

Solving Bivariate Polynomial Systems on a GPU

Marc Moreno Maza and Wei Pan

University of Western Ontario and Intel Corporation

E-mail: moreno@csd.uwo.ca; wei.pan@intel.com

Abstract. We present a CUDA implementation of dense multivariate polynomial arithmetic based on Fast Fourier Transforms over finite fields. Our core routine computes on the device (GPU) the subresultant chain of two polynomials with respect to a given variable. This subresultant chain is encoded by values on a FFT grid and is manipulated from the host (CPU) in higher-level procedures.

We have realized a bivariate polynomial system solver supported by our GPU code. Our experimental results (including detailed profiling information and benchmarks against a serial polynomial system solver implementing the same algorithm) demonstrate that our strategy is well suited for GPU implementation and provides large speedup factors with respect to pure CPU code.

1. Introduction

Graphics processing units (GPUs) are becoming very attractive for scientific computing applications that are targeting high performance. Many application software have gained massive speedup factors by integrating GPU support, in areas like Monte Carlo simulation, weather forecasting, climate research, molecular modeling, quantum mechanical physics and astrophysics. See for instance the research areas in the website of GPGPU.org [2] or the application section in the Wikipedia page for GPGPU [1], where many references and links are provided. Symbolic computation is also entering the area of many-core computing, but few reports have been published so far [4, 8, 10, 17].

In this paper, and up to our knowledge, we report on the first GPU implementation of a polynomial system solver over finite fields. A GPU implementation of a bivariate solver over the real numbers is reported in [3]. Solving polynomial systems is a driving subject in symbolic computation with many successful results from theoretical to practical aspects. Adapting this knowledge and acquired experience is, however, very challenging.

The difficulty starts at the level of dense multivariate polynomial arithmetic. For instance, techniques that have appeared to be very effective for multicore implementation [18, 19] do not apply to GPU implementation and had to be revisited [17].

Efficient multicore implementation relies on a good usage of the different levels of cache memories (L1, L2, L3) and in minimizing the impact of performance bottlenecks due to phenomena like true sharing, false sharing, memory contention, etc. On the other hand, concurrency platforms (such as OpenMP, Intel Cilk Plus, KAAPI) for multicore architectures are generally based on the *fork-join parallelism model* and provide tools for automatic scheduling; this helps programmers focusing on exposing parallelism for their targeted applications.

The GPU programming model is based on *data-parallelism* and a large part of the scheduling is done statically by the programmer through the decomposition of the application into GPU kernels. Efficient GPU implementation relies on optimizing memory usage for maximum bandwidth, maximizing occupancy of the device in order to hide memory transfer latency and optimizing instruction usage for maximum throughput.

The motivation of our work is to support polynomial system solvers based on the notion of a *regular chains* [11] where the core algorithms [16] often rely on the theory of polynomial subresultants [9]. In [12], we have shown that the dominant cost of those algorithms can be essentially reduced to that of subresultant chain computations.

In Section 2, following an idea proposed by Collins in [6], we explain how we compute a subresultant chain by evaluation-interpolation, based on FFT techniques. Once we obtain a so-called FFT grid on which the input polynomials are sampled, we employ the Brown’s Algorithm to compute subresultant chains of univariate polynomials. All these computations are performed on the device for which we have designed a specific implementation of Brown’s Algorithm, see Section 3.

This sampled subresultant chain can then be exploited by a high-level algorithm running on the host. A simple case is the resultant computation described in Section 3. A more advanced one is that of the bivariate solver algorithm of [13] that we report in Section 4. Actually, we implemented this algorithm with and without GPU code support. Our pure CPU implementation is serial C code. The specifications of the GPU cards used in our experimentation are listed in Appendix A while profiling information for our GPU kernels appear in Appendix C.

Our experimental results show that, for resultant computation our GPU-supported code outperform our pure CPU code by a speedup factor of 34 (resp. 69) on sufficiently large input bivariate (resp. trivariate) polynomials. For the largest examples we tested our GPU-supported bivariate solver outperform its CPU counterpart by a factor of 7.5. For this latter experimentation, we should insist on the fact that a significant part of the computation (univariate polynomial GCDs) are still performed by CPU code. Removing this bottleneck on the critical path of the whole application is work in progress and could be an opportunity to use two GPU cards concurrently.

One may ask whether the algorithms implemented in this work could also lead to a successful multicore implementation. We actually tried and the answer is no for sufficiently large input data. Indeed, our most powerful GPU card can efficiently handle a sampled subresultant chain with a size in the order of 1 GB. Since the construction of the sampled subresultant chain essentially consists of many successive traversals of this data structure, a multicore implementation will suffer from high rate of cache misses due to the fact that L2 to L3 caches are today in the order of several MB.

2. Subresultant computations

This section is devoted to a high level description of polynomial subresultant computation, by means of an FFT-based modular algorithm. Let \mathbb{K} be a field and $n \geq 1$. In the sequel, we consider $P, Q \in \mathbb{K}[x_1, \dots, x_{n+1}]$ two non-constant polynomials with the same main variable $y := x_{n+1}$. We define $p := \deg(P, y) \geq q := \deg(Q, y)$. We denote by \mathbb{B} the ring $\mathbb{K}[x_1, \dots, x_n]$ and by S_j the j -th subresultant of P, Q in $\mathbb{B}[y]$, for $0 \leq j < q$.

FFT grid. Given n positive integers m_1, \dots, m_n , we call the following finite set Θ a *grid* in \mathbb{K}^n

$$\begin{aligned} \Theta &= \Theta_1 \times \Theta_2 \times \dots \times \Theta_n \\ &= \{(u_1, \dots, u_n) \mid u_i \in \Theta_i \text{ for each } i\} \end{aligned} \tag{1}$$

where Θ_i is a finite subset of \mathbb{K} with size m_i , for each i . We say that the grid Θ is *valid* for a polynomial $f \in \mathbb{K}[x_1, \dots, x_n][y]$, if the leading coefficient of f in y does not vanish at any point of Θ . Assume furthermore that, for each i , the integer m_i is a power of 2 and that there exists $\omega_i \in \mathbb{K}$ which is a m_i -th primitive root of unity. If Θ_i is chosen as follows:

$$\Theta_i = \{\omega_i^j \mid j = 0 \cdots m_i - 1\},$$

then we say that Θ a *FFT grid* in \mathbb{K}^n . The *format* of Θ is (m_1, \dots, m_n) and its *size* is the product $m_1 \cdots m_n$. The *evaluation-interpolation method* for computing subresultants S_j , for $0 \leq j < q$, proposed by Collins in [6], proceeds as follows.

- (S₁) Compute an upper bound for the degree $\deg(S_0, x_i)$ and set m_i to it, for $i = 1 \cdots n$.
- (S₂) Construct a grid Θ of format (m_1, \dots, m_n) , valid for both P and Q .
- (S₃) Evaluation: compute $P(u, y)$ and $Q(u, y)$ for each $u \in \Theta$.
- (S₄) For all $u \in \Theta$ and all $0 \leq j < q$, compute the subresultant $S_j(u)$ of $P(u, y)$, $Q(u, y)$ in y .
- (S₅) Interpolation: for each $0 \leq j < q$, construct S_j from the image set $\{S_j(u) \mid u \in \Theta\}$.

For Step (S₁), a well-known degree bound can be derived from the Sylvester matrix of P and Q , see the paper of Monagan [15] for detailed discussions. For Step (S₂), let h be the product of the leading coefficients of P and Q in y , which is a nonzero polynomial in $\mathbb{K}[x_1, \dots, x_n]$. Constructing a valid grid for h deterministically is difficult. In practice, one can generate a grid at random and check whether the grid is valid or not. Step (S₄) is equivalent to computing the subresultant chains of m univariate polynomial pairs. A standard tool is Brown's subresultant algorithm [5].

