A complete modular resultant algorithm targeted for realization on graphics hardware

Pavel Emeliyanenko
Max-Planck Institute for Informatics
Saarbrücken, Germany
asm@mpi-sb.mpg.de

ABSTRACT

This paper presents a complete modular approach to computing bivariate polynomial resultants on Graphics Processing Units (GPU). Given two polynomials, the algorithm first maps them to a prime field for sufficiently many primes, and then processes each modular image individually. We evaluate each polynomial at several points and compute a set of univariate resultants for each prime in parallel on the GPU. The remaining "combine" stage of the algorithm comprising polynomial interpolation and Chinese remaindering is also executed on the graphics processor. The GPU algorithm returns coefficients of the resultant as a set of Mixed Radix (MR) digits. Finally, the large integer coefficients are recovered from the MR representation on the host machine. With the approach of displacement structure [16] and efficient modular arithmetic [8] we have been able to achieve more than 100x speed-up over a CPU-based resultant algorithm from Maple 13.[1]

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Algebraic algorithms*

General Terms

Algorithms, Experimentation, Measurement, Performance, Theory

Keywords

CUDA, GPU, modular algorithm, parallel computing, polynomial resultants

1. INTRODUCTION

Resultants is a powerful algebraic tool in the quantifier elimination theory. Among their numerous and widespread applications, resultants play an important role in topological study of algebraic curves and computer graphics. However, despite the fact that this problem received a good deal of attention in the literature, resultants still constitute a major bottleneck for many geometric algorithms. This is mainly due to the rapid growth of coefficient bit-length and the degree of the resultant polynomial with respect to the initial parameters as well as the necessity to work in a complicated domain. We find it, therefore, very advantageous to try to utilize the incredible computational horsepower of Graphics Processing Units (GPUs) for this problem.

A classical resultant algorithm is the one of Collins [6]. The algorithm employs modular and evaluation homomorphisms to deal with expression swell during computation of resultants. Following the "divide-conquer-combine" strategy, it reduces the coefficients of input polynomials modulo sufficiently many primes. Then, several evaluation homomorphisms are applied recursively reducing the problem to the univariate case. Finally, a set of univariate resultants are computed using polynomial remainder sequences (PRS), see [11]. The final result is recovered by means of polynomial interpolation and the Chinese remainder algorithm (CRA).

This idea gave rise to the whole spectrum of modular algorithms: including sequential [20, 17], and parallel ones specialized for workstation networks [4] or shared memory machines [21, 15]. However, neither of these algorithms admits a straightforward realization on the GPU. The reason for that is because the modular approach exhibits only a *coarse-grained* parallelism since the PRS algorithm, lying in its core, hardly admits any parallelization. This is a good choice for traditional parallel platforms such as workstation networks or multi-core machines but not for massively-threaded architecture like that of GPU. That is why, we have decided to use an alternative method to solve the problem in the univariate case, namely, the approach of *displacement structure* [16]. In the essence, this method reduces computation of the resultant to the triangular factorization of a structured matrix. Operations on matrices generally map very well to the GPU's threading model. The displacement structure approach is traditionally applied in floating-point arithmetic, however using square-root and division-free modifications [10] we have been able to adapt it for a prime field. It is worth mentioning that *even though the PRS algorithm can also be made division-free* (which is probably the method of choice for a modular approach) this does not facilitate its the realization on the GPU.

In this work we extend our previous results [9] by porting the remaining stages of the resultant algorithm (polynomial interpolation and partly the CRA) to the graphics processor, thereby, minimizing the amount of work to be done on

[1] www.maplesoft.com

PASCO 2010, 21–23 July 2010, Grenoble, France.
Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ...$10.00.

the CPU. For the sake of completeness, we present the full approach here. We also use an improved version of the univariate resultant algorithm given in [9] which is based on a modified displacement equation, see Section 2.2. Additionally, we have developed an efficient stream compaction algorithm based on parallel reductions in order to eliminate "bad" evaluation points right on the GPU, see Section 4.1. What concerns the CRA, we compute the coefficients of the resultant polynomial using Mixed Radix (MR) representation [22] on the GPU without resorting to multi-precision arithmetic, and finally recover the large integers on the host machine. Foundation of our algorithm is a fast 24-bit modular arithmetic developed in [8]. The arithmetic rests on mixing floating-point and integer computations, and widely exploits the GPU multiply-add capabilities.

The organization of the paper is as follows. In Section 2 we formulate the problem in a mathematically concise way and give an introduction to displacement structure and the generalized Schur algorithm. Section 3 describes the GPU architecture and CUDA framework. Section 4 focuses on the algorithm itself including the main aspects of the GPU realization. In Section 5 we present experimental results and draw conclusions.

2. THEORETICAL BACKGROUND

In this section we give a definition of the resultant of two polynomials, and describe the displacement structure approach in application to univariate resultants and polynomial interpolation. We also briefly consider Chinese remaindering and the Mixed Radix representation of the numbers.

2.1 Polynomial resultants

Let f and g be two polynomials in $\mathbb{Z}[x, y]$ of y-degrees p and q respectively: $f(x, y) = \sum_{i=0}^{p} f_i(x)y^i$ and $g(x, y) = \sum_{i=0}^{q} g_i(x)y^i$. Let $R = res_y(f, g)$ denote the resultant of f and g with respect to y. The resultant R is the determinant of $(p + q) \times (p + q)$ Sylvester matrix S:

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

From the definition it follows that the resultant R is a polynomial in $\mathbb{Z}[x]$. However, using modular techniques one can reduce the problem to the univariate case. Computing univariate resultants is discussed in the next section.

2.2 Displacement structure of Sylvester matrix and the generalized Schur algorithm

Suppose we are given two polynomials $f, g \in \mathbb{Z}[x]$ of degrees p and q respectively ($p \geq q$) and the associated Sylvester matrix $S \in \mathbb{Z}^{n \times n}$ ($n = p + q$). Let us first assume that S is a strongly regular matrix.[1] The matrix S is structured with a *displacement rank* 2 because it satisfies the *displacement equation*:

$$S - FSA^T = GB^T \text{ where } F = Z_n \text{ and } A = Z_q \oplus Z_p.$$

Here $Z_n \in \mathbb{Z}^{n \times n}$ is a down-shift matrix zeroed everywhere except for 1's on the first subdiagonal. Accordingly, *generators* $G, B \in \mathbb{Z}^{n \times 2}$ are matrices whose entries can be deduced from the matrix S directly by inspection:

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

Consequently, the matrix S can fully be described by its generators. Our goal is to obtain an LDU^T-factorization of S where the matrices L and U are triangular with unit diagonals, and D is a diagonal matrix. Having this factorization, the resultant is: $det(S) = det(D) = \prod_i^n d_{ii}$ (the product of diagonal entries of D).

