Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra

Pavel Emeliyanenko

Max-Planck Institute for Informatics, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 14 May 2011
Received in revised form
18 May 2012
Accepted 30 July 2012
Available online xxxx

Keywords:
Symbolic algorithms
Resultants
Modular computations
GPU
CUDA
Parallel computing

ABSTRACT

In this article we report on our experience in computing resultants of bivariate polynomials on Graphics Processing Units (GPU). Following the outline of Collins' modular approach [6], our algorithm starts by mapping the input polynomials to a finite field for sufficiently many primes m. Next, the GPU algorithm evaluates the polynomials at a number of fixed points $x \in \mathbb{Z}_m$, and computes a set of univariate resultants for each modular image. Afterwards, the resultant is reconstructed using polynomial interpolation and Chinese remaindering. The GPU returns resultant coefficients in the form of Mixed Radix (MR) digits. Finally, large integer coefficients are recovered from the MR representation on the CPU. All computations performed by the algorithm (except for, partly, Chinese remaindering) are outsourced to the graphics processor thereby minimizing the amount of work to be done on the host machine. The main theoretical contribution of this work is the modification of Collins' modular algorithm using the methods of matrix algebra to make an efficient realization on the GPU feasible. According to the benchmarks, our algorithm outperforms a CPU-based resultant algorithm from 64-bit Maple 14 by a factor of 100.
© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Resultants is a fundamental algebraic tool in computer algebra systems, and has found many applications in various areas of science and engineering. For instance, they play an important role in the topological study of algebraic curves and computer graphics. Despite the fact that this problem has been extensively studied both from a theoretical and practical perspective, many scientific applications still suffer from a high computational complexity of resultants. The reason for this is because the degree and bitlength of a resultant polynomial grow wildly with respect to those of original polynomials. In the past years, Graphics Processing Units (GPUs) have evolved to programmable general-purpose processors with outstanding computational horsepower. As of now, they are becoming the new standard platform for high-performance scientific computing. The aim of this work is to accelerate the computation of resultants of bivariate polynomials with integer coefficients on the GPU.

A classical approach to computing resultants is the one proposed by Collins [6]. This algorithm employs modular and evaluation homomorphisms to deal with *expression swell* – the problem shared by all symbolic algorithms – during the computation of resultants. Using a "divide–conquer–combine" strategy, it first reduces the coefficients of input polynomials modulo sufficiently many primes. Then, several evaluation homomorphisms are applied recursively further reducing the problem to the univariate case. Finally, a set of univariate resultants are computed using polynomial remainder sequences (PRS), see [14]. The final result is recovered via polynomial interpolation and Chinese remaindering. This idea was exploited by the number of modular algorithms, including sequential [25,20], and parallel ones specialized for workstation networks [4] or shared memory machines [26,17]. Unfortunately, the modular approach in its original form is not suitable for realization on the GPU due to the lack of fine-grained parallelism since the PRS algorithm, used in its core to compute univariate resultants, is resistant to parallelization. To overcome this difficulty, we have adopted an algorithm based on structured matrices [18] to solve the problem in the univariate case. Here the main idea is that matrix computations inherently bear some *data-level* parallelism which can be readily used on the GPU. In the essence, this method computes the resultant by direct factorization of Sylvester's matrix (see Section 2.1 for definitions). In addition, we have augmented the factorization algorithm itself with division-free computations [13] to make it suitable for realization in a finite field. We should emphasize that, even though the PRS algorithm can also be made division-free (which is probably the method of choice for the modular algorithm), this fact alone does not facilitate its the realization on the GPU due to the reasons mentioned above.

We have successfully evaluated our algorithm on real problems: for example, to dramatically speed-up the solution of a system

E-mail address: asm@mpi-inf.mpg.de.

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.07.015

of bivariate polynomial equations [2] and the computation of topology of an algebraic curve [1]. This paper serves as a summary in which, besides presenting the algorithm itself, we also report on details and improvements gathered in the course of practical application of the algorithm which were not covered in the original works [12,10,11]. At the highest level, our approach is based the same "divide–conquer–combine" principle as the one introduced in the work of Collins. In the implementation we have strived to minimize the amount of work to be done on the host machine: at the current state of development, all algorithm steps, excluding (partly) Chinese remaindering, are outsourced to the GPU. In order to interpolate the resultant polynomial over a prime field, by analogy to the resultant computations, we again rely on matrix algebra based methods. Additionally, we use an efficient stream compaction approach to eliminate "bad" evaluation points right on the GPU, see Section 5.5. To eventually "combine" all modular images, we compute a Mixed-Radix (MR) representation of the resultant coefficients on the GPU without resorting to multi-precision arithmetic. For this, we have developed a block-structured Mixed-Radix conversion algorithm suitable for realization on the GPU. Potentially, this approach can handle any number of modular residues, loaded and processed in chunks. Finally, large integer coefficients are recovered from the MR representation on the CPU by evaluating a Horner scheme.

From the realization point of view, our algorithm takes advantage of NVIDIA's Fermi GPU architecture [21]. Particularly, this relates to the extensive use of double-precision arithmetic in modular computations, and the need for new voting intrinsics to speed-up the stream compaction subalgorithm. Yet, our algorithm is also backward compatible with previous generation Tesla cards [19] by enabling reduced-precision modular arithmetic. For the sake of completeness, we consider both realizations here.

The remaining part of the paper is structured as follows. In Section 2 we formulate the problem in a mathematically concise way and give an introduction to the theory of structured matrices. Section 3 is devoted to the GPU architecture and CUDA framework. Section 4 presents the structure of the algorithm at a high-level and discusses some practical ways to deal with unlucky homomorphisms, such as "bad" primes and "bad" evaluation points. In Section 5 we focus on the main aspects of the GPU realization. Finally, in Section 6 we evaluate the performance of our algorithm and draw conclusions.

2. Theoretical background

In this section, we briefly introduce the theory of structured matrices and the generalized Schur algorithm. Next, using the methods of matrix algebra we derive the algorithms to compute univariate resultants and polynomial interpolation which are the key ingredients of the modular approach.

Throughout this section, we shall also agree that matrix elements are indexed starting from $(0, 0)$ to be consistent with the indexing of polynomial coefficients.[1] In other words, the first diagonal element of an $n \times n$ matrix M will be denoted by $M_{0,0}$ and the last one—by $M_{n-1,n-1}$.

2.1. Resultant of bivariate polynomials

Let $f, g \in \mathbb{Z}[x, y]$ be two bivariate polynomials with integer coefficients of y-degrees p and q, respectively:

$$f(x, y) = \sum_{i=0}^{p} f_i(x)y^i \quad \text{and} \quad g(x, y) = \sum_{i=0}^{q} g_i(x)y^i, \tag{1}$$

where $f_i \in \mathbb{Z}[x]$ and $g_i \in \mathbb{Z}[x]$. Then, *Sylvester's matrix* $S^{(y)} \in \mathbb{Z}[x]^{n\times n}$, $n = p + q$, associated with f and g is defined as follows:

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

Accordingly, the *resultant* of f and g with respect to the variable y, $R^{(x)} := \mathrm{res}_y(f, g)$, is the *determinant* of $S^{(y)}$. It follows that $R^{(x)}$ is a univariate polynomial in x. In a completely analogous manner, we can write the resultant $R^{(y)} := \mathrm{res}_x(f, g)$ of f and g with respect to the variable x if, instead of (1), we write f and g as univariate polynomials in the *outermost* variable x.

The goal of Collins' modular approach is to reduce the computation of $R^{(x)}$ (or $R^{(y)}$) to the operations over the integer domain since the latter ones can be easily carried out on the computer. In the next section, we discuss the computation of univariate resultants in a prime field.

2.2. Resultants and factorization of structured matrices

Suppose we are given Sylvester's matrix $S \in \mathbb{Z}^{n\times n}$ $(n = p + q)$ associated with two univariate polynomials $f, g \in \mathbb{Z}[x]$ of degrees p and q, respectively. For the time being, we assume that S is a *strongly regular* matrix.[2] The matrix S has a special structure (known as quasi-Toeplitz) which is formalized via the following *displacement equation* [18]:

$$S - FSA^T = GB^T \quad \text{where } F = Z_n \text{ and } A = Z_q \oplus Z_p, \tag{2}$$

where $Z_n \in \mathbb{Z}^{n\times n}$ denotes a down-shift matrix zeroed everywhere except for 1's on the first subdiagonal; and $\oplus$ is a Kronecker sum. $G, B \in \mathbb{Z}^{n\times 2}$ are the so-called *generator matrices* such that the product GB^T has a constant rank 2 called a *displacement rank* of S. The generators provide us a compact representation of a matrix using $\mathcal{O}(n)$ parameters. Moreover, G and B can be easily expressed in terms of the coefficients of f and g, hence we do not need to construct S explicitly:

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

Our goal is to obtain a triangular factorization of S. It then follows from the elementary properties of determinants that the resultant will be equal to the product of diagonal elements obtained in matrix factorization. To achieve this, we shall use the *generalized Schur algorithm* [18,5] which yields matrix factorization by iteratively computing the *Schur complements* of leading submatrices.[3] Owing to the low-rank displacement representation of a matrix, the factorization can be computed using $\mathcal{O}(n^2)$ arithmetic operations, see [18, p. 323].

One step of the Schur algorithm can be written in terms of the generator recursions. Let G_i, B_i be the generator matrices of size $(n-i) \times 2$ in step i of the algorithm $(0 \le i < n)$. Formally speaking, (G_i, B_i) is a compact representation of the Schur complement of an $i \times i$ leading block of S. Then, the next generator pair (G_{i+1}, B_{i+1})

[1] In standard notation, for a polynomial f of degree n, f_0 denotes a *trailing* coefficient while f_n is a *leading* coefficient of f.

