Modular Resultant Algorithm for Graphics Processors

Pavel Emeliyanenko

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

Abstract. In this paper we report on the recent progress in computing bivariate polynomial resultants on Graphics Processing Units (GPU). Given two polynomials in $\mathbb{Z}[x, y]$, our algorithm first maps the polynomials to a prime field. Then, each modular image is processed individually. The GPU evaluates the polynomials at a number of points and computes univariate modular resultants in parallel. The remaining "combine" stage of the algorithm is executed sequentially on the host machine. Porting this stage to the graphics hardware is an object of ongoing research. Our algorithm is based on an efficient modular arithmetic from [1]. With the theory of displacement structure we have been able to parallelize the resultant algorithm up to a very fine scale suitable for realization on the GPU. Our benchmarks show a substantial speed-up over a host-based resultant algorithm [2] from CGAL (www.cgal.org).

Keywords: polynomial resultants, modular algorithm, parallel computations, graphics hardware, GPU, CUDA.

1 Overview

Polynomial resultants play an important role in the quantifier elimination theory. They have a comprehend applied foreground including but not limited to topological study of algebraic curves, curve implitization, geometric modelling, etc. The original modular resultant algorithm was introduced by Collins [3]. It exploits the "divide-conquer-combine" strategy: two polynomials are reduced modulo sufficiently many primes and mapped to homeomorphic images be evaluating them at certain points. Then, a set of univariate resultants is computed independently for each prime, and the result is reconstructed by means of polynomial interpolation and the Chinese Remainder Algorithm (CRA). A number of parallel algorithms have been developed following this idea: those specialized for workstation networks [4] and shared memory machines [5, 6]. In the essence, they differ in how the "combine" stage of the algorithm (polynomial interpolation) is realized. Unfortunately, these algorithms employ polynomial remainder sequences [7] (PRS) to compute univariate resultants. The PRS algorithm, though asymptotically quite fast, is sequential in nature. As a result, the Collins' algorithm in its original form admits only a *coarse-grained* parallelization which is suitable for traditional parallel platforms but not for systems with the massively-threaded architecture like GPUs (Graphics Processing Units).

C.-H. Hsu et al. (Eds.): ICA3PP 2010, Part I, LNCS 6081, pp. 427–440, 2010.

That is why, we have decided to use an alternative approach based on the theory of *displacement structure* [8] to compute univariate resultants. This method reduces the problem to matrix computations which commonly map very well to the GPU's threading model. The displacement structure approach is traditionally applied in a floating-point arithmetic, however, using square-root and division-free modifications [9], we have been able to adapt it to work in a prime field. As of now, the research is carried out to port the remaining algorithm stages (polynomial interpolation and the CRA) to the GPU. Modular computations still constitute a big challenge on the GPU, see [10,11]. Our algorithm uses the fast modular arithmetic developed in [1] which is based on mixing floating-point and integer computations, and is supported by the modified CUDA [12] compiler[1]. This allowed us to benefit from the multiply-add capabilities of the graphics hardware and minimize the number of instructions per modular operation, see Section 4.3.

The rest of the paper is structured as follows. In Section 2 we state the problem in mathematically rigorous way and give an overview of the displacement structure which constitutes the theoretical background for our algorithm. Section 3 surveys the GPU architecture and CUDA programming model. In Section 4 we present the overall algorithm and discuss how it maps to the graphics hardware. Finally, Section 5 provides an experimental comparison of our approach with a host-based algorithm and discusses feature research directions.

2 Problem Statement and Mathematical Background

In this section we define the resultant of two polynomials and give an introduction to the theory of displacement structure which we use to compute univariate resultants.

2.1 Bivariate 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 defined as the determinant of $(p+q)\times(p+q)$ Sylvester matrix S:

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

[1] `http://www.mpi-inf.mpg.de/~emeliyan/cuda-compiler`

Accordingly, the resultant of two monic polynomials f/f_p and g/g_q relates to $res_y(f,g)$ as follows:

$$res_y(f,g) = f_p^q g_q^p \cdot res_y(f/f_p, g/g_q).$$

Note that, the resultant is a polynomial in $\mathbb{Z}[x]$. Using modular and evaluation homomorphisms one can effectively avoid the arithmetic in polynomial domain as discussed in Section 4.

2.2 Displacement Structure and the Generalized Schur Algorithm in Application to Polynomial Resultants

We consider a strongly regular matrix $M \in \mathbb{Z}^{n\times n}$[2]. The matrix M is said to have a *displacement structure* if it satisfies the displacement equation:

$$\Omega M \Delta^T - F M A^T = GJB^T,$$

where $\Omega, \Delta, F, A \in \mathbb{Z}^{n\times n}$ are lower-triangular matrices, $J \in \mathbb{Z}^{r\times r}$ is a signature matrix, G and $B \in \mathbb{Z}^{n\times r}$ are *generator matrices*, such that GJB^T has a *constant* rank $r < n$. Then, r is called a *displacement rank* of M. We refer to [8, 13] on the algorithms for general displacement structure and focus our attention on resultants.

