mathfrak{B}(f(\alpha,\beta),\rho)$ (and $\mathfrak{B}(g(\alpha,\beta),\rho)$) is directly related to the absolute error induced by the interval arithmetic. In order to bound this error, we briefly outline how the interval arithmetic is performed and refer the reader to (KS11a, Section 4) for more details; cf. (MOS11, Theorem 18) for an alternative approach when using floating point evaluation instead. For a precision $\rho \in \mathbb{N}$ and $x \in \mathbb{R}$, we define:

$$\begin{aligned} \text{down}(x,\rho) &= \{k \cdot 2^{-\rho} \in \mathbb{R} : k = \lfloor x \cdot 2^{\rho} \rfloor\}, \\ \text{up}(x,\rho) &= \{k \cdot 2^{-\rho} \in \mathbb{R} : k = \lceil x \cdot 2^{\rho} \rceil\}. \end{aligned} \tag{5.23}$$

That is, x is included in the interval $\mathfrak{B}(x,\rho) := [\text{down}(x,\rho), \text{up}(x,\rho)]$. For simplicity, we omit the precision parameter ρ and write $\text{up}(x)$ or $\mathfrak{B}(x)$. Arithmetic operations on approximate numbers obey the rules of classical interval arithmetic; for $x, y \in \mathbb{R}$, we define:

$$\begin{aligned} \mathfrak{B}(x) + \mathfrak{B}(y) &:= [\text{down}(x) + \text{down}(y), \text{up}(x) + \text{up}(y)], \\ \mathfrak{B}(x) - \mathfrak{B}(y) &:= [\text{down}(x) - \text{up}(y), \text{up}(x) - \text{down}(y)], \\ \mathfrak{B}(x) \cdot \mathfrak{B}(y) &:= \left[\text{down}(\min_{i,j=\{1,2\}}\{H_i(x)H_j(y)\}), \text{up}(\max_{i,j=\{1,2\}}\{H_i(x)H_j(y)\})\right], \end{aligned}$$

where $H_1(x) = \text{down}(x)$, and $H_2(x) = \text{up}(x)$. Using these rules for $F \in \mathbb{R}[x]$ and $x_0 \in \mathbb{R}$, $\mathfrak{B}(F(x_0),\rho)$ can be evaluated using the Horner's scheme:

$$\mathfrak{B}(F(x_0)) = \mathfrak{B}(F_0) + \mathfrak{B}(x_0) \cdot (\mathfrak{B}(F_1) + \mathfrak{B}(x_0) \cdot (\mathfrak{B}(F_2) + \ldots)).$$

The next lemma provides a bound on the error that is induced by polynomial evaluation with precision ρ.

Lemma 5.2.1: Let $F \in \mathbb{R}[x]$ be a polynomial of degree N with coefficients of absolute value less than 2^{μ}, $c \in \mathbb{R}$ with $|c| \leq 2^{\nu}$, and $\rho \in \mathbb{N}$. Then,

$$|F(c) - H(F(c), \rho)| \leq 2^{-\rho+1} 2^{\mu} 2^{N\nu} (N+1)^2,$$

where $H = \{\text{down}, \text{up}\}$. In particular, $\mathfrak{B}(F(c), \rho)$ has width $2^{-\rho+2}(N+1)^2 2^{\mu+N\nu}$ or less. For a proof, see (KS11a, Lem. 3). ◇

In particular, this lemma asserts that the absolute error which results from approximate polynomial evaluation is *linear* in $2^{-\rho}$ and *of degree n* in the absolute value of the input.

We now aim to bound the error for evaluating $f(\alpha,\beta)$ using fixed-point arithmetic with precision ρ. Using Lemma 5.2.1, we obtain the following estimate for each coefficient $f_i(\alpha)$ of $f(\alpha, y)$:

$$|f_i(\alpha) - \mathfrak{B}(f_i(\alpha), \rho)| \leq 2^{-\rho+1}(n+1)^2 2^{\tau} \max\{1, |\alpha|\}^n,$$

while the maximal magnitude of the values $\mathfrak{B}(f_i(\alpha), \rho)$ is estimated as $2^{\tau} \max\{1, |\alpha|\}^n$. Applying Lemma 5.2.1 second time, we conclude evaluating $f(\alpha,\beta)$ with precision ρ induces an absolute error of less than

$$2^{-\rho+1}(n+1)^2 2^{\tau} \max\{1, |\alpha|, |\beta|\}^n.$$

Thus, the width of $\mathfrak{B}(f(\alpha,\beta), \rho)$ is bounded by $2^{-\rho+2}(n+1)^2 2^{\tau} \max\{1, |\alpha|, |\beta|\}^n$. The same bound also applies to $\mathfrak{B}(g(\alpha,\beta), \rho)$. It follows that our inclusion/exclusion test must succeed for any precision ρ less than

$$\rho(\mathcal{B}) := \log(8(n+1)^2 2^{\tau} \max\{1, |\alpha|, |\beta|\}^n \delta(\mathcal{B})^{-1})$$

because, then, both intervals $\mathfrak{B}(f(\alpha,\beta), \rho)$ and $\mathfrak{B}(f(\alpha,\beta), \rho)$ have width less than $\delta(\mathcal{B})/2$. Since we double the working precision ρ in each step, we eventually succeed with

$$\rho < 2\rho(\mathcal{B}) = O(\log n + \tau + n \log \max\{1, |\alpha|, |\beta|\} - \log \delta(\mathcal{B})) = \tilde{O}(n^4 + n^3\tau).$$

In addition, we have to refine the isolating intervals $I(\alpha)$ and $I(\beta)$ to a width $2^{-\rho} = 2^{-\tilde{O}(n^4+n^3\tau)}$. In our analysis of Separate, we have already seen that refining the isolating intervals for all real roots of $R^{(y)}$ (and $R^{(x)}$) to a width of $2^{-\tilde{O}(n^4+n^3\tau)}$ demands for $\tilde{O}(n^8 + n^7\tau)$ many bit operations. It remains to bound the cost for evaluating $\mathfrak{B}(f(\alpha,\beta), \rho)$ and $\mathfrak{B}(f(\alpha,\beta), \rho)$: Due to the fact that we have to perform $O(n^2)$ many multiplications and additions with dyadic numbers whose binary representations need $O(\tau + n \log \max\{1, |\alpha|, |\beta|\} - \delta(\mathcal{B}(\alpha,\beta)))$ many bits, the latter computation requires

$$\tilde{O}(n^2\{\tau + n \log \max\{1, |\alpha|, |\beta|\} - \rho(\mathcal{B}(\alpha,\beta))\}) \tag{5.24}$$

many bit operations. Hence, for the bit complexity of the polynomial evaluations at all (α,β), we obtain the bound

$$\begin{aligned} &\sum_{\alpha,\beta} \tilde{O}(n^2\{\tau + n \log \max\{1, |\alpha|, |\beta|\} - \delta(\mathcal{B}(\alpha,\beta))\}) \\ &\quad = \tilde{O}(n^6\tau + n^3 \sum_{\alpha,\beta} \log \max\{1, |\alpha|, |\beta|\} - n^2 \sum_{\alpha,\beta} \delta(\mathcal{B}(\alpha,\beta))) \\ &\quad = \tilde{O}(n^7 + n^6\tau - n^2 \sum_{\alpha,\beta} \delta(\mathcal{B}(\alpha,\beta))), \end{aligned} \tag{5.25}$$

where we use the same argument as in (5.20) to bound the sum of all $\log\max\{1,|\alpha|,|\beta|\}$. The following computation further shows that $-\sum_{\alpha,\beta}\log\delta(\mathcal{B}(\alpha,\beta)) = \tilde{O}(n^6+n^5\tau)$: Using the upper bound (5.17) for $LB(\alpha)$ and $LB(\beta)$ yields

$$\begin{aligned}\log(\min(LB(\alpha),LB(\beta)))^{-1} &\leq \log LB(\alpha)^{-1} + \log LB(\beta)^{-1}\\ &+ 2n^2\cdot\log\max\{1,|\alpha|,|\beta|\} + O((s_\alpha+s_\beta)(n^2+n\tau)),\end{aligned}$$

where s_α denotes the multiplicity of α as a root of $R^{(y)}$, and s_β the multiplicity of β as a root of $R^{(x)}$. Hence, the bound $\tilde{O}(n^6+n^5\tau)$ for the sum over all $\log(\min(LB(\alpha),LB(\beta)))^{-1}$ follows from

$$\begin{aligned}&\sum_{\alpha,\beta}\log LB(\alpha)^{-1}+\log LB(\beta)^{-1} = \sum_{\beta}\sum_{\alpha}\log LB(\alpha)^{-1}+\\ &\quad+\sum_{\alpha}\sum_{\beta}\log LB(\beta)^{-1} \leq -n^2(\sum_{\alpha}\log LB(\alpha)+\sum_{\beta}\log LB(\beta))\\ &\quad= -n^2(\log\prod_{\alpha}LB(\alpha)+\log\prod_{\beta}LB(\beta)) = \tilde{O}(n^6+n^5\tau),\end{aligned}$$

and

$$\begin{aligned}&\sum_{\alpha,\beta}2n^2\log\max\{1,|\alpha|,|\beta|\} + O((s_\alpha+s_\beta)(n^2+n\tau))\\ &= \tilde{O}(n^6+n^5\tau+(n^4+n^3\tau)\cdot(\sum_{\alpha}s_\alpha+\sum_{\beta}s_\beta)) = \tilde{O}(n^6+n^5\tau).\end{aligned}$$

In addition, the result from (5.20) shows that

$$\begin{aligned}\sum_{\alpha,\beta}\log\max_{w\in\{u^{(x)},u^{(y)},v^{(x)},v^{(y)}\}}UB(\alpha,\beta,w) &\leq \sum_{\alpha,\beta}\log UB(\alpha,\beta,u^{(x)}) + \sum_{\alpha,\beta}\log UB(\alpha,\beta,u^{(y)})\\ +\sum_{\alpha,\beta}\log UB(\alpha,\beta,v^{(x)}) + \sum_{\alpha,\beta}\log UB(\alpha,\beta,v^{(y)}) &= \tilde{O}(n^6+n^5\tau).\end{aligned} \tag{5.26}$$

Thus, the claimed bound for $-\sum_{\alpha,\beta}\log\delta(\mathcal{B}(\alpha,\beta))$ follows from our definition of $\delta(\mathcal{B}(\alpha,\beta))$. Substituting this into (5.25), we obtain

$$\tilde{O}(n^8+n^7\tau) \tag{5.27}$$

which bounds the overall bit complexity of polynomial evaluations.