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

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

Please cite this article in press as: P. Emeliyanenko, Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra, J. Parallel Distrib. Comput. (2012), doi:10.1016/j.jpdc.2012.07.015

obeys the following recursive relation [18, p. 356]:

$$\begin{bmatrix}\mathbf{0}\\ G_{i+1}\end{bmatrix} = \left\{G_i + (F_i - I_{n-i})G_i\frac{b_i^T g_i}{g_i b_i^T}\right\}\Theta_i,\qquad \begin{bmatrix}\mathbf{0}\\ B_{i+1}\end{bmatrix} = \left\{B_i + (A_i - I_{n-i})B_i\frac{g_i^T b_i}{b_i g_i^T}\right\}\Gamma_i, \tag{3}$$

where I_r is an $r \times r$ identity matrix; F_i (and A_i) is a matrix obtained from F (and A) after deleting the first i rows and columns; g_i and b_i are the top rows of G_i and B_i, respectively; and Θ_i, Γ_i are arbitrary 2×2 matrices satisfying: $\Theta_i\Gamma_i^T = I$. Observe that, the size of the generators (the number of rows) reduces by one in each step of the Schur algorithm.

Here, the usual choice of Θ_i and Γ_i is to reduce the generators G_i and B_i to the so-called *proper form* in which the corresponding matrices have a single non-zero entry in their top rows. In other words, the transformations $\overline{G}_i = (G_i\Theta_i)$ and $\overline{B}_i = (B_i\Gamma_i)$ can be written as follows:

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

For proper form generators $(\overline{G}_i, \overline{B}_i)$, the relation (3) can be largely simplified. Namely, to compute G_{i+1} (and B_{i+1}) we multiply the first column of $\overline{G}_i$ (and $\overline{B}_i$) by the corresponding down-shift matrix F_i (and A_i) while keeping the other column intact. Moreover, from (2) we deduce that diagonal elements d_{ii} in the factorization of S are obtained as: $d_{ii} = \delta^i\zeta^i$.

2.3. Division-free matrix transformations

To bring the generators to a proper form in each step of the algorithm, we need to find matrices Θ and Γ satisfying:

$$\begin{bmatrix}a_0 & b_0\end{bmatrix}\Theta = \begin{bmatrix}\delta & 0\end{bmatrix}\quad\text{and}\quad\begin{bmatrix}c_0 & d_0\end{bmatrix}\Gamma = \begin{bmatrix}\zeta & 0\end{bmatrix},$$

in addition with $\Theta\Gamma^T = I$. It is easy to check that it holds for the following matrices:

$$\Theta = \begin{bmatrix}c_0 & b_0\\ d_0 & -a_0\end{bmatrix},\qquad \Gamma = \frac{1}{D}\begin{bmatrix}a_0 & d_0\\ b_0 & -c_0\end{bmatrix}, \tag{4}$$

with $D = a_0c_0 + b_0d_0$. Next, recall that, our algorithm is to be used in a finite field. Therefore, for reasons of efficiency, it is desirable to minimize the number of finite field divisions. Following the idea of [13], we introduce a *common* denominator for each generator column in order to collect all the division factors. Thereby, we can defer the actual division until the end of the algorithm. In other words, if we express the generators as follows:

$$G^T = \begin{bmatrix}(a_0\, a_1\, \cdots)/l_a\\ (b_0\, b_1\, \cdots)/l_b\end{bmatrix},\qquad B^T = \begin{bmatrix}(c_0\, c_1\, \cdots)/l_c\\ (d_0\, d_1\, \cdots)/l_d\end{bmatrix},$$

the proper form generators $(\overline{G}, \overline{B}) = (G\Theta, B\Gamma)$ can be written in a division-free form:

$$\overline{G}_i^T = \begin{bmatrix}l_a(a_ic_0 + b_id_0)\\ l_b(a_ib_0 - b_ia_0)\end{bmatrix}\qquad \overline{B}_i^T = \begin{bmatrix}l_c(c_ia_0 + d_ib_0)\\ l_d(c_id_0 - d_ic_0)\end{bmatrix}. \tag{5}$$

Besides, we need to update the column denominators, which are shown to be *pairwise* equal:

$$\bar{l}_a = \bar{l}_d = \bar{a}_0 \quad\text{and}\quad \bar{l}_c = \bar{l}_b = l_al_c^2. \tag{6}$$

Now, we are ready to devise a univariate resultant algorithm.

2.4. Univariate resultant algorithm

Let G, B be the generator matrices associated with two polynomials f and g as defined in Section 2.2. For convenience, we write them as a pair of column vectors: $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$.

Algorithm 1 Univariate resultant algorithm

```
1:  procedure RESULTANT_UNIVARIATE(f : Polynomial, g : Polynomial)
2:     p = degree(f), q = degree(g), n = p + q
3:     f ← f/f_p                                  ▷ convert f to monic polynomial
4:     let G = (a, b), B = (c, d)                 ▷ set up the generators
5:     for j = 0 to q − 1 do                      ▷ first q "lite" iterations
6:        for i = j + 1 to p + j do               ▷ multiply by rotation matrix
7:           b_i = b_i − a_i b_j
8:        od
9:        c_{2q−j} = b_j                          ▷ update a single entry of c
10:       a_{i+1} ← a_i  for ∀i = j . . . n − 2   ▷ shift down the first column
11:    od                  ▷ set the common denominators and resultant to one:
12:    l_a = 1, l_c = 1, res = 1, l_res = 1
13:    for j = q to n − 1 do                      ▷ the remaining p "full" iterations
14:       for i = j to n − 1 do                   ▷ multiply with the rotation matrix
15:          a_i = l_a(a_i c_j + b_i d_j), b_i = l_c(a_i b_j − b_i a_j),
16:          c_i = l_c(c_i a_j + d_i b_j), d_i = l_a(c_i d_j − d_i c_j)
17:       od                  ▷ update the denominators and the resultant:
18:       l_c = l_a l_c^2, l_a = a_j, res = res · c_j, l_res = l_res · l_c
19:                           ▷ shift down the first columns of G and B
20:       a_{i+1} ← a_i, c_{i+1} ← c_i  for ∀i = j . . . n − 2
21:    od
22:    return res · (f_p)^q / l_res               ▷ compensate for monic form of f
23: end procedure
```

In each iteration of the algorithm we update these matrices according to (5) and collect one factor of the resultant. After n iterations ($n = p + q$) the generators vanish completely, and the product of all factors yields the resultant. The pseudocode is given in Algorithm 1.

The algorithm is split in two parts: lines 5–11 where only a single column $\mathbf{b}$ of G is updated; and lines 13–21 where all four columns participate. In what follows, we will refer to these parts as "lite" and "full" iterations, respectively. The reason for this lies in a particular structure of B having only two non-zero entries: $c_0 = d_q = 1$. Furthermore, if we ensure that the polynomial f is monic (that is, $a_0 = f_p \equiv 1$), it follows that $D = a_0c_0 + b_0d_0 = a_0 \equiv 1$, see (4). As a result, the denominators are identically 1 throughout "lite" iterations. Substituting this into (5) leads to rather simplified generator recursions. By the same token, the resultant factors a_0c_0 are *unit* during "lite" iterations, and hence not need to be collected. In line 22 we multiply the resultant by $(f_p)^q$ to compensate for monic f. To realize finite field division in lines 3 and 22 we use Montgomery modular inverse algorithm, see Section 5.1.2.

Observe that, *strong regularity* assumption, introduced in the beginning of Section 2.2, guarantees that denominators l_a and l_c do not vanish in the course of the algorithm. Yet, this is not always true for Sylvester's matrix. In Section 4.2 we elaborate on how to deal with this problem.

2.5. Polynomial interpolation in a finite field

The task of polynomial interpolation is to find a polynomial $f(x)$ of degree less than n, satisfying a set of equations: $f(x_i) = y_i$, for $0 \le i < n$. The coefficients $\{a_i\}$ of f are the solutions of the following system:

$$V\mathbf{a} = \mathbf{y},\quad \text{with } V_{ij} = x_i^j \text{ for } 0 \le i, j < n - 1, \tag{7}$$

where $V \in \mathbb{Z}^{n\times n}$ is called a Vandermonde matrix. We can apply the Schur algorithm to the matrix $M \in \mathbb{Z}^{2n\times(n+1)}$ which is embedding of V:

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

After n steps we obtain a Schur complement R of submatrix V in M, such that: $R = \mathbf{0} - I_nV^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$ which is precisely the solution of (7). The matrix M has a displacement rank 2 and satisfies the displacement equation:

$$M - FMA^T = GB^T, \tag{8}$$

where $A = Z_n \oplus 0 \in \mathbb{Z}^{(n+1)\times(n+1)}$, $F = \mathrm{diag}(x_0 \cdots x_{n-1}) \oplus Z_n \in \mathbb{Z}^{2n\times 2n}$, $Z_n \in \mathbb{Z}^{n\times n}$ is a down-shift matrix, and $\oplus$ denotes a Kronecker sum. Since M has a rectangular form, the generators $G \in \mathbb{Z}^{2n\times 2}$ and $B \in \mathbb{Z}^{(n+1)\times 2}$ are of different dimensions. They can be expressed as follows:

$$G^T = \begin{bmatrix} 1 & \cdots & 1 & 1 & 0 & \cdots & 0 \\ y_0 & \cdots & y_{n-1} & 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$B \equiv 0 \quad \text{except for } B_{0,0} = 1, \quad B_{n,1} = -1. \tag{9}$$

Again, in each step of the algorithm we update the matrices as in (5). Having the proper form generators $(\overline{G}_i, \overline{B}_i)$ in step i, the recursion in matrix form can be written as follows (for $0 \le i < n$):

$$\begin{bmatrix} \mathbf{0} \\ G_{i+1} \end{bmatrix} = \Phi_i \overline{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$
$$\begin{bmatrix} \mathbf{0} \\ B_{i+1} \end{bmatrix} = \Psi_i \overline{B}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{B}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \tag{10}$$

which should read as: "take the first column of $\overline{G}_i$ (or $\overline{B}_i$) and multiply it with Φ_i (or Ψ_i), keeping the second column unchanged". Here, $\Phi_i = F_i - f_i I_i$, $\Psi_i = A_i(I_i - f_i A_i)^{-1}$, where F_i, A_i and I_i are obtained by deleting the first i rows and columns from matrices F, A and I_n, respectively; and f_i is an i-th diagonal element of F. After n steps of the algorithm, the product GB^T yields the solution of (7).