Step (S₃) and Step (S₅) are instances of the so-called *multipoint evaluation* and *multipoint interpolation* problems, respectively. In general, these operations can be performed by means of subproduct tree techniques [9]. We do not analyze this point of view further since our focus is on FFT grids. If Θ is a FFT grid, then FFT-based multipoint evaluation and interpolation run in $O(m \log(m))$ operations in \mathbb{K} .

The approach used in our implementation is summarized by Figure 1, where all the computations are converted into the bivariate case, by means of Kronecker's substitutions. This turns multivariate FFT computations into univariate ones, which simplifies both the analysis and the implementation. In practice, it is wiser to conduct large univariate FFTs (or large bivariate FFTs, see [18] for details) rather than multivariate FFTs with small sizes along one or more dimensions. Another feature of our implementation is the use of random translations of the form $\phi_a : x \mapsto x + a$ (applied to the polynomials $F', G' \in \mathbb{Z}_c[x, y]$, see Figure 1) in order to find a valid FFT grid. Details, including a probability analysis can be found in [20].

3. Brown's algorithm on the GPU

One of the most important algebraic tools for solving polynomial system symbolically is the polynomial subresultants. We refer to [7, 21] for the theory of polynomial subresultants. In this section, after briefly reviewing some properties of polynomial subresultants, we present our CUDA FFT-based implementation of Brown's subresultant algorithm for bivariate (or trivariate) polynomials over a prime field $\mathbb{K} = \mathbb{Z}_c$ of characteristic $c > 2$.

Let \mathbb{D} be an integral domain and let $P, Q \in \mathbb{D}[y]$ be non-constant polynomials of degree p, q respectively. We assume $p \geq q$. The polynomials computed by the subresultant algorithm form a sequence, called the *subresultant chain* and denoted by $S = \text{src}(P, Q, y)$. This sequence consists of $q + 1$ polynomials (S_q, \dots, S_0) in $\mathbb{D}[y]$, starting at $S_q = \text{lc}(Q)^{p-q}Q$ and ending at the resultant $S_0 = \text{res}(P, Q, y)$. The polynomial S_j is called the subresultant of index j . For each index $0 \leq j \leq q$, if S_j is not zero, its degree is at most j and S_j is said regular if $\deg(S_j, y) = j$. Hence, S_q is a regular subresultant by definition. See [12] for a complete and formal definition of a subresultant chain.

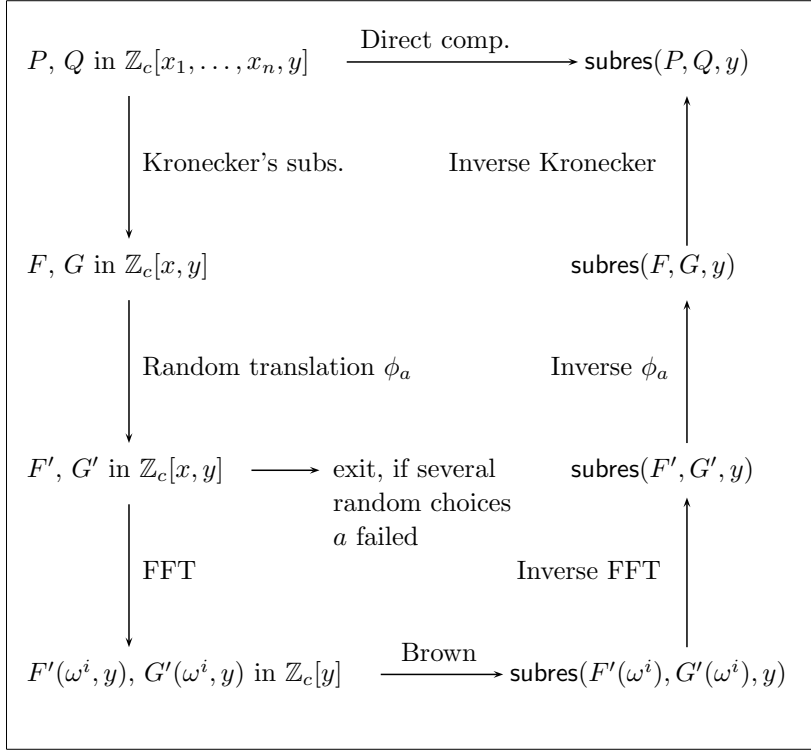


Figure 1. Compute the subresultant chain via an FFT based modular algorithm

3.1. Brown's algorithm

Algorithm 1: Brown's subresultant algorithm

Input : polynomials $F, G \in \mathbb{K}[x]$ such that $\deg(F) \geq \deg(G) > 0$

Output : the subresultant chain of F and G

- 1 $S_i \leftarrow 0$ for $0 \leq i < \deg(G)$;
 - 2 $B \leftarrow \text{prem}(F, -G, x)$, $A \leftarrow G$, $\alpha \leftarrow \deg(F) - \deg(G)$;
 - 3 **while** $B \neq 0$ **do**
 - 4 $d \leftarrow \deg(A)$, $e \leftarrow \deg(B)$, $\delta \leftarrow d - e$;
 - 5 $S_{d-1} \leftarrow B$;
 - 6 $S_e \leftarrow \text{lc}(A)^{\alpha(1-\delta)} \text{lc}(B)^{\delta-1} B$;
 - 7 **if** $e = 0$ **then break**;
 - 8 $B \leftarrow \text{lc}(A)^{-\alpha\delta-1} \text{prem}(A, -B, x)$, $A \leftarrow S_e$, $\alpha \leftarrow 1$;
 - 9 **return** S_i for $0 \leq i < \deg(G)$;
-

The main subroutine involved in Brown's Algorithm is *pseudo-remainder* computation. Recall that given $f, g \in \mathbb{K}[x]$, the *pseudo-division with remainder* of f by g computes $q, r \in \mathbb{K}[x]$ with

$$\text{lc}(g)^{1+\deg(f)-\deg(g)} f = qg + r, \quad \deg(r) < \deg(g), \quad (2)$$

assuming $g \neq 0$. The polynomials q and r are uniquely determined by Equation (2) and computed by Algorithm 2. These polynomials are called respectively the *pseudo-quotient* and *pseudo-remainder* of $f(x)$ w.r.t. $g(x)$. We also denote by $\text{prem}(f, g, x)$ the pseudo-remainder r .

Algorithm 2: The pseudo-remainder algorithm

Input : polynomials $f, g \in \mathbb{K}[x]$ such that $\deg(f) \geq \deg(g) > 0$

Output : the pseudo-remainder of f by g in x

```

1  $r \leftarrow f$ ;
2 for  $i \leftarrow \deg(f) - \deg(g)$  down to 0 do
3    $r \leftarrow \text{lc}(g)r - \text{lc}(f)x^i g$ ;
4 return  $r$ ;

```

We present our CUDA implementation for constructing FFT-based subresultant chain of bivariate polynomials. The first step is to evaluate the input polynomials $F, G \in \mathbb{Z}_c[x, y]$. For our implementation, univariate polynomials are all dense, encoded as a vector of coefficients. Multivariate dense polynomials with n variables are encoded recursively as a univariate polynomial with $(n - 1)$ -variable polynomials as coefficients. For example, $F = 1 + 2x + 3xy + 4x^2 + 5y^2$ can be encoded as a vector $[1, 2, 4, 0, 3, 0, 5, 0, 0]$, read as

$$F = (1 + 2x + 4x^2) + (0 + 3x + 0x^2)y + (5 + 0x + 0x^2)y^2.$$

Our subroutine `list_fft_univariate` computes a list of univariate n -point FFTs on a list of polynomials of the same size q . For $n = 2^k$ and $q \geq 1$, the Stockham FFT factorization implies

$$I_q \otimes \text{DFT}_n = \prod_{i=0}^{k-1} (I_q \otimes \text{DFT}_2 \otimes I_{2^{k-1}})(I_q \otimes D_{2,2^{k-i-1}} \otimes I_{2^i}) (I_q \otimes L_2^{2^{k-i}} \otimes I_{2^i}), \quad (3)$$

which can be efficiently realized from three types of CUDA kernels, similar to the Stockham FFT, as presented in [17]. More details appear in our article [17] reporting on a GPU implementation of FFT over finite fields.