Let $f, g \in \mathbb{Z}[x]$ be two polynomials of degrees p and q respectively, and $S \in \mathbb{Z}^{n\times n}$ be the associated Sylvester matrix ($n = p + q$). The matrix S is structured and has a displacement rank 2. It satisfies the displacement equation: $S - ZSZ^T = GJB^T$, where Z is a down-shift matrix zeroed everywhere except for 1's on its subdiagonal, $J = I \oplus -I \in \mathbb{Z}^{2\times 2}$. Accordingly, $G, B \in \mathbb{Z}^{n\times 2}$ are generators defined as follows:

$$B^T = \begin{bmatrix} f_p \ldots f_{p-q+1} & f_{p-q} & f_{p-q-1} \ldots f_0 & 0 \ldots 0 \\ g_q \ldots g_1 & g_0 - f_p & -f_{p-1} \ldots & -f_1 \end{bmatrix} \quad \begin{array}{c} G \equiv 0 \text{ except for} \\ G_{0,0} = 1,\ G_{q,1} = -1 \end{array}$$

Our goal is to obtain an LDU^T-factorization of the matrix S, where the matrices L and U are lower 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* computes the matrix factorization by iteratively computing the *Schur complements* of leading submatrices. The Schur complement R of a submatrix A in M arises in the course of a block *Gaussian elimination* performed on the rows of matrix M, and is defined as:

$$R = M - CA^{-1}B, \quad \text{where} \quad M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}.$$

The main idea of the algorithm is to operate on low-rank matrix generators instead of the matrix itself giving an asymptotically fast solution. After n iterations the algorithm returns the Schur complement of an $n \times n$ leading submatrix

[2] In other words, a matrix whose leading principal minors are non-singular.

expressed in terms of matrix generators. In each step it brings the generators to a *proper* form. Let (G_i, B_i) denote the generators in step i. A proper form generator $\overline{G}_i$ 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[3] to G_i and B_i respectively:

$$(G_i\Theta_i)^T = \overline{G}_i^T = \begin{bmatrix} \delta^i & a_1^i & a_2^i & \dots \\ 0 & b_1^i & b_2^i & \dots \end{bmatrix} \text{ and } (B_i\Gamma_i)^T = \overline{B}_i^T = \begin{bmatrix} \zeta^i & c_1^i & c_2^i & \dots \\ 0 & d_1^i & d_2^i & \dots \end{bmatrix}.$$

Once the generators are in proper form, it follows from the displacement equation that: $d_{ii} = \delta^i\zeta^i$. The next generator G_{i+1} is obtained from $\overline{G}_i$ by shifting down the column with first non-zero entry while keeping the other column intact (for explanations please refer to [8]):

$$\begin{bmatrix} 0 \\ G_{i+1} \end{bmatrix} = Z\overline{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \text{ where } Z \text{ is a down-shift matrix.}$$

The generator B is processed by analogy. Remark that, the size of generators is *decreased* by one in each step of the algorithm.

2.3 Non-Hermitian Division-Free Rotations

Here the term non-Hermitian means that we apply rotation to a non-symmetric generator pair (G, B). Our task is to find matrices Θ and Γ satisfying: $\begin{bmatrix} a & b \end{bmatrix}\Theta = \begin{bmatrix} \alpha & 0 \end{bmatrix}$, $\begin{bmatrix} c & d \end{bmatrix}\Gamma = \begin{bmatrix} \beta & 0 \end{bmatrix}$ with $\Theta J\Gamma^T = J$. It is easy to check that these equations hold for the following matrices:

$$\Theta = \begin{bmatrix} c & -b/D \\ -d & a/D \end{bmatrix}, \quad \Gamma = \begin{bmatrix} a/D & -d \\ -b/D & c \end{bmatrix}, \quad \text{where} \quad D = ac - bd.$$

Note that, these formulae contain divisions which is undesirable as we are going to apply the algorithm in a finite field. Similar to Givens rotations [9], we use the idea to defer the division until the end of the algorithm by keeping a common denominator for each generator column. In other words, we express the generators in the following way:

$$G^T = \begin{bmatrix} 1/l_a & 0 \\ 0 & 1/l_b \end{bmatrix} \begin{bmatrix} a_0 & a_1 & \dots \\ b_0 & b_1 & \dots \end{bmatrix} \text{ and } B^T = \begin{bmatrix} 1/l_c & 0 \\ 0 & 1/l_d \end{bmatrix} \begin{bmatrix} c_0 & c_1 & \dots \\ d_0 & d_1 & \dots \end{bmatrix},$$

Then, the generator update $(\overline{G}, \overline{B}) = (G\Theta, B\Gamma)$ proceeds as follows:

$$\begin{array}{ll} \overline{a}_i = l_a(a_ic_0 - b_id_0) & \overline{b}_i = l_b(b_ia_0 - a_ib_0) \text{ , where } G = (a_i, b_i), \\ \overline{c}_i = l_c(c_ia_0 - d_ib_0) & \overline{d}_i = l_d(d_ic_0 - c_id_0) \text{ , where } B = (c_i, d_i). \end{array}$$

It can be shown that the denominators are *pairwise* equal, thus, we can keep only two of them. They are updated as follows: $\overline{l}_a = \overline{l}_d = \overline{a}_0$, $\overline{l}_c = \overline{l}_b = l_a l_c^2$.