Although, the equations look complicated at first glance, in the next section we see that the matrix B is obsolete due to its trivial structure, leaving us with a single generator G.

2.6. Polynomial interpolation

Let $G = (\mathbf{a}, \mathbf{b})$, $B = (\mathbf{c}, \mathbf{d})$ be the generators as defined in (9), where we again associate the matrices with column vectors. We will show that it suffices to work with a *single* generator G which leads to an efficient interpolation algorithm. The key to understanding is the fact that, in contrast to univariate resultants, we are not interested in *intermediate* results (i.e., diagonal elements) of matrix factorization. Instead, our goal is to obtain the *final* Schur complement R of M as defined in Section 2.5. Such R is given by the product GB^T of the generators after n steps of the Schur algorithm.[4]

Observe that, at the beginning $B := B_0$ has only two non-zero entries ($c_0 = 1$ and $d_n = -1$) and is already in a proper form (since $d_0 = 0$). Now, using (10), straightforward computations show that:

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

where $B_1 \in \mathbb{Z}^{n\times 2}$, $B_2 \in \mathbb{Z}^{(n-1)\times 2}$, and by $(*)$ we denote unimportant entries. A crucial observation here is that the leading rows of matrices B_i $(0 \le i < n)$ equal identically to $\begin{bmatrix} 1 & 0 \end{bmatrix}$ which can be verified by expanding the respective formulae. Next, provided that only the leading rows of G and B are needed to set up the rotation matrices in (4), we conclude that the matrix B does *not* affect the update of the matrix G. After n steps of the Schur algorithm, it follows that the last generator B_n has the form: $B_n = \begin{bmatrix} 0 & 1 \end{bmatrix}$,[5] which implies that the desired product $G_n B_n^T$ is simply given by the second column of G_n.

Now, we try to simplify the recursion for the generator G. By the above observations, we have $(c_0, d_0) = (1, 0)$ throughout the whole algorithm. Then, (5) implies that: $\tilde{a}_j = l_a(a_j c_0 + b_j d_0) \equiv l_a a_j$. Hence, only the column vector $\mathbf{b}$ needs to be multiplied by the rotation matrix, and we can omit the denominator l_a for the column $\mathbf{a}$. Besides, observe that only n entries of the generator G are *non-zero* at a time. Thus, we can use a sort of "sliding window" approach where only n relevant entries of G get updated in each iteration of the Schur algorithm. The pseudocode is given in Algorithm 2. Here, lines 10–12 are the effect of multiplying Φ_j from (10) by the column vector $\mathbf{a}$. Note that, we use auxiliary variables s and t to write the update of $\mathbf{a}$ in a "uniform" way. This is done with the intention to later avoid excessive branching in the GPU code.[6]

Algorithm 2 Polynomial interpolation

```
1:  procedure INTERPOLATE(x : Vector, y : Vector, n : Integer)
2:                  ▷ returns the coefficients of f(x), s.t., f(x_i) = y_i, 0 ≤ i < n
3:    let G = (a, b), l_int = 1        ▷ set up the generator G and denominator
4:    for j = 0 to n − 1 do
5:      for i = j + 1 to j + n − 1 do          ▷ multiply b by rotation matrix
6:        b_i = b_i a_j − a_i b_j
7:      od
8:      l_int = l_int · a_j, s = 0, t = 0            ▷ update the denominator
9:      for i = j + 1 to j + n do            ▷ multiply the matrix Φ_j with a
10:       if (i < n) then s = a_i, t = x_i          ▷ consider different cases
11:       elif (i > n and i ≤ j + n) then s = a_{i−1}, t = 1 fi
12:       a_i = s · t − a_i · x_j
13:     od            ▷ update the last non-zero entries of a and b:
14:     b_{j+n} = −b_j, a_{n+j+1} = 1
15:   od                 ▷ divide the coefficients by the denominator:
16:   b_i ← −b_i/l_int  for ∀i = n . . . 2n − 1
17:   return (b_n . . . b_{2n−1})          ▷ return the coefficients of f
18: end procedure
```

As a last remark, observe that, the Vandermonde matrix is strongly regular by definition, provided that interpolation points x_i are pairwise distinct. Hence, there is no need for special treatment as in the case of Sylvester's matrix.

3. General purpose computing on GPUs

In this overview, we particularly focus on the GPUs supporting CUDA framework, nevertheless the main ideas and principles are common to other architectures as well. With the release of Tesla architecture [19], NVIDIA GPUs no longer have separated fragment and vertex processors. Instead, they are unified in Multiprocessors (MPs) capable of running shader programs as well as general purpose parallel programs. The number of MPs per GPU can vary in a broad range. In its turn, each multiprocessor comprises a set of scalar in-order processors which execute the same instruction at a time in parallel using threads. A minimal scheduling entity on the GPU is a *warp* consisting of 32 threads. Threads of a warp are assigned to the processor cores of an MP and always execute synchronously. When a data-dependent branch causes a warp to *diverge* (follow different execution paths), all taken branch paths are processed serially. Such a model of execution is known as SIMT (Single Instruction Multiple Thread) which is similar to SIMD with an exception that the vector organization is not exposed to the software allowing a programmer to write thread-level parallel code. On the other hand, should performance become a critical factor, one can always consider a physical organization of threads into warps.

CUDA [8] is a heterogeneous serial–parallel programming model which means that the serial execution on the host machine is interleaved with parallel execution on the GPU. A program running on the GPU is referred to as *kernel*. Kernel is launched on a large number of parallel threads organized in a *grid* of

[4] Here n stands for the number of interpolation points.

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

[6] Short conditional statements are likely to be replaced by predicated instructions to avoid branching in the GPU code.

Please cite this article in press as: P. Emeliyanenko, Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra, J. Parallel Distrib. Comput. (2012), doi:10.1016/j.jpdc.2012.07.015

thread blocks. A thread block contains up to 512 threads that can communicate via shared memory and synchronize memory access with barriers. The more recent Fermi architecture [21] increases a block size to 1024 threads. Different thread blocks run without synchronization and cannot communicate within one kernel call. Therefore, sequentially dependent algorithms must be split in two or more kernels to provide data flow. Block independence is one of the cornerstones of the GPU programming model which enables a binary program to run unchanged on devices with a different number of multiprocessors.

Each MP has a large set of physical *registers* (8–16 kb per MP) which are statically allocated to threads of a block when it is scheduled for execution on a multiprocessor. Static register allocation has an advantage that context switching induces no overhead since each thread has its own private copy of registers.

To allow interthread communication within a block, each multiprocessor has a 16–48 kb block of fast *shared memory* serviced via 16 banks (32 banks on Fermi) to speed-up concurrent access by threads of a block. Another three memory spaces, including read-only *texture* and *constant* memory as well as read–write *global* memory, are simultaneously accessible by the entire grid of thread blocks and have a lifetime of an application. Texture and constant memory are cached on the device and optimized for different data locality. Global memory resides in external DRAM and is of much higher latency than on-chip shared memory. It is, therefore, highly recommended to use *coalescing* access optimizations to utilize memory bandwidth more efficiently (even though it is cached on the devices with Fermi architecture).

Overall GPU optimization strategies are: 1. maximize parallel execution; 2. minimize global memory access by storing intermediate results in shared/register memory; 3. keep high arithmetic intensity of computations to hide memory access latencies; 4. control register and shared memory allocation to improve thread occupancy,[7] which also includes favoring small thread blocks over large ones.

4. Structure of the algorithm

We start with a high-level description of the algorithm. Next, we discuss some practical methods on how to deal with a non-strong regularity issue raised in Section 2.4.

4.1. High-level structure

At the highest our approach follows the ideas of the classical modular algorithm by Collins. Given two bivariate polynomials $f, g \in \mathbb{Z}[x, y]$, the algorithm consists of five steps:

(a) apply modular homomorphism reducing the coefficients of f and g modulo sufficiently many primes: $\mathbb{Z}[x, y] \to \mathbb{Z}_m[x, y]$;
(b) for each modular image, choose a set of points $\alpha_m^{(i)} \in \mathbb{Z}_m$ and evaluate the polynomials at $x = \alpha_m^{(i)}$ (evaluation homomorphism): $\mathbb{Z}_m[x, y] \to \mathbb{Z}_m[x, y]/(x - \alpha_m^{(i)})$;
(c) compute a set of univariate resultants in $\mathbb{Z}_m[x]$ using a matrix-based approach:
$\mathrm{res}_y(f, g)|_{\alpha_m^{(i)}} : \mathbb{Z}_m[x, y]/(x - \alpha_m^{(i)}) \to \mathbb{Z}_m[x]/(x - \alpha_m^{(i)})$;
(d) interpolate the resultant polynomial for each prime m using a matrix-based approach:
$Z_m[x]/(x - \alpha_m^{(i)}) \to \mathbb{Z}_m[x]$;
(e) lift the resultant coefficients by means of Chinese remaindering: $\mathbb{Z}_m[x] \to \mathbb{Z}[x]$.