The generalized Schur algorithm [16, 5] computes the matrix factorization by iteratively computing the *Schur complements* of leading submatrices. The Schur complement R of a submatrix M_{00} in $M = [M_{ij}]$, $(i, j = \{0, 1\})$ is defined as: $R = M_{11} - M_{10}M_{00}^{-1}M_{01}$. The idea of the algorithm is to rely on a low-rank displacement representation of a matrix. The displacement equation, preserved under all transformations, allows us to derive the matrix factorization by operating solely on matrix generators. As a result, *the matrix factorization can be computed in $\mathcal{O}(n^2)$ arithmetic operations*, see [16, p. 323].

In each step, the generators are transformed to a *proper* form. Let us denote the generator matrices in step i by (G_i, B_i). A generator $\overline{G}_i$ is said to be in a proper form if it has only *one* non-zero entry in its first row. The transformation is done by applying non-Hermitian rotation matrices Θ_i and Γ_i[2] such that $\overline{G}_i = (G_i\Theta_i)$ and $\overline{B}_i = (B_i\Gamma_i)$:

$$\overline{G}_i^T = \begin{bmatrix} \delta^i & a_1^i & a_2^i & \dots \\ 0 & b_1^i & b_2^i & \dots \end{bmatrix}, \overline{B}_i^T = \begin{bmatrix} \zeta^i & c_1^i & c_2^i & \dots \\ 0 & d_1^i & d_2^i & \dots \end{bmatrix}.$$

Once the generators are in proper form, the displacement equation yields: $d_{ii} = \delta^i\zeta^i$. To obtain the next generator G_{i+1} (and by analogy B_{i+1}) from $\overline{G}_i$ (or $\overline{B}_i$) we multiply its first column (the one with the entry δ^i or ζ^i) by corresponding down-shift matrix (F for $\overline{G}_i$ and A for $\overline{B}_i$) while keeping the other column intact, see [16]. In case of $\overline{G}_i$ this corresponds to shifting down all elements of the first column by one position, while for $\overline{B}_i$, in addition to the down-shift, the q-th entry of the first column (c_q^i) must be zeroed.[3] Accordingly, the length of generators decreases by one in each step of the algorithm.

2.3 Division-free rotations in non-Hermitian case

Note that the term non-Hermitian stands for the fact that we apply rotations to an asymmetric generator pair. To bring the generators to a proper form we need to find matrices Θ and Γ that satisfy: $\begin{bmatrix} a_0 & b_0 \end{bmatrix} \Theta = \begin{bmatrix} \delta & 0 \end{bmatrix}$, $\begin{bmatrix} c_0 & d_0 \end{bmatrix} \Gamma = \begin{bmatrix} \zeta & 0 \end{bmatrix}$, with $\Theta\Gamma^T = I$. This holds for the following matrices:

$$\Theta = \begin{bmatrix} c_0 & b_0 \\ d_0 & -a_0 \end{bmatrix}, \Gamma = \frac{1}{D} \begin{bmatrix} a_0 & d_0 \\ b_0 & -c_0 \end{bmatrix}, D = a_0c_0 + b_0d_0.$$

To get rid of expensive divisions, we use an idea similar to the one introduced for Givens rotations [10]. Namely, we postpone the division until the end of the algorithm by

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

[2] These matrices must satisfy: $\Theta\Gamma^T = I$, which preserves the displacement equation since: $G\Theta(B\Gamma)^T = GB^T$.

[3] This is because the matrix $A = Z_q \oplus Z_p$ is formed of two down-shift matrices.

keeping a common denominator for each generator column. Put it differently, we express the generators as follows:

$$\begin{array}{ll} \mathbf{a}^T = 1/l_a \cdot (a_0,\ a_1,\ \dots) & \mathbf{b}^T = 1/l_b \cdot (b_0,\ b_1,\ \dots), \\ \mathbf{c}^T = 1/l_c \cdot (c_0,\ c_1,\ \dots) & \mathbf{d}^T = 1/l_d \cdot (d_0,\ d_1,\ \dots), \end{array}$$

where $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$. Accordingly, the generator update $(\overline{G}, \overline{B}) = (G\Theta, B\Gamma)$ proceeds in the following way:

$$\begin{array}{ll} \overline{a}_i = l_a(a_i c_0 + b_i d_0) & \overline{b}_i = l_b(a_i b_0 - b_i a_0), \\ \overline{c}_i = l_c(c_i a_0 + d_i b_0) & \overline{d}_i = l_d(c_i d_0 - d_i c_0), \end{array}$$

here $\overline{G} = (\overline{\mathbf{a}}, \overline{\mathbf{b}})$ and $\overline{B} = (\overline{\mathbf{c}}, \overline{\mathbf{d}})$. Moreover, it follows that the denominators are *pairwise* equal, thus, we can keep only two of them. They are updated as follows: $\overline{l}_a = \overline{l}_d = \overline{a}_0$, $\overline{l}_c = \overline{l}_b = l_a l_c^2$.

Note that the denominators must be non-zero to prevent the algorithm from breaking down. It is guaranteed by a *strong-regularity* assumption introduced at the beginning. Yet, this is not always the case for Sylvester matrix. In Section 4.1 we discuss how to alleviate this problem.

2.4 Polynomial interpolation

The task of polynomial interpolation is to find a polynomial $f(x)$, $deg(f) < n$, satisfying the set of equations: $f(x_i) = y_i$, for $0 \le i < n$. The coefficients a_i of f are given by the solution of the system: $V\mathbf{a} = \mathbf{y}$, where $V \in \mathbb{Z}^{n\times n}$ is a Vandermonde matrix: $V_{ij} = x_i^j$ $(i, j = 0, \dots, n-1)$. If we apply the generalized Schur algorithm to the following matrix $M \in \mathbb{Z}^{2n\times(n+1)}$:

$$M = \begin{bmatrix} V & -\mathbf{y} \\ I_n & \mathbf{0} \end{bmatrix}, \text{ where } I_n \text{ is } n \times n \text{ identity matrix,}$$

then after n steps we obtain the Schur complement R of V, such that: $R = \mathbf{0} - I_n V^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$, i.e., the desired solution. The matrix M has a displacement rank 2 and satisfies the equation:

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

here $F = diag(x_0 \dots x_{n-1}) \oplus Z_n$, $A = Z_{n+1}$, with $Z_n \in \mathbb{Z}^{n\times n}$ is a down-shift matrix. The generators $G \in \mathbb{Z}^{2n\times 2}$ and $B \in \mathbb{Z}^{(n+1)\times 2}$ have the following form:

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

Again, in each step of the algorithm we bring the generators to a proper form as outlined in Section 2.3. The next generator pair $(G_{i+1},\ B_{i+1})$ is computed in a slightly different manner (see [16]):

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i \overline{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i \overline{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$

here $\Phi_i = F_i - f_i I_i$, $\Psi_i = A_i(I_i - f_i A_i)^{-1}$, where F_i, A_i and I_i are obtained by deleting the first i rows and columns of matrices F, A and I_n respectively, f_i is an i-th diagonal element of F. Although, the equations look complicated at first glance, it turns out that the generator B does *not* need to be updated due to its trivial structure, see Section 4.3. After n steps of the algorithm, the product GB^T yields the solution of the system.