[3] Such matrices must satisfy: $\Theta J\Gamma^T = J$ to ensure that the displacement equation holds after transformation. In other words, we get: $G\Theta J(B\Gamma)^T = GJB^T$.

Apparently, the denominators must be non-zero to prevent the algorithm from the failure. This is guaranteed by the *strong-regularity* assumption introduced in the beginning. However, this is not always the case for Sylvester matrix. In Section 4.4 we discuss how to deal with this problem.

3 GPU Architecture and CUDA Framework

In this section we consider GPUs with NVIDIA Tesla architecture. The GPU comprises a set of Streaming Multiprocessor (SMs) which can execute vertex and fragment shaders as well as general purpose parallel programs. As an example, the GTX 280 contains 30 SMs. The GPU execution model is known as *single-instruction multiple-thread* or SIMT. It means that the SM applies an instruction to a group of 32 threads called *warps* which are always executed synchronously. If thread code paths within a warp diverge, the SM executes all taken paths serially. Different warps can execute disjoint paths without penalties.

On the top level, threads are grouped in a programmer defined *grid* of *thread blocks*. Such a model creates potentially unlimited parallel resources exploited dynamically by the target hardware. A thread block can contain up to 512 threads which can communicate using fast on-chip shared memory and synchronization barriers. The code running on the GPU is referred to as a *kernel* which is launched on a grid of thread blocks. Different blocks run completely independent from each other: data movement between thread blocks can be realized by splitting a program in two or more kernel launches[4].

CUDA memory model is built on five memory spaces. Each thread has a statically allocated fast local storage called *register file*. Registers is a scarce resource and should be used carefully to prevent spilling. The SM has a fixed amount of per-block on-chip *shared memory* (16 Kb). Shared memory is divided in 16 banks to facilitate concurrent access. The GPU has two cached memory spaces – read-only *constant* and *texture* memory – that are visible to all thread blocks and have a lifetime of an application. The remaining read-write *global memory* is also visible to the entire grid but is not cached on the device. It is crucial to stick to *memory coalescing* patterns in order to use the global memory bandwidth effectively.

4 Mapping Resultants Algorithm to Graphics Hardware

In this section we consider the algorithm step-by-step. We start with a high-level overview, then consider computation of univariate resultants and 24-bit modular arithmetic. Finally, we discuss the main implementation details and outline some ideas about the polynomial interpolation on the GPU.

4.1 Algorithm Overview

Our approach follows the "divide-conquer-combine" strategy of Collins' modular algorithm. At the beginning, the input polynomials are mapped to a prime field

[4] Block independence guarantees that a binary program will run unchanged on the hardware with any number of SMs.

for sufficiently many primes. The number of primes depends on the height of the resultant coefficients which is given by Hadamard's bound, see [14]. For each prime m_i we compute resultants at $x = \alpha_0, x = \alpha_1, \ldots \in \mathbb{Z}_{m_i}$. The degree bound (the number of evaluation points α_k) can be devised from the entries of Sylvester matrix [14]. The resultant $r \in \mathbb{Z}[x]$ is reconstructed from modular images using polynomial interpolation and the CRA. The first part of the algorithm is run on the GPU: we launch a kernel on a 2D grid $N \times S$,[5], where one thread block evaluates polynomials and computes one univariate resultant. The univariate resultant algorithm will be discussed in Section 4.2.

In order for the algorithm to work properly, we need to handle "bad" primes and evaluation points adequately. For two polynomials $f, g \in \mathbb{Z}[x, y]$ as defined in Section 2.1, *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 $g_q(\alpha) \equiv 0 \bmod m$.* "Bad" primes can be discarded quite easily: we do this during the initial modular reduction of polynomial coefficients prior to the grid launch. To deal with "bad" evaluation points we *enlarge* the grid by a small amount of excessive points (1–2%) such that, if for some points the algorithm fails, we still have enough information to reconstruct the result. The same technique is used to deal with *non-strongly regular* Sylvester matrices, see Section 2.3. In fact, non-strong regularity corresponds to the case where polynomial coefficients are related via some non-trivial equation which occurs rarely in practise and is confirmed by our tests (see Section 5). Indeed, if for some α_k Sylvester matrix is ill-conditioned, instead of using intricate methods, we simply ignore the result and take another evaluation point. In a very "unlucky" case when we cannot reconstruct the resultant due to the lack of points, we launch another grid to compute extra information.