Overall complexity and concluding remarks. Finally, combining the complexity bounds (5.12), (5.15) and (5.27) for the three stages of the algorithm, we conclude that

$$\tilde{O}(n^8+n^7\tau) \tag{5.28}$$

determines the total complexity of the algorithm. To the best of our knowledge, the latter bound is a major step forward in terms of improving asymptotic complexity for this fundamental task. We would also like to stress the fact that the new complexity bound is not only the merit of improved asymptotic complexity of real root isolation and root refinement but also due to the use of the novel *validation procedure* which completely avoids any symbolic operations traditionally used in analogous algorithms (such as computing signed remainder sequences or SRs for short). This fact also finds a confirmation in the experiments (see Section 5.1.4) because, typically, evaluating the SRs constitutes a major performance bottleneck. To see how our bound compares to the previously known estimates, we shall briefly discuss several analogous works.

A quite early result on the complexity analysis can be found in (GVK96). This work analyzes the complexity of computing the topology of an algebraic curve – the problem closely related to that of solving a bivariate polynomial system. The derived complexity bound for the algorithm Top is $\tilde{O}(N^{14})$ bit operations where $N = \max(n, \tau)$.

Another paper (DET09) proposes three algorithms to solving a bivariate polynomial system which are based on the evaluation of SRs. The first method, Grid, projects the solutions onto orthogonal axes and, then, matches them by means of a Sign_at procedure. The complexity of Grid is bounded by $\tilde{O}(N^{14})$ bit operations, where the overall cost is dominated by that of Sign_at operations. The second approach called M_rur assumes that the system is in generic position, i.e., no two solutions share a common x-coordinate, and achieves a bit complexity of $\tilde{O}(n^{10}(n^2 + \tau^2)) = \tilde{O}(N^{12})$. The remaining approach, G_rur, has the same bit complexity as M_rur but relies on computing the GCDs of the square-free parts of $f(\alpha, y)$ and $g(\alpha, y)$, with α being a projected solution of a polynomial system. Although, the improved bounds for univariate real root isolation and refinement can lead to better overall complexity of M_rur (only in a sheared system) and G_rur, the so obtained results would be considerably weaker than those achieved by Bisolve. For example, the dependence on n in the final steps of M_rur and G_rur is by a factor n^2 larger when n is dominating. In addition, the analysis of the lifting step in G_rur is based on the study of a modular GCD algorithm over an extension field from (vHM02). To improve the total complexity, the authors of (DET09) propose to *augment* the original approach by assuming asymptotically fast arithmetic which, for instance, would imply the use of a *subquadratic time* Chinese remainder algorithm (CRA). However, upon a closer look, it appears that the algorithm (vHM02) applies CRA *incrementally* (along with trial division), in which case the asymptotically fast methods do not apply. In our complexity analysis we have tried to stay away from such speculative assumptions.

In his PhD thesis, M. Kerber describes randomized algorithms to analyze the topology of a single algebraic curve and to compute the arrangements of such curves. The latter problem can be regarded as a subproblem of solving a bivariate polynomial system. The detailed analysis shows that it can be solved in an *expected number* of $\tilde{O}(n^{10}(n + \tau)^2)$ bit operations; see (Ker09, Section 3.3.4). At the time of writing, a recent work (KS11b) improves upon the complexity of computing the topology of an algebraic curve. The paper analyzes a *deterministic* approach whose complexity evaluates to $\tilde{O}(n^8\tau(n + \tau))$ bit operations.

5.3 Rasterization of implicit algebraic curves

The problem of accurate visualization of implicitly defined algebraic curves in $\mathbb{R}^2$ has been studied for years. Such curves have a number of prominent features making them very attractive in many of scientific fields: for instance, to represent various geometric objects in a compact form. In this section, we describe an algorithm (EBS09) for geometrically correct visualization of algebraic curves, and discuss how to use the graphics hardware to greatly speed-up the visualization process. As it turned out, the main bottleneck of the original approach was computing a full Cylindrical Algebraic Decomposition (CAD) of an algebraic curve (see also Section 5.1.3). We show that, under some mild assumptions,[1] the algorithm can proceed by just considering the *projections* of all "topological events" (self-intersection and extremal points as to be defined later). The latter task can be easily achieved by the resultant and GCD computation (along with real root isolation), see also Section 5.1.2 (Project). Note that, the original algorithm has been integrated to the interactive web application available at `http://exacus.mpi-inf.mpg.de`, where we can visualize and explore the arrangements induced by 2D algebraic curves; see (EK08) for the video presentation about it.

In our discussion, we shall stick to the following outline: first, we state the problem in a mathematically formal way and shortly overview the existing methods for curve drawing. Then, we describe the algorithm itself omitting some technical details that are covered in the original paper. Finally, we run the algorithm on a number of examples that suppose to create challenges for visualization software: many of such can be found in (Lab10). To showcase the attained visual quality as well as the efficiency of parallel processing, we shall compare our algorithm with another visualization tools currently available.

5.3.1 Problem introduction and overview

A real algebraic plane curve can be defined as the zero set of a possibly reducible bivariate polynomial:

$$C = \{(x, y) \in \mathbb{R}^2 : f(x, y) = 0\}, \quad \text{where} \ \ f \in \mathbb{Z}[x, y].$$

Accordingly, the degree of an algebraic curve C is informally defined as the degree of the supporting polynomial f. Points $\mathbf{p} \in \mathbb{R}^2$ along the curve can be classified in two disjoint sets as *regular* and *non-regular*. Let $\nabla f = (f_x, f_y) \in (\mathbb{Z}[x, y])^2$ be a gradient vector of curve, where $f_x = \frac{\partial f}{\partial x}$ and $f_y = \frac{\partial f}{\partial y}$. Non-regular points are further divided into *x-critical* for which $f(\mathbf{p}) = f_y(\mathbf{p}) = 0$, *y-critical* when $f(\mathbf{p}) = f_x(\mathbf{p}) = 0$, and *singular* when $f(\mathbf{p}) = f_x(\mathbf{p}) = f_y(\mathbf{p}) = 0$. We define a *curve arc* as a single connected component of C which has no singularities in the interior and can be bounded by two non-regular endpoints (note that, a curve arc is not necessarily bounded). Additionally, an *x-monotone* curve arc is an arc having no x-critical points in the interior. Finally, an *isolated* singularity (or solitary point) is a singular point which has no curve arcs attached to it.[2]

As a task of a *geometrically correct* visualization of an algebraic curve we ask for a correct image of the given curve in a rectangular domain $\mathcal{D} \subset \mathbb{R}^2$ up to pixel level or, more precisely, we request a polyline connecting the colored pixels to lie within a given Hausdorff distance from the curve. In this sense, our notion of correct visualization is *weaker* than another natural one asking for visualization which reflects the whole curve topology. That is, in our case, we disregard any possible curve geometry in the regions which fall

[1] If we refrain from rasterizing isolated singularities, see Section 5.3.1.

[2] For instance, the one given by an implicit equation: $x^2 + (y - 3)^2 = 0$.

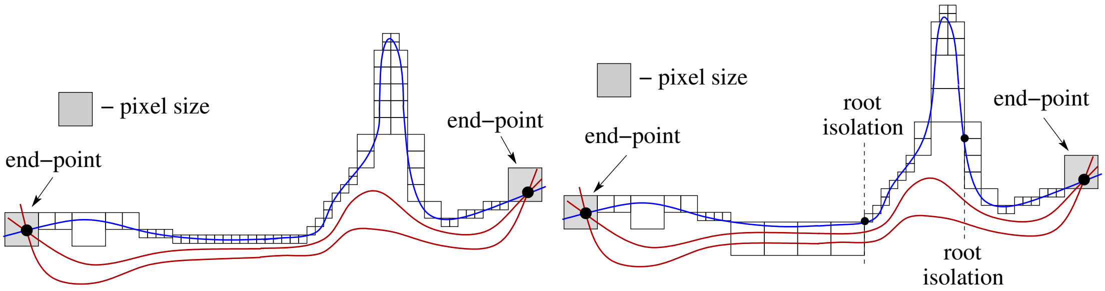

Figure 5.2: Curve arcs can pass very close to each other causing difficulties for accurate curve visualization even if the exact topology is computed (*left*); our method does not subdivide pixels once the direction of motion along the curve can be determined uniquely (*right*).

beyond the pixel level. Existing methods for curve rasterization can be attributed to one of the following families or a combination of them.

Space covering. These numerical methods are based on *interval arithmetic* to decide which parts of a rasterization domain $\mathcal{D}$ can be effectively discarded as not intersected by a curve and which require further subdivision. Classical algorithms can guarantee geometric correctness of the obtained rasterization but typically fail for singular curves. More recent works (AM07, BCGY12) subdivide the initial domain in a set of xy-regular boxes where the topology of a curve is known and a set of isolating boxes of size $\leq \varepsilon$ enclosing possible singularities. Yet, both algorithms have to reach the root separation bounds to certify the correctness of the output.

Continuation methods are efficient because only points surrounding a curve arc need to be considered. They typically find one or more seed points lying on a curve, and then follow the curve through adjacent pixels/plotting cells. Some algorithms consider a small pixel neighbourhood and obtain the next pixel based on sign evaluations (Cha88). Other approaches (MG04, RR05) use Newton-like iterations to compute the points along the curve. Continuation methods commonly break down at singularities or can identify only particular ones.