Example 1 Let $F = a(x) + b(x)y + c(x)y^2 + d(x)y^3$ be a bivariate polynomial where a, b, c, d , have degree less than 8. Let ω be a 8-th primitive root of unity. After evaluating $F(x, y)$ at $x = (1, \omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6, \omega^7)$, the layout of the evaluation data is

$$M = \begin{bmatrix} a(1) & a(\omega) & a(\omega^2) & \cdots & a(\omega^6) & a(\omega^7) \\ b(1) & b(\omega) & b(\omega^2) & \cdots & b(\omega^6) & b(\omega^7) \\ c(1) & c(\omega) & c(\omega^2) & \cdots & c(\omega^6) & c(\omega^7) \\ d(1) & d(\omega) & d(\omega^2) & \cdots & d(\omega^6) & d(\omega^7) \end{bmatrix}$$

The i -th column corresponds to the univariate polynomial $F(\omega^i, y)$ in y for $i = 0 \cdots 7$. Transposing matrix M gives

$$M^t = \begin{bmatrix} a(1) & b(1) & c(1) & d(1) \\ a(\omega) & b(\omega) & c(\omega) & d(\omega) \\ a(\omega^2) & b(\omega^2) & c(\omega^2) & d(\omega^2) \\ a(\omega^3) & b(\omega^3) & c(\omega^3) & d(\omega^3) \\ a(\omega^4) & b(\omega^4) & c(\omega^4) & d(\omega^4) \\ a(\omega^5) & b(\omega^5) & c(\omega^5) & d(\omega^5) \\ a(\omega^6) & b(\omega^6) & c(\omega^6) & d(\omega^6) \\ a(\omega^7) & b(\omega^7) & c(\omega^7) & d(\omega^7) \end{bmatrix}.$$

Assuming that the leading coefficient $d(x)$ in y of $F(x, y)$ does not vanish at any power of ω . Then each row of M^t can be regarded as a univariate polynomial of degree 3.

As illustrated in the above example, the second step is to transpose the evaluated F and G for preparing the subresultant constructions. The major cost is to compute the subresultant chains at all evaluation points. To accelerate the overall performance, we implemented Brown's subresultant algorithm in CUDA for computing a sequence of subresultant chains in a highly parallel manner and we present two different approaches for this task.

Coarse-grained approach. The most direct way is to let each CUDA thread run a univariate Brown's subresultant algorithm. This approach works all the time and for practical problems where the number of threads can reach a big value, say 2^{12} , this can still bring a reasonable amount of speedup. However, this approach incurs a large amount of memory accesses to the GPU global memory within a thread block, which limits the peak performance of the implementation.

Fine-grained approach. The second approach is to break a list of univariate Brown's subresultant computations into a sequence of lists of univariate polynomial pseudo-divisions. Provided that at each step all A_j 's have the same degree and all B_j 's have the same degree¹, Algorithm 1 can be turned into the "list version", namely Algorithm 3. Initially, all F_j 's have the same degree according to the choice of ω , (as all G_j 's do).

For two univariate polynomials $P(x)$ and $Q(x)$, let $S_{k_1}, \dots, S_{k_\ell}$ be all the regular subresultants of P and Q with $0 \leq k_1 < \dots < k_\ell < \deg(Q)$. Then (k_1, \dots, k_ℓ) is called the degree sequence of P and Q . In fact, they are the degrees of all remainders in the Euclidean algorithm with input P, Q . Therefore, the above generic assumption says: the degree sequence of the subresultant chain of $F(\omega^i, y), G(\omega^j, y)$ is independent of j .

Algorithm 3: List of subresultant chains

Input : A list of pairs of polynomials $F_j, G_j \in \mathbb{K}[x]$ for $0 \leq j < m$ such that $\deg(F_j) \geq \deg(G_j) > 0$, all F_j having the same degree, and all G_j having the same degree.

Output : the subresultant chain of F_j and G_j , for $0 \leq j < m$.

```

1 for  $0 \leq i < \deg(G_j), 0 \leq j < m$  do
2    $S_i^j \leftarrow 0$ ;
3 for  $0 \leq j < m$  do
4    $B_j \leftarrow \text{prem}(F_j, -G_j, x)$ ;
5    $A_j \leftarrow G_j$ ;
6  $\alpha \leftarrow \deg(F_j) - \deg(G_j)$ ;
7 for  $0 \leq j < m$  do
8   while  $B_j \neq 0$  do
9      $d \leftarrow \deg(A_j), e \leftarrow \deg(B_j), \delta \leftarrow d - e$ ;
10     $S_{d-1}^j \leftarrow B_j$ ;
11     $S_e^j \leftarrow \text{lc}(A_j)^{\alpha(1-\delta)} \text{lc}(B_j)^{\delta-1} B_j$ ;
12    if  $e = 0$  then break;
13     $B_j \leftarrow \text{lc}(A_j)^{-\alpha\delta-1} \text{prem}(A_j, -B_j, x), A_j \leftarrow S_e^j, \alpha \leftarrow 1$ ;
14 for  $0 \leq j < m$  do
15   return  $\{S_i^j \mid 0 \leq i < \deg(G_j)\}$ ;

```

In Algorithm 3, the generic assumption implies that

¹ A nonzero constant has degree 0, and 0 has degree -1 .

- at Line 8, if $B_j = 0$ holds for some $0 \leq j < m$, then all B_j are zero;
- at Line 9, all A_j have the same degree d , and all B_j have the same degree e ;
- at Line 11, all S_e^j can be computed in a *Single Instruction Multiple Data (SIMD)* fashion.
- Similarly, at Line 13, all pseudo-divisions can be performed in a SIMD fashion.

Consequently, each of the lines 9 to 13 in the **while** loop can be executed in a SIMD fashion. Moreover, several threads can cooperate in computing one S_e^j or one B_j . In particular, our core subroutine performs a list of pseudo-divisions in a fine-grained way.

Example 2 Let $f = a_3x^3 + a_2x^2 + a_1x + a_0$ and $g = b_2x^2 + b_1x + b_0$. To obtain the pseudo-remainder $\text{prem}(f, -g, x)$ of f and g , we compute

- (1) $h_2 = -b_2f + a_3xg = c_2x^2 + c_1x + c_0$,
- (2) $h_1 = -b_2h_2 + c_2g = d_1x + b_0$.

Alternatively, the pseudo-remainder $\text{prem}(f, -g, x)$ can be computed in two steps:

$$(S_1) \quad c_2 = \begin{vmatrix} a_3 & a_2 \\ b_2 & b_1 \end{vmatrix}, \quad c_1 = \begin{vmatrix} a_3 & a_1 \\ b_2 & 0 \end{vmatrix}, \quad c_0 = \begin{vmatrix} a_3 & a_0 \\ b_2 & 0 \end{vmatrix};$$

$$(S_2) \quad d_1 = \begin{vmatrix} c_2 & c_1 \\ b_2 & b_1 \end{vmatrix}, \quad d_0 = \begin{vmatrix} c_2 & c_0 \\ b_2 & b_0 \end{vmatrix}.$$

As illustrated in the above example, the basic unit is to perform a single reduction step, called `list_reduce`, which takes two lists of univariate polynomials LF and LG as input, and computes

$$h_i = \text{lc}(g_i)f_i - \text{lc}(f_i)x^{\deg(f_i) - \deg(g_i)}g_i,$$

where f_i is the i -th polynomial in LF and g_i is the i -th polynomial in LG. We assume that polynomials in LG have the same degree dG , and polynomials in LF have the same degree dF . The result LH consists of the computed h_i 's. The source code of this kernel is included in Appendix B. In terms of the kernel `list_reduce`, a list of pseudo-remainders can be computed as in Algorithm 4 using a double-buffer method, where operation `swap` only exchanges two pointers not their contents.