It is worth mentioning, that the another interesting approach to compute polynomial resultants is given in [15]. It is based on modular arithmetic and linear recurring sequences. Although, this algorithm seemingly has a connection to the PRS, it is yet unclear whether it can be a good candidate for realization on the GPU.

4.2 Univariate Resultant Algorithm

The resultant $res(f, g) \bmod m_i$ at $x = \alpha_j$ is computed using the method explained in Section 2.2. In each iteration the algorithm multiplies the generators by rotation matrices collecting one factor of the resultant per iteration. Then, the generator columns are shifted down to obtain the generators for the next iteration. After $n = p + q$ iterations (notations are as in Section 2.2), the product of the factors yields the resultant.

The original algorithm can largely be improved. First, we write the generators as pairs of column vectors: $G = (a, b)$, $B = (c, d)$. Now, remark that, at the beginning $G \equiv 0$ except for two entries: $a_0 = 1,\ b_q = -1$. If we run the algorithm on monic polynomials[6], we can observe that the vectors a, b and c stay *constant*

[5] Here N denotes the number of moduli, and S – the number of evaluation points.

[6] In other words, on polynomials with unit leading coefficients.

during the first q iterations of the algorithm (except for a single entry a_q). Indeed, because polynomials are monic, c_0 and d_0 are initially ones, and so is the denominator of the rotation matrices (see Section 2.3): $D = a_0c_0 - b_0d_0 = c_0 \equiv 1$. Thus, we can get rid of the denominators completely which greatly simplifies the rotation formulae. Moreover, the first q factors of the resultant returned by the algorithm are *unit*, therefore we can skip them. However, we need to multiply the resultant by $f_p^q g_q^p$ as to compensate for running the algorithm on monic polynomials. The pseudocode is given below:

```
 1: procedure RESULTANT_UNIVARIATE(f : Polynomial, g : Polynomial)
 2:     p = degree(f),  q = degree(g),  n = p + q
 3:     f ← f/f_p,  g ← g/g_q                          ▷ convert polynomials to monic form
 4:     G = (a, b),  B = (c, d)                ▷ set up generators: see Section 2.2 for details
 5:     for j = 0 to q − 1 do                              ▷ first q iterations are simplified
 6:         d_i ← d_i − c_i d_j for ∀i = j + 1 ... n − 1          ▷ multiply by the rotation matrix
 7:         a_q = d_j                                                ▷ update a single entry of a
 8:         a_{i+1} ← a_i, c_{i+1} ← c_i for ∀i = j + 1 ... n − 2          ▷ shift down the generators
 9:     end for
10:     l_a = 1,  l_c = 1,  res = 1,  l_res = 1          ▷ denominators and resultant are set to 1
11:     for j = q to n − 1 do
12:         for i = j to n − 1 do              ▷ multiply the generators by rotation matrices
13:             s = l_a(a_i c_j − b_i d_j),  b_i = l_c(b_i a_j − a_i b_j),  a_i = s
14:             t = l_c(c_i a_j − d_i b_j),  d_i = l_a(d_i c_j − c_i d_j),  c_i = t
15:         end for
16:         l_c = l_a l_c^2,  l_a = a_j, res = res · c_j,  l_res = l_res · l_c      ▷ update the denominators
17:         a_{i+1} ← a_i, c_{i+1} ← c_i for ∀i = j ... n − 2                ▷ shift down the generators
18:     end for
19:     return res · f_p^q · g_q^p / l_res                                  ▷ return the resultant
20: end procedure
```

We will refer to iterations $j = 0 \dots q-1$ and $j = q \dots n-1$ as type S and T iterations respectively. For division in lines 3 and 19 we use the modified Montgomery modular inverse [16] with improvements from [17]. The number of iterations of this algorithm is bounded by moduli bitlength (24 bits), see Appendix A.

4.3 24-Bit Modular Arithmetic on the GPU

Modular multiplication is a challenging problem due to the limited hardware support for integer arithmetic. The GPU natively supports only 24-bit integer multiplication realized by mul24.lo and mul24.hi instructions[7]. However, the latter instruction is *not* exposed by CUDA API. To overcome this limitation, the authors of [10] propose to use slow 32-bit multiplication, while the tests from [11] show that 12-bit arithmetic is faster because modular reduction can be done in floating-point without overflow concerns.