Symbolic methods use projection techniques to capture topological events (tangents and singularities) along a vertical line. This can be done by computing signed remainder sequences (EKW07) or Gröbner bases (CLP$^+$09a). While knowing the exact curve topology certainly makes the visualization process easier, it does not immediately lead to an efficient algorithm because symbolic methods do not account for the size of the rasterization domain $\mathcal{D}$ due to their "symbolic" nature: for example, it might happen that the curve arcs are very tightly packed in $\mathcal{D}$ making the rasterization prohibitively inefficient, see Figure 5.2 (left).

Our method inherits the ideas from all the above classes of rasterization algorithms. At the beginning, the solutions (α,β) of $f(x,y) = f_y(x,y) = 0$ are projected onto the x-axis by means of the resultant computation, partitioning the curve (implicitly) into x-monotone parts. Note that, we do not require computing the entire CAD since connectivity information between arcs is not important for visualization. Next, the algorithm computes the so-called *seed points* (point representatives) for each x-monotone arc of the curve. This is done by isolating the real roots of $f(x_0,y) \in \mathbb{Q}[x]$, where $x_0 \in \mathbb{Q}$ is chosen arbitrarily between every two projected solutions α and α'. Accordingly, the number of real roots along $x = x_0$ gives the total number of arcs for this x-monotone part. Having a seed

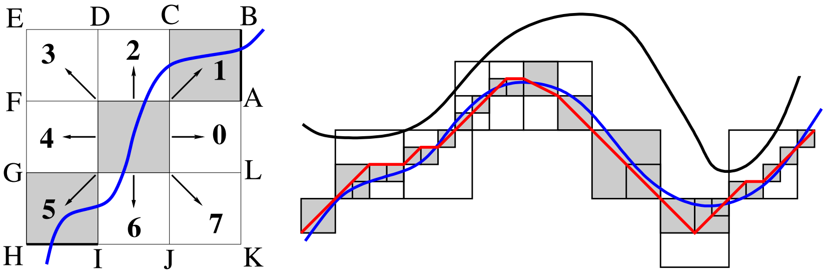

Figure 5.3: an 8-pixel stepping scheme with numbered directions, plotted pixels are shaded (*left*); adaptive approximation of a curve arc (lower one) by a polyline connecting shaded witness (sub-)pixels (*right*)

point, we proceed by tracing each arc *separately* in two opposite directions towards the end-points. In our hybrid approach, in contrast to (RR05), the roles of subdivision and curve-tracking are interchanged: that is, the curve arcs are traced in the original domain while subdivision is employed in tough cases. In each step we examine 8 neighbouring pixels of a current pixel and choose the one crossed by an arc. In case of a tie, the pixel is subdivided recursively into 4 parts. Local subdivision stops by reaching a certain threshold and when all curve arcs appear to leave a considered sub-pixel in one unique direction. From this point on, the arcs can be traced jointly until one of them goes apart. When it happens, we pick up a correct arc (the one which we rasterize at the moment) again by isolating the real roots at the pixel boundary, see Figure 5.2 (right).

According to our experiences, we can trace the majority of curves without resorting to exact computations even if root separation bounds are very tight. To handle exceptional cases, we switch to more accurate interval methods or increase the arithmetic precision.

5.3.2 Algorithm details

We begin with a high-level description of the algorithm which is based on the original work (Eme07), see also conference paper (EBS09). Through long-term practical experience, we have identified and applied a number of small optimizations aimed to improve the performance and numerical stability of the algorithm. In its core, the algorithm is based on an 8-way stepping scheme introduced in (Cha88), see Figure 5.3 (left). That is, from a current pixel we can step to one of its 8 neighbours (in 8 possible directions) depending on which part of the boundary of an *8-pixel neighbourhood* (a large rectangle *EBKH* in the figure) is cut by the curve arc. The decision is based on checking this boundary using interval arithmetic, see (EBS09, Section 2) for details.

The algorithm processes each x-monotone curve arc separately. The correctness of the rasterization is verified by introducing the notion of *witness* (sub-)pixel: that is, such a (sub-)pixel whose boundary is crossed only *twice* by the curve arc to be rasterized and is not crossed by any other arc. We implicitly assign a witness (sub-)pixel to each pixel in a curve approximation. Then, if we connect those (sub-)pixels by straight lines, we obtain a piecewise linear approximation of a curve arc which lies within a fixed Hausdorff distance from the actual curve image (vanishing locus), see Figure 5.3 (right).

The algorithm starts by picking up a seed point on a curve arc and covering it by a witness (sub-)pixel such that the arc leaves this (sub-)pixel in two different directions.

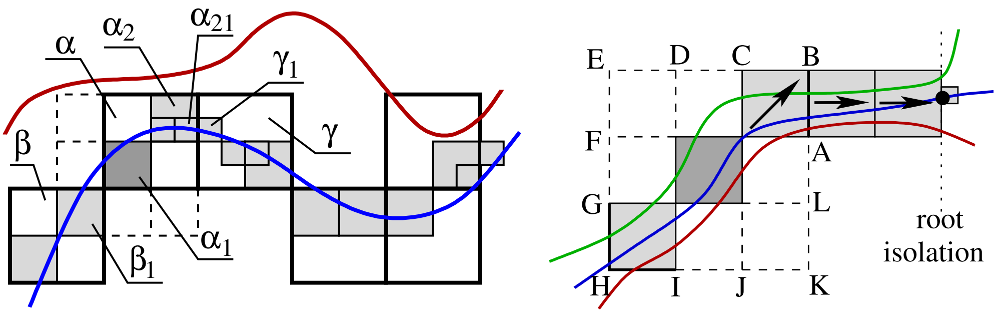

Figure 5.4: Rasterization of a curve arc using adaptive curve tracing (*left*); processing closely located arcs: the neighbourhood test for the dark-shaded pixel in the middle succeeds even in the presence of other arcs nearby (*right*)

We trace the arc in these two directions from the seed point towards the end-points. In each step, the algorithm examines an 8-pixel neighbourhood of a current pixel, see Figure 5.3 (left). If its boundaries are crossed only twice by the arc, we say that the *neighbourhood test* succeeds, see (EBS09, Section 3.2). In this case, we move to the next pixel using the direction returned by the test. Otherwise, there are two possibilities:

- the current pixel is *itself* a witness (sub-)pixel: then we subdivide it recursively into 4 even parts until the test succeeds for one of its sub-pixels or we reach the maximal subdivision depth.[1] In the latter case, the algorithm is restarted with increased arithmetic precision, see (EBS09, Section 3.5);
- the current pixel has an assigned witness (sub-)pixel: we proceed by tracing from this witness (sub-)pixel. In both situations tracing at a sub-pixel level is continued until the pixel boundary is met and we step to the *next* pixel. The last sub-pixel we encounter becomes a *witness* of the newly found pixel.

To give an example, consider plotting a lower arc in Figure 5.4 (left). Curve tracing begins from a seed point covered by a witness (sub-)pixel α_1 in the figure. Its 8-pixel surrounding box is depicted with dashed lines. The pixel it belongs to, namely α, is immediately added to the arc approximation. The curve arc leaves α_1 in two diagonal directions which correspond to the sub-pixels α_2 and β_1 in the figure. Suppose, we choose an upper-diagonal direction from α_1 and move to the sub-pixel α_2. The neighbourhood test fails for α_2 because another curve arc (red one) comes close at this location. Thus, we subdivide the sub-pixel α_2 in 4 pieces, one of them (α_{21}) intersecting the arc is taken.[2] We resume tracing from α_{21}, its neighbourhood test succeeds and we find the next "witness" sub-pixel (γ_1), a pixel it belongs to (γ) is added to the curve trace. We then check the neighbourhood of the sub-pixel γ_1, and so on. The process stops as soon as a termination condition for the curve end (which will be introduced in the following sections) is satisfied. Then, we proceed by tracing the arc towards another end-point starting from the saved sub-pixel β_1.

Note that, the neighbourhood test, which lies in the heart of the algorithm, is based on a number of heuristics to further speed-up the curve tracing process. For instance, in

[1] We define a subdivision depth k as the number of pixel subdivisions, that is, a pixel consists of 4^k depth-k sub-pixels.

[2] To choose such a sub-pixel we evaluate a polynomial at the corners of α_2 since we know that there is only one curve arc going through it.

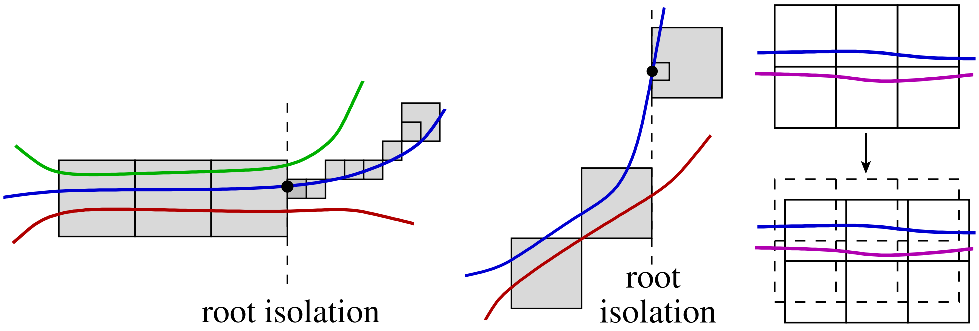

Figure 5.5: Arcs are traced together until one of them goes apart: from that point on the tracing can be resumed on subpixel level after real root isolation (*left*); an arc suddenly changes the slope at the location of root isolation (*middle*); grid perturbation (*right*)

fact, it is not necessary to check all the boundaries of an 8-pixel neighbourhood, as given in Figure 5.3 (left), because some directions are prohibited by the x-monotony constraint, while others can be discarded since we know the direction we came from to the current pixel. More detailed explanations can be found in (EBS09).

