

High-performance Polynomial GCD Computations on Graphics Processors

Pavel Emeliyanenko
Max-Planck-Institut für Informatik,
Saarbrücken, Germany
asm@mpi-inf.mpg.de

ABSTRACT

We propose an algorithm to compute a greatest common divisor (GCD) of univariate polynomials with large integer coefficients on Graphics Processing Units (GPUs). At the highest level, our algorithm relies on modular techniques to decompose the problem into subproblems that can be solved separately. Next, we employ resultant-based or matrix algebra methods to compute a GCD of each modular image in parallel. Our approach exhibits block structure to distribute the computation of a single modular GCD over several thread blocks, and thus to remove any hardware limitations on the maximal size of polynomials that can be handled. To “combine” all modular results, we have adopted Mixed-Radix Conversion (MRC) algorithm running on the GPU. Our approach shows a significant speed-up over host-based GCD algorithm from Maple 13.¹

KEYWORDS: Parallel computing, GPGPU, CUDA, symbolic algorithms, GCD, modular techniques

1. INTRODUCTION

Computing a greatest common divisor (GCD) of two polynomials with integer coefficients is a highly-demanding operation in computer algebra systems. It is also well-known for its resistance to effective parallel solution, and as a result the GCD can easily become the bottleneck of many basic applications. A classical approach to computing GCD in a polynomial domain is the one of Brown [3]. The algorithm uses modular homomorphism to reduce the problem to computations modulo several primes, process the modular images separately, and then recover a final result via Chinese remaindering. The modular approach inhibits the growth of intermediate results, and already bears some parallelism. However, Euclid’s algorithm, used in its core to compute a GCD in a finite field, hardly admits any parallelization. Such a modular structure might

be adequate for traditional multicore platforms but does not map well to the GPU due to the lack of fine-grained parallelism. Apart from Euclid’s algorithm, there is a class of matrix algebra methods [18, 19] to compute a GCD. These algorithms are traditionally used for polynomials with inexact coefficients arising in various numeric applications, and require more arithmetic operations compared to that of Euclid’s algorithm. Yet, their ability to describe the problem in a neatly structured way, makes them very attractive for realization on massively parallel architectures.

In this work we develop an algorithm to compute a GCD of univariate polynomials with integer coefficients on Graphics Processing Units (GPUs). The requirements is to have a common solution for polynomials of *arbitrary* degrees and very large coefficient bit-lengths. Our algorithm exploits the same “divide-conquer-combine” principle as the modular algorithm of Brown. Based upon previous experience with resultants computation [8, 6], we applied the theory of structured matrices [12] to develop an algorithm which works in a finite field and permits efficient realization on the GPU. However, we want to emphasize that the resultants and GCDs are symbolic algorithms of *different nature*, and cannot be viewed as a simple extension of one another. Furthermore, *the novelty of our approach is that it relies on block structure to distribute the computation of a single GCD across numerous thread blocks.*² In the essence, this allows us to handle the inputs of unlimited size, restricted only by the hardware resources.

To finally recover the coefficients of a GCD polynomial, we use Mixed-Radix (MR) conversion algorithm. This approach is suited for GPU realization in terms of controlled thread occupancy and fine-grained parallelism. Potentially, it can handle *any* number of modular residues, loaded and processed in chunks. At the end, large integers are restored from their MR representation on the host machine. In our implementation, we extensively use double-precision computations to realize efficient finite field arithmetic

¹www.maplesoft.com

²For definition of a thread block, see Section 3.

(see Section 4.2) assuming Fermi architecture [1]. However, our algorithm is backward-compatible with previous generation Tesla GPUs [15] by enabling reduced precision modular arithmetic developed in [8, 6].

The rest of the paper is organized as follows. In Section 2, we give an overview to the theory of structured matrices and design a sequential GCD algorithm. Section 3 is devoted to GPU architecture and CUDA framework. Then, in Section 4 we discuss the realization of modular GCD and MRC subalgorithms, we also briefly consider the modular computations on the GPU. Finally, in Section 5 we demonstrate the performance of our method by comparing it with a GCD algorithm from Maple 13.

2. PRELIMINARIES

In this section we establish the connection between certain matrices and a GCD of two polynomials. We briefly discuss the theory of structured matrices and the generalized Schur algorithm, and finally derive a sequential algorithm which will be the main object of manipulations in the following sections of the paper. Throughout the paper, we assume that f and g are two univariate polynomials with integer coefficients ($f, g \in \mathbb{Z}[x]$) of degrees p and q , respectively. Without loss of generality, we further assume that $p \geq q$.

2.1. Sylvester's Matrix and GCD Theorem

To begin with, let us define a matrix S whose entries are formed by coefficients of f and g :

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

The matrix $S \in \mathbb{Z}^{n \times n}$ ($n = p + q$) is called *Sylvester's matrix* associated with f and g . The following theorem relates S and a GCD of two polynomials.

Theorem 1 *The last nonzero row of Sylvester's matrix put in echelon form, using row transformations only, gives the coefficients of a GCD. For the proof, see [13].*

In other words, if we triangulate the matrix S , for instance, using Gaussian elimination, we obtain a GCD in the last nonzero row of the triangular factor. In the next section, we discuss the algorithms for triangular factorization of structured matrices.

2.2. Factorization of Structured Matrices

Let $S \in \mathbb{Z}^{n \times n}$ be Sylvester's matrix as defined in Section 2.1. The matrix S is shift-structured (*quasi-Toeplitz*) because it satisfies a *displacement equation* [12]:

$$S - ZSZ^T = GB^T.$$

Here $Z \in \mathbb{Z}^{n \times n}$ is a downshift matrix zeroed everywhere except for 1's on its first subdiagonal; $G, B \in \mathbb{Z}^{n \times 2}$ are the *generator matrices* implying that GB^T has a constant rank 2 which is called a *displacement rank* of S . From the equation above, it is clear that we need $\mathcal{O}(n)$ parameters to fully describe the matrix S .