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

We use the arithmetic based on mixing floating-point and integer computations [1] which is supported by the patched CUDA compiler[8]. In what follows, we will refer to umul24 and umul24hi as intrinsics for mul24.lo and mul24.hi respectively. The procedure MUL_MOD in Algorithm 1 computes $a \cdot b \bmod m$ for two 24-bit residues. The idea is to split the product as follows: $a \cdot b = 2^{16}hi + lo$ (32 and 16 bits), and then use a congruence ($0 \leq \lambda < m$):

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

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

The next procedure SUB_MUL_MOD is an extended version of MUL_MOD which is used to implement matrix rotations: it evaluates $(x_1y_1 - x_2y_2) \bmod m$ (see Section 2.3). The algorithm computes the products x_1y_1 and x_2y_2, and subtracts partially reduced residues. Adding $m \cdot 100$ in line 14 is necessary to keep the intermediate result positive since umul24 operation in line 16 cannot handle negative operands. In total, line 14 is compiled in 4 multiply-add (MAD) instructions[9]. The remaining part is an inlined REDUCE_MOD operation (see [1]) with a minor change. Namely, in line 15 we use the mantissa trick [18] to multiply by $1/m$ and round the result down using a single MAD instruction.

4.4 Putting It All Together

Having all the ingredients at hand, we can now discuss how the algorithm maps to the GPU. The GPU part of the algorithm is realized by two kernels, see Figure 1 (a). The first kernel calculates modular resultants, while the second one eliminates zero denominators from the input sequence and multiplies resultants by respective modular inverses $\mathsf{l}_{\mathsf{res}}^{-1}$. Grid configuration for each kernel launch is shown to the left.

The number of threads per block for the **first kernel** depends on the maximal y-degree of polynomials being processed. We will use the notation: $p = deg_y(f)$, $q = deg_y(g)$, where $f, g \in \mathbb{Z}[x, y]$ and $p \geq q$. We provide three kernel instantiations for different polynomial degrees: kernel **A** with 64 threads per block for $p \in [32, 63]$; **B** – 96 threads for $p \in [64, 95]$; and **C** – 128 threads for $p \in [96, 127]$. One reason behind this configuration is that we use $p + 1$ threads to evaluate coefficients of f at $x = \alpha_i$ in parallel using Horner form (the same for g). The resultant algorithm consists of one outer loop split up in iterations of type S and T, see Section 4.1. The inner loop of the algorithm is completely vectorized: this is another reason why we need the number of threads to match the polynomial degree. Remark that, in each iteration the size of the generators decreases by

[8] `http://www.mpi-inf.mpg.de/~emeliyan/cuda-compiler`

[9] The graphics hardware supports 24-bit integer as well as floating-point MADs. The compiler aggressively optimizes subsequent multiply and adds to use MAD instructions.

Algorithm 1. 24-bit modular arithmetic on the GPU

```
 1: procedure MUL_MOD(a, b, m, invm)          ▷ invm = 2^16/m (in floating-point)
 2:     hi = umul24hi(a, b)                          ▷ high 32 bits of the product
 3:     prodf = fmul_rn(hi, invm)                    ▷ multiply in floating-point
 4:     l = float2uint_rz(prodf)                     ▷ integer truncation: l = ⌊hi · 2^16/m⌋
 5:     r = umul24(a, b) − umul24(l, m)      ▷ now r ∈ [−2m + ε; m + ε] with 0 ≤ ε < m
 6:     if r < 0 then r = r + umul24(m, 0x1000002) fi    ▷ multiply-add: r = r + m · 2
 7:     return umin(r, r − m)                            ▷ return r = a · b mod m
 8: end procedure
 9: procedure SUB_MUL_MOD(x1, y1, x2, y2, m, invm1, invm2)
10:     h1 = umul24hi(x1, y1), h2 = umul24hi(x2, y2)          ▷ two inlined MUL_MOD's
11:     pf1 = fmul_rn(h1, invm1), pf2 = fmul_rn(h2, invm1)        ▷ invm1 = 2^16/m
12:     l1 = float2uint_rz(pf1), l2 = float2uint_rz(pf2)
13:             ▷ compute an intermediate product r, mc = m · 100:
14:     r = mc + umul24(x1, y1) − umul24(l1, m) − umul24(x2, y2) + umul24(l2, m)
15:     rf = uint2float_rn(r) ∗ invm2 + e23      ▷ invm2 = 1/m, e23 = 2^23, rf = ⌊r/m⌋
16:     r = r − umul24(float_as_int(rf), m)                ▷ r = r − ⌊r/m⌋ · m
17:     return (r < 0 ? r + m : r)
18: end procedure
```

1, and so is the number of working threads, see Figure 1 (b). To achieve higher thread occupancy, we unroll the type S iterations by the factor of 2. 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. We keep the column vectors a and c of the generators $G = (a, b)$ and $B = (c, d)$ in shared memory because they need to be shifted down in each iteration. The vectors b and d are located in register space. Accordingly, each iteration (of type S or T) consists of fetching current first rows of G and B (these are shared by all threads), transforming the generators using SUB_MUL_MOD operation, saving computed factors in shared memory (only for type T iterations), and shifting down the columns a and c, see Figure 1 (b). Also, during type T iterations we keep track of the size of generators and switch to iterations *without* sync on crossing the *warp* boundary[10]. The final result is given by the product $f_p^q g_q^p \cdot \prod_i d_{ii}$ (see Section 2.2). We compute this product efficiently using "warp-sized" reductions [19]. The idea is to run prefix sums for different warps separately omitting synchronization barriers, and then combine the results in a final reduction step, see Figure 1 (c). The **second kernel**, launched with 128 threads, runs stream compaction for each modulus in parallel, and then computes modular inverses for the remaining entries. Stream compaction algorithm is also based on "warp-sized" reductions[11].