Rasterization of closely located curve arcs. One of the challenges for a curve visualization algorithm, already identified in Section 5.3.1, relates to accurate rasterization of arcs that lie very close to each other. To deal with such arcs, we augmented the neighbourhood test (the test that determines in which out of 8 directions a curve arc leaves a current pixel) in a way that we allow a pixel to pass the test as soon as an outgoing direction can be determined uniquely *even* if the number of arcs crossing a sub-segment of the boundary is more than one, see sub-segments AB or GH in Figure 5.4 (right). In this figure, while tracing the middle arc (colored in blue), the neighbourhood test for the dark-shaded pixel succeeds because we can determine the direction of motion along the curve (indicated by an arrow). From this point on, the three arcs can be traced *all together* without the need for actually separating them. At the location where one of them goes apart, we need to pick up the correct arc which can be achieved by real root isolation at the pixel boundary, see also Figure 5.5 (left). Note that, such a "collective" arc tracing does not violate our requirement on a fixed Hausdorff distance from the curve because the arc is guaranteed to lie within an 8-pixel neighbourhood even though this neighbourhood does not necessarily contain only one arc. In a situation when the arc suddenly changes the slope at the the point of real root isolation, as shown in Figure 5.5 (middle), we simply connect the disjoint pixels by a straight line because, by x-monotony constraint, the arc cannot "escape" anywhere else. Typically, we enable tracing arcs collectively by reaching a certain subdivision depth (certain subpixel level) which implicitly indicates that the arcs can be "tightly packed" near this location.

Another difficult situation is depicted in Figure 5.5 (right), where the arcs lie on different sides of the pixel grid, thereby prohibiting a "collective tracing". For example, consider the curve $f(x, y) = y^2 - 10^{-12}$ (two horizontal lines) when the grid origin is at $(0, 0)$. A simple remedy against this is to *shift* the grid origin by an arbitrary sub-pixel amount (grid perturbation) from its initial position before the algorithm starts. In this way, we can ensure with a high probability that the neighbouring arcs are not separated by the pixel grid.

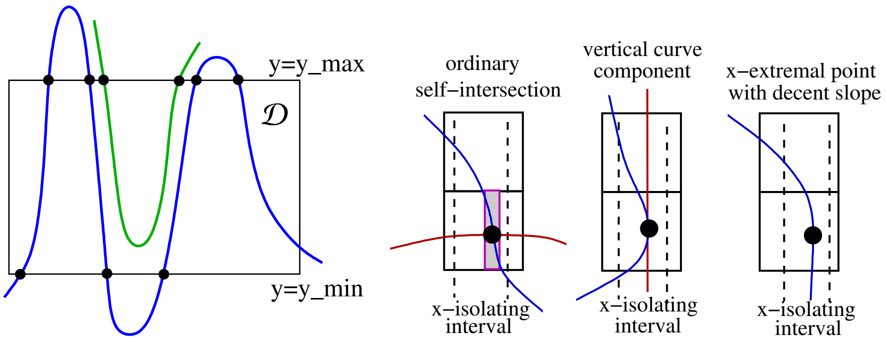

Figure 5.6: Clipping a curve arc against the horizontal boundaries of the rasterization domain $\mathcal{D}$ (*left*); possible situations while detecting the arc's end-point (*right*)

Arc clipping and the stopping criteria. Observe that, the x-monotonicity alone does not allow us to plot curve arcs efficiently because arcs can leave and enter a rasterization domain $\mathcal{D}$ several times as shown in Figure 5.6 (left). Apparently, we do not want to trace an arc outside the drawing window. To avoid this, we isolate the real roots of $f(x, y_0)$ at $y_0 = \{y_{\min}, y_{\max}\}$, the horizontal boundaries of $\mathcal{D}$, to compute the intersection points with the curve.[1] Next, the algorithm does matching to assign each point $x = \alpha_i$ to a particular arc. This can be done as follows: we use the bitstream Descartes algorithm (EKK+05) for each polynomial $f(\alpha_i, y)$ to find those roots y_i which lie within one pixel from the horizontal boundaries $\{y_{\min}, y_{\max}\}$. Note that, we allow a certain tolerance when matching the arcs with intersection points because this can only produce an error within a pixel size. Once the intersection points are processed, each arc can be rasterized using disjoint segments between each two points: it is only required to determine whether the first segment lies inside or outside the window (again using root isolation) since "inside" and "outside" segments are alternating. Finally, remark that, arcs clipping enables us to treat the arcs of different types (bounded, unbounded or asymptotic) *uniformly* without the need for explicitly checking the type. The only exception is vertical arcs which, however, can be easily detected as outlined below.

To make the algorithm complete, we need work out the missing details of the stopping criteria. Indeed, recall that, we do not have the actual end-points of a curve arc: only the projections onto the x-axis are available. That is why, we again need to exploit the x-monotonicity to decide where to stop tracing the arc. Namely, the tracing can be terminated as soon as we reach a (sub-)pixel containing the x-isolating interval of an end-point, and there exists such a box inside this (sub-)pixel that the curve crosses its *vertical boundaries* only. This last condition is necessary to prevent a premature stopping alarm for arcs with a decent slope, see Figure 5.6 (right). Unfortunately, this solution is not working when a curve has a *vertical line* as a component at some x-coordinate α. Luckily, the vertical lines can be easily detected by checking if the coefficients of $f(\alpha, y)$ vanish identically. In the latter case, we simply do not check the vertical boundaries of a terminating (sub-)pixel but use only the x-monotony constraint as a termination condition.

Finally, there are situations when an arc has a very decent slope with an x-extremal

[1] To be precise, before root isolation, we extract the square-free part of $f(x, y_0)$ because it can happen that there are double roots along a horizontal line.

Instance	y-degree	# x-arcs	Project	Rasterize	Axel	Maple
the_sun	19	181	1.35	4.1	0.69	2.37
inf_plus_der	24	80	0.22	5.33	205	4.7
mtaylor_grid	28	280	6.5	4.3	1.85	0.97
dfold_10_6	32	120	0.29	28.5	?	3.9
flower	16	24	0.06	0.79	0.26	9.6
FTT_3_5_5	30	200	0.5	58.5	104	1.4
octagon	14	436	1.7	8.6	535	22.9
spider	28	304	11.2	26.1	709	8.4
kushnirenko	47	84	18.1	25	?	6.1

Table 5.4: Execution time in seconds for curve rasterization. 'y-degree': degree of a defining polynomial in y-variable; '# x-arcs': the total number of x-monotone arcs of a curve; 'Project': projecting the event points of a curve onto the x-axis; 'Rasterize': visualization using our method; the last two columns: visualization using Axel and Maple 14, respectively.

end-point (the rightmost picture in Figure 5.6). Here, it might happen that the algorithm overlooks the actual end-point and stops tracing later because, for our stopping criteria to work, a terminating pixel must to be subdivided sufficiently many times to ensure that the vertical boundaries being checked are quite narrow. However, the algorithm is not forced to subdivide a pixel if nothing "special" happens around an x-extremal point. Still, this fact certainly does not violate our constraint on a fixed Hausdorff distance from the curve.

5.3.3 Benchmarks and comparison

The experiments have been conducted on our desktop machine with 2.8GHz 8-Core Intel Xeon W3530 having 8 MB of L2 cache and GeForce GTX580 graphics processor under 64-bit Linux platform. Similar to Bisolve benchmarks in Section 5.1.4, we have used Gmp 5.0.1 library for exact number types and RealSolving (Rs) package[1] to speed-up the univariate root solving. Note that, the original approach (EBS09) has been implemented within Cgal (Computational Geometry Algorithms Library, `www.cgal.org`). Owing to generic programming style, we have been able to easily exchange the relevant parts of the algorithm to replace a curve analysis (computing a full CAD) with a "lite" version where only the projections of topological events of a curve are computed (see Section 5.3.1).

We have compared our approach with the algorithm (AM07) from Axel 0.5.3[2] and implicitplot from Maple 14. Although, the both competing algorithms are, in fact, numeric subdivision methods,[3] it was nevertheless interesting to compare the resulting visual quality and the running times with those for our approach. To obtain smooth curve images with Axel, we have varied the accuracy parameter ε between $5 \cdot 10^{-4}$ and $5 \cdot 10^{-7}$ depending on the curve, while the "feature size" asr has been set to 10^{-2}. Maple's implicitplot has been configured with the parameters: resolution = 500 and numpoints between 10^5 and 10^6, also to get a satisfiable image quality. Note that, neither of the subdivision algorithms can set the precision parameters *adaptively*, thus we had to play around with the options to find some compromise between image quality and the running time. For our algorithm the resolution (the number of pixels in a rasterization domain $\mathcal{D}$) has been set

[1] Gmp: `http://gmplib.org`, Rs: `http://www.loria.fr/equipes/vegas/rs`

[2] Subversion repository is available at `svn://scm.gforge.inria.fr/svn/axel`.

[3] The symbolic approach described in (AM07) is not available in Axel.

k	Project	Rasterize	Cxy
7	0.25	0.16	14.2
8	0.37	0.17	22.1
9	0.54	0.25	27.6
10	0.82	0.2	< 180
11	1.18	0.34	n/a
12	1.67	0.29	n/a

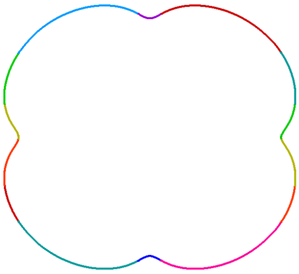

Table 5.5: Execution time in seconds to plotting the curve $(x^2 + y^2)^k - 4x^2y^2 - 0.01 = 0$ inside a box $[-1, 1] \times [-1, 1]$ for different values of k using our approach and the algorithm Cxy, respectively. Cxy timings are taken from (LY11). The curve plot for $k = 11$ is shown to the right.

to 640 × 480. We have also partially compared our algorithm with a certified subdivision approach, called Cxy, from (LY11). Here, the main concern was the computing time because the algorithm Cxy computes isotopic approximation of (non-singular) algebraic curves. Unfortunately, we have not been able to compile the source code,[1] and that is why we have considered only the examples mentioned in the authors' work.