Algorithm 4: List of pseudo-remainders

Input : Two lists LF, LG of polynomials such that $\text{LF}[i] = F_i$, $\text{LG}[i] = G_i$ for $0 \leq i < m$, $\deg(F_i) \geq \deg(G_i) > 0$, all F_i having the same degree dF , and all G_i having the same degree dG .

Output : The list LH of polynomials such that $\text{LH}[i] = H_i = \text{prem}(F_i, -G_i, x)$ for $0 \leq i < m$.

- 1 `list_reduce(LX, dF, LF, dG, LG, p)`;
 - 2 **for** $d \leftarrow \text{dF} - 1$ **down to** dG **do**
 - 3 `list_reduce(LY, d, LX, dG, LG, p)`;
 - 4 `swap(LX, LY)`;
 - 5 Copy out the result from LX to LH;
-

d	t_0	t_1	t_1/t_0
30	0.23	0.29	1.3
40	0.23	0.43	1.9
50	0.27	1.14	4.2
60	0.27	1.53	5.7
70	0.31	3.95	12.7
80	0.32	4.88	15.3
90	0.35	5.95	17.0
100	0.50	19.10	38.2
110	0.53	17.89	33.8
120	0.58	19.72	34.0

Figure 2. Computing resultants for bivariate dense polynomials in seconds

3.2. Computing resultants

The data computed from Brown’s Algorithm, either with the coarse-grained method or with the fine-grained method, is called a *subresultant chain cube* or simply an *scube*. It consists of the images of all the subresultants of two polynomials on a FFT grid. Any subresultant or any coefficient of a subresultant can be interpolated from the scube, by means an inverse FFT computation inside the GPU.

For two bivariate polynomials of partial degree d in both x and y , the size of their scube is in the order of d^4 . For instance, this size is approximately 632MB for $d = 100$. For applications like computing the resultant or several subresultants (as in our bivariate solver), it is natural to keep the scube within the GPU memory and copy only the desired result (say a subresultant) back to the main main, whenever necessary.

Throughout this paper, our code is compiled with gcc 4.2.4 and nvcc 2.2. All the benchmarks were conducted on a desktop with the processor Intel Core 2 Quad CPU Q9400 @ 2.66GHz and 6 GB main memory. In Appendix A, we briefly list the specifications of two NVIDIA’s graphics cards used in our benchmarking: Geforce GTX 285 and Telsa C2050. Without any modifications, our code constructs a scube up to 2 times faster on the latter GPU, mainly due to the better support double-precision floating point calculations. Please see Appendix B for detailed description on integer multiplications modulo a prime number using floats.

In Figure 2, we report the timing for computing resultant $\text{res}(F_1, F_2, y)$ with bivariate random dense polynomials $F_1, F_2 \in \mathbb{Z}_c[x, y]$ such that $c = 469762049$ and $d = \deg(F_i, x) = \deg(F_i, y)$ for $i = 1, 2$. In the figure, the first column, labeled by d , shows the partial degree d . The second one, labeled by t_0 , is the timing for GPU implementation of the FFT-based scube method, which includes the time for moving result back to the main memory. The third column, labeled by t_1 , shows the results for the CPU implementation of the FFT-based scube serial C code in the `modpn` library [14]. The last column reports the ratio between the two implementations. Note that all the resultants in these experiments are computed with the fine-grained method. The maximal speedup we achieve is approximately 38.

In addition to the one-dimensional Stockham FFT, we have also implemented a two-dimensional Stockham FFT for handling trivariate polynomials. This can be applied to compute the scube for a trivariate input. Figure 3 lists our experimental results for computing resultants for trivariate random dense polynomials in $\mathbb{K}[x, y, z]$. The first column shows the common partial degree d in x, y and z . The other three column have the same meaning as in the bivariate case. Note that all but $d = 15$ and $d = 18$ are based on the fine-grained approach. When the coarse-grained method is forced to be used, the speedup it achieves drops significantly. This confirms experimentally the merit of the fine-grained approach.

We observe that the GPU-based implementation achieves a much larger speedup factor in the

d	t_0	t_1	t_1/t_0
7	0.22	0.16	0.7
8	0.23	0.76	3.3
9	0.24	0.85	3.5
10	0.25	0.98	3.9
11	0.24	1.10	4.6
12	0.30	4.96	16.5
13	0.31	5.52	17.8
14	0.32	6.07	19.0
15	0.78	8.95	11.5
16	0.65	31.65	48.7
17	0.66	34.55	52.3
18	3.46	47.54	13.7
19	0.73	51.04	69.9
20	0.75	43.12	57.5

Figure 3. Computing resultants for trivariate dense polynomials in seconds

trivariate case (approximately 70 for the best cases) than in the bivariate case (approximately 38). We are currently investigating the phenomenon.

4. Bivariate Solver

In this section, we discuss the GPU acceleration towards solving bivariate polynomials systems over finite fields by means of triangular decompositions. This work extends the papers [12, 14]. Throughout this section, we consider two bivariate polynomials F_1 and F_2 are in $\mathbb{K}[x, y]$, with ordered variables $x < y$ and with coefficients in a field \mathbb{K} .

4.1. Preliminaries

The key idea behind solving bivariate polynomials follows from the fact that the common roots of F_1 and F_2 are likely to be described by the common roots of a single triangular set:

$$\begin{cases} A(x) = 0 \\ B(x, y) = 0 \end{cases}$$

Generically, the univariate polynomial $A(x)$ describes the projection of $V(F_1, F_2)$ on the x -axis, whereas the bivariate polynomial $B(x, y)$ is expected to have partial degree 1 in y . We usually obtain $A(x)$ from the resultant of F_1 and F_2 in y , and compute $B(x, y)$ in a polynomial GCD computation.

Example 3 ([14]) *Consider $F_1 = x^2 + y + 1$ and $F_2 = x + y^2 + 1$. We have $\text{res}(F_1, F_2, y) = x^4 + 2x^2 + x + 2$, which is a squarefree polynomial when $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{Z}_c$ with $c \neq 3, 7$. It is not hard to show that $A(x) = x^4 + 2x^2 + x + 2$ and $B(x, y) = y + x^2 + 1$ are a desired pair of polynomials. That is, the common solutions of $F_1(x)$ and $F_2(x)$ are exactly those of $A(x)$ and $B(x, y)$. In fact, the polynomial $B(x, y)$ is a gcd of F_1 and F_2 modulo $A(x)$ in a technical sense introduced in [16] and that we illustrate here. For a formal definition and algorithms, please refer to [12].*

Observe that $A(x)$ writes $(x^2 + x + 1)(x^2 - x + 2)$, thus we have

$$\begin{cases} F_1 = y - x, \\ F_2 = (y - x)(y + x), \end{cases} \text{ if } x^2 + x + 1 = 0,$$

$$\begin{cases} F_1 = y + x - 1, \\ F_2 = (y - x + 1)(y + x - 1), \end{cases} \text{ if } x^2 - x + 2 = 0.$$

Hence, using the Chinese Remaindering Theorem, we obtain

$$B(x, y) = \gcd(F_1, F_2) \bmod A(x) = y + x^2 + 1.$$

4.2. Algorithm

Algorithm 7 is an improved version of the bivariate solver routine presented in [13]. It can also be viewed as a specialized version of the general regular gcd algorithm presented in [12]. The improvement is two folds. First Algorithm 7 makes no genericity assumptions on the input polynomials F_1, F_2 , such as the resultant $\text{res}(F_1, F_2, y)$ is not zero. Secondly, in some non-generic situations, Algorithm 7 avoids unnecessary computations. However, and in order to avoid technical details irrelevant to this paper's objectives, Algorithm 7 computes a decomposition into triangular sets, rather than regular chains. Regular chains have additional algebraic properties which are often desirable in practice. Deriving a decomposition into regular chains from a decomposition into triangular sets is easy and we refer to [16] for related procedures.