[10] Warp, as a minimal scheduling entity, is always executed synchronously, hence, shared memory access not need to be synchronized.

[11] Stream compaction can be regarded to as an exclusive prefix sum of 0's and 1's where 0's correspond to elements being eliminated.

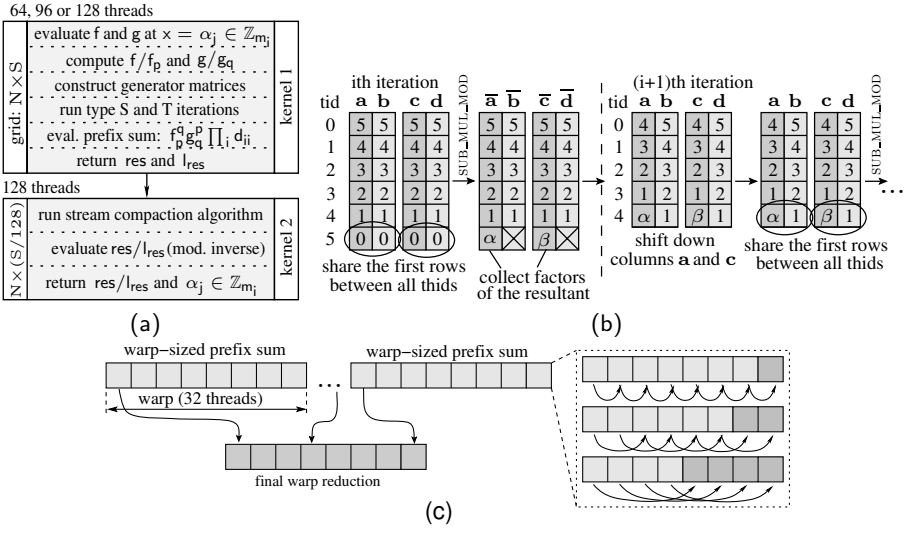

Fig. 1. (a) GPU part of the algorithm consisting of two kernel launches (N – number of moduli, S – number of eval. points); (b) vector updates during the type T iterations (**tid** denotes the thread ID); (c) warp-sized reduction (prefix sum)

4.5 Polynomial Interpolation

Now we sketch some ideas on how to realize polynomial interpolation efficiently on the GPU. The task of interpolation is to find a polynomial $f(x)$, $deg(f) \leq n$, satisfying the following set of equations: $f(\alpha_i) = y_i$, for $0 \leq i \leq n$. The polynomial coefficients a_i are given by the solution of an $(n+1) \times (n+1)$ *Vandermonde* system:

$$V\mathbf{a} = \mathbf{y}, \text{ where } V \text{ is a Vandermonde matrix: } V_{ij} = \alpha_i^j \ (i, j = 0, \ldots, n).$$

Vandermonde matrix is structured and has a displacement rank 1. Thus, we can adapt the generalized Schur algorithm to solve the linear system in a small parallel time. Namely, if we apply the algorithm to the following matrix embedding:

$$M = \begin{bmatrix} V & -\mathbf{y} \\ I & \mathbf{0} \end{bmatrix},$$

then after $n+1$ steps we obtain the Schur complement R of a submatrix V which is equal to: $R = 0 - IV^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$, i.e., the solution of a Vandermonde system.

5 Experiments and Conclusion

We have tested our algorithm on the *GeForce GTX 280* graphics processor. As a reference implementation we have use the resultant algorithm [2] from CGAL[12]

[12] `www.cgal.org`

Table 1. Timings. *First column.* p and q: polynomial y-degrees; **points:** # of evaluation points; **moduli:** # of 24-bit moduli; *Second column.* **eval:** polynomial evaluation; **res:** univariate resultants; **interp:** polynomial interpolation; **CRA:** Chinese remaindering; **ERR:** # of wrong entries computed