The running times for curve rasterization are listed in Table 5.4. For our approach, we have measured the time for projecting the "event points" and actual visualization separately. The number of x-monotone arcs rasterized by our algorithm is given in the 3rd column of the table. The curve images are shown in Figures 5.7 and 5.9. In the figures, different x-monotone arcs are rasterized with different colors. Additionally, Figure 5.10 provides zooming at certain singular points to demonstrate that our algorithm indeed preserves geometric correctness of a curve plot at any resolution. From Table 5.4, we see that Axel is particularly fast for curves having simple singularities (where just four curve branches meet at one point), but it experiences a large slowdown when the curve has a complicated topology or curve arcs are very badly separated. Also, we did not manage to produce a plot for 'dfold_10_6' curve since Axel was constantly reporting floating-point error. What concerns Maple's implicitplot, it works reasonably fast for all instances we tried. It is not surprising because, unlike Axel, this algorithm is based on a pure numerical subdivision and does not attempt to recover any topological information of a curve. This fact also find a confirmation in the obtained curve images: we see that Maple is unable to adequately rasterize the curves 'octagon' and 'spider', see Figure 5.9. Zooming in Figure 5.10 reveals a hidden structure of these curves where each arc is essentially composed of *two* arcs passing very close to each other: this makes an accurate visualization very challenging. In contrast, Axel produces more accurate plots (which, however, are still not free from some visual artifacts). The last curve listed in Table 5.4, 'kushnirenko', corresponds to the zero set of $\mathcal{A}$-Discriminant appearing in (DRRS07) for the purpose of disproving the Kushnirenko's conjecture.[2] Rasterization using our algorithm and Maple (at increasing resolution) is provided in Figure 5.8. The comparison of visual quality again shows the superiority of our approach. Axel again has failed to visualize this curve (reporting floating-point error).

To compare with the Cxy approach, we have rasterized a curve $f(x, y) = (x^2 + y^2)^k - 4x^2y^2 - 0.01$ for different values of k from 7 to 12. The running times together with the

[1] It is available at `http://cs.nyu.edu/exact/papers/cxy`

[2] The curve equation is available at: `http://www.math.tamu.edu/~rojas/haas3disc`

curve image for $k = 11$ are provided in Table 5.5. Note that, in (LY11, p. 22) it was reported that our approach times out for $k \geq 7$. This was certainly caused by expensive symbolic operations used in the original approach (EBS09) to compute the CAD since, by looking at the curve image, it is clear that the visualization should not be of any challenge. Indeed, from the timings we now see that this example presents no difficulties to our rasterization algorithm.

Still, one of the shortcomings of our approach relates to the fact that it has to process x-monotone arcs of a curve *separately*: meaning that, we have to find a seed point for each of them, refine it sufficiently many times and, in addition, check if an arc crosses the boundaries of a rasterization domain. Now, imagine that the curve decomposes into hundreds of small arcs, which, however, are not small enough to be replaced by a single pixel. Then, our algorithm would have to use all the complicated algebraic machinery (real root isolation, refinement, etc.) for each of them. This situation is well-observed in the experiments for the curves 'FTT_3_5_5', 'the_sun' and 'mtaylor_grid'. Here, we notice that the rasterization complexity (see column 5 in Table 5.4) is not the result of a complicated curve topology but is mainly determined by the fact that the algorithm spends too much time for the above mentioned operations. One possible solution to this problem would be to enable tracing arcs *across* the actual end-points: that is, stop only by hitting the boundary of a rasterization domain or when tracing is "no longer possible". Certainly, the meaning of "no longer possible" would require further clarification. Additionally, we would have to keep track of those pixels which have already been plotted. Another reasonable idea is to have an algorithm which already returns a "minimal set" of x-monotone arcs: indeed, many arcs computed in the projection step can be merged together to produce larger arcs which still do not violate x-monotony constraint. However, looking from the other perspective, our visualization algorithm can directly profit from multi-core processing due to the fact that each arc is traced individually.

In summary, we see that the combination of symbolic-numeric methods used by our algorithm works best in practice since it allows us to obtain geometrically correct rasterization in all situations while, at the same time, does not lead to large performance loss (as in the case of Axel). From the timings in the 'Project' column in Table 5.4, we also observe that the "symbolic phase" of our approach has become negligibly cheap, even for complicated instances: compare with the original benchmarks in (EBS09, Section 4). This is because we are no longer required to compute the whole topology graph of a curve, and thus can take a full advantage of the GPU algorithms and fast real root isolation.

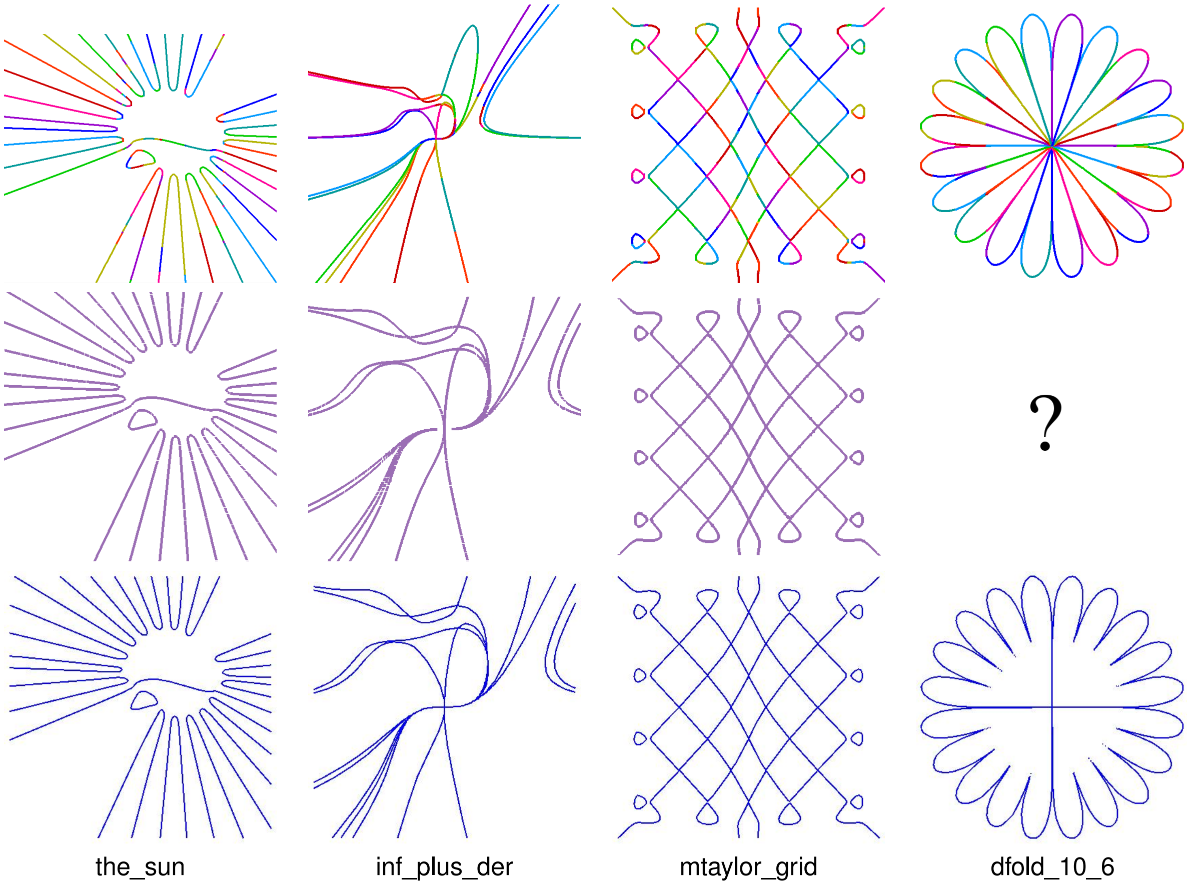

Figure 5.7: From top to bottom: curve images rendered using our approach, Axel and Maple's implicitplot, respectively.

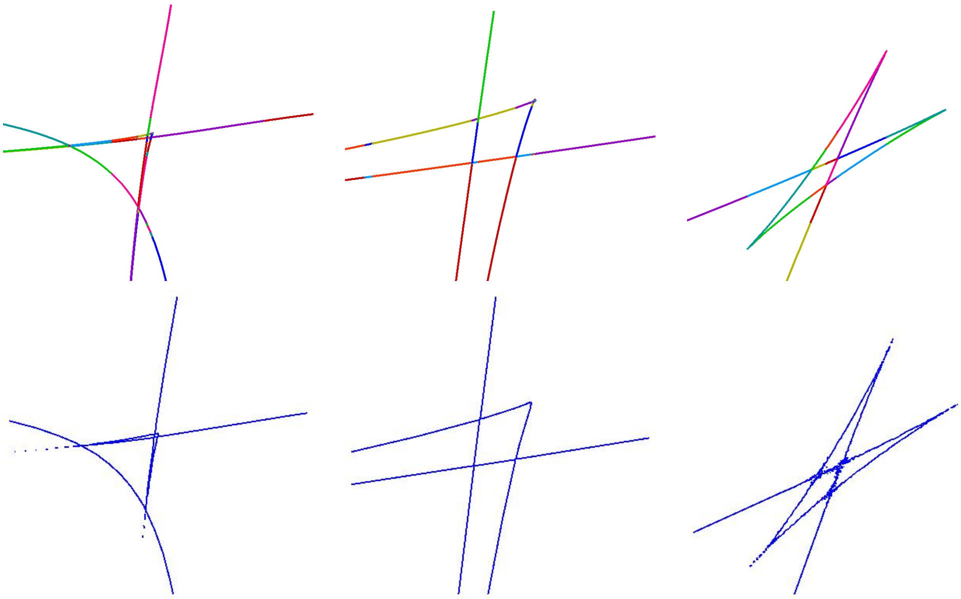

Figure 5.8: Visualization of $\mathscr{A}$-Discriminant appearing in the counter-example of Kushnirenko's Conjecture, see also (DRRS07). Top row: our method; bottom row: Maple's implicitplot.