2.5 Chinese Remainder Algorithm (CRA)

The task of CRA is to reconstruct a number X from its residues $(x_1, x_2, \dots, x_k)$ modulo a set of primes $(m_1, m_2, \dots, m_k)$. A classical approach is to associate X with the *mixed-radix* (MR) digits $(\alpha_1, \dots, \alpha_k)$:

$$X = \alpha_1 V_1 + \alpha_2 V_2 + \dots + \alpha_k V_k,$$

where $V_1 = 1$, $V_j = m_1 m_2 \dots m_{j-1}$ $(2 \le j \le k)$. We use the algorithm [22] to compute the digits α_i as follows $(1 \le i \le k)$:

$$\begin{aligned} \alpha_1 &= x_1,\ \alpha_2 = (x_2 - \alpha_1)c_2 \bmod m_2, \\ \alpha_3 &= ((x_3 - \alpha_1)c_3 - (\alpha_2 V_2 c_3 \bmod m_3)) \bmod m_3,\ \dots \\ \alpha_i &= ((x_i - \alpha_1)c_i - (\alpha_2 V_2 c_i \bmod m_i) - \dots \\ &\qquad -(\alpha_{i-1} V_{i-1} c_i \bmod m_i)) \bmod m_i, \end{aligned}$$

where $c_i = (m_1 m_2 \dots m_{i-1})^{-1} \bmod m_i$ can be precomputed in advance. It is easy to see that one can compute the MR digits on the GPU because the algorithm exposes some parallelism.

3. CUDA PROGRAMMING MODEL

Starting with the G80 series, NVIDIA GPUs do not have separated fragment and vertex processors. Instead, they are unified in Streaming Multiprocessors (SMs) capable of running shader programs as well as general purpose parallel programs. For instance, the GTX 280 contains 30 SMs. The SM can be regarded as a 32-lane SIMD vector processor because a group of 32 threads called a *warp* always executes same instruction. Accordingly, a data-dependent branch causes a warp to *diverge*, i.e., the SM has to serialize execution of all taken branch paths. Different warps are free from the divergence problem.

CUDA [1] is a heterogeneous serial-parallel programming model. In other words, a CUDA program executes serial code on the host interleaved with parallel threads execution on the GPU. To manage large number of threads that can work cooperatively, CUDA groups them in *thread blocks*. Each block contains up to 512 threads that can share data in fast on-chip memory and synchronize with barriers. Thread blocks are arranged in a *grid* that is launched on a single CUDA program or *kernel*. The blocks of a grid execute independently from each other. Hence, sequentially dependent algorithms must be split up in two or more kernels.

CUDA memory hierarchy is built on 6 memory spaces. These include: *Register file* which is a set of physical registers (16Kb per SM) split evenly between all active threads of a block.[1] *Local memory* is a private space used for per-thread temporary data and register spils. Each SM has 16Kb of low-latency on-chip *shared memory* can be accessed by all threads of a block and is organized in 16 banks to speed-up concurrent access. Bank conflicts are resolved by warp serialization and broadcast mechanism. Read-only *texture* and *constant* memory spaces as well as read-write *global* memory have lifetime of an application and are visible to all thread blocks of a grid. Texture and constant memory spaces are cached on the device. Global memory has no on-chip cache and is of much higher latency than shared memory. To use bandwidth efficiently, the graphics hardware tries to combine

[1] The advantage of static register allocation is that context switching comes almost for free, however this incurs register pressure – a formidable problem in GPU programming.

separate thread memory accesses to a single wide memory access which is also known as *memory coalescing*. The second generation NVIDIA Tesla cards (GT200 series) are much less restrictive in what concerns the memory access patterns for which coalescing can be achieved.

4. THE ALGORITHM

We start with a high-level description of the algorithm. Then, we consider subalgorithms for univariate resultants and polynomial interpolation in detail. Next, we briefly discuss the realization of fast modular arithmetic on the GPU. Finally, in the last subsection we outline the main implementation details of the algorithm.

4.1 Algorithm overview

As mentioned in the beginning, our approach follows the ideas of Collins' algorithm. To compute the resultant of two polynomials in $\mathbb{Z}[x,y]$, we map them to a prime field using several modular homomorphisms. The number of prime moduli in use depends on the resultant coefficients' bit-length given by Hadamard's bound [17]. For each modular image (for each prime m_i) we compute resultants at $x = \alpha_0, x = \alpha_1, \dots \in \mathbb{Z}_{m_i}$. Each univariate resultant is computed using the displacement structure approach, see Section 4.2. The degree bound (or the number of evaluation points) can be obtained using the rows and columns of Sylvester matrix. As shown in [17], one can use both lower and upper bounds for the resultant degree which is advantageous for sparse polynomials. In the next step, resultants over $\mathbb{Z}_{m_i}[x]$ are recovered through polynomial interpolation, see Section 4.3. Afterwards, the modular images of polynomials are lifted using the Chinese remaindering giving the final solution.

An important issue is how to deal with "bad" primes and evaluation points. With the terminology from [17], *a prime m is said to be bad if $f_p \equiv 0 \bmod m$ or $g_q \equiv 0 \bmod m$.* Similarly, *an evaluation point $\alpha \in \mathbb{Z}_m$ is bad if $f_p(\alpha) \equiv 0 \bmod m$ or $g_q(\alpha) \equiv 0 \bmod m$.* Dealing with "bad" primes is easy: we can eliminate them right away during the initial modular reduction of polynomial coefficients performed on the CPU. To handle "bad" evaluation points, we run the GPU algorithm with an *excess* amount of points (typically 1–2% more than required). The same idea is applied to deal with *non-strongly regular* Sylvester matrices.[1] In the essence, non-strong regularity indicates that there is a non-trivial relation between polynomial coefficients which, as our tests confirm, is a rare case on the average(see [9, Section 5] for experiments).

That is why, if for some $\alpha_k \in \mathbb{Z}_m$ the denominators vanish, instead of using some sophisticated methods, we simply *ignore* the result and take another evaluation point. In a very "unlucky" case when we cannot reconstruct the resultant because of the lack of points, we launch another grid to compute extra information. This can be exemplified as follows. Consider two polynomials:

$$\begin{aligned} f &= y^8 + y^6 - 3y^4 - 3y^3 + (x+6)y^2 + 2y - 5x \\ g &= (2x^3 - 13)y^6 + 5y^4 - 4y^2 - 9y + 10x + 1. \end{aligned}$$

Now, if we evaluate them at points $x = 0 \dots 100000$, it is easy to check that the corresponding Sylvester matrix is *non-strongly regular* only for a single point $x = 2$. Hence, we have enough information to recover the result.