Steps (a)–(d) and partly (e) are implemented on the graphics processor. In essence, what remains to be done on the host machine is to filter out "bad" primes (to be discussed below) and, at the end of the algorithm, convert the resultant coefficients from the mixed-radix representation to positional number system.

The number of primes and evaluation points needed by the algorithm is usually estimated using Hadamard's bound on resultant's height and degree, respectively. We refer to [20] where some practical methods are outlined: for instance, we can compute the bounds over columns and rows of Sylvester's matrix, and then pick up a minimal one. It might be possible to obtain sharper estimates by investigating the theory of *sparse resultants*. It can be shown that the resultant degree equals *mixed volume* of Newton polytopes of two polynomials: see [7] for some basic facts, and [27] for the height of a sparse resultant.

To recover the resultant from its homomorphic images, we need to deal "unlucky homomorphisms". Using terminology from [20]: *a prime m is said to be bad if $f_p \equiv 0 \mod m$ or $g_q \equiv 0 \mod m$.* Similarly, *an evaluation point $\alpha \in \mathbb{Z}_m$ is bad if $f_p(\alpha) \equiv 0 \mod m$ or $g_q(\alpha) \equiv 0 \mod m$.* Dealing with "bad" primes is relatively easy: we can discard them in advance during the initial modular reduction of polynomials. To account for "bad" evaluation points, we propose to run the algorithm with an *excess* amount of points (typically 1%–2% more than required). Then, in case the computation fails for some $\alpha_m^i \in \mathbb{Z}_m$, we simply *ignore* the result and take another evaluation point. This situation is easy to detect since "bad" evaluation points result in a zero denominator $\mathrm{l_{res}}$ in Algorithm 1. In a very "unlucky" case when we cannot reconstruct the resultant due to the lack of points, we restart the algorithm to compute extra information. Yet, this is not the whole story because the algorithm can also fail due to non-strongly regular Sylvester's matrix, which we consider in the next section.

Schematic view of the algorithm running on the graphics processor is depicted in Fig. 1. The computations begin on the host machine, where we search and remove "bad" primes from the moduli set, that is, the primes dividing the leading coefficients of either input polynomial. Then, the control is transferred to the GPU executing six CUDA kernels corresponding to the steps of Collins' modular algorithm. The kernels are listed with the respective grid and block configurations in the figure, we consider each one of them in detail in Section 5. Yet, prior to this, we need to take care of some unfinished business with non-strongly regular Sylvester's matrices.

4.2. Dealing with non-strong regularity

In this section we outline some practical ways how to prevent the algorithm's failure. First, remark that, non-strong regularity indicates a presence of some non-trivial relation between polynomial coefficients which occurs quite rarely in practice. Furthermore, the majority of such situations can be handled in exact same way as "bad" evaluation points.[8] Indeed, if the computation fails for some evaluation point, there is, in general, a large selection of other points to choose from. That is, suppose we are given the following polynomials:

$$f = y^8 + y^6 - 3y^4 - 3y^3 + (x + 6)y^2 + 2y - 5x,$$
$$g = (2x^3 - 13)y^6 + 5y^4 - 4y^2 - 9y + 10x + 1.$$

Then, for $\alpha = \{0, \ldots, 100\,000\}$, Sylvester's matrix of $f(\alpha, y)$ and $g(\alpha, y)$ is non-strongly regular only for a single point $\alpha = 2$. However, the problem occurs if, for polynomials $f, g \in \mathbb{Z}[x, y]$, some minors of Sylvester's matrix $S^{(y)}$ (defined in Section 2.1) vanish

7 Occupancy is one of the main GPU performance indicators, it is the ratio of the number of resident threads to the maximum number of resident threads per MP.

8 In fact, *both* non-strong regularity of Sylvester's matrix and "bad" evaluation points yield a zero denominator $\mathrm{l_{res}}$ in Algorithm 1, and therefore are not distinguishable from the algorithm's perspective.

Please cite this article in press as: P. Emeliyanenko, Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra, J. Parallel Distrib. Comput. (2012), doi:10.1016/j.jpdc.2012.07.015

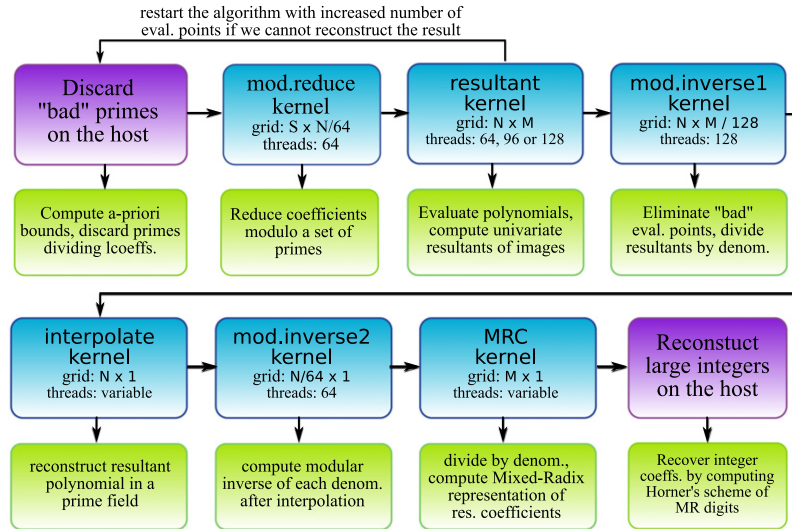

Fig. 1. Structure of the modular algorithm running on the GPU. *Abbrev.*: S: total number of scalar coefficients of both polynomials; N: number of moduli; M: number of evaluation points.

identically. According to large empirical evidence, it mostly happens when one of the polynomials has a *zero* trailing coefficient. This can be exemplified as follows. Let

$$f = y^8 + (4x^2 - 12)y^6 + (12x^3 + 2)y^4 + (20x^4 - 28x^2 + 12)y^2 - 18x^4 - 3, \quad \text{and} \quad g = f'_y,$$

hence, we have $g_0 \equiv 0$, and every second principal minor of $S^{(y)}$ (skipping the first seven) vanishes identically. One simple way to fix this is consider the resultant of *swapped* polynomials, i.e., $\text{res}_y(f, g) = -\text{res}_y(g, f)$. Indeed, in the example above, the matrix $S^{(y)}$ for g and f (swapped) is strongly-regular. Although, this sometimes does not work. A more general approach is to divide out a factor y of g (which causes the algorithm to break down), compute a "reduced" resultant, and then compensate for the factor y. In other words, suppose $f_0 \neq 0$ and $g = h \cdot y^k$, then it holds that[9]: $\text{res}_y(f, g) = \text{res}_y(f, h) \cdot f_0^k$.

Our long-term practical experiences show that the above two cases cover 99% of all "bad" situations. However, in extremely pathological cases (when neither of the above works), we can further exploit resultant properties in the attempt to "randomize" Sylvester's matrix. For example, consider the following:

$$\text{res}_y(f, g) = \text{res}_y(h, g) \quad \text{if } h = f + g \cdot q, \text{ and } \deg_y h = \deg_y f,$$

$$\text{res}_y(f, g)^2 = \text{res}_y(f^2, g) = \text{res}_y(f, g^2).$$

In the latter case, we also need to compute a polynomial square root to extract the actual resultant which can be achieved by a linear-time algorithm.

5. Realization

In this section we outline the main aspects of the GPU implementation. We begin with the realization of modular arithmetic, and then proceed with an in-depth discussion of the GPU kernels corresponding to the steps of the modular algorithm.

5.1. Modular arithmetic

The efficiency of our algorithm largely depends on how good the realization of modular operations is. Performing fast modular computations on the GPU is not an easy task to accomplish because the graphics hardware is heavily optimized for floating-point performance while, for example, integer division and modulo ('%') operations are particularly slow and should be avoided. Furthermore, GPUs with Tesla architecture [19] support natively only 24-bit integer multiplication while 32-bit multiplication is demoted in a more primitive operations. Luckily, the next generation Fermi GPUs support 32-bit integer arithmetic fully in hardware. Below, we consider the realization of relevant modular operations for *both* architectures.