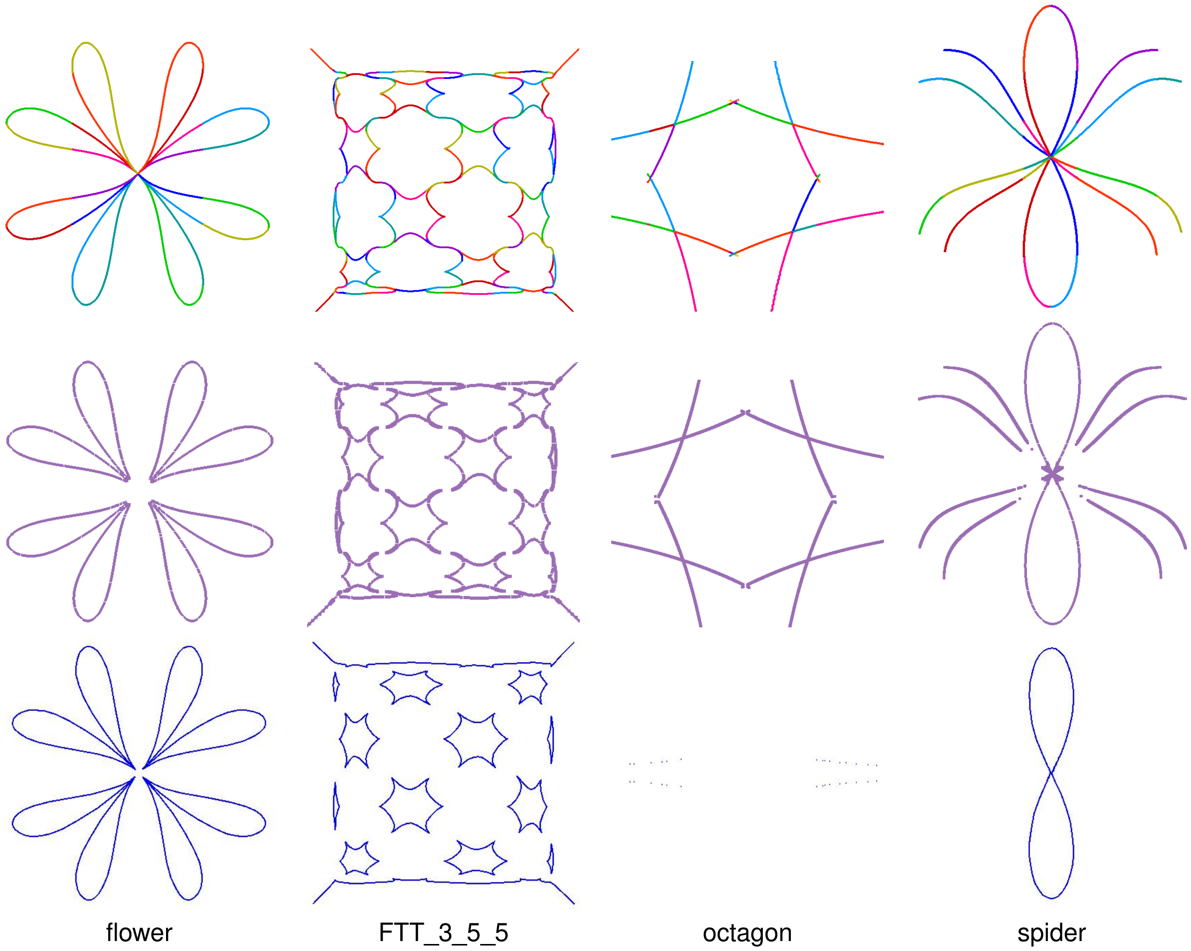

Figure 5.9: From top to bottom: curve images rendered using our approach, Axel and Maple's implicitplot, respectively.

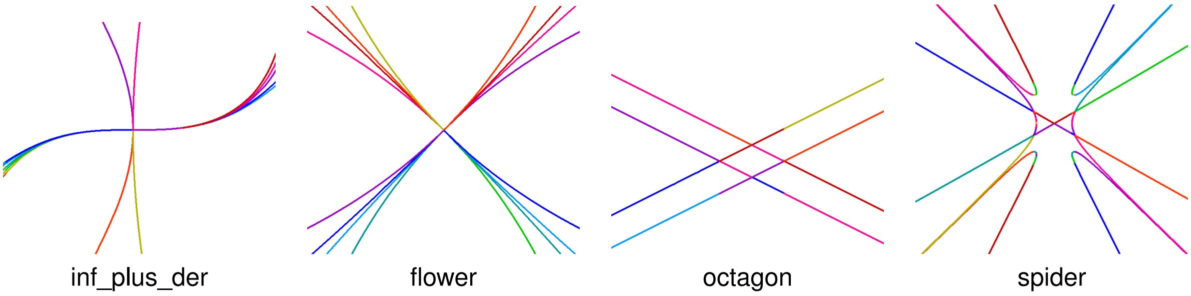

Figure 5.10: Zooming at singularities for selected curves listed in Table 5.4.

6 Conclusion and open problems

Doing exact computations with polynomials on the GPU is not as widely practiced as, for instance, parallel computing with floating-point arithmetic adopted in many engineering fields. This application area of graphics accelerators is a quite new and very fast evolving. Without repeating what has already been said for concluding remarks in the previous chapters, we can identify the following directions for future work.

First, there is a quite natural demand in extending our algorithms to higher dimensions in order to be able to handle algebraic objects of greater complexity. Though, this seems to be quite straightforward at first glance, if we recall that the modular approach already works for multivariate polynomials of arbitrary dimension, we anticipate that the actual realization on the GPU could be a matter of large technical difficulty. This is because the modular approach is based on the recursive application of homomorphisms which, in CUDA language, would be equivalent to a series of kernel calls with intricate CPU control and data flow. Furthermore, for multivariate polynomials with three or more variables, an efficient "sparse" representation already becomes necessary because such polynomials quite often have many zero terms.[1] On the CPU, this problem is "transparently" solved using conventional STL data structures and C++ template mechanism. While in the plain memory of a graphics accelerator, this would imply development of some versatile data structures with shifts and offsets to mark the locations of non-zero polynomial coefficients which are hard to maintain and debug.

Additionally, there are many scientific problems, especially in computational geometry and related fields, where performing *all* computations with exact arithmetic is, albeit feasible, proven to be an overkill. One of the classical examples from computer algebra is the task of real root isolation. Therefore, a natural question comes up as to whether it is possible to perform multi-precision (BigFloat) computations on the GPU ? Although, the general answer is negative, there is a number of ways in which we can "model" the operations on multi-precision numbers on the graphics card. In our opinion, one of the most promising approaches relates to the use of a *Hybrid Number System* (HNS). The latter one can be viewed as the generalization of RNS (Residue Number System, see also Section 2.2.2) where numbers are represented using both weighted and residue notations and the expansion/contraction of either part of the representation is provided; see, e.g., (HS04, BP92). In such a way, operations on mantissas (most significant bits) of BigFloat numbers are realized in the RNS while magnitude comparison, sign detection, etc. can best be performed on weighted representation. Moreover, we can add/drop any number of bits of the weighted part (to control the precision) without disturbing the RNS.

[1] Recall that, a dense multivariate polynomial in d variables of total degree bounded by n can have up to n^d monomials.

In effect, this approach has many parallels with the *dynamic RNS* widely adopted in cryptography, where one is allowed to change the number of moduli (dynamic range) used in the representation at a runtime; see (Gon91, Sod86, BDK01). Besides that, there are problems where the set of required operations on BigFloats are restricted to that of addition, subtraction, bit-shifts and comparison. In the latter case, we can do well without using residue arithmetic at all. Indeed, addition/subtraction operations are straightforward to parallelize if we employ some redundant representation to accumulate carry/borrow propagations and, in the end, use parallel reduction to update the result in one sweep. However, this approach is closely tied to a concrete problem at hand.

It is also quite clear that, in order to exploit the advantages of parallel processing at full, the target algorithms must *augmented* in such a way to minimize the cost of sequential steps (for instance, using cheap approximate computations whenever possible) while letting the GPU do the hard work. Often enough, it requires looking at a seemingly well-studied problem from a new perspective. In this respect, our approaches for the solution of a system of polynomial equations, topology computation of algebraic curves and curve visualization provide a good exposition. That is why, another important research direction would be developing algorithms that can outsource the main computationally expensive tasks to the GPU and, at the same time, concentrate on the optimization of the remaining sequential routines. We also wish to continue our work on the complexity analysis: in particular, we are quite confident that our recent complexity estimates obtained for Bisolve (see Section 5.2) can be further extended and generalized to BiCurveAnalysis.

Lastly, it probably makes sense not to become attached to the GPUs only but consider other architectures for massively-parallel computing: luckily, this sector of computer industry grows quite rapidly now. Among possible alternatives, we distinguish the Intel's MIC (Many Integrated Core) architecture (Ska10) which inherits many design decisions from the former Larrabee project. This architecture is built upon x86 in-order processor cores equipped with 16-lane VPUs (Vector Processor Unit, somewhat similar to the GPU's Multiprocessor). Intel's MIC is supposed to be interfaced through OpenCL and has several features, lacked on the current GPUs, including cache coherency among all processor cores and support for scatter/gather operations. Cache coherency, for example, can greatly simplify the use of block-level parallelism since the results of computation are transparently shared between all cores, while scatter/gather functionality allows concurrent data accesses at non-contiguous addresses without a noticeable overhead. A commercial release of the chips is planned for late 2012.

Bibliography

[Abb06] J. Abbott. Quadratic Interval Refinement for Real Roots. Poster presented at ISSAC '06, 2006.

[Abb09] J. Abbott. "Bounds on Factors in $\mathbb{Z}[x]$". *ArXiv e-prints*, April 2009.

[AM07] L. Alberti and B. Mourrain. Visualisation of Implicit Algebraic Curves. *Computer Graphics and Applications, Pacific Conference on*, pages 303–312, 2007.

[AMW08] L. Alberti, B. Mourrain, and J. Wintz. Topology and Arrangement Computation of Semi-Algebraic Planar Curves. *Computer Aided Geometric Design*, 25(8):631–651, 2008.

[Bai88] D.H. Bailey. A High-Performance FFT Algorithm for Vector Supercomputers. *International Journal of Supercomputer Applications*, 2:82–87, 1988.