Observe that Algorithm 7 takes three polynomials as input. Indeed, in addition to F_1 and F_2 , it takes a third polynomial g , which is univariate in x and which plays the role of a "book keeper". Algorithm 7 computes the common zeros of these three polynomials via a triangular decompositions into triangular sets. Observes also that Algorithm 7 is essentially a wrapper function for Algorithm 6. This second algorithm takes as input two bivariate (non-univariate, non-constant) polynomials F_1, F_2 and two univariate polynomials g, h . Algorithm 6 computes the common zeros of F_1, F_2, g that do not cancel h . It can be seen as the core procedure of our bivariate solver. This is precisely in this procedure that the subresultant chain of F_1, F_2 is computed and then manipulated to obtain the necessary subresultant polynomials. Once calculated, the subresultant chain of F_1, F_2 is traversed in a bottom-up manner as in the algorithm of [12].

Observe that Algorithm 6 starts by forwarding to Algorithm 5 the case where the resultant of F_1, F_2 w.r.t. y is zero. After Line 7, the polynomial R is a squarefree factor of the resultant. In the **while** loop, we maintain a level counter i , initialized to 1. During the loop execution, we have the following invariant: R is a non-constant univariate polynomial and R divides the coefficient in y^v of S_v for all $0 < v < i$. The inner **while** loop from Line 12 to Line 16 searches for the first regular subresultant S_j such that R does not divide $\text{lc}(S_j, y)$. This search skips all the non-regular subresultants. The leading coefficient of S_j splits the squarefree polynomial R into two parts $G = \gcd(R, \text{lc}(S_j, y))$ and $R' = R/G$. The pair (R', S_j) is added to the output, and the algorithm continues with G to the level $j + 1$.

Finally, Algorithm 5 can be seen as another wrapper function which handles the case where the input polynomials F_1 and F_2 have a non-trivial factor, and thus a zero resultant.

4.3. Complexity analysis

We analyze the algebraic complexity of Algorithm 6 for the generic dense bivariate input. To further simplify the analysis, we assume $\deg(F_1, x) = \deg(F_1, y) = \deg(F_2, x) = \deg(F_2, y) = d$. That is, both F_1 and F_2 are random dense square polynomials with partial degrees d . We also assume $g = 0$ and $h = 1$.

The degree of the resultant is bounded by $B = 2d^2 + 1$, which can be read from the Sylvester matrix of F_1 and F_2 . Hence the FFT size n is the smallest power of 2 such that $n = 2^e \geq B$.

Algorithm 5: ModularGenericSolve2ZeroResultant(F_1, F_2, g, h)

Input : $F_1, F_2 \in \mathbb{K}[x, y]$ and $g, h \in \mathbb{K}[x]$ such that $h(x) \neq 0$ implies $\gcd(\text{lc}(F_1, y), \text{lc}(F_2, y))(x) \neq 0$ and such that F_1, F_2 have positive degree in y and such that F_1, F_2 have a zero resultant in y
Output : a set of triangular sets $\{(A_1, B_1), \dots, (A_e, B_e)\}$ such that

$$V(F_1, F_2, g) \setminus V(h) = \bigcup_{i=1}^e V(A_i, B_i).$$

```
1  $G \leftarrow \gcd(F_1, F_2)$ ;
2  $result \leftarrow \{(g, G)\}$ ;
3  $F_1 \leftarrow F_1 \text{ quo } G$ ;
4  $F_2 \leftarrow F_2 \text{ quo } G$ ;
5 if  $F_1 \in \mathbb{K}[x]$  then
6    $F_1 \leftarrow \text{squarefreePart}(F_1)$ ;
7    $F_1 \leftarrow F_1 \text{ quo } \gcd(F_1, h)$ ;
8    $F_1 \leftarrow \gcd(g, F_1)$ ;
9 if  $F_1 \in \mathbb{K}$  then return  $\emptyset$ ;
10 if  $F_2 \in \mathbb{K}[x]$  then
11    $F_2 \leftarrow \text{squarefreePart}(F_2)$ ;
12    $F_2 \leftarrow F_2 \text{ quo } \gcd(F_2, h)$ ;
13    $F_2 \leftarrow \gcd(g, F_2)$ ;
14 if  $F_2 \in \mathbb{K}$  then return  $\emptyset$ ;
15 if  $F_1, F_2 \in \mathbb{K}[x]$  then
16    $g \leftarrow \gcd(F_1, F_2)$ ;
17   if  $g \notin \mathbb{K}$  then return  $result \cup \{(g, 0)\}$  ;
18 if  $F_1 \in \mathbb{K}[x]$  then return  $result \cup \{(F_1, F_2)\}$ ;
19 if  $F_2 \in \mathbb{K}[x]$  then return  $result \cup \{(F_2, F_1)\}$ ;
20 return  $result \cup \text{ModularGenericSolve2}(F_1, F_2, g, h)$ 
```

Let ω be a proper n -th primitive root of unity, any power of which does not cancel the leading coefficients of F_1 and F_2 in y . Then for each $0 \leq i < n$, evaluating F_1 and F_2 over ω^i gives a pair of univariate polynomials of degree d . The subresultant algorithm on each pair costs $\Theta(d^2)$ operations in \mathbb{Z}_c , from which we derive the cost to construct the FFT based subcube $\Theta(d^4)$.

For generic input F_1, F_2 , we expect that the loop from Line 7 to Line 11 find $j = 1$. That is, the expected gcd of F_1 and F_2 modulo their resultant in y is the subresultant of index 1. This also implies that $G = 1$ at Line 19. In this case, the return value of Algorithm 6 is the pair (A, B) where A is the squarefree part of the resultant and B is the subresultant of index 1. Hence, the major steps are

- subresultant chain construction at Line 5,
- computation of the squarefree part at Line 7,
- interpolation of the subresultant S_1 at Line 13,
- remainder computation at Line 14,
- gcd computation at Line 19.

The cost to interpolate the subresultant S_1 is $O^\sim(d^2)$, which is the cost to interpolate

Algorithm 6: ModularGenericSolve2(F_1, F_2, g, h)

Input : $F_1, F_2 \in \mathbb{K}[x, y]$ and $g, h \in \mathbb{K}[x]$ such that $h(x) \neq 0$ implies that $\gcd(\text{lc}(F_1, y), \text{lc}(F_2, y))(x) \neq 0$ and such that F_1, F_2 have positive degree in y
Output : a set of triangular sets $\{(A_1, B_1), \dots, (A_e, B_e)\}$ such that

$$V(F_1, F_2, g) \setminus V(h) = \bigcup_{i=1}^e V(A_i, B_i).$$

```
1  $h \leftarrow \text{squarefreePart}(h)$ ;
2  $g \leftarrow \text{squarefreePart}(g)$ ;
3  $g \leftarrow g \text{ quo } \gcd(g, h)$ ;
4 if  $g \in \mathbb{K} \setminus \{0\}$  then return  $\emptyset$ ;
5 Compute the subresultant chain  $S$  of  $F_1, F_2$  in  $y$ ;
6 if  $S_0 = 0$  then return  $\text{ModularGenericSolve2ZeroResultant}(F_1, F_2, g, h)$  ;
7  $R \leftarrow \text{squarefreePart}(S_0)$ ;
8  $R \leftarrow R \text{ quo } h$ ;
9  $R \leftarrow \gcd(R, g)$ ;
10  $result \leftarrow \emptyset, i \leftarrow 1$ ;
11 while  $R \notin \mathbb{K}$  and  $i \leq \deg(F_2, y)$  do
12   while  $i \leq \deg(F_2, y)$  do
13     Let  $S_j$  be the regular subresultant with  $i \leq j \leq \deg(F_2, y)$  with  $j$  minimum;
14     if  $R$  divides  $\text{lc}(S_j, y)$  then  $i \leftarrow i + 1$ ;
15     else
16       break;
17   if  $i > \deg(F_2, y)$  then
18     return  $result \cup \{(R, F_1)\}$ ;
19    $G \leftarrow \gcd(R, \text{lc}(S_j, y))$ ;
20   if  $G \in \mathbb{K}$  then return  $result \cup \{(R, S_j)\}$ ;
21    $result \leftarrow result \cup \{(R \text{ quo } G, S_j)\}$ ;
22    $R \leftarrow G, i \leftarrow j + 1$ ;
23 return  $result$ ;
```