4.2 Computing univariate resultants

Let $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ be the generator matrices associated with two polynomials f and g as defined in Section 2.2. In each iteration we update these matrices and collect one factor of the resultant at a time. After n iterations ($n = p + q$) the generators vanish completely, and the product of collected factors yields the resultant.[2] The optimized algorithm is given below:

```
1: procedure resultant_univariate($f$ : Polynomial, $g$ : Polynomial)
2:     $p = \mathrm{degree}(f),\ q = \mathrm{degree}(g),\ n = p + q$
3:     $f \leftarrow f/f_p$                              ▷ convert f to monic form
4:     let $G = (\mathbf{a}, \mathbf{b}),\ B = (\mathbf{c}, \mathbf{d})$        ▷ set up the generators
5:     for $j = 0$ to $q - 1$ do                      ▷ simplified iterations
6:         $\mathbf{b}_i \leftarrow \mathbf{b}_i - \mathbf{a}_i\mathbf{b}_j$   for $\forall i = j+1 \dots p+j$   ▷ rotations
7:         $\mathbf{c}_{2q-j} = \mathbf{b}_j$                   ▷ update a single entry of $\mathbf{c}$
8:         $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i$   for $\forall i = j \dots n-2$             ▷ shift-down
9:     end for
10:    $l_a = 1,\ l_c = 1$                             ▷ denominators and
11:    $res = 1,\ l_{res} = 1$                         ▷ the resultant are set to 1
12:    for $j = q$ to $n - 1$ do        ▷ the remaining p iterations
13:        for $\forall i = j \dots n-1$    ▷ multiply by rotation matrices
14:            $\mathbf{a}_i \leftarrow l_a(\mathbf{a}_i c_j + \mathbf{b}_i d_j),\ \mathbf{b}_i \leftarrow l_c(\mathbf{a}_i b_j - \mathbf{b}_i a_j)$,
15:            $\mathbf{c}_i \leftarrow l_c(\mathbf{c}_i a_j + \mathbf{d}_i b_j),\ \mathbf{d}_i \leftarrow l_a(\mathbf{c}_i d_j - \mathbf{d}_i c_j)$
16:
17:        $l_c = l_a l_c^2,\ l_a = \mathbf{a}_j$               ▷ update denominators and
18:        $res = res \cdot \mathbf{c}_j,\ l_{res} = l_{res} \cdot l_c$               ▷ the resultant
19:                           ▷ shift-down the first generator columns
20:        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i, \mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$   for $\forall i = j \dots n-2$
21:    end for
22:    return $res \cdot (f_p)^q / l_{res}$                 ▷ return the resultant
23: end procedure
```

In what follows, we will denote the iterations $j = 0 \dots q-1$ and $j = q \dots n-1$ as type S and T iterations respectively. Note that, the division in lines 3 and 22 is realized by the Montgomery inverse algorithm [7] with improvements from [19]. Though the algorithm is serial, the number of iterations is bounded by the moduli bit-length (24 bits), for details see [9, Appendix A].

Our algorithm is based on the observation that $B \equiv 0$ at the beginning of the algorithm, except for $c_0 = d_q = 1$. Moreover, if we ensure that the polynomial f is monic, i.e., $f_p \equiv 1$, we can observe that the vectors $\mathbf{a}$, $\mathbf{c}$ and $\mathbf{d}$ remain *unchanged* during the first q iterations of the algorithm (with the exception of a single entry c_q). Indeed, if f is monic we have that: $D = a_0c_0 + b_0d_0 = a_0 \equiv 1$ (see Section 2.3), hence we can get rid of the denominators completely which greatly simplifies the vector update. By the same token, the computed resultant factors are *unit* during the first q iterations. Thus, we do not need to collect them. At the end, we have to multiply the resultant by $(f_p)^q$ as to compensate for running the algorithm on monic f.

4.3 Polynomial interpolation

Suppose $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ are the generators defined in Section 2.4. Again, we take into account that B has only two non-zero entries: $c_0 = 1$ and $d_n = -1$.[3] As a result, we can skip updating B throughout all n iterations of the algorithm. Furthermore, the vector $\mathbf{a}$ does not need to be multiplied by the rotation matrix because:

[1] In fact, both cases correspond to zero denominator, and therefore, are indistinguishable from the algorithm's perspective.

[2] Recall that, in each iteration the size of generators decreases by 1.

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

Listing 1 24-bit modular arithmetic on the GPU

```
 1: procedure mul_mod(a, b, m, invm)                                   ▷ computes $a \cdot b \bmod m$
 2:     hf = uint2float_rz(umul24hi(a, b))             ▷ compute 32 MSB of the product, convert to floating-point
 3:     prodf = fmul_rn(hf, invm)                            ▷ multiply by invm=$2^{16}/m$ in floating-point
 4:     l = float2uint_rz(prodf)                                 ▷ integer truncation: $l = \lfloor hi \cdot 2^{16}/m \rfloor$
 5:     r = umul24(a, b) − umul24(l, m)                       ▷ now $r \in [-2m + \varepsilon; m + \varepsilon]$ with $0 \le \varepsilon < m$
 6:     if r < 0 then r = r + umul24(m, 0x1000002) fi     ▷ single multiply-add instruction: $r = r + m \cdot 2$
 7:     return umin(r, r − m)                                              ▷ return $r = a \cdot b \bmod m$
 8: end procedure
 9: procedure sub_mul_mod(x1, y1, x2, y2, m, invm1, invm2)           ▷ computes $(x_1y_1 - x_2y_2) \bmod m$
10:     hf1 = uint2float_rz(umul24hi(x1, y1))
11:     hf2 = uint2float_rz(umul24hi(x2, y2))                               ▷ two inlined MUL_MOD operations
12:     pf1 = fmul_rn(hf1, invm1), pf2 = fmul_rn(hf2, invm1)    ▷ multiply by $invm1 = 2^{16}/m$ in floating-point
13:     l1 = float2uint_rz(pf1), l2 = float2uint_rz(pf2)                ▷ truncate the results to nearest integer
14:     r = mc + umul24(x1, y1) − umul24(l1, m) − umul24(x2, y2) + umul24(l2, m)   ▷ intermediate product r, $mc = m \cdot 100$
15:     rf = uint2float_rn(r) ∗ invm2 + e23     ▷ multiply by $invm2 = 1/m$ and truncate, $e23 = 2^{23}$, $rf = \lfloor r/m \rfloor$
16:     r = r − umul24(float_as_int(rf), m)
17:     return r < 0 ? r + m : r
18: end procedure
```