To compute the factorization of S , we can use the Schur algorithm [12, 4] that operates entirely on matrix generators. Unfortunately, the algorithm works under assumption that matrix is *strongly-regular*,¹ which is often not the case for Sylvester's matrix. To get around this difficulty, we can triangulate a symmetric matrix $M = S^T S$ instead (which is strongly-regular by definition) and arrive at orthogonal factorization of S . Indeed, if M is factored in the form: $M = R^T R$, then it follows that R is a triangular factor in QR-factorization of S :

$$M = S^T S = (QR)^T QR = R^T Q^T QR = R^T R,$$

where we use the fact that: $Q^T Q = I$ since Q is orthogonal. The matrix M is structured with respect to the following equation:

$$M - ZMZ^T = GJG^T \text{ with } G \in \mathbb{Z}^{n \times 4}, J = I_2 \oplus -I_2,$$

where I_r is an $r \times r$ identity matrix, and \oplus denotes the Kronecker sum. Remarkably, we do not need to compute the entries of M explicitly because its generator G can be expressed in terms of the coefficients of the original polynomials:

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

Hence, to compute a GCD, we can apply the Schur algorithm to the matrix M . The Schur algorithm yields matrix factorization by iteratively computing the *Schur complements* of leading submatrices.² Due to low-rank displacement representation of a matrix used by the algorithm, the factorization can be computed in $\mathcal{O}(n^2)$ arithmetic operations, see [12, p. 323].

¹Meaning that its leading principal minors are non-singular.

²Schur complement arises in the course of block Gaussian elimination. For a submatrix H_{00} in $H = [H_{ij}]$, ($i, j = \{0, 1\}$), the Schur complement Y is defined as: $Y = H_{11} - H_{10}H_{00}^{-1}H_{01}$.

2.3. Derivation of the Algorithm

One step of the Schur algorithm can be written in terms of generator recursions. Let G_i be the generator matrix of size $(n-i) \times 4$ in step i ($i = 0, \dots, n-1$). Formally speaking, G_i is a compact representation of the Schur complement of an $i \times i$ leading block of M . The generator G_{i+1} in the next step obeys the following relation ([12, p. 341]):

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \left\{ G_i - (I_{n-i} - Z_i)G_i \frac{Jg_i^T g_i}{g_i Jg_i^T} \right\} \Theta_i, \quad (2)$$

where Z_i is a matrix obtained from Z by deleting the first i rows and columns; g_i is a top row of the generator matrix G_i ; and Θ_i is an arbitrary 4×4 matrix satisfying: $\Theta_i J \Theta_i^T = J$. A usual choice for Θ_i is to reduce the top row of G_i to the special form: $g_i \Theta_i = (\nu_i \ 0 \ 0 \ 0)$ which can be achieved by orthogonal transformations (Givens or Householder). Substituting this Θ_i into (2) leads to simple generator recursions. Yet, in order for this to work in a prime field, we need to adopt square root and division free transformations in the spirit of [9]. This approach works well for asymmetric generator pair [6] but is costly in our ‘‘symmetric’’ case. Hence, we propose to set $\Theta_i = I$ and use the relation (2) directly.

We proceed by defining $L^i = G_i J g_i^T$ in step i of Schur algorithm. Observe that, L^i is precisely an $(i+1)$ th row of the triangular factor of M ([12, p. 341]). Hence, the goal of the algorithm is to extract the last nonzero L^i . Next, to minimize the number of finite field divisions, we can collect all factors $g_i J g_i^T$ in a common denominator. Moreover, it follows that: $L^i = g_i J g_i^T$, hence we do not need to compute the divisors separately.

By unfolding the matrix recursions (2), we arrive at Algorithm 1. In the pseudocode, 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 18–32 where all four columns participate. This is because the first q rows ($p \geq q$) of G in (1) have only *two* non-zero entries, hence there is no need to update \mathbf{c} and \mathbf{d} in the first q steps. Henceforth, we will refer to these parts of the algorithm as ‘‘lite’’ and ‘‘full’’ iterations, respectively. Lines 14–16 are needed 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 the GCD in lines 29–31, 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 due to obvious reasons since the degree of $\text{GCD}(f, g)$ is at most as large as the degree of either polynomial.

Algorithm 1 Serial GCD algorithm

```

1: procedure GCD( $f$  : Polynomial,  $g$  : Polynomial)
2:    $p = \text{degree}(f)$ ,  $q = \text{degree}(g)$ ,  $n = p + q$ ,  $\text{det} = 1$ 
3:   let  $G = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$   $\triangleright$  setup the matrix generator as in (1)
4:   for  $i = 0$  to  $q - 1$  do  $\triangleright$  initial  $q$  iterations are ‘‘lite’’
5:     for  $j = i$  to  $n - 1$  do  $\triangleright$  the  $G$ ’s top row:  $g_i = \{a_i, b_i, 0, 0\}$ 
6:        $L_j^i = a_i a_j + b_i b_j$   $\triangleright$  triangular factor:  $L^i = G_i J g_i^T$ 
7:     od
8:     for  $j = i + 1$  to  $n - 1$  do  $\triangleright$  compute the next gen.:  $G_{i+1}$ 
9:        $F_j^i = L_j^i - L_{j-1}^i$   $\triangleright$  matrix prod.:  $F^i = (I_{n-i} - Z_i) \cdot L^i$ 
10:       $a_j = a_j \cdot L_j^i - a_i \cdot F_j^i$ ,  $b_j = b_j \cdot L_j^i - b_i \cdot F_j^i$ 
11:    od
12:     $\text{det} = \text{det} \cdot L_j^i$   $\triangleright$  collect the denominators
13:  od
14:  for  $j = q$  to  $n - 1$  do  $\triangleright$  bring to the common denom.
15:     $c_j = c_j \cdot \text{det}$ ,  $d_j = d_j \cdot \text{det}$ 
16:  od  $\triangleright$  the remaining  $p$  ‘‘full’’ iterations:
17:  for  $i = q$  to  $n - 1$  do  $\triangleright$  the  $G$ ’s top row:  $g_i = \{a_i, b_i, c_i, d_i\}$ 
18:    for  $j = i$  to  $n - 1$  do  $\triangleright$  triangular factor:  $L^i = G_i J g_i^T$ 
19:       $L_j^i = a_i a_j + b_i b_j - c_i c_j - d_i d_j$ 
20:    od
21:     $\triangleright$  the next gen.:  $G_{i+1} = G_i \cdot L^i - (I_{n-i} - Z_i) \cdot L^i \cdot g_i$ 
22:    for  $j = i + 1$  to  $n - 1$  do
23:       $F_j^i = L_j^i - L_{j-1}^i$   $\triangleright$  matrix prod.:  $F^i = (I_{n-i} - Z_i) \cdot L^i$ 
24:       $a_j = a_j \cdot L_j^i - a_i \cdot F_j^i$ ,  $b_j = b_j \cdot L_j^i - b_i \cdot F_j^i$ 
25:       $c_j = c_j \cdot L_j^i - c_i \cdot F_j^i$ ,  $d_j = d_j \cdot L_j^i - d_i \cdot F_j^i$ 
26:    od
27:     $\triangleright$  if generator columns are not linearly independent:
28:    if ( $a_{i+1} = c_{i+1}$  and  $b_{i+1} = d_{i+1}$ ) then
29:      return  $(L_1^1, L_{i+1}^1, \dots, L_{n-1}^1) / L_1^1$   $\triangleright$  return monic gcd
30:    fi
31:  od
32:  return 1  $\triangleright$  polynomials are coprime
33: end procedure