[BCC$^+$09] D.J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on Graphics Cards. In *EUROCRYPT '09*, Berlin, Heidelberg, 2009. Springer-Verlag.

[BCGY12] M. Burr, S.W. Choi, B. Galehouse, and C. Yap. Complete Subdivision Algorithms, II: Isotopic Meshing of Singular Algebraic Curves. *Journal of Symbolic Computation*, 47(2):131–152, 2012.

[BDK01] J.-C. Bajard, L.-S. Didier, and P. Kornerup. Modular Multiplication and Base Extensions in Residue Number Systems. In *In 15th IEEE Symposium on Computer Arithmetic*, pages 59–65. IEEE, 2001.

[BEKS11a] E. Berberich, P. Emeliyanenko, A. Kobel, and M. Sagraloff. Arrangement computation for planar algebraic curves. In *SNC '11*, San Jose, USA, June 2011. ACM.

[BEKS11b] E. Berberich, P. Emeliyanenko, A. Kobel, and M. Sagraloff. Exact Symbolic-Numeric Computation of Planar Algebraic Curves. *CoRR*, abs/1201.1548v1, 2011.

[BES11] E. Berberich, P. Emeliyanenko, and M. Sagraloff. An elimination method for solving bivariate polynomial systems: Eliminating the usual drawbacks. In *ALENEX '11*, pages 35–47, San Francisco, USA, January 2011. SIAM.

[BKS10] E. Berberich, M. Kerber, and M. Sagraloff. An Efficient Algorithm for the Stratification and Triangulation of Algebraic Surfaces. *Computational Geometry: Theory and Applications*, 43:257–278, 2010. Special issue on SoCG'08.

[BMO$^+$10] D. Bini, V. Mehrmann, V. Olshevsky, E. Tyrtyshnikov, and M. van Barel, editors. *Numerical Methods for Structured Matrices and Applications: The Georg Heinig Memorial Volume*. Operator Theory: Advances and Applications. Birkhäuser Verlag, 2010.

[BO88] M. Ben-Or. A deterministic algorithm for sparse multivariate polynomial interpolation. In *STOC '88*, pages 301–309, New York, NY, USA, 1988. ACM.

[BOA09] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG '09*, pages 159–166, New York, NY, USA, 2009. ACM.

[BP92] F. Barsi and M.C. Pinotti. Adding Flexibility to Hybrid Number Systems. *Comput. J.*, 35(6):630–635, 1992.

[BPR06] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Bro71] W. S. Brown. "On Euclid's algorithm and the computation of polynomial greatest common divisors". In *SYMSAC '71*, pages 195–211, New York, NY, USA, 1971. ACM.

[Bro78] W. S. Brown. The Subresultant PRS Algorithm. *ACM Trans. Math. Softw.*, 4:237–249, 1978.

[BZ10] R. Brent and P. Zimmermann. Modern Computer Arithmetic (version 0.5.1). *CoRR*, abs/1004.4710, 2010.

[CA76] G.E. Collins and A.G. Akritas. Polynomial real root isolation using Descarte's rule of signs.

In *SYMSAC '76*, pages 272–275, New York, NY, USA, 1976. ACM.

[CGL09] J.-S. Cheng, X.-S. Gao, and J. Li. Root isolation for bivariate polynomial systems with local generic position method. In *ISSAC '09*, pages 103–110, New York, NY, USA, 2009. ACM.

[Cha88] R. Chandler. A tracking algorithm for implicitly defined curves. *IEEE Computer Graphics and Applications*, 8, 1988.

[CLO98] D. Cox, J. Little, and D. O'Shea. *Using Algebraic Geometry*, volume 185 of *Undegraduate Texts in Mathematics*. Springer-Verlag, New York, 1998.

[CLP$^+$09a] J. Cheng, S. Lazard, L. Pe naranda, M. Pouget, F. Rouillier, and E. Tsigaridas. On the topology of planar algebraic curves. In *SCG '09*, pages 361–370, New York, NY, USA, 2009. ACM.

[CLP$^+$09b] J. Cheng, S. Lazard, L. Penaranda, M. Pouget, F. Rouillier, and E. Tsigaridas. On the topology of planar algebraic curves. In *SCG '09: Proc. of the 25th Annual Symposium on Computational Geometry*, pages 361–370, New York, NY, USA, 2009. ACM.

[Coh03] J.S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. AK Peters, Natick, 2003.

[Col71] G.E. Collins. The calculation of multivariate polynomial resultants. In *SYMSAC '71*, pages 212–222. ACM, 1971.

[Col74] G.E. Collins. The Computing Time of the Euclidean Algorithm. *SIAM J. on Computing*, 3(1):1–10, 1974.

[Col75] G.E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, pages 134–183, 1975.

[Cor04] J.-S. Coron. Finding small roots of bivariate integer polynomial equations revisited. In *EUROCRYPT '04*, pages 492–505. Springer-Verlag, 2004.

[CS98] S. Chandrasekaran and A. Sayed. A fast stable solver for nonsymmetric toeplitz and quasi-toeplitz systems of linear equations. *SIAM J. Matrix Anal. Appl.*, 19:107–139, 1998.

[CUD10] CUDA. CUDA Compute Unified Device Architecture. Programming Guide. Version 3.2, 2010. NVIDIA Corp.

[CUF10] CUFFT. CUDA CUFFT library. Version 3.0, 2010. NVIDIA Corp.

[dDBQ04] G.M. de Dormale, P. Bulens, and J.-J. Quisquater. An improved Montgomery modular inversion targeted for efficient implementation on FPGA. In *FPT '04. IEEE International Conference on*, pages 441–444, 2004.

[DET09] D.I. Diochnos, I.Z. Emiris, and E.P. Tsigaridas. On the asymptotic and practical complexity of solving bivariate systems over the reals. *J. Symb. Comput.*, 44(7):818–835, 2009.

[dKMW05] J. de Kleine, M. Monagan, and A. Wittkopf. "Algorithms for the non-monic case of the sparse modular GCD algorithm". In *ISSAC '05*, pages 124–131, New York, NY, USA, 2005. ACM.

[DRRS07] A. Dickenstein, J.M. Rojas, K. Rusek, and J. Shih. Extremal Real Algebraic Geometry and A-Discriminants. *CoRR*, abs/math/0609485v2, 2007.

[EBS09] P. Emeliyanenko, E. Berberich, and M. Sagraloff. Visualizing Arcs of Implicit Algebraic Curves, Exactly and Fast. In *ISVC '09*, pages 608–619, Berlin, Heidelberg, 2009. Springer-Verlag.

[EK08] P. Emeliyanenko and M. Kerber. Visualizing and exploring planar algebraic arrangements: a web application. In *SCG '08*, pages 224–225, New York, NY, USA, 2008. ACM.

[EKK$^+$05] A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A Descartes algorithm for polynomials with bit-stream coefficients. In *CASC '05*, volume 3718 of *LNCS*, pages 138–149, 2005.

[EKW07] A. Eigenwillig, M. Kerber, and N. Wolpert. Fast and exact geometric analysis of real algebraic plane curves. In *ISSAC '07*, pages 151–158, New York, NY, USA, 2007. ACM.

[Eme07] P. Emeliyanenko. Visualization of Points and Segments of Real Algebraic Plane Curves. Master's thesis, Universität des Saarlandes, February 2007.

[Eme09] P. Emeliyanenko. Efficient Multiplication of Polynomials on Graphics Hardware. In *APPT '09*, pages 134–149, Berlin, Heidelberg, 2009. Springer-Verlag.

[Eme10a] P. Emeliyanenko. Accelerating Symbolic Computations on NVIDIA Fermi, 2010. Poster presentation. Available at: `http://www.nvidia.com/object/research_summit_posters_2010.html`.

[Eme10b] P. Emeliyanenko. A complete modular resultant algorithm targeted for realization on graphics hardware. In *PASCO '10*, pages 35–43, New York, NY, USA, 2010. ACM.

[Eme10c] P. Emeliyanenko. Modular Resultant Algorithm for Graphics Processors. In *ICA3PP '10*, pages 427–440, Berlin, Heidelberg, 2010. Springer-Verlag.

[Eme11] P. Emeliyanenko. High-performance polynomial GCD computations on graphics processors. In *High Performance Computing and Simulation (HPCS '11)*, pages 215–224. IEEE Press, July 2011.

[ER83] D.F. Elliott and K.R. Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Academic Press, Inc., Orlando, FL, USA, 1983.

[Fer10] Fermi. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2010. Whitepaper, NVIDIA Corp.

[FL94] E.N. Frantzeskakis and K.J.R. Liu. A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing. *Signal Processing, IEEE Transactions on*, 42:2455–2469, Sep 1994.

[Fuj09] N. Fujimoto. High throughput multiple-precision GCD on the CUDA architecture. In *Signal Processing and Information Technology (ISSPIT), 2009 IEEE International Symposium on*, pages 507–512, 2009.

[GCL92] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for computer algebra*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.

[GLD$^+$08] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC'08*, pages 1–12. IEEE Press, 2008.

[Gon91] J. Gonnella. The application of core functions to residue number systems. *Signal Processing, IEEE Transactions on*, 39(1):69–75, jan 1991.

[GPU] GPUFFTW. GPUFFTW: High Performance Power-of-Two FFT Library using Graphics Processors. `http://gamma.cs.unc.edu/GPUFFTW`.

[GR74] A.J. Goldstein and Graham R.L. A Hadamard-Type Bound on the Coefficients of a Determinant of Polynomials. *SIAM Review*, 16:394–395, 1974.

[GS72] I. Gohberg and A. Semencul. On the inversion of finite Toeplitz matrices and their continuous analogs. *Mat.Issled.*, 2:201–233, 1972.

[GVK96] L. González-Vega and M.E. Kahoui. An Improved Upper Complexity Bound for the Topology Computation of a Real Algebraic Plane Curve. *Journal of Complexity*, 12(4):527–544, 1996.

[Har07] M. Harris. Parallel Prefix Sum (Scan) with CUDA, 2007. NVIDIA Corp.