$\overline{a}_i = l_a(a_ic_0 + b_id_0) \equiv l_aa_i$ (see Section 2.3). Also, observe that only n entries of the generator G are *non-zero* at a time. Thus, we can use some sort of a "sliding window" approach, i.e., only n relevant entries of G are updated in each iteration. The pseudocode for polynomial interpolation is given below:

```
 1: procedure interpolate(x : Vector, y : Vector, n : Integer)
 2:         ▷ returns a polynomial f, s.t., $f(x_i) = y_i$, $0 \le i < n$
 3:     let G = (a, b)                      ▷ set up the generator matrix
 4:     $l_{int} = 1$                             ▷ set the denominator to 1
 5:     for j = 0 to n − 1 do
 6:                                   ▷ multiply by the rotation matrix
 7:         $b_i \leftarrow b_ia_j - a_ib_j$   for $\forall i = j + 1 \dots j + n - 1$
 8:         $l_{int} = l_{int} \cdot a_j$                  ▷ update the denominator
 9:               ▷ update the last non-zero entries of a and b
10:         $b_{j+n} = -b_j$, $a_{n+j+1} = 1$, $s = 0$, $t = 0$
11:         if (i > j and i < n) then $s = a_i$, $t = x_i$
12:         elif (i > n and $i \le j + n$) then $s = a_{i-1}$, $t = 1$ fi
13:                                    ▷ multiply a by the matrix $\Phi_j$
14:         $a_i \leftarrow s \cdot t - a_i \cdot x_j$   for $\forall i = j + 1 \dots j + n$
15:     end for           ▷ divide the result by the denominator
16:     $b_i \leftarrow -b_i/l_{int}$   for $\forall i = n \dots 2n - 1$
17:     return $b_n \dots b_{2n-1}$      ▷ return the coefficients of f(x)
18: end procedure
```

Note that, we write the update of $\mathbf{a}$ in line 13 in a "linearized" form (using s and t) to avoid thread divergence because in the GPU realization one thread is responsible for updating a single entry of each of vectors $\mathbf{a}$ and $\mathbf{b}$.[1]

4.4 Modular arithmetic

Modular multiplication still constitutes a big problem on modern GPUs due to the limited support for integer arithmetic and particularly slow modulo ('%') operation. The GPU natively supports 24-bit integer multiplication realized by two instructions: mul24.lo and mul24.hi.[2] CUDA only provides an intrinsic for mul24.lo while the latter instruction is not available in a high-level API. To overcome this limitation, the authors of [13] suggest to use 12-bit residues because the reduction can proceed in floating-point without overflow concerns. In the other paper [3], 280-bit residues are partitioned in 10-bit limbs to facilitate multiplication. Hence, neither paper exploits the GPU multiplication capabilities at full. We access the "missing intrinsic" directly using the PTX inline assembly [2] and realize the modular reduction in floating-point, see also [8].

The procedure MUL_MOD in Listing 1 computes $a{\cdot}b \bmod m$. Here umul24 and umul24hi denote the intrinsics for mul24.lo and mul24.hi respectively. First, we partition the product as follows: $a \cdot b = 2^{16}hi + lo$ (32 and 16 bits parts), and use the congruence:

$$2^{16}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = 2^{16}hi + lo - \\ -m \cdot l = a \cdot b - l \cdot m = r, \text{ where } 0 \le \lambda < m.$$

It can be shown that $r \in [-2m + \varepsilon; m + \varepsilon]$ for $0 \le \varepsilon < m$. As a result, r fits into a 32-bit word. Thus, we can compute it as the difference of 32 *least significant bits* of the products $a \cdot b$ and $m \cdot l$ (see line 5). Finally, the reduction in lines 6–7 maps r to the valid range $[0; m - 1]$.

The next procedure SUB_MUL_MOD evaluates the expression: $(x_1y_1 - x_2y_2) \bmod m$ used in rotation formulas (see Section 2.3). It runs two MUL_MOD's in parallel with the difference that the final reduction is deferred until the subtraction in line 13 takes place. The advantage is that line 13 produces 4 multiply-add (MAD) instructions.[3] Lines 14–16 are taken from REDUCE_MOD procedure in [8] with a minor change: namely, in line 14 we use a mantissa trick [14] to multiply by $1/m$ and round the result down in a single multiply-add instruction. We have studied the efficiency of our realization using the `decuda` tool[4]. According to disassembly, the SUB_MUL_MOD operation maps to 16 GPU instructions where 6 of them are *multiply-adds*.

4.5 GPU realization

In this section we go through the main aspects of our implementation. Suppose we are given two polynomials $f, g \in \mathbb{Z}[x, y]$ with y-degrees p and q respectively, and $p \ge q$. The algorithm comprising four kernel launches is depicted in Figure 1. Grid configuration for each kernel is shown to the left.

Before going through the realization details of each GPU kernel separately we would like to outline some features

[1] Short conditional statements are likely to be replaced by predicated instructions which do not introduce branching in the GPU code.

[2] They return 32 least and most significant bits of the product of 24-bit integer operands respectively.

[3] The compiler favors MAD instructions by aggressively merging subsequent multiply and adds.

[4] `http://wiki.github.com/laanwj/decuda`

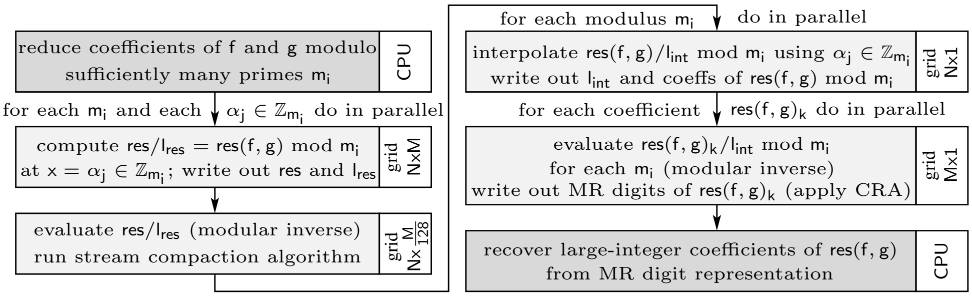
reduce coefficients of f and g modulo sufficiently many primes m_i
CPU
for each m_i and each $\alpha_j \in \mathbb{Z}_{m_i}$ do in parallel
compute $res/l_{res} = res(f,g) \bmod m_i$ at $x = \alpha_j \in \mathbb{Z}_{m_i}$; write out res and l_{res}
grid N×M
evaluate res/l_{res} (modular inverse) run stream compaction algorithm
grid N× M/128
for each modulus m_i do in parallel
interpolate $res(f,g)/l_{int} \bmod m_i$ using $\alpha_j \in \mathbb{Z}_{m_i}$ write out l_{int} and coeffs of $res(f,g) \bmod m_i$
grid Nx1
for each coefficient $res(f,g)_k$ do in parallel
evaluate $res(f,g)_k/l_{int} \bmod m_i$ for each m_i (modular inverse) write out MR digits of $res(f,g)_k$ (apply CRA)
grid Mx1
recover large-integer coefficients of res(f, g) from MR digit representation
CPU

Figure 1: Schematic view of the resultant algorithm with N is the number of moduli, and M is the number of evaluation points

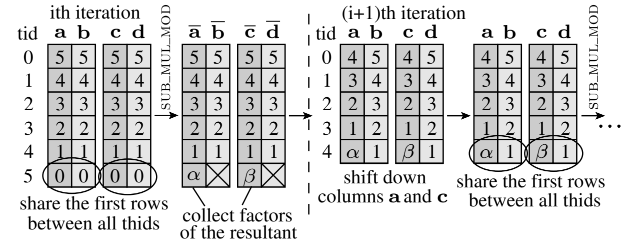
ith iteration
tid
a b c d
SUB_MUL_MOD
$\bar{a}$ $\bar{b}$ $\bar{c}$ $\bar{d}$
(i+1)th iteration
share the first rows between all thids
collect factors of the resultant
shift down columns a and c
share the first rows between all thids
α
β