```

3. GPU ARCHITECTURE

In this work we primary focus on GPUs supporting CUDA framework, although the ideas and principles formulated are common to other architectures as well. At the highest level, the GPU consists of a set of Multiprocessors (MPs), the number of which can vary in a broad range. Each multiprocessor, in its turn, has a number of scalar, in-order processors which execute the same program in parallel using threads. Threads running on the GPU are scheduled in groups called *warps* consisting of 32 threads. In other words, threads of a warp always execute the same instruction. Accordingly, when a branch condition causes a warp to diverge, all taken branch paths are executed serially. This model of execution is called SIMT (Single Instruction Multiple Thread) which is akin to SIMD with a difference that it does not expose the vector organization to the software allowing the users to write thread-level parallel code.

Threads running on the GPU are organized in a grid of *thread blocks*, in a way that threads from the same block can communicate via fast shared memory and synchroniza-

tion barriers. There is no explicit communication provided between different blocks. A single CUDA program running on the GPU is referred to as *kernel*. The kernel is launched on a grid of thread blocks, and runs until all blocks terminate execution.

Each MP has a large *register file*. The registers are statically allocated to the threads scheduled on a multiprocessor. Besides, there is a small amount of *shared memory* (16–48 Kb) per MP that can be used for interthread communication. The main memory of the GPU – *global memory* – can be accessed simultaneously by all thread blocks. Its latency is much higher than that of shared memory, therefore it is highly recommended to use *coalesced access* optimizations to save the memory bandwidth.

4. IMPLEMENTATION

We begin with a general framework of the algorithm. Next, we briefly discuss the modular arithmetic on the GPU. Afterwards, we focus our attention on the implementation of modular GCDs, and conclude with the Mixed-Radix conversion algorithm.

4.1. Algorithm Structure

From the high-level perspective, our algorithm resembles the modular approach of Brown [3] and Collins. We start by mapping two polynomials f and g in $\mathbb{Z}[x]$ to a prime field for sufficiently many primes. The number of primes N can be estimated using bounds on the height of polynomial divisors.¹ A comprehensive discussion on this subject can be found in [2]. Despite the fact, that these bounds are usually quite pessimistic, we can exploit special knowledge we have about a GCD. First, the modular algorithm allows us to compute the degree of a GCD, hence we can apply *degree-aware* bounds. Next, we know the leading and trailing coefficients of a GCD polynomial which can greatly improve the accuracy of Binomial bound [2, Section 3.1]. Moreover, the reversal trick, appeared in the same source, can produce very good estimates when the trailing coefficient of a GCD is small. At last, since a GCD is a divisor of both f and g , we can apply the bounds to both polynomials and pick the minimal one. To sum up, the outline is the following: we start with the number of primes given by the height of original polynomials. Then, once modular GCDs are computed, we apply degree-aware bounds and enlarge the moduli set in case of need.

To be able to recover the GCD from modular images, we need to deal with “unlucky” primes which can be of

¹The height of a polynomial is a maximal bitlength of its coefficients.

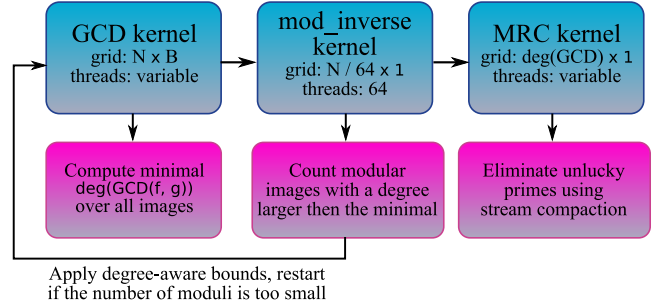


Figure 1. High-level view of a GCD algorithm. N : number of moduli; B : number of thread blocks per modular GCD, see Section 4.5

two kinds [20, Lecture IV]. “Unlucky” primes of the first kind, that is, those which divide the leading coefficients of either f or g , can be discarded in advance during the modular reduction of the polynomials on the host machine. To further refine the set of primes, we use the idea of Brown [3], which claims that we can keep those primes m for which the degree of a modular image $\text{GCD}(f|_m, g|_m)$ is *minimal* and discard the remaining ones.²

The GPU algorithm comprising three kernels is shown in Figure 1. The first GCD kernel is launched on a 2D grid and computes a set of modular GCDs. Additionally, it accumulates a minimal GCD degree among all modular images using CUDA atomic primitives. The next `mod_inverse` kernel is responsible for computing modular inverses of leading coefficients for each GCD image (see line 30 in Algorithm 1). We use a GPU implementation of Montgomery inverse algorithm, see [8, Appendix A]. The input is processed in chunks of size 64. In addition, in `mod_inverse` kernel we count the number of “unlucky” primes using parallel reduction and atomic primitives. Once this is done, we apply degree-aware bounds on the CPU to check if the number of primes (discarding “unlucky” ones) suffices to recover the result. If not, the algorithm is restarted with extended moduli set. The last `MRC_kernel`, discussed in Section 4.6, computes the mixed-radix (MR) representation of GCD coefficients. In this kernel, we also eliminate the detected “unlucky” primes using stream compaction subalgorithm, see [7]. In the last step, we reconstruct the large integer coefficients by evaluating the Horner form on the MR digests using the CPU.