Listing 1 24-bit modular arithmetic for Tesla GPUs

```
1:  procedure MUL_MOD(a, b, m, invm)                 ▷ computes a · b mod m
2:     hf = uint2float_rz(umul24hi(a, b))             ▷ 32 MSB of the product
3:     prodf = fmul_rn(hf, invm)                      ▷ invm = (float)(1 ≪ 16)/m
4:     l = float2uint_rz(prodf)                       ▷ truncate towards zero
5:     r = umul24(a, b) − umul24(l, m)                ▷ r ∈ [−2m + ε; m + ε]
6:     if r < 0 then                                  ▷ adjust the result if negative sign
7:        r = r + umul24(m, 0x1000002)                ▷ r = r + m · 2
8:     fi
9:     return umin(r, r − m)                          ▷ return r = a · b mod m
10: end procedure
11:                                                   ▷ computes (x1y1 − x2y2) mod m
12: procedure SUB_MUL_MOD(x1, y1, x2, y2, m, inv1, inv2)
13:    h1 = uint2float_rz(umul24hi(x1, y1))           ▷ two inlined MUL_MOD's
14:    h2 = uint2float_rz(umul24hi(x2, y2))
15:    l1 = float2uint_rz(fmul_rn(h1, inv1))          ▷ inv1 = 65536.0f/m
16:    l2 = float2uint_rz(fmul_rn(h2, inv1))          ▷ mul. and truncate
17:    r = mc + umul24(x1, y1) − umul24(l1, m)        ▷ mc = m · 100
18:    r = r − umul24(x2, y2) + umul24(l2, m)         ▷ diff. of MUL_MOD's
19:                                   ▷ inv2 = 1.0f/m, e23 = (float)(1 ≪ 23)
20:    rf = uint2float_rn(r) ∗ inv2 + e23             ▷ rf = ⌊r/m⌋
21:    r = r − umul24(float_as_int(rf), m)
22:    return (r < 0 ? r + m : r)
23: end procedure
```

5.1.1. Tesla architecture

For Tesla GPUs, we restrict ourselves to 24-bit modular arithmetic which reflects the hardware capabilities. Besides, 24-bit residue fits in the mantissa of a single-precision floating-point number, and therefore we can replace expensive integer division with operations in floating-point.

GPU provides two instructions, mul24.lo and mul24.hi, for integer multiplication,[10] which can be accessed directly from CUDA

[9] We assume that the polynomials are coprime, that is why $f_0 \neq 0$.

[10] They compute 32 least and most significant bits of 48-bit product of 24-bit integer operands, respectively.

Listing 2 31-bit modular multiplication on Fermi cards

```
1: procedure MUL_MOD(a, b, m, inv)                      ▷ computes a · b mod m
2:    hi = umulhi(a ∗ 2, b ∗ 2)                   ▷ compute 32 MSB of the product
3:                          ▷ mul. and truncate, inv = (double)(1 ≪ 30)/m
4:    rf = uint2double_rn(hi) ∗ inv + (double)(3 ≪ 51)          ▷ rf = ⌊r/m⌋
5:    r = a ∗ b − double2loint(rf) ∗ m                      ▷ partial residue
6:    return (r < 0 ? r + m : r)                ▷ adjust by m if negative sign
7: end procedure
```

Listing 3 31-bit Montgomery modular inverse

```
1: procedure MONTGOMERY_INVERSE(x, m, mu)                   ▷ res.: x^−1 mod m
2:   v = x, u = m, s = 1, r = 0, k = 0
3:   while (v! = 0) {                 ▷ first stage: compute r = x^−1 2^k mod m
4:     rs = r
5:     if (v & 1) {
6:       uv = v
7:       if ((v xor u) < 0) { v = v + u } else { v = v − u }
8:       if ((v xor uv) < 0) { u = uv, rs = s }
9:       s = s + r
10:    }
11:    v = v/2, r = rs ∗ 2, k = k + 1
12:  }
13:  r = m − r                         ▷ second stage: get rid of 2^k factor
14:  if (r < 0) { r = r + m }               ▷ r = x^−1 2^k mod m, 31 ≤ k ≤ 62
15:                  ▷ Montgomery mul.: r = r · (2^−m) = x^−1 2^(k−m)   (mod m)
16:  if (k >= 32) {
17:    s = r ∗ mu                                    ▷ mu = −m^−1 mod 2^32
18:    u = s ∗ m, v = umulhi(s, m)                   ▷ (v, u) = s · m (63 bits)
19:    u = u + r, r = v + (u < r), k = k − 32        ▷ r = ((v, u) + r)/2^32
20:  }
21:  if (k == 0) return r
22:            ▷ Montgomery mul.: r = r · (2^(m−k))(2^−m) = x^−1   (mod m)
23:  s = r ≪ (32 − k), d = s ∗ mu                    ▷ mu = −m^−1 mod 2^32
24:  u = d ∗ m, v = umulhi(d, m)                     ▷ (v, u) = d · m (63 bits)
25:  d = r ≫ k, u = u + s
26:  r = v + d + (u < s)                             ▷ r = ((v, u) + s)/2^32 + d
27:  return r
28: end procedure
```

using inline PTX assembly [22]. We denote the corresponding intrinsics by umul24 and umul24hi. The algorithm computing $a \cdot b \bmod m$ for two residues a and b is given by MUL_MOD in Listing 1. The algorithm works as follows. First, we split the product $a \cdot b$ in 32- and 16-bit parts, and apply a congruence:

$$\begin{aligned} a \cdot b &= 2^{16}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo \\ &= (2^{16}hi - m \cdot l) + lo = a \cdot b - m \cdot l = r, \end{aligned}$$

where $0 \le \lambda < m$. It holds that $r \in [-2m + \varepsilon; m + \varepsilon]$ for $0 \le \varepsilon < m$. Hence, r fits in 32 bits and we can compute it by considering only 32 least significant bits of the products $a \cdot b$ and $m \cdot l$, see line 5 in Listing 1. Finally, in lines 6–9 we further reduce r to a valid residue range.

The subroutine SUB_MUL_MOD in Listing 1 evaluates an expression: $(x_1y_1 - x_2y_2) \mod m$ which is frequently used in rotation formulas, see Section 2.3. The procedure consists of two inlined MUL_MOD operations with the difference that we subtract partial residues in lines 17–18 before to the final reduction. The advantage is that the compiler merges subsequent *multiply* and *add* instructions producing more efficient code. The remaining lines 20–23 are needed to bring r to the valid residue range. In line 20 we also use a mantissa trick [15] to multiply by $1/m$ and truncate the result using one multiply–add instruction.

5.1.2. Fermi architecture

On Fermi cards, integer multiplication is no longer a problem, and we can enjoy full-precision 31-bit modular arithmetic. Additionally, we can benefit from the performance of double-precision arithmetic which is now only two times slower than single-precision. The algorithm mul_mod in Listing 2 is analogous to its 24-bit counterpart. The main difference is that we use umulhi intrinsic returning 32 MSB of 64-bit integer product. For two 31-bit residues a and b, we again partition the product $a \cdot b$ in 32- and 30-bit parts, and use the following:

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

where $0 \le \lambda < m$. Here, $r \in [-m + \varepsilon; \varepsilon]$ for $0 \le \varepsilon < m$. Hence, it remains to adjust r by m in case of negative sign. In line 4 of Listing 2, we also use "magic number" 3^{51} to truncate a floating-point value to the nearest integer [15]. Note that, in the 31-bit case, we do provide a special realization of the SUB_MUL_MOD routine.

Lastly, we consider a modular inverse operation which is used by resultant and interpolation algorithms (see lines 3 and 22 in Algorithm 1, and line 16 in Algorithm 2). The realization is similar on both architectures, that is why we discuss Fermi's algorithm only. Listing 3 gives the pseudocode of the Montgomery modular inverse algorithm which is a combination of the ideas from [9,24]. The algorithm consists of two stages. In the first stage, lines 3–12, we compute iteratively $x^{-1}2^k \mod m$, where x is a residue modulo m, and $31 \le k \le 62$. The number of iterations is bounded by the moduli bit-length (31 bits). Next, in lines 13–26, we divide out the factor 2^k using two Montgomery multiplications by the powers of two. Comments in the code should help further understanding of the algorithm.

5.2. Realization: general remarks

In what follows, we will consider implementation of each GPU kernel as shown in Fig. 1. Yet, before going through the realization details, we give an overview of some common optimization techniques we have applied to improve the performance.

Among standard tools, we use constant propagation via templates, loop unrolling, exploiting warp-level parallelism (to compute a parallel prefix sum), and favoring small thread blocks over large ones. To decrease register pressure, we also declare frequently used variables with `volatile` keyword. This qualifier forces the compiler to keep these variables in physical registers and reuse them, instead of inlining the corresponding expressions. Besides, we provide *launch bounds* for each GPU kernel. These parameters, first available in CUDA version 3.0, provide a hint to the compiler to optimize register allocation. Alone, the latter technique gave us about 30% of additional speed-up.

As a last remark, we keep a moduli set and corresponding reciprocals (`inv`) needed for modular reduction in *constant* memory space. This is because one thread block (except the MRC kernel) uses a *single* modulus throughout all computations. Therefore, all threads of a block read the *same* value and the data is loaded via constant memory cache. Also, direct reference to the data in constant memory has a positive effect on reducing register pressure.

5.3. Modular reduce kernel

The first 'mod.reduce' kernel in Fig. 1 performs modular reduction of polynomial coefficients. According to some empirical evidence, the cost of this operation might become very high, if not dominating, when the modular algorithm is applied to polynomials with large coefficients but moderate degrees. Therefore, we have decided to perform these computations on the GPU as well.

Here, the grid configuration is chosen to be $S \times N/64$, where S is a total number of scalar coefficients of both polynomials and N is the size of moduli set. In other words, we partition the moduli set in chunks of size 64 primes each and assign one CUDA block to compute 64 residues of some polynomial coefficient. We dedicate one thread to compute a residue modulo some prime m_i using a

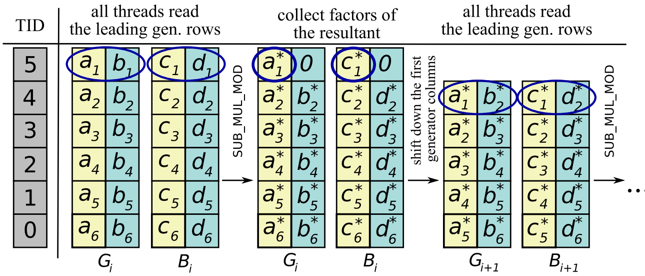

Fig. 2. The workflow of univariate resultant algorithm running with 6 threads indexed by *TID*.