two coefficients of S_1 with inverse FFTs. The classic algorithm to compute the gcd of two univariate polynomial of degree d^2 costs $O(d^4)$ field operations. However, by means of the half-gcd technique, this can be reduced to $O^\sim(d^2)$. Hence, the squarefree part at Line 2 and the gcd computation at Line 14 both cost $O^\sim(d^2)$ operations in \mathbb{Z}_c . By means of Newton iteration, the fast remainder computation (Ch. 9 of [9]) costs $O^\sim(d^2)$ field operations at Line 14. In our implementation, the subresultant construction and polynomial interpolations, of cost $\Theta(d^4)$, are performed in parallel inside the GPU; the remaining portion costs $O^\sim(d^2)$ operations in \mathbb{Z}_c .

4.4. Experimentation

In this section, we report our benchmarking results for solving bivariate systems. The dense polynomials F_1 and F_2 are taken from $\mathbb{Z}_c[x, y]$ randomly, with $c = 469762049$. In Figure 4, the partial degrees of F_1 and F_2 are all d , indicated in the first column. The following four columns indicate timing as follows:

t_0 : GPU implementation of FFT-based scube construction,

Algorithm 7: ModularSolve2(F_1, F_2, g)

Input : $F_1, F_2 \in \mathbb{K}[x, y]$ and $g \in \mathbb{K}[x]$ **Output** : a set of triangular sets $\{(A_1, B_1), \dots, (A_e, B_e)\}$ such that

$$V(F_1, F_2, g) = \bigcup_{i=1}^e V(A_i, B_i).$$

```
1 if  $g = F_1 = F_2 = 0$  then return  $\{(0, 0)\}$  ;
2 if  $g, F_1, F_2 \in \mathbb{K}$  then return  $\emptyset$ ;
3 if  $F_1 \in \mathbb{K}[x]$  and  $F_2 \in \mathbb{K}[x]$  then
4   |  $A \leftarrow \gcd(F_1, F_2, g)$ ;
5   | if  $A \in \mathbb{K}$  then return  $\emptyset$ ;
6   | return  $\{(A, 0)\}$ ;
7 if  $F_1 \in \mathbb{K}[x]$  then
8   | return ModularSolve2( $F_1 + F_2, F_2, g$ );
9 if  $F_2 \in \mathbb{K}[x]$  then
10  | return ModularSolve2( $F_1, F_1 + F_2, g$ );
11  $h \leftarrow \gcd(\text{lc}(F_1, y), \text{lc}(F_2, y))$ ;
12  $G \leftarrow \text{ModularGenericSolve2}(F_1, F_2, g, h)$ 
13 if  $h \in \mathbb{K}$  then return  $G$ ;
14  $F_1 \leftarrow \text{reductum}(F_1, y)$ ,  $F_2 \leftarrow \text{reductum}(F_2, y)$ ;
15 return  $G \cup \text{ModularSolve2}(F_1, F_2, \gcd(g, h))$ ;
```

d	t_0	t_1	t_2	t_3	t_2/t_0	t_3/t_1
25	0.23	0.30	0.10	0.17	0.4	0.6
30	0.25	0.35	0.14	0.25	0.6	0.7
35	0.23	0.42	0.34	0.51	1.5	1.2
40	0.25	0.46	0.42	0.64	1.7	1.4
45	0.24	0.51	0.48	0.77	2.0	1.5
50	0.28	0.67	1.14	1.56	4.1	2.3
55	0.27	0.76	1.33	1.84	4.9	2.4
60	0.29	0.88	1.54	2.20	5.3	2.5
65	0.30	1.13	3.50	4.44	11.7	3.9
70	0.31	1.20	3.94	4.94	12.7	4.1
75	0.31	1.34	4.42	5.56	14.3	4.2
80	0.32	1.42	4.84	6.06	15.1	4.3
85	0.34	1.74	5.40	6.95	15.9	4.0
90	0.33	1.80	5.94	7.54	18.0	4.2
95	0.45	2.47	13.09	15.44	29.0	6.3
100	0.48	2.56	14.23	16.66	29.7	6.5
105	0.49	2.74	15.53	18.14	31.7	6.6
110	0.52	2.93	16.78	19.58	32.1	6.7
115	0.54	3.35	18.05	21.26	33.4	6.3
120	0.55	3.80	24.41	28.60	44.4	7.5

Figure 4. Solving bivariate systems in seconds

t_1 : total time for solving with GPU support,
 t_2 : CPU implementation of FFT-based subcube construction,
 t_3 : total time for solving without any GPU support.

The fifth column shows the speedup for the subresultant chain construction and the last column shows the speedup for solving bivariate random dense systems. Figure 5 shows the comparison of the running time with and without GPU acceleration. The largest speedup we achieve is approximately 7.5.

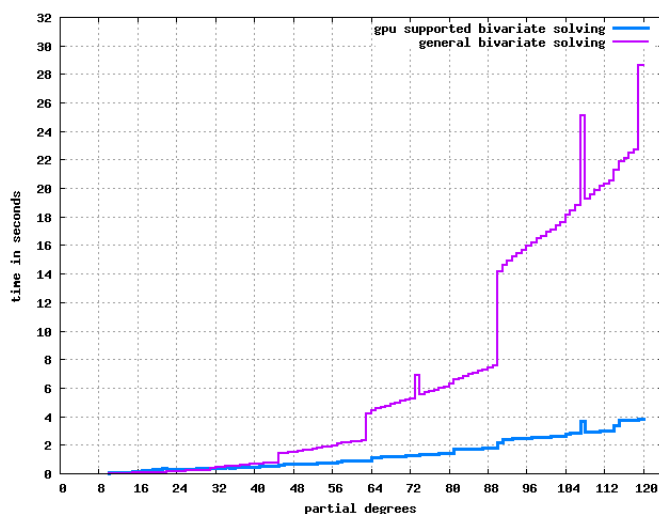


Figure 5. Solving bivariate random dense polynomial systems over a finite field

When asymptotically fast arithmetic is used for univariate polynomials, in terms of algebraic complexity, the dominate part of solving generic bivariate system is the subresultant construction. This could be observed from Figure 6. In the histogram, the left column (red) indicates the time spent for subresultant chain construction, and the right column (green) indicates the total time. After plugging the GPU support in the FFT-based subresultant chain computations into the bivariate solver, the univariate polynomial arithmetic (for large polynomials) dominates the total running time as shown in Figure 7. This also limits our speedup factor, since univariate polynomials GCDs are currently all computed serially by the CPU.

5. Conclusions and remarks

In this paper, we have reported a GPU implementation of dense multivariate polynomial arithmetic focusing on the computation of subresultant chains. With respect to a pure CPU code, we obtain very large speedup factors for this task. We have also reported on the implementation of a bivariate polynomial system solver, based on our GPU code. By speeding up subresultant computations, our GPU-supported solver has substantially increased its performance w.r.t. its pure CPU code counterpart.