4.2. Modular Multiplication

The performance of the algorithm to a large extent relies on the efficiency of modular arithmetic. This could become quite challenging providing that GPUs have partic-

²Here, $f|_m$ denotes a polynomial with coefficients reduced modulo m .

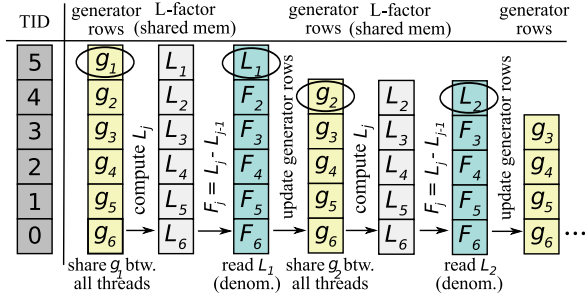


Figure 2. Schematic view of a simple GCD algorithm running with 6 threads indexed by TID.

ularly slow integer division and modulo (%) operations which should be avoided. In this work, we use 31-bit modular arithmetic optimized for Fermi architecture: unlike the previous generation graphics cards, Fermi GPUs natively support 32-bit integer multiplication, they also improve on the performance of double-precision arithmetic allowing to replace expensive division by floating-point operations.¹ Nevertheless, our implementation is backward-compatible: we refer the reader to [8, 6] where we have developed modular arithmetic for Tesla cards. One method to multiply two residues, which takes advantage of floating-point, appears in [16]. Yet, this method is not optimal on the graphics hardware as it was borrowed from the CPU code (and our benchmarks confirm that). To achieve better performance, we can use umulhi intrinsic available in CUDA.² The algorithm works as follows. For two 31-bit residues a and b and modulus m , we partition the product $a \cdot b$ as $2^{30}hi + lo$, and use a congruence:

$$\begin{aligned} a \cdot b &= 2^{30}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = \\ &= (2^{30}hi - m \cdot l) + lo = a \cdot b - l \cdot m = r, \end{aligned}$$

where $0 \leq \lambda < m$. It can be shown that $r \in [-m + \varepsilon; \varepsilon]$ for $0 \leq \varepsilon < m$. Hence, it only remains to adjust r by m in case of negative sign. Pseudocode is given below where we also use mantissa trick [11] to multiply two floating-points and truncate the result to a nearest integer in a single multiply-add instruction:

```

1: procedure MUL_MOD(a, b, m, inv) ▷ inv = (double)(1 << 30)/m
2:   ▷ given 31-bit residues a and b, computes a · b mod m
3:   hi = umulhi(a * 2, b * 2) ▷ compute 32 MSB of the product
4:   ▷ multiply in floating-point and truncate:
5:   rf = (double)(hi) * inv + (double)(3 << 31)
6:   r = a * b - double2loint(rf) * m ▷ partial residue
7:   return (r < 0 ? r + m : r) ▷ adjust by m if negative sign
8: end procedure

```

¹ Still, double-precision performance on Fermi is about 2 times lower than that of single-precision arithmetic, see [1].

² It computes 32 most significant bits of a product of two 32-bit integer operands. Interestingly enough, it does not have an x86 analogue.

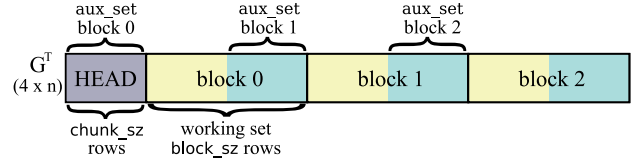


Figure 3. Partitioning of the matrix G ; aux_set: additional matrix rows loaded by a thread block.

4.3. Modular GCD: the Naive Approach

Algorithm 1 maps quite straightforwardly to the graphics hardware. In what follows, we assume that: $L_j^i = a_i a_j + b_i b_j - c_i c_j - d_i d_j$, and $F_j^i = L_j^i - L_{j-1}^i$, 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 the generator G . We designate one thread to process a single row g_j . In this way, the inner loop of the algorithm is completely vectorized.

Accordingly, the number of working threads decreases with the size of the generator matrix in each step. An iteration begins by choosing a thread which holds a current leading row of the generator matrix, and broadcasting the data to other threads. Each thread computes L_j^i and writes the results back to shared memory. Next, we perform the “downshift” by evaluating F_j^i , see lines 9 and 24 in Algorithm 1. Finally, participating threads update respective generator rows which completes the iteration. Two steps of the algorithm are shown in Figure 2. Note that, in the implementation we also distinguish between “lite” and “full” steps, however we omit these details for the time being.

4.4. Modular GCD: Improved Version

Apparently, the naive approach works fine as long as the generator matrix is small enough to fit in a single CUDA block because threads need to cooperate. To exploit block-level parallelism, we introduce some *data redundancy*, so that different thread blocks can run several algorithm steps without the need for communication.

For this, we partition the rows of matrix G in chunks of size $chunk_sz^3$ as in Figure 3. The first $chunk_sz$ rows of G will be denoted by HEAD. We select thread block to be of size $block_sz := chunk_sz \cdot 2$ 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 goal of

³Chunk size is chosen to be divisible by 32 (warp size) for reasons of efficiency.