Figure 2: Vector updates during the type T iterations of the resultant algorithm, where tid denotes the thread ID

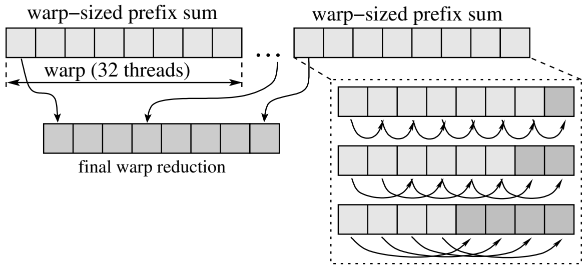

Figure 3: Warp-sized parallel reduction (prefix sum)

shared by all kernels. In our implementation we have used a number of standard optimization techniques including constant propagation via templates, loop unrolling, exploiting warp-level parallelism (parallel prefix sum), favoring small thread blocks over the large ones, etc.

An important aspect of GPU optimization is dealing with *register pressure*. We decrease register usage partly by kernel templetizations and by declaring frequently used local variables with `volatile` keyword. The latter technique forces the compiler to really keep those variables in registers and reuse them instead of inlining the expressions.

We keep the moduli set m_i and corresponding reciprocals `invm` used for reduction in *constant memory space*. This is because one thread block (except for CRA kernel) works with a *single* modulus. Hence, all threads of a block read the same value which is what the constant memory cache optimized for. Moreover, direct access from constant memory has a positive effect on reducing register usage.

4.5.1 Resultant kernel

The resultant kernel evaluates polynomials at $x = \alpha_j \in \mathbb{Z}_{m_i}$ and computes univariate resultants. It is launched on a 2D grid $N \times M$, see Figure 1. The evaluation points are implicitly given by the block indices. Four kernel instantiations with 32×2, 64, 96 and 128 threads per block are specialized for different polynomial degrees. A kernel specialization with P threads per block covers the range of degrees: $p \in [P/2;\ P-1]$.[1]

This configuration is motivated by the fact that we use $p+1$ threads to substitute x in each of $p+1$ coefficients of f (and the same for g) in parallel. The resultant algorithm consists of one outer loop split up in iterations of types S and T, see Section 4.2. In each iteration the generators are transformed using the SUB_MUL_MOD procedure, see Figure 2. The inner loop is completely vectorized: this is another reason for our block configuration. The type S iterations are unrolled by the factor of 2 for better thread occupancy, so that each thread runs two SUB_MUL_MOD operations in a row. In this way, we double the maximal degree of polynomials that can be handled, and ensure that all threads are occupied. Moreover, at the beginning of type T iterations we can guarantee that not less than half of threads are in use in the corner case ($p = P/2$).

The column vectors a and c of generators $G = (a, b)$ and $B = (c, d)$ are stored in shared memory because they need to be shifted down in each iteration. The vectors b and d reside in a register space. Observe that, the number of working threads decreases with the length of generators in the outer loop. We run the type T iterations until at least half of all threads enter an idle state. When this happens, we rebalance the workload by switching to a "lighter" version where all threads do half of a job. Finally, once the generator size descends below the warp boundary, we switch to iterations *without* sync.[2] The factors of the resultant and the denominator are collected in shared memory, then the final product $(f_p)^q \cdot \prod_i d_{ii}$ is computed efficiently using "warp-sized" parallel reduction based on [12]. The idea of reduction is to run prefix sums for several warps separately omitting synchronization barriers, and then combine the results in a final reduction step, see Figure 3. Listing 2 computes a prefix sum of 256 values stored in registers. It slightly differs from that of used in [12]. Namely, our algorithm requires less amount of shared memory per warp ($\mathsf{WS} + \mathsf{HF} + 1 = 49$ words in-

[1] The first kernel instantiation – 32×2 – calculates two resultants at a time.

[2] Recall that, the warp, as a minimal scheduling entity, is always executed synchronously on the GPU.

Listing 2 "warp-sized" parallel reduction with 256 threads

```
procedure warp_scan(x)    ▷ parallel prefix of 256 values x
            ▷ abbrev.: OP: prefix operation, TID: thread-id
                    ▷ WS: warp-size (32), HF: half-warp (16),
                ▷ initialize shared memory space for reduction:
    volatile type ∗scan = data + HF+
        +(TID%32) + (TID/32) ∗ (WS + HF + 1)
    scan[−HF] = Ident     ▷ ident. symbol: OP(Ident,x)=x
    scan[0] = x                ▷ save elements to shared memory
                            ▷ run warp-scans independently:
    t = OP(t, scan[−1]), scan[0] = t
    t = OP(t, scan[−2]), scan[0] = t
    t = OP(t, scan[−4]), scan[0] = t
    t = OP(t, scan[−8]), scan[0] = t
    t = OP(t, scan[−HF]), scan[0] = t
                ▷ "post-scan" leading elements of each warp
    volatile type ∗postscan = data + HF+
        +(WS ∗ 8/32) ∗ (WS + HF + 1)
    syncthreads()                         ▷ thread synchronize
    if TID < 8 then          ▷ post-scan 8 leading elements
        volatile type ∗scan2 = postscan + TID
        scan2[−HF] = Ident                ▷ put identity symbol
                ▷ load high-order elements from shared mem
        t = data[HF + WS − 1 + TID ∗ (WS + HF + 1)]
        scan2[0] = t, t = OP(t, scan2[−1])
        scan2[0] = t, t = OP(t, scan2[−2])
        scan2[0] = t, t = OP(t, scan2[−4])
        scan2[0] = t
    fi
    syncthreads()                         ▷ thread synchronize
    t = scan[0]    ▷ read out and update scanned elements
    t = OP(t, postscan[TID/32 − 1]▷ postscan[-1] = Ident
    return t
end procedure
```

stead of 64 words) and uses fewer number of *sync* operations (2 instead of 3).

4.5.2 Modular inverse and stream compaction kernel

This kernel is also launched on a 2D grid with 128 threads per block, see Figure 1. For each modulus m_i it eliminates the resultants corresponding to "bad" evaluation points,[1] and computes the quotient res/l_{res} for each non-zero denominator l_{res} using the modular inverse. The input sequence (the set of M evaluation points for each modulus) is partitioned into 128-element chunks such that one thread computes one Montgomery modular inverse. In this way, we achieve the full occupancy due to sequential nature of the Montgomery algorithm.