The subresultant construction also illustrates the following pattern, which is frequent in symbolic computation and which is well-suitable for GPU computing. More precisely, the total amount of work (in terms of number of coefficient operations) is essentially proportional (may be up to logarithmic factors) to the size of the output, This type of algorithms seem to have sufficient parallelism for current multicore architectures. However, due the output data size,

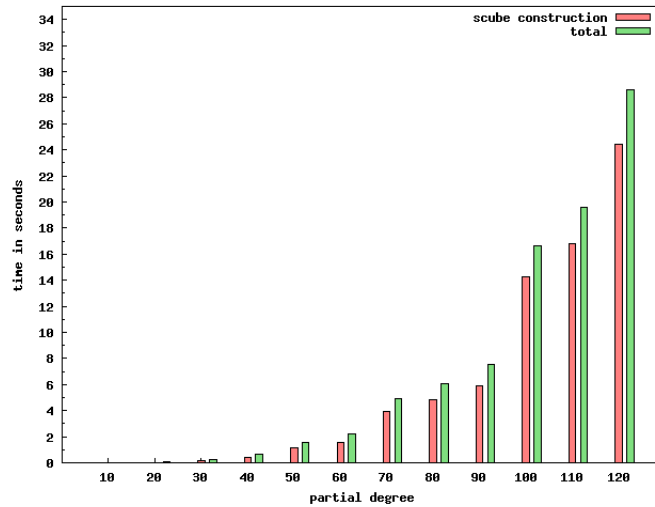


Figure 6. Subresultant computations inside bivariate dense solver, C code only

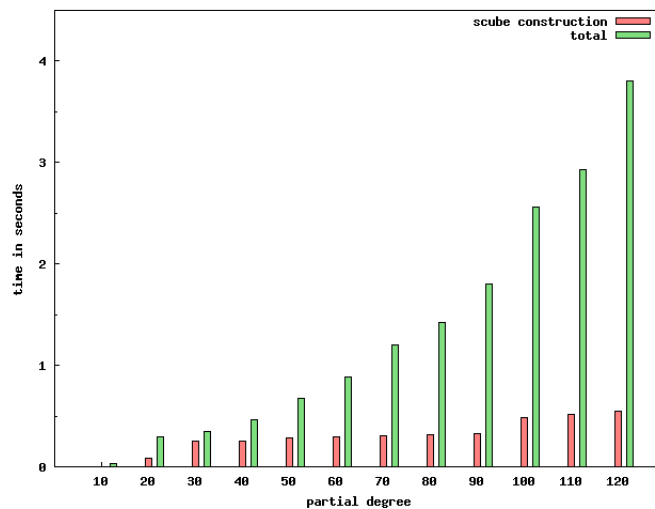


Figure 7. Subresultant computations inside bivariate dense solver, C with GPU code

which is often much larger than than the L2 or L3 cache size, performance can be limited by the memory bandwidth between the L2 (or L3) cache and the main memory.

As shown in Section 4.4, the speedup we can achieve in the bivariate solver is largely limited by the computation of univariate polynomial gcds of large degree. In the future work, we would like to improve this part and in turn fully utilize the speedup from our GPU scube construction.

References

- [1] *Wikipedia page for GPGPU*. <http://en.wikipedia.org/wiki/GPGPU>.
- [2] *GPGPU.org*. <http://gpgpu.org/category/research>.
- [3] E. Berberich, P. Emeliyanenko, and M. Sagraloff. An elimination method for solving bivariate polynomial systems: Eliminating the usual drawbacks. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2011.
- [4] B. Boyer, J.-G. Dumas, and P. Giorgi. Exact sparse matrix-vector multiplication on gpu's and multicore architectures. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, PASCO '10*, pages 80–88, New York, NY, USA, 2010. ACM.

- [5] W. Brown. The subresultant PRS algorithm. *Transaction on Mathematical Software*, 4:237–249, 1978.
- [6] G. Collins. *The Calculation of Multivariate Polynomial Resultants*, volume 18, pages 515–532. Journal of the ACM, 1971.
- [7] M. El Kahoui. An elementary approach to subresultants theory. *J. Symb. Comp.*, 35:281–292, 2003.
- [8] P. Emeliyanenko. A complete modular resultant algorithm targeted for realization on graphics hardware. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 35–43. ACM, 2010.
- [9] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.
- [10] L. Jacquín, V. Roca, J.-L. Roch, and M. Al Ali. Parallel arithmetic encryption for high-bandwidth communications on multicore/gpgpu platforms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 73–79, New York, NY, USA, 2010. ACM.
- [11] M. Kalkbrener. *A generalized Euclidean algorithm for computing triangular representations of algebraic varieties*, volume 15, pages 143–167. *J. Symb. Comp.*, 1993.
- [12] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *Proc. ISSAC'09*, pages 239–246, New York, NY, USA, 2009. ACM Press.
- [13] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. In *MICA'08*, pages 73–80, 2008.
- [14] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple, 2008. To appear in the *J. of Symbolic Computation*.
- [15] M. Monagan. Probabilistic algorithms for computing resultants. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 245–252, New York, NY, USA, 2005. ACM.
- [16] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. <http://www.csd.uwo.ca/~moreno>.
- [17] M. Moreno Maza and W. Pan. *Fast polynomial arithmetic on a GPU*, volume 256. *J. of Physics: Conference Series*, 2010.
- [18] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. In *Proc. of PDCAT'09*. IEEE Computer Society, 2009.
- [19] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In D. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCIS*, Heidelberg, 2010. Springer-Verlag Berlin.
- [20] W. Pan. *Algorithmic Contributions to the Theory of Regular Chains*. PhD thesis, University of Western Ontario, 2010.
- [21] C. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1993.

Appendix A. Specifications of our GPU cards

Our CUDA code has been run for the two NVIDIA GPUs and we briefly list their specifications below.

GPU	GTX 285	Tesla C2050
Capability	1.3	2.0
Multiprocessors	30	14
Cores	240	448
Clock Rate	1.15G GHz	1.15 GHz
Memory Bandwidth	159 GB/sec	144 GB/sec
Double Floating Point	partially	fully
Global Memory	1GB	3GB
Shared Memory	16KB	48KB or 16KB
L1 Cache	none	48KB or 16KB
L2 Cache	none	768KB
Concurrent Kernels	no	up to 16

The first card belongs to the GeForce 200 series, which starts to support double-precision floating point numbers in hardware, but only one double-precision floating point unit (FPU) per multiprocessor (a group of 8 or 32 cores). The second card belongs to the Fermi series released in April 2010, in which each core is integrated with a double-precision FPU.

Appendix B. Source code

The following CUDA kernel performs a list of eliminations over a finite field \mathbb{Z}_c , which takes two lists of univariate polynomials LF and LG, and computes

$$h_i = \text{lc}(g_i)f_i - \text{lc}(f_i)x^{\deg(f_i)-\deg(g_i)}g_i,$$

where f_i is the i -th polynomial in LF and g_i is the i -th polynomial in LG. It requires that polynomials in LG have the same degree dG, and polynomials in LF have the same degree dF. The result LH consists of h_i 's computed.

```
__global__ void list_reduce(int *LH, int dF, int *LF,
    int dG, int *LG, int p)
{
    int bid = blockIdx.x;
    int tid = bid * blockDim.x + threadIdx.x;
    int qtid = tid / dF;    // pair index
    int rtid = tid % dF;

    int *F = LF + qtid * (dF + 1); // first poly
    int *G = LG + qtid * (dG + 1); // second poly
    int *H = LH + qtid * dF;       // output poly

    // The configuration is the following
    //      u ..... a
    //      v ..... b
    // where a is the current coefficient to eliminate,
    // b is the current leading coefficient,
    // and u, v are coefficients to be adjusted.
    // For each pair (u, v), compute
    //      a * v - u * b mod p
    // and store it to H[rtid];

    int dgap = dF - dG;
    int a = F[dF]; // the leading coefficient of F
    int b = G[dG]; // the leading coefficient of G
    int u = F[rtid];
    int v = ((rtid >= dgap) ? G[rtid - dgap] : 0);

    int t1 = mul_mod(a, v, p); // t1 = a * v mod p
    int t2 = mul_mod(b, u, p); // t2 = b * u mod p
    H[rtid] = sub_mod(t1, t2, p);
}
```

The modular multiplication `mul_mod` uses the following method in our code, assuming that the inverse $\text{pinv} = 1/p$ of p has been pre-computed.

```
int mul_mod(int a, int b, int p, double pinv) {
    int q = (int)((((double)a) * ((double)b)) * pinv);
    int result = a * b - q * p;
    if (result < 0) result += p;
    return result;
}
```

The above code requires two double-precision floating point multiplications. Up to our knowledge, this is the simplest and best method to do the modular multiplications, even for those GPUs which only partially support double-precision floats, like Nvidia GTX 285.