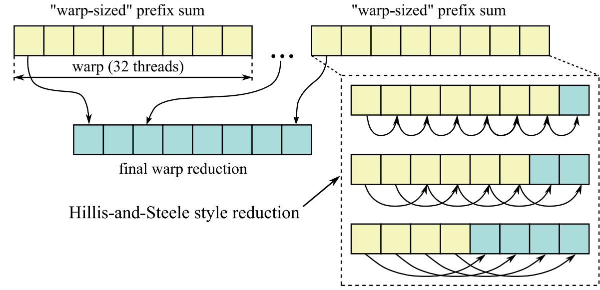

Fig. 3. "Warp-sized" parallel reduction (prefix sum).

simple iterative algorithm, known as single-word division, which can be found in many textbooks; see, e.g., [3]. Thread workload is perfectly balanced because each thread does the same job. Here, only one complication arises: namely, how to access the input data in an optimal way since polynomial coefficients might be too large (several thousand bits) for allocating them entirely in shared memory. We deal with this problem by "streaming" data through the kernel: i.e., the large integers are processed in chunks of size just enough to fit the GPU's shared memory.

5.4. Resultant kernel

Resultant kernel constitutes the core of the algorithm. We designate one thread block to compute a univariate resultant modulo a prime m_i for some evaluation point $\alpha_j \in \mathbb{Z}_{m_i}$. Accordingly, a grid configuration for this kernel is $N \times M$, see Fig. 1. We provide four kernel specializations for 32×2, 64, 96 and 128 threads per per block. The number of threads depends on the degree of input polynomials. A kernel with P threads handles polynomials with degrees $P/2 \le d < P$.[11]

First, polynomials are evaluated at $x = \alpha_j$. We use a simple Horner scheme where each thread iteratively computes one coefficient of $f(\alpha_j, y)$ and $g(\alpha_j, y)$. Next, we run univariate resultant algorithm from Section 2.4 The algorithm maps quite straightforward to the GPU: the outer loop is split up in "lite" and "full" iterations while the inner loop is vectorized. In other words, we associate one thread with four data elements: $(\mathbf{a}_i, \mathbf{b}_i)$ and $(\mathbf{c}_i, \mathbf{d}_i)$ which correspond to one row of each of the matrices $G = (\mathbf{a}, \mathbf{b})$ and $B = (\mathbf{c}, \mathbf{d})$. In each iteration, the current top generator rows are shared between all threads. Then, each thread applies rotation formulas from Section 2.3 (or SUB_MUL_MOD operations) to its data elements. At the end, the first columns **a** and **c** are "shifted down" preparing for the next iteration, and the resultant factors are saved in shared memory. One step of the algorithm during "full" iterations is depicted in Fig. 2.

For "lite" iterations, we also unroll the outer loop by the factor of two for higher arithmetic intensity, such that one thread now processes two rows of each of the matrices G and B at a time. In this way, we double the maximal degree of polynomials that can be handled, and ensure that all threads are occupied. Besides, at the beginning of "full" iterations, we can guarantee that at least half of threads are busy in the corner case ($d = P/2$). Observe that, the number of working threads decreases with the length of generators in the outer loop. Therefore, we use a load balancing strategy to improve thread occupancy: when at least half the threads enter the idle state, we switch to another code subroutine where computations are organized in a way that threads only do half a job. Eventually, once the generator size descends below the warp boundary, the remaining algorithm steps are run *without* thread synchronization because warp, as a minimal scheduling entity, always executes synchronously on the GPU.

Finally, the product of the collected factors (as well as the product of respective denominators l_{res}) is computed using "warp-sized" parallel reduction based on a "Hillis-and-Steele" style algorithm [16]. The idea of this algorithm is to run prefix sums independently for several warps without synchronization, and then combine the results in a final reduction step, see Fig. 3.

To sum up, since the inner loop of Algorithm 1 is now vectorized, the *parallel complexity* of computing univariate resultants becomes *linear* in the polynomial degrees.

5.5. Modular inverse 1 and stream compaction kernel

The purpose of this kernel is to divide the univariate resultants computed in the previous step by respective denominators l_{res} (see Algorithm 1), and eliminate "bad" evaluation points.[12] For each modulus m_i, we partition the set of computed resultants in 128-element chunks and assign one chunk to a block with 128 threads. This is why, the kernel is launched on a grid of size $N \times M/128$, see 'mod.inverse1' in Fig. 1. Division is performed using the Montgomery modular inverse algorithm, see Listing 3.

"Bad" evaluation points are discarded using a stream compaction algorithm. Our realization is based on the following remarks. 1. The number of evaluation points M per modulus can be quite large (generally about 1–2 thousand), hence it is inefficient/infeasible to process all points by one thread block. 2. Running hierarchical stream compaction in global memory (several kernel launches) does not pay off because "bad" evaluation points occur quite rarely on the average.[13] 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. Remark that, on Fermi we can use ballot voting primitive and popc intrinsic to efficiently compute the prefix sum, see [8,11].

Finally, a compacted sequence is written back to global memory. The current writing position is controlled by a global variable which gets updated (atomically) each time a block outputs its results to global memory, see Fig. 4.

5.6. Interpolation kernel

Interpolation kernel realizes the algorithm from Section 2.6. Here, one thread block is dedicated to interpolating a polynomial for some modulus m_i. Similarly to the resultant kernel, the

11 The first kernel with 32×2 threads computes two resultants at a time.

12 The points for which denominators vanish, see Section 4.1.

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

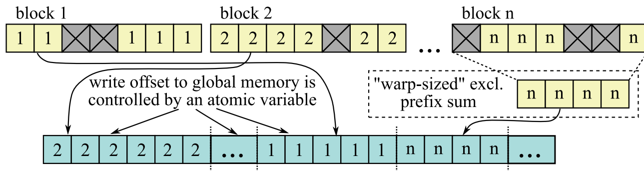

Fig. 4. Stream compaction across several thread blocks, 'x' marks elements being eliminated. Results for each block are written back to global memory in unspecified order.

inner loop of the algorithm is vectorized. However, since the interpolation algorithm is simpler than that of resultants, one thread is used assigned to process *two* or *four* rows (depending on the number of evaluation points M) of the generator matrix $G = (\mathbf{a}, \mathbf{b})$ (see Algorithm 2). This enables us to keep high arithmetic intensity of computations. One important remark is that, the size of the matrix G does *not* decrease throughout the algorithm: instead, in each iteration we update only M relevant entries of G in a "sliding window" fashion (see discussion in Section 2.6).

The number of threads per block is selected according to the number of evaluation points M. Meaning that, the maximal resultant degree is limited to 2048 on Tesla and 4096 on Fermi (chosen according to the maximal number of threads per block on respective architectures). According to our experiences, these limitations would satisfy the demands of the most practical applications.

For reasons of efficiency, we also found it advantageous to parameterize the kernel by *data parity* (in other words, by $M \bmod 2$ or $M \bmod 4$ depending on the loop unrolling factor) instead of the data size itself. By parameterizing the kernel in such a way, we can substantially reduce branching because most of the branch decisions are resolved at compile time. Besides, kernel parameterization also lowers register usage and has a positive effect on performance. At the end of the algorithm, the product of denominators $\mathsf{l}_{\mathrm{int}}$ is evaluated using parallel prefix sum.

Again, our interpolation algorithm executes in *linear parallel time* on the GPU since the inner loop of the algorithm is vectorized.

5.7. Modular inverse kernel 2

In this kernel, 'mod.inverse2' in Fig. 1, we precompute the modular inverses for subsequent division of polynomial coefficients by respective denominators $\mathsf{l}_{\mathrm{int}}$ produced in the course of the interpolation. Its realization is rather straightforward and closely resembles that of the 'mod.inverse1' kernel considered above (with the exception that stream compaction is no longer used). Therefore we omit further discussion.

5.8. MRC (Mixed-Radix conversion) kernel

The remaining MRC kernel reconstructs integer coefficients of a resultant in the form of Mixed-Radix (MR) digits. The main property of the Mixed-Radix conversion (MRC) algorithm [28] is that MR digits can be computed *without* resorting to multi-precision arithmetic (unlike conventional Chinese remainder algorithm). This fact becomes decisive for choosing this algorithm for realization on the GPU.

Speaking precisely, for a given set of primes $(m_1, m_2, \ldots, m_N)$ and respective residues $(x_1, x_2, \ldots, x_N)$, the MRC algorithm associates a large integer X with mixed-radix digits $\{\alpha_i\}$ in the following way:

$$X = \alpha_1 M_1 + \alpha_2 M_2 + \cdots + \alpha_N M_N,$$

where $M_1 = 1, M_j = m_1 m_2 \cdots m_{j-1}$ $(j = 2, \ldots, N)$. Having the set of MR digits, it is clear that, we can obtain a large integer X by simply evaluating a Horner scheme:

$$X = \alpha_1 + m_1(\alpha_2 + m_2(\alpha_3 + m_3(\cdots + \alpha_N)\cdots)).$$

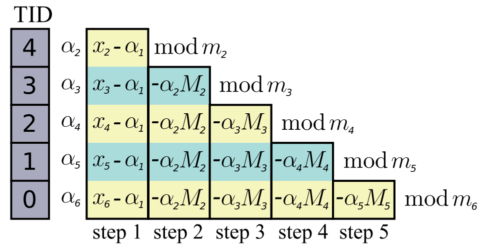

Fig. 5. Simple MRC algorithm running with 5 threads indexed by *TID*. Besides digits α_j, in step i, each thread computes $M_{i+1} = M_i m_i \bmod m_j$.