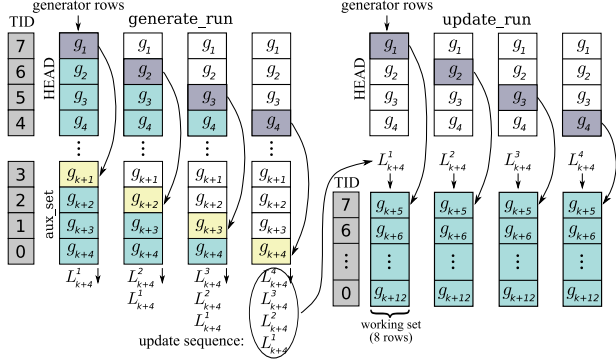


Figure 4. Sample workflow of the block algorithm with $\text{block_sz} = 8$. First, we generate a sequence $\{L_{k+4}^i\}$, and then apply it to the working set. White boxes mark elements not touched by the algorithm.

this data partitioning is to run chunk_sz iterations of the algorithm without communication. 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 needs the set of leading rows (HEAD). Second, for generator update, one computes $F_j^i = L_j^i - L_{j-1}^i$, hence we need additional chunk_sz rows before block's working set to compute the "preceding" elements L_{j-1}^i of L^i in each step.

The core of the algorithm comprises two phases: **gen_run** where we prepare the update sequence using the rows of HEAD and **aux_set**; and **update_run** where this sequence is applied to the working set. The workflow of the algorithm is shown in Figure 4. 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 leading row g_1 , and apply it to **aux_set** in order to compute the L-factors: $L_{k+1}^1, \dots, L_{k+4}^1$. Then, we update the subsequent rows g_2, \dots, g_4 of HEAD, as well as the rows g_{k+2}, \dots, g_{k+4} of **aux_set**. The row g_{k+1} is left *unchanged* because it needs a *previous* element L_k^1 for the update which is not available (for this reason, g_{k+1} is marked with a yellow box in the figure). We also keep L_{k+4}^1 which is required to update the first row g_{k+5} of the working set. Analogously, in the second step, we use g_2 to compute $L_{k+2}^2, \dots, L_{k+4}^2$, 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 L-factor, and the element L_{k+4}^2 is saved. As one can see from Figure 4, the rows of G are processed in a "stair-like" fashion. At the end of **gen_run**, we have a full sequence $L_{k+4}^4, \dots, L_{k+4}^1$.

In the next stage, **update_run**, the update sequence is applied to the working set. Note that, we do not need to process the rows of HEAD over again because the results are available from the previous stage. In each step, we take the leading row of HEAD and extract the last element of

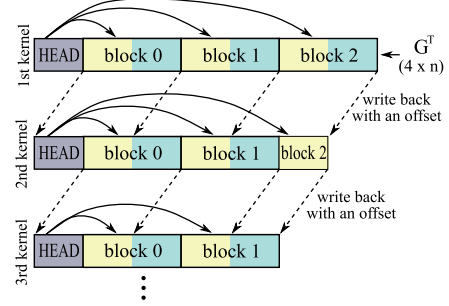


Figure 5. Block GCD algorithm running with several thread blocks. Each kernel corresponds to chunk_sz steps of a serial algorithm. Dashed arrows mark global memory regions where blocks write the results.

the update sequence to process the working set. It can be seen as though the update sequence is "pushed" onto the working set from the top: that is, in step 1 we take L_{k+4}^1 to compute F_{k+5}^1 , in step 2 we take L_{k+4}^2 to compute F_{k+5}^2 , etc. Remark that, in each step *all rows* of the working set $(g_{k+5}, \dots, g_{k+12})$ get updated, hence we achieve the full thread occupancy here.

4.5. Modular GCD: the Overall Approach

Having the basic routine at hand, we can now present the overall approach working with a number of thread blocks. From the host perspective, the algorithm consists of several kernel calls applying repeatedly the procedure from Section 4.4 where each call is equivalent to chunk_sz steps of Algorithm 1. 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 grid launch, see Figure 5. *Our experiments confirm that the optimal choice for the parameter chunk_sz is 128. This value we use in Section 5 for benchmarks.*

The host part of the algorithm is given in Algorithm 2. Here, we provide a template argument to the CUDA kernel in lines 8 and 17 to distinguish between "lite" and "full" iterations (as defined in Section 2.3). The number of "lite" iterations is controlled by `start_i` counter advanced by chunk_sz in each step. We also use double-buffering (`p_in` and `p_out`) to avoid data corruption due to simultaneous memory access. The kernel is launched on a 2D grid $N \times B$ with N is the number of moduli (see Section 4.1); and B is the number of blocks per modular GCD. At the beginning, we set: $B = (\text{p} + \text{chunk_sz}) / (2 * \text{chunk_sz})$. This number of blocks suffices because, by looking at the matrix G in (1), we see that the first two columns relevant during "lite" iterations have no more than $p+1$ *nonzero* entries each ($p \geq q$). Note that, in "full" iterations, B decreases every two steps

Algorithm 2 Host part of the block GCD algorithm

```

1: procedure GCD_HOST_PART(Polynomial f, Polynomial g)
2:   p = f.degree(), q = g.degree(), n = p + q ▷ assume: p ≥ q
3:   start_i = 0 ▷ setup iteration counter
4:   B = (p + chunk_sz)/(2 * chunk_sz)
5:   dim3 thids(chunk_sz) ▷ # of threads per block
6:   dim3 grid(N, B) ▷ grid of thread blocks
7:   while (1) { ▷ first run "lite" iterations
8:     gcd_block_kernel<chunk_sz, false><<< grid, thids >>>
9:     (p_out, p_in, start_i, 0) ▷ kernel launch
10:    start_i += chunk_sz ▷ increase iteration counter
11:    swap(p_in, p_out) ▷ "ping-pong" memory access
12:    if (start_i ≥ q) break ▷ finished "lite" iterations
13:  }
14:  ofs = (q - start_i + chunk_sz) ▷ compute mem. ofs
15:  while (1) { ▷ run "full" iterations
16:    dim3 grid(N, B) ▷ # of blocks B decreases
17:    gcd_block_kernel<chunk_sz, true><<< grid, thids >>>
18:    (p_out, p_in, start_i, ofs) ▷ kernel launch
19:    start_i += chunk_sz ▷ increase iteration counter
20:    swap(p_in, p_out) ▷ "ping-pong" memory access
21:    sz = n - start_i ▷ the current size of generator matrix
22:    if (sz ≤ 3 * chunk_sz) break ▷ break if sz is small
23:    B = (sz + chunk_sz - 1)/(2 * chunk_sz)
24:  }
25:  ▷ simple kernel runs the remaining iterations
26:  gcd_simple_kernel<<< grid, thids >>>(p_in, p_out)
27: end procedure

```

of the algorithm.