To realize the efficient stream compaction we have made the following observations. **1.** The number of evaluation points M per modulus can be quite large (typically on the order of one thousand), hence it is inefficient/impossible to process all of them in one thread block. **2.** Running hierarchical stream compaction in global memory (several kernel launches) seems to be unreasonable because "bad" evaluation points occur quite rarely on the average.[2] **3.** The actual order of evaluation points does not matter for interpolation. As a result, we found the following solution optimal: Each block runs the stream compaction on its 128-element chunk using warp-sized reduction in shared memory. The reduction computes an exclusive prefix sum of a sequence of 0's and 1's where 0's correspond to elements being eliminated. Finally, the compacted sequence is written out 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. Note that, such a memory access pattern does not cause a severe performance degradation on GT200 hardware. This is because on the devices of compute capability 1.2 (SM12) and higher memory coalescence is also achieved for misaligned access, see [1].

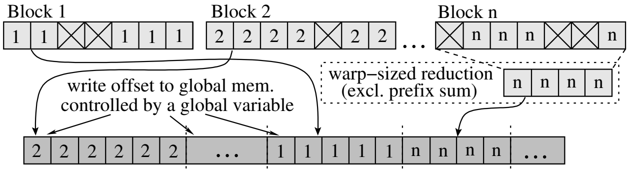

Figure 4: Stream compaction in shared memory

4.5.3 Interpolation kernel

The realization is based on the algorithm from Section 4.3. One thread block is responsible for interpolating one polynomial for some prime modulus m_j. Similarly to the resultant kernel, the inner loop is unrolled and vectorized. Again, vector updates are performed using SUB_MUL_MOD operation.

We have chosen to unroll the inner loop to have high arithmetic intensity and to decrease the number of shared memory accesses per iteration because running one SUB_MUL_MOD operation per thread results in far too low thread's workload. On the other hand, unrolling increases register pressure. Therefore, in order to keep the computations fast we unroll the loop by the factor of 2 for low resultant degrees ($n \leq 256$) and by the factor of 4 for the remaining ones ($n > 256$). Our tests have shown that the kernel with 128 threads and unrolling factor of 2 runs faster than that of 64 threads and the factor of 4. Accordingly, one thread is responsible for updating 2 or 4 elements of generator G. The number of threads per block is adjusted to estimated degree n of the resultant. Hence, in the current implementation the maximal degree is limited to 2048 which corresponds to 512 threads.

The major difference is that the generator's size stays constant throughout the algorithm. That is why, in each iteration we process only n relevant entries of G in a "sliding window" fashion. We have found it advantageous to parameterize the kernel by the "data parity", that is, by n mod 2 or n mod 4 depending on the loop unrolling factor, in place of the data size n itself. This allowed us to substantially reduce branching inside the outer loop which is a big performance issue for GPU algorithms. Again, the collected factors of the denominator $\mathsf{l_{int}}$ are multiplied using the prefix sum.

4.5.4 CRA kernel

The remaining CRA kernel processes each resultant coefficient $res(f,g)_k$ independently, see Figure 1. It first divides the residues $res(f,g)_k$ mod m_j by respective denominators $\mathsf{l_{int}}$ computed during the interpolation (using Montgomery inverse), and runs the CRA to recover the Mixed-radix representation of the coefficient. The algorithm is rather straightforward realization of formulas from Section 2.5. It consists

[1] In other words, those points for which denominators vanish, see Section 4.1.

[2] Our experiments show that for random polynomials typically 3–4 evaluation points are "bad" out of 10–50 thousand.

Table 1: Timing the resultants of f and $g \in \mathbb{Z}[x,y]$. *1st column*: **instance number;** *2nd column*: deg$_y$(f/g) : **polynomials' y-degree;** deg$_x$(f/g) : **polynomials' x-degree;** bits : **coefficient bit-length;** sparse/dense: **varying density of polynomials;** *3rd column*: **resultant degree**

#	Configuration	degree	GPU	Maple	speed-up
1-2.	deg$_y$(f) : **20**, deg$_y$(g) : **16** deg$_x$(f) : **7**, deg$_x$(g) : **11**, bits : **32** / **300**	332 / 332	0.015 s / 0.14 s	1.8 s / 16.3 s	120x / 116x
3-4.	deg$_y$(f) : **29**, deg$_y$(g) : **20** deg$_x$(f) : **32**, deg$_x$(g) : **25**, bits : **64** / **250**	1361 / 1361	0.3 s / 0.89 s	36.6 s / 143.9 s	122x / 161x
5-6.	deg$_y$(f) : **62**, deg$_y$(g) : **40**, bits : **24** deg$_x$(f) : **12**, deg$_x$(g) : **10** (sparse/dense)	1088 / 1100	0.28 s / 0.33 s	25.8 s / 34.8 s	92x / 105x
7-8.	deg$_y$(f) : **90**, deg$_y$(g) : **80**, bits : **20** deg$_x$(f) : **10**, deg$_x$(g) : **10** (sparse/dense)	1502 / 1699	1.06 s / 1.4 s	76.7 s / 148.8 s	72x / 106x
9-10.	deg$_y$(f) : **75**, deg$_y$(g) : **60** deg$_x$(f) : **15**, deg$_x$(g) : **7**, bits : **32** / **100**	1425 / 1425	1.2 s / 3.1 s	100.6 s / 298.4 s	84x / 96x
11-12.	deg$_y$(f) : **126**, deg$_y$(g) : **80**, bits : **16** deg$_x$(f) : **4**, deg$_x$(g) : **7** (sparse/dense)	1150 / 1202	1.29s / 1.53 s	87.9 s / 121.3 s	68x / 79x

of one outer loop while the inner loop is vectorized. In each iteration one mixed-radix digit α_i is computed and the remaining ones $\alpha_{i+1} \dots \alpha_M$ get updated. The intermediate variables V_i are computed on-the-fly while c_i's are preloaded from the host because modular inverse is expensive.

It is worth noting that we simplify the computations by processing moduli in *increasing order*, i.e., $m_1 < m_2 < \dots < m_M$. Indeed, suppose x_i and x_j are residues modulo m_i and m_j respectively ($j < i$). Then, the expressions of the form $(x_i - x_j) \cdot c_i \bmod m_i$ can be evaluated *without* initial modular reduction of x_j because $x_j < m_i$. The modular multiplication is performed using MUL_MOD procedure from Listing 1.

The block size is adjusted to the number of moduli M. Again, for performance reasons we have unrolled the outer loop by the factor of 2 or 4 and parameterized the kernel by "data parity" (M mod 2 or M mod 4).

As a very last step, we reconstruct the multi-precision coefficients from their MR representation by evaluating the Horner form on the host machine.

5. PERFORMANCE EVALUATION AND CONCLUSIONS

We have run experiments on the *GeForce GTX 280* graphics card. The resultant algorithm from Maple 13 (32-bit version) has been tested on a *2.8Ghz Dual-Core AMD Opteron 2220SE* with 1MB L2 cache comprised in a four-processor cluster with total of 16Gb RAM under Linux platform. We have configured Maple to use *deterministic* algorithm by setting `EnvProbabilistic` to 0. This is because our approach uses Hadamard's bounds both for resultant's height and degree while the default Maple algorithm is probabilistic, in other words it uses as many moduli as necessary to produce a "stable" solution, see [17].