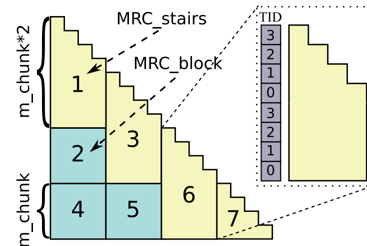

Fig. 6. Improved MRC algorithm based on two subalgorithms running with m_chunk := 4 threads. Numbers inside the triangle indicate the order in which the subalgorithms are applied.

MR digits can be evaluated as follows $(i = 1, \ldots, N)$:

$$\alpha_1 = x_1, \qquad \alpha_2 = (x_2 - \alpha_1)c_2 \bmod m_2,$$
$$\alpha_3 = ((x_3 - \alpha_1)c_3 - (\alpha_2 M_2 c_3 \bmod m_3)) \bmod m_3, \ldots$$
$$\alpha_i = ((x_i - \alpha_1)c_i - (\alpha_2 M_2 c_i \bmod m_i) - \cdots - (\alpha_{i-1} M_{i-1} c_i \bmod m_i)) \bmod m_i,$$

where $c_i = (m_1 m_2 \cdots m_{i-1})^{-1} \bmod m_i$ are precomputed in advance, and M_i are evaluated "on-the-fly". To further simplify the computations, we can arrange the moduli in *increasing order*, that is: $m_1 < m_2 < \cdots < m_N$. Then, the expressions of the form $\alpha_j M_j c_i \bmod m_i$ for $j < i$ can be evaluated without preceding modular reduction of α_j since $\alpha_j < m_i$. The same argument applies to updating M_i's. With this observation, one can conceive of a simple parallel algorithm computing MR digits, see Fig. 5. Here, in step i $(i = 1, \ldots, N - 1)$, we use a previously computed digit α_i to update all the remaining digits α_{i+1} through α_N. Beside the assigned digit α_j, in step i, each thread also computes $M_{i+1} = M_i m_i \bmod m_j$. Certainly, the *parallel complexity* of this algorithm is again *linear* in the number of residues x_i. Sadly, this solution does not work when the "capacities" of a CUDA thread block are exceeded because threads need to work cooperatively. We could still process several digits by one thread. However, this solution would show very bad occupancy because now the number of working threads *decreases* in each iteration.

To figure out an optimal solution, we have observed that, 2–3 thousand 31-bit moduli (60–90 thousand bits per resultant coefficient) is more than enough for the majority applications. Hence, we propose to compute MR digits in a single thread block by processing them in *chunks* of size m_chunk. 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 is illustrated in Fig. 6 where we fill in a triangle using two "shapes" (subalgorithms): MRC_stairs and MRC_block. The procedure MRC_stairs closely resembles the

Table 1
Timing the resultants of f and g w.r.t. variable y. 1st *column*: instance number; 2nd *column*: $\deg_y/\deg_x$: degree of polynomials w.r.t. y- and x-variables, resp.; bits: coefficient bit-length; sparse/dense: varying density of polynomials; 3rd *column*: resultant degree; 6th *column*: grid configuration of the resultant kernel ($N \times M$), see Section 5.4.

#	Configuration	Degree	GPU	Maple	Blocks executed
1–2.	$\deg_y(f)$: **20**, $\deg_y(g)$: **16** (sparse) $\deg_x(f)$: **7**, $\deg_x(g)$: **11**, bits: **32**/**300**	548/548	16 ms/174 ms	1.16 s/11.9 s	353 × 628
3–4.	$\deg_y(f)$: **19**, $\deg_y(g)$: **17** (dense) $\deg_x(f)$: **40**, $\deg_x(g)$: **39**, bits: **32**/**100**	1664/1664	66 ms/189 ms	6.9 s/16.6 s	122 × 1519
5–6.	$\deg_y(f)$: **62**, $\deg_y(g)$: **40**, bits: **24** $\deg_x(f)$: **12**, $\deg_x(g)$: **10** (sparse/dense)	1107/2777	146 ms/498 ms	11.9 s/45.7 s	96 × 2902
7–8.	$\deg_y(f)$: **31**, $\deg_y(g)$: **20**, bits: **100** $\deg_x(f)$: **40**, $\deg_x(g)$: **30** (sparse/dense)	1689/2432	294 ms/486 ms	28.3 s/57.5 s	174 × 2491
9–10.	$\deg_y(f)$: **80**, $\deg_y(g)$: **90**, bits: **32** $\deg_x(f)$: **10**, $\deg_x(g)$: **10** (sparse/dense)	1736/3210	854 ms/1.94 s	78 s/189 s	187 × 1784
11–12.	$\deg_y(f)$: **60**, $\deg_y(g)$: **75** (sparse) $\deg_x(f)$: **20**, $\deg_x(g)$: **23**, bits: **32**/**200**	2981/2981	1.11 s/7.01 s	114 s/663 s	888 × 3032
13–14.	$\deg_y(f)$: **31**, $\deg_y(g)$: **23** (dense) $\deg_x(f)$: **42**, $\deg_x(g)$: **33**, bits: **24**/**400**	2027/2027	96 ms/1.93 s	12.7 s/201 s	705 × 2109
15–16.	$\deg_y(f)$: **41**, $\deg_y(g)$: **29**, bits: **900** $\deg_x(f)$: **20**, $\deg_x(g)$: **11** (sparse/dense)	1011/2910	3.9 s/14.3 s	247 s/? (>15 min)	2045 × 2854

algorithm in Fig. 5, with the exception that now we process *twice* as many digits per thread (which also justifies the shape of MRC_stairs in the figure). By assigning threads as shown in Fig. 6, we ensure that all threads are occupied in each step of the procedure. The purpose of MRC_block is to (optionally) preload and update a set of m_chunk digits using the ones computed in the preceding MRC_stairs call(s). By applying these subroutines in a specified order, one obtains an MRC algorithm which runs across a single block with m_chunk threads.

Since the number of processed chunks needed for MRC_block calls increases in each step, the main challenge of the algorithm is where to store the computed 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 register space and shared memory. By reaching a certain threshold, the data is to be placed in GPU's local memory.[14] Selection of a concrete kernel depends on the number of moduli N, and is based on heuristics favoring small blocks to large ones by adjusting the parameter m_chunk.

For a short summary, we would like to add that *all steps of the modular algorithm discussed above execute in $O(n)$ parallel time on the GPU with n threads, thereby making the final complexity also linear in the input parameters.* In the next final section, we provide detailed benchmarks showing the efficiency of the proposed approach. This will also justify the correctness of our complexity analysis.

6. Performance evaluation and conclusions

For experiments we have used a desktop machine with 2.8 GHz 8-Core Intel Xeon W3530 (8 MB L2 cache) CPU and GeForce GTX580 graphics card (Fermi core). To reconstruct resultant coefficients from mixed-radix digits on the host machine, we have employed the GMP 5.0.1 library.[15] As a main contestant we have taken a host-based modular resultant algorithm from 64-bit Maple 14.[16] For polynomials with integral coefficients, Maple provides a very efficient *builtin* implementation of Collins' modular approach. In other words, for integer polynomials, this implementation is available in a precompiled binary form. For large integer arithmetic Maple also uses the GMP library. We further note that, Maple's internal precision parameter (`Digits`) has no influence on the resultant algorithm since the latter one is based on modular computations where the integer precision (the number of primes) is chosen according to Hadamard's bounds to deliver the *exact* result. Maple's approach further relies on probabilistic techniques to minimize the number of prime moduli in use [20].

The timings are reported in Table 1. In the GPU column we account for all stages of the algorithm including recovering the large integer coefficients on the CPU. We have varied different parameters such as polynomials' x- and y-degrees, coefficient bit-length and density (the number of non-zero scalar coefficients). Each row in the table presents *two* experiments where a varying parameter is written with slash '/'. The last column 'blocks executed' specifies a grid configuration for the resultant kernel.

Observe that, Maple's algorithm performs better for sparse polynomials which indicates that, internally, Maple uses the PRS algorithm to compute univariate resultants while our matrix-based approach is less sensitive to the density parameter. On the other hand, our algorithm is faster for polynomials with high x-degree. This behavior is expected since, with increasing the x-degree, the number of thread blocks increases (thereby, leading to better hardware utilization) while the size of Sylvester's matrix stays the same. On the contrary, increasing the y-degree penalizes performance as it causes the number of threads per block to increase. Similarly, for larger bit-lengths, the attained performance is typically better again because of increased degree of parallelism. However, for moderate-degree polynomials with very large coefficients the time for the CPU routines (reconstructing large integers from MR digits) becomes noticeably large, see last row in Table 1.

A histogram in Fig. 7 shows relative contribution of different algorithm stages to the overall time. The numbers along x-axis correspond to configurations in Table 1. Clearly, the time for computing resultants is dominating since this is a key part of the algorithm. The second largest time is either reconstructing large integers on the host 'MR recover' (in case of large coefficient bitlength), or polynomial interpolation (in case of high degrees). The remaining two graphs in Fig. 7 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 increasing the polynomials' y-degree affects both the number of moduli and evaluation points and, as a

[14] Observe that, on Fermi accesses to local memory are cached.
[15] http://gmplib.org.
[16] `modp1(Prime(1))` is 3 037 000 453 which verifies 64-bit Maple.