4.6. Mixed Radix Conversion

When the set of modular GCDs is ready, we apply the Chinese remaindering to recover the coefficients of a resulting polynomial. To reconstruct a number X from its residues (x_1, x_2, \dots, x_N) modulo primes (m_1, m_2, \dots, m_N) we use Mixed Radix Conversion (MRC) algorithm [21]. This algorithm associates X with a set of mixed-radix (MR) digits $\{\alpha_i\}$: $X = \alpha_1 M_1 + \alpha_2 M_2 + \dots + \alpha_N M_N$, where $M_1 = 1$, $M_j = m_1 m_2 \dots m_{j-1}$ ($j = 2, \dots, N$). The MR digits can be computed as follows ($i = 1, \dots, N$):

$$\begin{aligned}
 \alpha_1 &= x_1, \alpha_2 = (x_2 - \alpha_1)c_2 \bmod m_2, \\
 \alpha_3 &= ((x_3 - \alpha_1)c_3 - (\alpha_2 M_2 c_3 \bmod m_3)) \bmod m_3, \dots \\
 \alpha_i &= ((x_i - \alpha_1)c_i - (\alpha_2 M_2 c_i \bmod m_i) - \dots \\
 &\quad - (\alpha_{i-1} M_{i-1} c_i \bmod m_i)) \bmod m_i,
 \end{aligned}$$

where $c_i = (m_1 m_2 \dots m_{i-1})^{-1} \bmod m_i$ are precomputed in advance, and M_i are evaluated "on-the-fly". It is worth noting that, we can arrange the moduli in *increasing order*, that is: $m_1 < m_2 < \dots < m_N$. So that, the expressions of the form $\alpha_j M_j c_i \bmod m_i$, for $j < i$, can be evaluated without the modular reduction of α_j since $\alpha_j < m_i$. The same argument applies to updating M_i 's. A simple parallel algorithm computing the MR digits is given in Figure 6. Yet, this solution does not work as soon as the "capacity" of thread block is exceeded. We can deal with this by process-

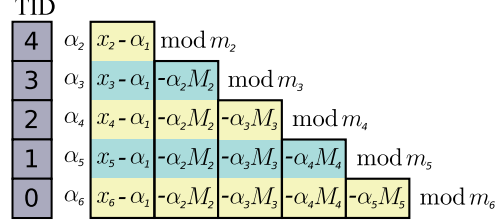


Figure 6. Basic MRC algorithm working with 5 threads indexed by TID. Besides digits α_j , in step i , each thread computes $M_{i+1} = M_i m_i \bmod m_j$.

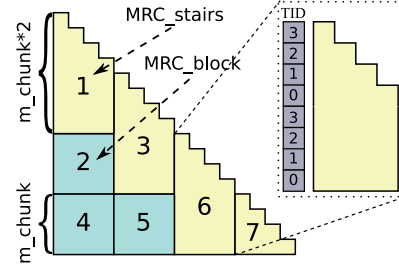


Figure 7. Improved MRC algorithm running with $m_chunk := 4$ threads. The numbers show the order in which the subalgorithms are applied.

ing several digits in one thread but this is not a complete solution, moreover it shows very bad thread occupancy in the last iterations when the number of idle threads increases.

For an optimal solution, we observe that, the number of 31-moduli needed by the majority of applications is quite moderate (2–3 thousand at most). Hence, we propose to split the inputs in *chunks* (of size m_chunk) and compute MR digits in a loop using a one thread block. This is a good compromise between a more general approach consisting of several kernel calls (since we save on memory transfers) and a simple parallel one (which is limited by the maximal block size).

Geometrically, our approach can be illustrated by covering a triangle using two shapes (subroutines): MRC_stairs and MRC_block, see Figure 7. The algorithm runs on the GPU with m_chunk threads per block, its pseudocode is given in Algorithm 3. MRC_stairs has the same structure as the basic algorithm in Figure 6, with the exception that 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 7, we ensure that all threads are *occupied* in each step. The purpose of the procedure MRC_block is to optionally preload and update a set of m_chunk MR digits using the ones computed in the preceding MRC_stairs call(s).

Since the number of processed chunks needed for

Algorithm 3 MRC block algorithm (GPU part)

```
1: procedure MRC_BLOCK_ALG( chunk[0 . . . n_chunks - 1] )
2:   MRC_load(chunk[0], chunk[1])      ▷ load first two chunks
3:   for i = 0 to n_chunks - 1 do
4:     MRC_stairs(chunk[i], chunk[i + 1])
5:     if (i < n_chunks - 1) then
6:       MRC_load(chunk[i + 1])      ▷ preload the next chunk
7:     fi
8:     for j = 0 to i - 1 do      ▷ update a new chunk i + 1 using
9:       MRC_block(chunk[i + 1], chunk[j])  ▷ previous ones
10:    od
11:  od
12:  MRC_stair_last(chunk[i + 1])
13: end procedure
```

MRC_block calls increases in each step, the main challenge of the algorithm is where to keep the data. We solve this by *parameterizing* the kernel with the number of chunks (which is usually quite small) while keeping the parameter `m_chunk` flexible. When the number of chunks is small, all data is stored in registers and shared memory. By reaching a certain threshold, the data is placed in GPU's local memory.¹ The selection of a concrete kernel depends on the actual number of moduli, and is based on heuristics favoring small blocks to large ones by adjusting the parameter `m_chunk`.