[Hec96] C. Hecker. Let's get to the (floating) point. *Game Developer Magazine*, pages 19–24, 1996.

[Hof89] C. Hoffmann. *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[HS86] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, 1986.

[HS04] R. Hashemian and B. Sreedharan. A Hybrid Number System And Its Application In FPGA-DSP Technology. *Information Technology: Coding and Computing, International Conference on*, 2:342, 2004.

[HS06] M. Hinek and D. Stinson. An inequality about factors of multivariate polynomials. Technical Report CACR Technical Report CACR 2006-15, Centre for Applied Cryptographic Research, University of Waterloo, 2006.

[HW09] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *AFRICACRYPT '09*, pages 350–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[J́92] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[J̈11] A. Jörg. *Matters Computational: Ideas, Algorithms, Source code*. Springer-Verlag, Berlin, Heidelberg, 1 edition, 2011. `http://www.jjj.de/fxt`.

[Kai87] T. Kailath. Signal processing applications of some moment problems. *Moments in Mathematics*, 37:71–109, 1987. Landau, editor.

[Kal95] B.S. Kaliski. The montgomery inverse and its applications. *Computers, IEEE Transactions on*, 44(8):1064–1065, 1995.

[KC94a] T. Kailath and J. Chun. Generalized displacement structure for block-toeplitz, toeplitz-block, and toeplitz-derived matrices. *SIAM J. Matrix Anal. Appl.*, 15:114–128, 1994.

[KC94b] T. Kailath and J. Chun. Generalized Displacement Structure for Block-Toeplitz,Toeplitz-Block, and Toeplitz-Derived Matrices. *SIAM J. Matrix Anal. Appl.*, 15:114–128, 1994.

[Ker09] M. Kerber. *Geometric Algorithms for Algebraic Curves and Surfaces*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2009.

[Knu97] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 2 edition, 1997.

[KS95] T. Kailath and A. Sayed. Displacement structure: theory and applications. *SIAM Review*, 37:297–386, 1995.

[KS11a] M. Kerber and M. Sagraloff. Efficient real root approximation. In *ISSAC '11*, pages 209–216, New York, NY, USA, 2011. ACM.

[KS11b] M. Kerber and M. Sagraloff. A worst-case bound for topology computation of algebraic curves. *CoRR*, abs/1104.1510, 2011. submitted.

[Lab10] O. Labs. A List of Challenges for Real Algebraic Plane Curve Visualization Software. In Ioannis Z. Emiris, Frank Sottile, and Thorsten Theobald, editors, *Nonlinear Computational Geometry*, volume 151 of *The IMA Volumes in Mathematics and its Applications*, pages 137–164. Springer New York, 2010.

[Lai69] M. A. Laidacker. "Another Theorem Relating Sylvester's Matrix and the Greatest Common Divisor". *Mathematics Magazine*, 42(3):126–128, May 1969.

[LF95] H.-C. Liao and R. J. Fateman. "Evaluation of the Heuristic Polynomial GCD". In *ISSAC '95*, pages 240–247. ACM Press, 1995.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[LR01] T. Lickteig and M.-F. Roy. Sylvester-Habicht Sequences and Fast Cauchy Index Computation. *Journal of Symbolic Computation*, 31(3):315–341, 2001.

[LY11] L. Lin and C. Yap. Adaptive Isotopic Approximation of Nonsingular Curves: the Parameterizability and Nonlocal Isotopy Approach. *Discrete Comput. Geom.*, 45:760–795, 2011.

[Man92] D. Manocha. *Algebraic and numeric techniques in modeling and robotics*. PhD thesis, Berkeley, CA, USA, 1992. UMI Order No. GAX93-05000.

[MG04] J. Morgado and A. Gomes. A Derivative-Free Tracking Algorithm for Implicit Curves with Singularities. In *Computational Science - ICCS 2004*, volume 3039 of *Lecture Notes in Computer Science*, pages 221–228. Springer Berlin / Heidelberg, 2004.

[MG11] N. Möller and T. Granlund. Improved Division by Invariant Integers. *Computers, IEEE Transactions on*, 60(2):165 –175, 2011.

[Mig74] M. Mignotte. An inequality about factors of polynomials. *Mathematics of Computation*, 28(128):1153–1157, 1974.

[Mon05] M. Monagan. Probabilistic algorithms for computing resultants. In *ISSAC '05*, pages 245–252. ACM, 2005.

[MOS11] K. Mehlhorn, R. Osbild, and M. Sagraloff. A general approach to the analysis of controlled perturbation algorithms. *Comput. Geom.*, 44(9):507–528, 2011.

[MP09] B. Mourrain and J.P. Pavone. Subdivision methods for solving polynomial equations. *J. Symb. Comput.*, 44(3):292–306, 2009.

[MP10] Marc Moreno Maza and Wei Pan. Fast polynomial multiplication on a GPU. *Journal of Physics: Conference Series*, 256(1):012009, 2010.

[MPS07] A. Moss, D. Page, and N. Smart. Toward acceleration of RSA using 3D graphics hardware. In *Proceedings of the 11th IMA international conference on Cryptography and coding*, pages 364–383, Berlin, Heidelberg, 2007. Springer-Verlag.

[Mun08] A. Munshi. The OpenCL Specification, 2008. Chronos OpenCL Working Group.

[NUF] NUFFT. Nukada FFT library. `http://matsu-www.is.titech.ac.jp/~nukada/nufft`.

[Pan97] V. Pan. Solving a polynomial equation: some history and recent progress. *SIAM Review*, 39(2):187–220, 1997.

[Pan01] V. Pan. *Structured matrices and polynomials: unified superfast algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[Pan10] W. Pan. *Algorithmic Contributions to the Theory of Regular Chains*. PhD thesis, University of Western Ontario, 2010.

[Pet99] S. Petitjean. Algebraic Geometry and Computer Vision: Polynomial Systems, Real and Complex Roots. *J. Math. Imaging Vis.*, 10(3):191–220, 1999.

[PTX10] PTX. PTX: Parallel Thread Execution. ISA Version 2.1, 2010. NVIDIA Corp.

[Rei97] D. Reischert. Asymptotically fast computation of subresultants. ISSAC '97, pages 233–240, New York, NY, USA, 1997. ACM.

[Rei05] J.H. Reif. Efficient parallel factorization and solution of structured and unstructured linear systems. *J. Comput. Syst. Sci.*, 71(1):86–143, 2005.

[Roo99] S.H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[RR05] H. Ratschek and J.G. Rokne. SCCI-hybrid Methods for 2D Curve Tracing. *Int. J. Image Graphics*, 5:447–480, 2005.

[RZ04] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial's real roots. *J. Comput. Appl. Math.*, 162(1):33–50, 2004.

[Sag11] Michael Sagraloff. When Newton Meets Descartes - A Simple and Fast Algorithm to Isolate the Real Roots of a Polynomial. *CoRR*, abs/1109.6279, 2011. submitted in parallel to ISSAC'12.

[Sch17] I. Schur. Über potenzreihen die im Inneren des Einheitskreises beschränkt sind. *Reine und Angewandte Mathematik*, 147:205–232, 1917.

[Sed83] T. Sederberg. *Implicit and parametric curves and surfaces for computer aided geometric design*. PhD thesis, West Lafayette, IN, USA, 1983. AAI8400421.

[Sho09] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2 edition, 2009.

[SK95] A. Sayed and T. Kailath. A Look-Ahead Block Schur Algorithm for Toeplitz-Like Matrices. *SIAM J. Matrix Anal. Appl.*, 16(2):388–414, 1995.

[SK00] E. Savas and C.K. Koc. The Montgomery modular inverse-revisited. *Computers, IEEE Transactions on*, 49(7):763–766, 2000.

[SK08] B. Siciliano and O. Khatib, editors. *Handbook of Robotics*. Springer, Berlin / Heidelberg, 2008.

[Ska10] K. Skaugen. Petascale to Exascale: Extending Intel's HPC commitment, 2010. ISC 2010 keynote, `http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf`.

[Sod86] M.A. Soderstrand. *Residue number system arithmetic: modern applications in digital signal processing*. IEEE press reprint series. Institute of Electrical and Electronics Engineers, 1986.

[Som04] M. Sombra. The Height of the Mixed Sparse Resultant. *American Journal of Mathematics*, 126(6):1253–1260, 2004.

[SS10] C. Stussak and P. Schenzel. Parallel Computation of Bivariate Polynomial Resultants on Graphics Processing Units. In *PARA '10*, University of Iceland, Reykjavik, 2010. extended abstract.

[Stu02] B. Sturmfels. *Solving systems of polynomial equations*, volume 97 of *Regional conference series in mathematics*. AMS, Providence, RI, 2002.

[SW05a] R. Seidel and N. Wolpert. On the Exact Computation of the Topology of Real Algebraic Curves. In *SoCG '05*, pages 107–115, 2005.

[SW05b] A.J. Sommese and C.W. Wampler. *The Numerical Solution of Systems of Polynomials Arising in Engineering and Science*. World Scientific, Singapore, 2005.

[SY09] M. Sagraloff and C. Yap. An efficient and exact subdivision algorithm for isolating complex roots of a polynomial and its complexity analysis, 2009.

[TY90] K. Thull and C. Yap. "A Unified Approach to HGCD Algorithms for polynomials and integers", 1990. Manuscript.

[vHM02] M. van Hoeij and M.B. Monagan. A modular GCD algorithm over number fields presented with multiple extensions. In *ISSAC '02*, pages 109–116, 2002.

[Vin08] J. Vince. *Geometric Algebra for Computer Graphics*. Springer, London, 2008.

[vzGG03] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[Yap00] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 2000.

[Yas91] M. Yassine. Matrix Mixed-Radix Conversion For RNS Arithmetic Architectures. In *Pro-*

ceedings of 34th Midwest Symposium on Circuits and Systems, 1991.

[ZV02] Z. Zilic and Z.G. Vranesic. A deterministic multivariate interpolation algorithm for small finite fields. *Computers, IEEE Transactions on*, 51(9):1100–1105, 2002.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken,

Pavel Emeliyanenko