parameters setup	CPU timing breakdown	CPU eval + resultant	GPU eval + resultant	ratio	ERR
$p:32,\ q:32$ points: 970 moduli: 96	eval: 29.5 s, res: 25.4 s interp: 27.6 s, CRA: 0.63 s total: 83.8 s	54.9 s	131.6 ms	**417x**	1
$p:50,\ q:43$ points: 940 moduli: 101	eval: 26.1 s, res: 47.0 s interp: 26.9 s, CRA: 0.69 s total: 101.0 s	73.1 s	175.7 ms	**415x**	0
$p:63,\ q:63$ points: 1273 moduli: 136	eval: 55.1 s, res: 148.4 s interp: 65.7 s, CRA: 1.32 s total: 271.1 s	203.5 s	393.5 ms	**517x**	3
$p:70,\ q:64$ points: 1354 moduli: 102	eval: 48.7 s, res: 129.2 s interp: 57.2 s, CRA: 1.0 s total: 236.1 s	177.9 s	492.6 ms	**361x**	4
$p:95,\ q:95$ points: 1152 moduli: 145	eval: 45.0 s, res: 292.5 s interp: 57.8 s, CRA: 1.3 s total: 397.1 s	337.5 s	752.5 ms	**448x**	2
$p:120,\ q:99$ points: 1549 moduli: 130	eval: 71.3 s, res: 461.2 s interp: 93.3 s, CRA: 1.53 s total: 627.9 s	532.5 s	1363.6 ms	**390x**	3

(Computational Geometry Algorithms Library) run on the 2.5Ghz *Quad-Core Intel Xeon E5420* with 12MB L2 cache and 8Gb RAM under 32-bit Linux platform. The code has been compiled with '*–DNDEBUG –O3 –march=core2*' options. We have benchmarked the first two stages of the host-based algorithm (evaluate + resultant) and compared them with our realization. Table 1 summarizes the running time for different configurations. The GPU timing includes the time for GPU–host data transfer for objective comparison. The number of evaluation points has been increased by 2% to accommodate "unlucky" primes. The total number of evaluation points for which the algorithm fails is given by the column **ERR** in the table. The tests confirm that, indeed, this occurs rarely on the average. Observe that, the maximal speed-up is attained for $p = \{63, 95\}$ (see Table 1): this is no surprise as these parameters correspond to the full thread occupancy. In total, one can see that our algorithm outperforms the CPU implementation by a large factor. Moreover, due to the vast amount of blocks executed[13], the algorithm achieves a full utilization of the GTX280 graphics card and we expect the performance to scale well on forthcoming GPUs.

The left graph in Figure 2 examines the performance depending on the polynomials y-degree (which are chosen to be equal) with the number of moduli and evaluation points fixed to 128 and 1000 respectively. One can see that the

[13] Recall that, the grid size equals to 'number of moduli' × 'number of points'.

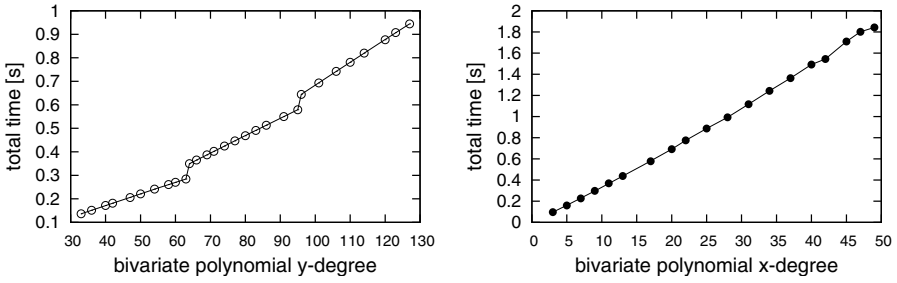

Fig. 2. The running time as a function of the polynomials' y-degree (left) and x-degree (right)

performance scales linearly with the degree. This is an expected behavior because the algorithm consists of one outer loop (while the inner loop is vectorized). Performance degradation at the warp boundary (64 and 96) is due to switching to a larger kernel once all thread resources are exhausted. It might be possible to smooth "stairs" by dynamically balancing the thread workload. The second graph in Figure 2 evaluates how the running time grows with the x-degree (and y-degree fixed). Linear dependency is because of the fact that the x-degree only causes the number of evaluation points (one grid dimension) to increase while the number of moduli remains the same.

To conclude, we have identified that with the approach of displacement structure we can harness the power of GPUs to compute polynomial resultants. Our algorithm has achieved a considerable speed-up which was previously beyond the reach of traditional serial algorithms. Certainly, this is only the first step in realization of a complete and robust resultant algorithm on graphics hardware. From Table 1 one can see that polynomial interpolation could be quite expensive. Nevertheless, our benchmarks clearly show that graphics processors have a great performance potential in such a not yet well-explored application domain. Moreover, with the ideas from Section 4.4, we are currently underway to realize polynomial interpolation on the GPU.