5. EVALUATION AND CONCLUSIONS

We have run experiments on a desktop with 2.8GHz 8-Core Intel Xeon W3530 (8 MB L2 cache) CPU and NVIDIA GTX580 graphics card under 64-bit Linux platform. To reconstruct large integers on the host machine, we have used GMP 5.0.1 library.² To optimize register usage, we have specified *launch bounds* for each GPU kernel, which gave us about 30% speed-up on the average.

We have compared our algorithm with a CPU-based algorithm from 64-bit Maple 13³ as we are not aware of any analogous GPU implementation. We remark that, *univariate GCD is a built-in algorithm in Maple*: meaning that, it is provided by a library compiled from C++ sources and also uses GMP for exact number types. Maple's implementation relies on several algorithms chosen according to the input size. These, for instance, include, heuristic GCD, EZGCD and sparse modular GCD algorithms, see [14] for comparison. For very large inputs, Maple employs asymptotically fast Half-GCD algorithm [17].

Benchmarks for single GCDs are given in Table 1. We have varied different parameters such as degrees, bitlength of coefficients, and density of polynomials (number of

¹Observe that, on Fermi all accesses to local memory are cached.

²<http://gmplib.org>

³www.maplesoft.com

Table 1. Benchmarks for Single Polynomial GCDs. deg(f / g) and bits(f / g): degrees and coefficient bitlength of polynomials *f* and *g*, respectively

configuration	deg(GCD)	GPU	Maple
deg(f / g): 923 / 412 bits(f / g): 300 / 200, sparse	100	5.4 ms	56 ms
deg(f / g): 1000 / 400 bits(f / g): 300 / 200, dense	100	5.6 ms	337 ms
deg(f / g): 2300 / 2100 bits(f / g): 35 / 1015, dense	1400	12.7 ms	1.9 s
deg(f / g): 3669 / 3957 bits(f / g): 3000 / 2000, -	3257	0.41 s	3.6 s
deg(f / g): 4900 / 4900 bits(f / g): 46 / 46, dense	2500	46 ms	5.4 s
deg(f / g): 10000 / 10000 bits(f / g): 162 / 165, -	5000	0.3 s	70 s
deg(f / g): 10000 / 10000 bits(f / g): 3733 / 748, -	5000	1.22 s	80.3 s

Table 2. Computing a Batch of N GCDs for Random Polynomials. Abbreviations as in Table 1

configuration	N	deg (GCD)	GPU	Maple
deg(f / g): 900 / 800 bits(f / g): 150 / 150, sparse	50	200 (avg.)	106 ms	2.26 s
deg(f / g): 900 / 800 bits(f / g): 150 / 150, sparse	100	200 (avg.)	233 ms	4.04 s
deg(f / g): 1000 / 900 bits(f / g): 170 / 180, sparse	200	100 (avg.)	0.72 s	10.0 s
deg(f / g): 900 / 700 bits(f / g): 150 / 100, dense	150	400 (avg.)	0.27 s	57.7 s

non-zero terms). One can see that Maple's algorithm performs worse for dense polynomials while our (matrix-based) approach is insensitive to polynomial density. On the other hand, our algorithm achieves smaller speed-up for polynomials with large coefficient bitlength. This is because the competing algorithms are of different nature: the modular approach needs to process more images when the bitlength increases while, for instance, heuristic GCD of Maple maps polynomials to large integers, and hence is not directly affected by coefficient growth. It is worth noting that, Maple's performance drops significantly for very large polynomial degrees: we anticipate that, in this case, CPU cache size might already become a bottleneck.

From Table 1, it becomes clear that for small inputs the GPU resources are underutilized (as a result, the performance gain is not very big). To this end, we have also carried out the experiments for a batch of low degree ran-

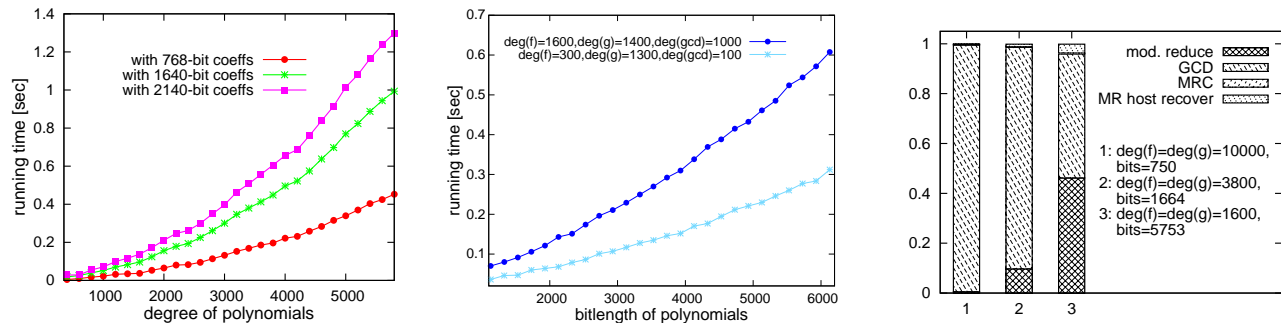


Figure 8. Execution Time vs. Polynomial Degree (left) and Coefficient Bitlength (middle); Breakdown of the Running Time (right), mod.reduce: modular reduction on the host, MR host recover: recover large integers from MR digits