Performance comparison is summarized in Table 1. The GPU timing covers all stages of the algorithm including initial modular reduction and recovering multi-precision results on the host. For large integer arithmetic we have used GMP-4.3.1 library.[1] We have varied different parameters such as polynomial's x- and y-degree, the number of moduli, the coefficient bit-length and the density of polynomials (the number of non-zero entries). One can see that our algorithm achieves better speed-up for dense polynomials. This implicitly indicates that Maple uses the PRS algorithm in its core: the PRS generally performs more iterations (divisions) for dense polynomials. Whereas our algorithm is indifferent to polynomial density as it is based on linear algebra.

Also, observe that, our algorithm is faster polynomials of high x-degree. This is expected because, with the x-degree, the number of thread blocks increase (thereby, leading to better hardware utilization) while the size of Sylvester matrix remains the same. On the contrary, increasing the y-degree penalizes the performance as it causes the number of threads per block to increase. Similarly, for larger bit-lengths, the attained performance is typically higher (again because of increased degree of parallelism), with the exception that for low-degree polynomials the time for CPU modular routines becomes noticeably large as compared to the resultant computation itself.

The histogram in Figure 5 shows how the different stages of the algorithm contribute to the overall timing. Apparently, the time for resultant kernel is dominating: this is no surprise because its grid size is much larger than that of other kernels, see Figure 1). The second largest time is either initial modular reduction ('mod. reduce' in the figure) for polynomials with large coefficients, or interpolation for high-degree polynomials. Also, observe that, the time for GPU–host data transfer ('data transfer' in the figure) is negligibly small. This indicates that our algorithm is *not* memory-bound, and therefore has a big performance potential on future generation GPUs. The remaining two graphs in Figure 5 examine the running time as a function of coefficients' bit-length and polynomial degree. The bit-length only causes the number of moduli to increase resulting in a linear dependency. While polynomial's y-degree affects both moduli and evaluation points, therefore the performance degrades quadratically.

We have presented the algorithm to compute polynomial resultants on the GPU. The displacement structure approach has been proved to be well-suited for realization on massively-threaded architectures. Our results indicate a significant performance improvement over a host-based implementation. We have achieved the high arithmetic intensity of the algorithm by dynamically balancing the thread's workload and taking into account hardware-specific features such as multiply-add instruction support. Moreover, our algorithm has a great scalability potential due to the vast amount of thread blocks used by the resultant kernel.

[1] `http://gmplib.org`

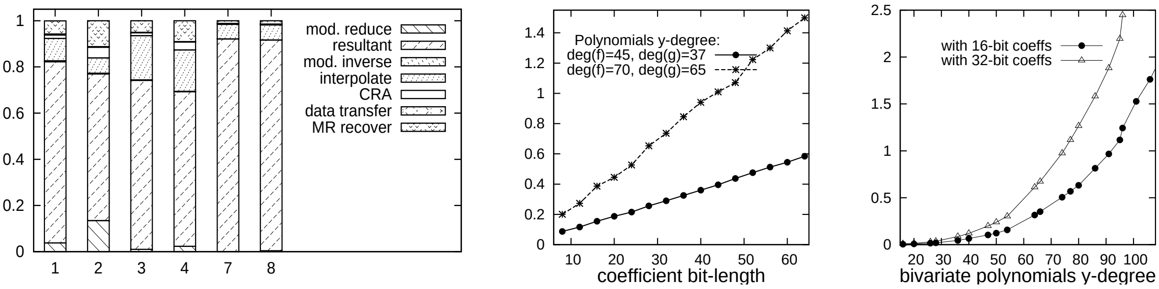

Figure 5: *Left:* **relative contribution of different stages to the overall time, x-axis: instance number in Table 1;** *Middle:* **the running time as a function of coefficient bit-length;** *Right:* **the running time as a function of polynomial y-degree. All timings are in seconds.**

As a future research directions, we would like to revisit implementation of our approach on the GPU in order to benefit from a block-level parallelism since the algorithm admits only a single-block parallelization, in that way limiting the size of input that can be handled. One possibility could be to adapt one of the recursive Schur-type algorithms for matrix factorization based on the block inversion formula, see for instance [18]. We would also like to extend our approach to other computationally-instensive symbolic algorithms that can be reformulated in matrix form: good candidates could be multivariate polynomial GCDs and subresultant sequences.

6. REFERENCES

[1] CUDA Compute Unified Device Architecture. Programming Guide. Version 2.3. NVIDIA Corp., 2009.

[2] PTX: Parallel Thread Execution. ISA Version 1.4. NVIDIA Corp., 2009.

[3] 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.

[4] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel. Distributed Symbolic Computation with DTS. In *IRREGULAR '95*, pages 231–248, London, UK, 1995. Springer-Verlag.

[5] 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.

[6] G. E. Collins. The calculation of multivariate polynomial resultants. In *SYMSAC '71*, pages 212–222. ACM, 1971.

[7] G. 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.

[8] P. Emeliyanenko. Efficient Multiplication of Polynomials on Graphics Hardware. In *APPT '09*, pages 134–149, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] P. Emeliyanenko. Modular Resultant Algorithm for Graphics Processors. In *ICA3PP '10*, pages 427–440, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] E. Frantzeskakis and K. 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.

[11] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for computer algebra*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.

[12] M. Harris, S. Sengupta, and J. D. Owens. CUDPP: CUDA Data Parallel Primitives Library. Version 1.1. http://gpgpu.org/developer/cudpp.

[13] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *AFRICACRYPT '09*, pages 350–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] C. Hecker. Let's get to the (floating) point. *Game Developer Magazine*, pages 19–24, 1996.

[15] H. Hong and H. W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In *CONPAR 94*, pages 325–336. Springer Verlag, 1994.

[16] T. Kailath and S. Ali. Displacement structure: theory and applications. *SIAM Review*, 37:297–386, 1995.

[17] M. Monagan. Probabilistic algorithms for computing resultants. In *ISSAC '05*, pages 245–252. ACM, 2005.

[18] J. H. Reif. Efficient parallel factorization and solution of structured and unstructured linear systems. *J. Comput. Syst. Sci.*, 71(1):86–143, 2005.

[19] E. Savas and C. Koc. The Montgomery modular inverse-revisited. *Computers, IEEE Transactions on*, 49(7):763–766, 2000.

[20] A. Schönhage and E. Vetter. A New Approach to Resultant Computations and Other Algorithms with Exact Division. In *ESA'94*, pages 448–459, London, UK, 1994. Springer-Verlag.

[21] W. Schreiner. Developing A Distributed System For Algebraic Geometry. In *EURO-CM-PAR'99*, pages 137–146. Civil-Comp Press, 1999.

[22] M. Yassine. Matrix Mixed-Radix Conversion For RNS Arithmetic Architectures. In *Proceedings of 34th Midwest Symposium on Circuits and Systems*, 1991.