References

1. Emeliyanenko, P.: Efficient multiplication of polynomials on graphics hardware. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 134–149. Springer, Heidelberg (2009)
2. Hemmer, M.: Polynomials, CGAL - Computational Geometry Algorithms Library, release 3.4. CGAL, Campus E1 4, 66123 Saarbrücken, Germany (January 2009)
3. Collins, G.E.: The calculation of multivariate polynomial resultants. In: SYMSAC 1971, pp. 212–222. ACM, New York (1971)
4. Bubeck, T., Hiller, M., Küchlin, W., Rosenstiel, W.: Distributed Symbolic Computation with DTS. In: IRREGULAR 1995, pp. 231–248. Springer, London (1995)
5. Schreiner, W.: Developing a distributed system for algebraic geometry. In: EUROCM-PAR 1999, pp. 137–146. Civil-Comp Press (1999)

6. Hong, H., Loidl, H.W.: Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In: Buchberger, B., Volkert, J. (eds.) CONPAR 1994 and VAPP 1994. LNCS, vol. 854, pp. 325–336. Springer, Heidelberg (1994)
7. Geddes, K., Czapor, S., Labahn, G.: Algorithms for computer algebra. Kluwer Academic Publishers, Dordrecht (1992)
8. Kailath, T., Ali, S.: Displacement structure: theory and applications. SIAM Review 37, 297–386 (1995)
9. Frantzeskakis, E., Liu, K.: A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing. IEEE Transactions on Signal Processing 42, 2455–2469 (1994)
10. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
11. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 350–367. Springer, Heidelberg (2009)
12. NVIDIA: CUDA Compute Unified Device Architecture. NVIDIA Corp. (2007)
13. Chandrasekaran, S., Sayed, A.H.: A Fast Stable Solver for Nonsymmetric Toeplitz and Quasi-Toeplitz Systems of Linear Equations. SIAM J. Matrix Anal. Appl. 19, 107–139 (1998)
14. Monagan, M.: Probabilistic algorithms for computing resultants. In: ISSAC 2005, pp. 245–252. ACM, New York (2005)
15. Llovet, J., Martínez, R., Jaén, J.A.: Linear recurring sequences for computing the resultant of multivariate polynomials. J. Comput. Appl. Math. 49(1-3), 145–152 (1993)
16. de Dormale, G., Bulens, P., Quisquater, J.J.: An improved Montgomery modular inversion targeted for efficient implementation on FPGA. In: IEEE International Conference on FPT 2004, pp. 441–444 (2004)
17. Savas, E., Koc, C.: The montgomery modular inverse-revisited. IEEE Transactions on Computers 49(7), 763–766 (2000)
18. Hecker, C.: Let's get to the (floating) point. Game Developer Magazine, 19–24 (1996)
19. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. Commun. ACM 29, 1170–1183 (1986)

A Montgomery Modular Inverse Algorithm

To realize efficient Montgomery modular inverse on the GPU, we have combined the algorithm from [16] with ideas from [17], and took advantage of the fast 24-bit integer multiplication supported by the GPU. The algorithm comprises two stages. In the first stage we iteratively compute $x^{-1}2^k \bmod m$[14], where $s \leq k \leq 2s$ and $s = \lceil \log_2 m \rceil$. Then, we run two Montgomery multiplications by the powers of two to get rid of 2^k factor. The pseudocode is given below:

```
1: procedure MONTGOMERY_INVERSE(x, m, mu)                    ▷ computes x^{-1} mod m
2:     v = x, u = m, s = 1, r = 0, k = 0                     ▷ x is a 24-bit residue modulo m
3:     repeat                          ▷ first stage: compute r = x^{-1}2^k mod m iteratively
4:         tmprs = r
5:         if v mod 2 = 1 then
6:             safeuv = v
7:             if (v xor u) < 0 then v = v + u else v = v − u fi
8:             if (v xor safeuv) < 0 then u = safeuv, tmprs = s fi
9:             s = s + r
10:        fi
11:        v = v/2, r = tmprs · 2, k = k + 1
12:    until v ≠ 0
13:    r = m − r                                  ▷ second stage: get rid of 2^k factor
14:    if r < 0 then r = r + m fi                 ▷ r = x^{-1}2^k mod m, 24 ≤ k ≤ 48
15:    if k > 24 then          ▷ first multiply: r = (x^{-1}2^k)(2^{-m}) = x^{-1}2^{k-m} (mod m)
16:        c = umul24(r, mu)                                  ▷ mu = −m^{-1} mod 2^24
17:        lo = umul24(c, m), hi = __umul24hi(c, m)           ▷ (hi, lo) = c · m (48 bits)
18:        lo = umul24(lo, 0x1000001) + r                     ▷ lo = (lo mod 2^24) + r
19:        r = hi/2^8 + lo/2^24, k = k − 24                   ▷ r = (lo, hi)/2^24
20:    fi
21:            ▷ second Montgomery multiply: r = (x^{-1}2^k)(2^{m-k})(2^{-m}) = x^{-1} (mod m)
22:    c = r · 2^{24-k}, d = umul24(c, mu)                    ▷ mu = −m^{-1} mod 2^24
23:    lo = umul24(d, m), hi = umul24hi(d, m)                 ▷ (hi, lo) = d · m (48 bits)
24:    d = r/2^k, lo = lo mod 2^24
25:    lo = umul24(c, 0x1000001) + lo                         ▷ lo = (c mod 2^24) + r
26:    r = hi/2^8 + d + lo/2^24
27:    return r
28: end procedure
```

[14] The number of iterations is bounded by moduli bit-length (24 bits).