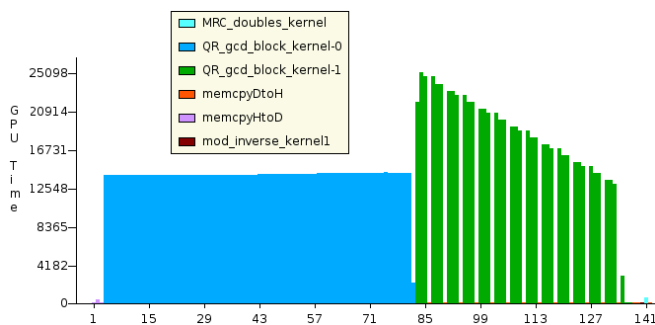


Figure 9. Kernel Statistics for the Last Entry in Table 1

dom polynomials, which is a *key routine in a multivariate GCD algorithm*. To support our statement, we recall that, the classical Brown’s multivariate GCD [10, Alg. 7.1] first applies modular and evaluation homomorphisms to reduce the problem to a *large set* of univariate GCDs, computes them separately, and then recovers a final result using interpolation. Its further developments [22, 5] exploit the same idea but use probabilistic techniques or more sophisticated “sparse” interpolation. The only exception is a heuristic GCD algorithm which, instead, maps a polynomial GCD to integer domain: however, in multivariate domain, it is generally inferior to modular algorithms, see [14]. The timings for batch GCD computations are listed in Table 2. Here, the batch size (the number GCDs computed) is given by the parameter N . From the first two rows of the table, one can see that the GPU hardware is saturated when N is somewhere between 50 and 100. Altogether, the speed-up attained for batch GCDs is clearly more impressive.

In Figure 8, we examine the performance versus polynomial degree and coefficient bitlength. In the left diagram, the dependency is *sublinear* since the complexity of the GPU algorithm is linear by design, however, when the degree increases, more thread blocks need to work cooperatively on a single GCD resulting in sublinear complexity. In contrast,

the bitlength only causes the number of moduli to increase (the number of independent thread blocks), and the performance scales linearly in the middle diagram. A histogram in Figure 8 shows how different stages of the algorithm contribute to the overall time: for high degrees the time for GCD kernel is dominating. However, for moderate degree and large bitlength, the time for modular reduction on the host machine becomes noticeably large. Figure 9 shows CUDA profiler output for the last example in Table 1. In the figure, blue and green shapes correspond to “lite” and “full” iterations of GCD algorithm, respectively. The height plot for “full” iterations has a trapezoid form because the number of thread blocks decreases in each step.

We have presented an algorithm to compute the GCD of univariate polynomials with integer coefficients. Our benchmarks show a considerable speed-up over GCD algorithm from Maple 13. To conclude, symbolic algorithms is an emerging area of application of graphics processors, and we see a lot of avenues for future work. As a primary objective, we would like to realize the modular reduction directly on the GPU since, as indicated by the timings above, it might become time-consuming for polynomials with large coefficient bitlengths. Another promising task is to extend the algorithm to *multivariate domain* by means of multi-point interpolation. We would also like to consider other symbolic algorithms which admit efficient realization on the GPU. These, for instance, include square-free factorization of polynomials, subresultants, real root isolation, etc. Besides, note that, our results on parallelization of Schur algorithm can be found useful in other areas since the factorization of structured matrices has a wide range of significant applications going far beyond the polynomial algebra.

REFERENCES

- [1] “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi”. Whitepaper, NVIDIA Corp., 2010.

- [2] J. Abbott. “Bounds on Factors in $\mathbb{Z}[x]$ ”. *ArXiv e-prints*, April 2009.
- [3] 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.
- [4] S. Chandrasekaran and A. H. Sayed. “A Fast Stable Solver for Nonsymmetric Toeplitz and Quasi-Toeplitz Systems of Linear Equations”. *SIAM J. Matrix Anal. Appl.*, 19:107–139, 1998.
- [5] 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.
- [6] 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.
- [7] P. Emeliyanenko. “Accelerating Symbolic Computations on NVIDIA Fermi”. In *GTC ’10*, 2010. Poster presentation.
- [8] P. Emeliyanenko. “Modular Resultant Algorithm for Graphics Processors”. In *ICA3PP ’10*, pages 427–440, Berlin, Heidelberg, 2010. Springer-Verlag.
- [9] 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.
- [10] K.O. Geddes, S.R. Czapor, and G. Labahn. *ALGORITHMS FOR COMPUTER ALGEBRA*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.
- [11] C. Hecker. “Let’s Get to the (Floating) Point”. *Game Developer Magazine*, pages 19–24, 1996.
- [12] T. Kailath and A. Sayed. “Displacement structure: theory and applications”. *SIAM Review*, 37:297–386, 1995.
- [13] M. A. Laidacker. “Another Theorem Relating Sylvester’s Matrix and the Greatest Common Divisor”. *Mathematics Magazine*, 42(3):126–128, May 1969.
- [14] H.-C. Liao and R. J. Fateman. “Evaluation of the Heuristic Polynomial GCD”. In *ISSAC ’95*, pages 240–247. ACM Press, 1995.
- [15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. *Micro, IEEE*, 28(2):39–55, 2008.
- [16] M. M. Maza and W. Pan. “Fast polynomial multiplication on a GPU”. *Journal of Physics: Conference Series*, 256(1):012009, 2010.
- [17] K. Thull and C. Yap. “A Unified Approach to HGCD Algorithms for polynomials and integers”, 1990. Manuscript.
- [18] D. Triantafyllou and M. Mitrouli. “Two Resultant Based Methods Computing the Greatest Common Divisor of Two Polynomials”. In *Numerical Analysis and Its Applications*, volume 3401, pages 519–526. Springer Berlin / Heidelberg, 2005.
- [19] D. Triantafyllou, M. Mitrouli, and N. Karcianas. “Resultant Based Methods Computing the Greatest Common Divisor of Several Polynomials”. In *Proceedings of Mediterrean Conference on Control and Automation*, pages 387–392, 2005.
- [20] C. K. Yap. *FUNDAMENTAL PROBLEMS IN ALGORITHMIC ALGEBRA*. Oxford University Press, 2000.
- [21] M. Yassine. “Matrix Mixed-Radix Conversion For RNS Arithmetic Architectures”. In *Proceedings of 34th Midwest Symposium on Circuits and Systems*, 1991.
- [22] R. Zippel, editor. *EFFECTIVE POLYNOMIAL COMPUTATION*. Kluwer Academic Publishers, 1993.