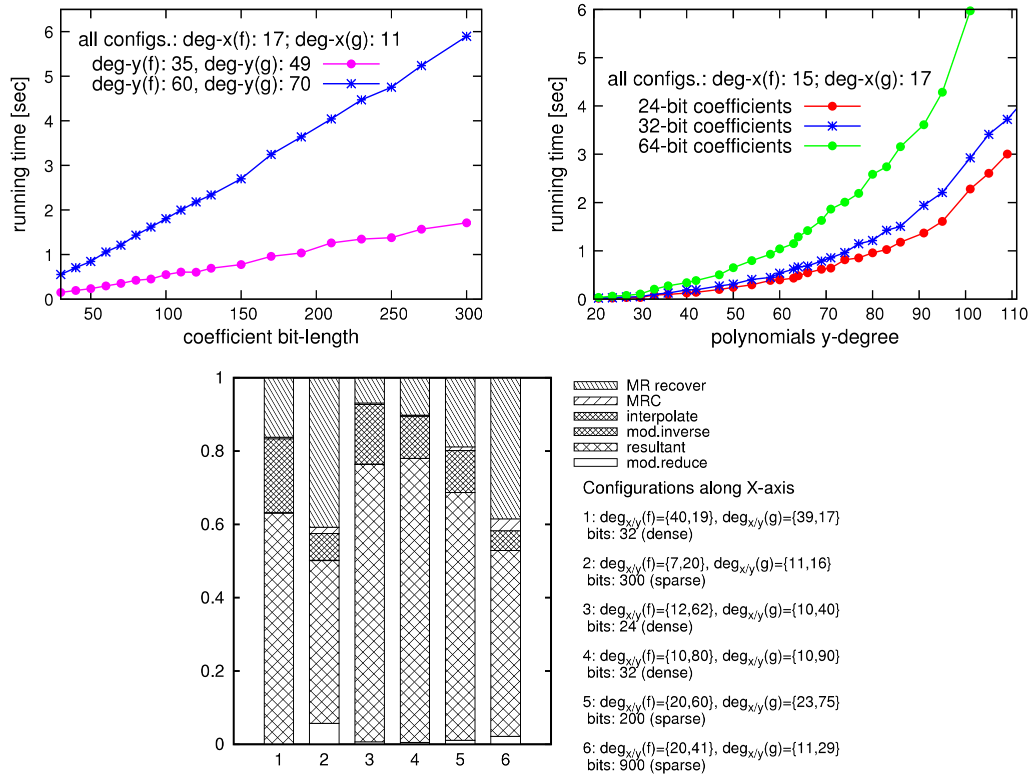

Fig. 7. *Top*: running time versus coefficient bitlength (*left*); and polynomial y-degree (*right*). *Bottom*: relative contribution of different algorithm stages to the overall time. *Source:* Various configurations are taken from Table 1.

result, performance degrades quadratically. Also, notice a jump of the running time between the y-degrees 90 and 100: this is because the algorithm switches to "larger" resultant kernel (the one with 128 threads per block) when all thread resources are occupied, see Section 5.4.

As a general remark, we have observed that even for moderate degree polynomials, our algorithm reaches hardware saturation very fast since the complexity of computing resultants increases rapidly with input parameters. Besides, due to high arithmetic intensity of computations, the time for GPU–host memory transfer was *negligibly* small for all instances we have tried. Altogether, this indicates that our approach should scale well on (future) GPUs with larger number of CUDA cores.

We have proposed an algorithm to computing resultants on the GPU. Our results indicate a significant performance improvement over a host-based implementation. Furthermore, our algorithm has a great scalability potential because of the large degree of (coarse-grained) parallelism inherent to the modular approach. Besides, we have shown that the matrix-based approach is well-suited for realization on massively-threaded architectures.

As future research directions, we would like to revisit our implementation in order to exploit block-level parallelism because, at the current state of development, our algorithm relies only on thread-level parallelism in that way limiting the size of polynomials that can be processed. One option would be to adapt the recursive Schur-type algorithm based on the block matrix inversion formula, see [23]. We would also like to move the remaining algorithm stage (MR recover) to the GPU since it can become relatively expensive for high degree polynomials. A final goal would be to extend this approach to other computationally-intensive tasks from computer algebra that can be reformulated in matrix form, thereby making a transition towards the fully GPU-accelerated computer algebra. Here, good candidates could be GCDs of multivariate polynomials or subresultants.

References

[1] E. Berberich, P. Emeliyanenko, A. Kobel, M. Sagraloff, Arrangement computation for planar algebraic curves, in: SNC'11, ACM, San Jose, USA, 2011.
[2] E. Berberich, P. Emeliyanenko, M. Sagraloff, An elimination method for solving bivariate polynomial systems: eliminating the usual drawbacks, in: ALENEX'11, SIAM, San Francisco, USA, 2011, pp. 35–47.
[3] R. Brent, P. Zimmermann, Modern computer arithmetic (version 0.5.1), CoRR abs/1004.4710.
[4] T. Bubeck, M. Hiller, W. Küchlin, W. Rosenstiel, Distributed symbolic computation with DTS, in: IRREGULAR'95, Springer-Verlag, London, UK, 1995, pp. 231–248.
[5] S. Chandrasekaran, A.H. Sayed, A fast stable solver for nonsymmetric Toeplitz and quasi-Toeplitz systems of linear equations, SIAM Journal on Matrix Analysis and Applications 19 (1998) 107–139.
[6] G.E. Collins, The calculation of multivariate polynomial resultants, in: SYMSAC'71, ACM, 1971, pp. 212–222.
[7] D. Cox, J. Little, D. O'Shea, Using Algebraic Geometry, in: Graduate Texts in Mathematics, vol. 185, Springer-Verlag, New York, 1998.
[8] CUDA Compute Unified Device Architecture, Programming guide. Version 3.2, nVIDIA Corp., 2010.
[9] G. de Dormale, P. Bulens, J.-J. Quisquater, An improved montgomery modular inversion targeted for efficient implementation on FPGA, in: FPT '04. IEEE International Conference on, 2004, pp. 441–444.
[10] P. Emeliyanenko, A complete modular resultant algorithm targeted for realization on graphics hardware, in: PASCO'10, ACM, New York, NY, USA, 2010, pp. 35–43.
[11] P. Emeliyanenko, Accelerating symbolic computations on NVIDIA Fermi, poster presentation, 2010. Available at: http://www.nvidia.com/object/research_summit_posters_2010.html.
[12] P. Emeliyanenko, Modular resultant algorithm for graphics processors, in: ICA3PP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 427–440.
[13] E. Frantzeskakis, K. Liu, A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing, IEEE Transactions on Signal Processing 42 (1994) 2455–2469.
[14] K. Geddes, S. Czapor, G. Labahn, Algorithms for Computer Algebra, Kluwer Academic Publishers, Boston, Dordrecht, London, 1992.
[15] C. Hecker, Let's get to the (floating) point, Game Developer Magazine (1996) 19–24.
[16] W.D. Hillis, G.L. Steele Jr., Data parallel algorithms, Communications of the ACM 29 (1986) 1170–1183.
[17] H. Hong, H.W. Loidl, Parallel computation of modular multivariate polynomial resultants on a shared memory machine, in: CONPAR 94, Springer Verlag, 1994, pp. 325–336.

Please cite this article in press as: P. Emeliyanenko, Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra, J. Parallel Distrib. Comput. (2012), doi:10.1016/j.jpdc.2012.07.015

[18] T. Kailath, S. Ali, Displacement structure: theory and applications, SIAM Review 37 (1995) 297–386.
[19] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA tesla: a unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.
[20] M. Monagan, Probabilistic algorithms for computing resultants, in: ISSAC'05, ACM, 2005, pp. 245–252.
[21] NVIDIA's next generation CUDA compute architecture: Fermi, Whitepaper, NVIDIA Corp., 2010.
[22] PTX: parallel thread execution, ISA Version 2.1, NVIDIA Corp., 2010.
[23] J.H. Reif, Efficient parallel factorization and solution of structured and unstructured linear systems, Journal of Computer and System Sciences 71 (1) (2005) 86–143.
[24] E. Savas, C. Koc, The montgomery modular inverse-revisited, IEEE Transactions on Computers 49 (7) (2000) 763–766.
[25] A. Schönhage, E. Vetter, A new approach to resultant computations and other algorithms with exact division, in: ESA'94, Springer-Verlag, London, UK, 1994, pp. 448–459.
[26] W. Schreiner, Developing a distributed system for algebraic geometry, in: EURO-CM-PAR'99, Civil-Comp Press, 1999, pp. 137–146.
[27] M. Sombra, The height of the mixed sparse resultant, American Journal of Mathematics 126 (6) (2004) 1253–1260.
[28] M. Yassine, Matrix mixed-radix conversion for RNS arithmetic architectures, in: Proceedings of 34th Midwest Symposium on Circuits and Systems, 1991.

Pavel Emeliyanenko is a post-doctoral researcher at Max-Plank Institute for Informatics, Saarbruecken, Germany.

He received his M.Sc. degree in 2007 and his Ph.D. degree in 2012 at Saarland University, Saarbruecken, Germany.

His research interests are non-linear algebraic geometry, software development, parallel computing, GPGPU programming. In particular, he is engaged in developing parallel algorithms to speed-up symbolic computations (polynomial multiplication, resultants, GCD, subresultants, etc.) which have diverse applications, for instance, in the field of algebraic geometry or computer graphics.

Please cite this article in press as: P. Emeliyanenko, Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra, J. Parallel Distrib. Comput. (2012), doi:10.1016/j.jpdc.2012.07.015