Appendix C. Profiling results

To visualize the performance of our implementations, we use the Nvidia’s visual profiler `cuda-prof` to analyze CUDA kernel calls. It is very helpful to find out the bottlenecks of an implementation.

Figure C1 shows the kernel statistics of the Stockham FFT. In this figure, the x-axis shows the kernel call indices in chronological order and the y-axis is proportional to the GPU time for each kernel. The first kernel copies the coefficient vector of the univariate polynomial into the GPU global memory, followed by a sequence of kernel calls to `double_expand_ker` to compute powers $\{1, \omega, \omega^2, \dots, \omega^{n/2-1}\}$. The last kernel copies the result back to the main memory. In the middle, it has basic butterflies, twiddling and data transpositions. Each of those kernel names uses `_a` or `_b` as a suffix, since we implement the same algorithm for handling input data in different ranges. This enables the kernels to efficiently utilize the shared memory space of the GPU, whenever necessary.

Figure C1 was generated using our first GPU GTX 285 and the FFT size is $n = 2^{26}$. The memory bandwidth will be the only bottleneck if the goal is to speedup a single univariate FFT. It is advisable to keep the data inside GPU global memory as long as possible, as we did for computing the subresultant chains.

Figure C2 was generated using our second GPU Tesla C2050 for constructing the scube of two random dense bivariate polynomials of partial degree 100 in each variable. There are three major components: FFT evaluations of a list of univariate polynomials (the region in a rectangle shape), initialization of the output (the highest single column), and the fine-grained Brown’s algorithm (the region in a triangle shape). The last one dominates the overall computation, i.e. the kernel `list_reduce`. We gain approximately an extra speedup factor of 2 by switching from the GTX 285 to Tesla C2050, due to the better support of double-precision floating point calculations.

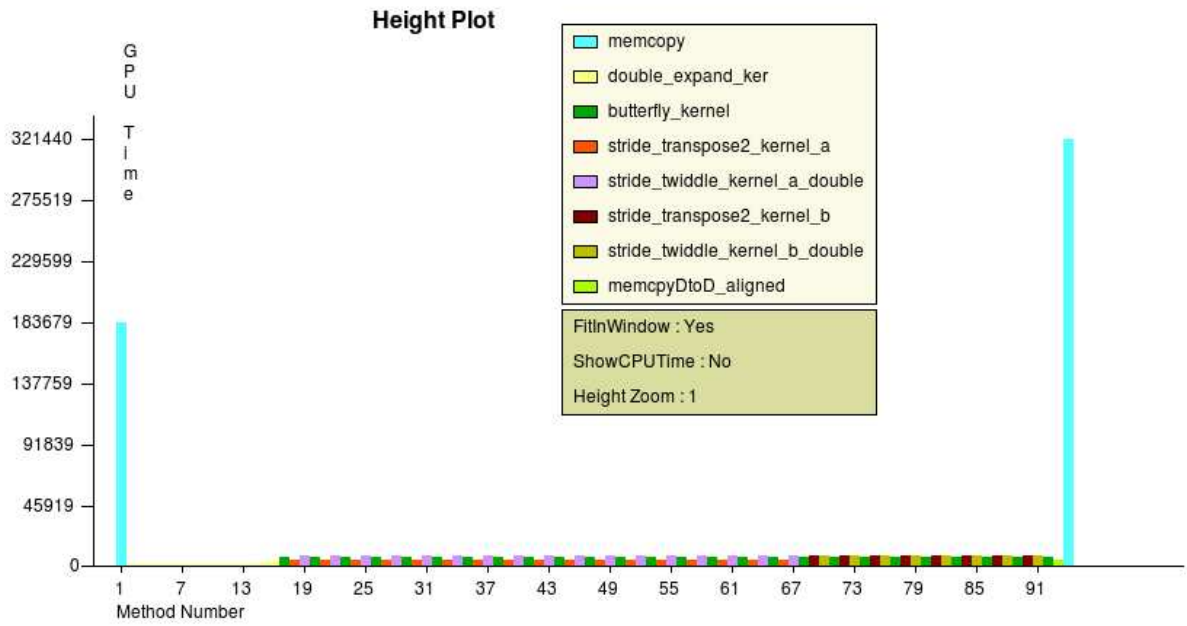


Figure C1. Kernel statistics of the Stockham FFT

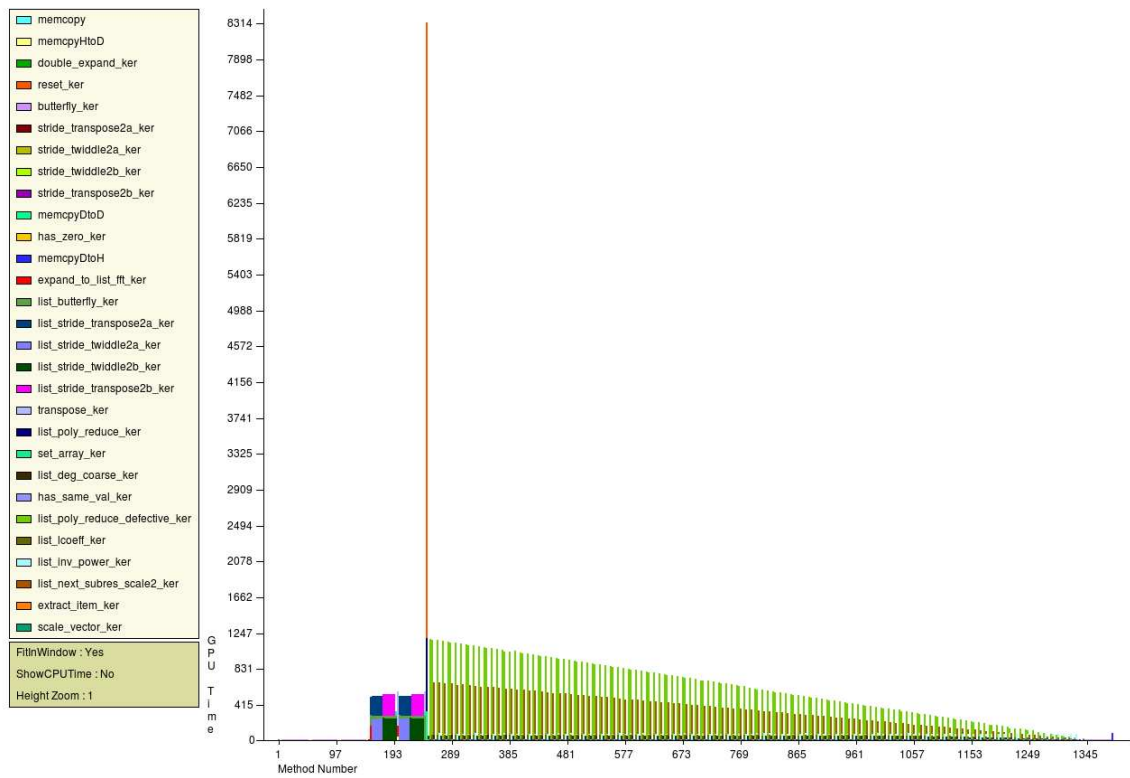


Figure C2. Kernel statistics of a scube construction