Accelerating Symbolic Computations on NVidia Fermi

Pavel Emeliyanenko

Max-Planck Institute for Informatics, Saarbrücken, Germany

asm@mpi-inf.mpg.de

Abstract: we present the first implementation of a modular resultant algorithm on GPUs [4,5]. With recent developments taking advantage of new NVidia Fermi GPU architecture and instruction set we have been able to achieve about 150x speedup over a CPU-based resultant algorithm from Maple 13.

Motivation

Resultants is a fundamental algebraic tool in the elimination theory. They have numerous applications, for instance in topological study of algebraic curves or computer graphics. Resultant of two bivariate polynomials **f** and **g** is the determinant of Sylvester matrix **S**:

$$f(x,y)=\sum_i f_i(x)y^i \quad g(x,y)=\sum_i g_i(x)y^i$$

$$res_y(f,g)=\det S_{f,g} \qquad S_{f,g}=\begin{bmatrix} f_4 & 0 & 0 & g_3 & 0 & 0 & 0\\ f_3 & f_4 & 0 & g_2 & g_3 & 0 & 0\\ f_2 & f_3 & f_4 & g_1 & g_2 & g_3 & 0\\ f_1 & f_2 & f_3 & g_0 & g_1 & g_2 & g_3\\ f_0 & f_1 & f_2 & 0 & g_0 & g_1 & g_2\\ 0 & f_0 & f_1 & 0 & 0 & g_0 & g_1\\ 0 & 0 & f_0 & 0 & 0 & 0 & g_0 \end{bmatrix}$$

Computing resultant involves a substantial amount of symbolic operations which rapidly becomes a bottleneck for many exact geometric algorithms

High-level structure of the algorithm

Following the ideas of classical "divide-conquer-combine" modular algorithm of Collins [1]:

- given two bivariate polynomials with large integer coefficients use modular and evaluation homeomorphisms to reduce the problem to a simpler domain:
  $$\mathbb{Z}[x,y]\to\mathbb{Z}_m[x,y]\to\mathbb{Z}_m[x,y]/(x-\alpha_m)$$
- compute univariate resultants over a prime field in parallel on the graphics hardware:
  $$z=res(f(y),g(y)):\mathbb{Z}_m[x,y]/(x-\alpha_m)\to\mathbb{Z}_m[x]/(x-\alpha_m)$$
- interpolate resultant polynomial over a prime field (GPU): $\mathbb{Z}_m[x]/(x-\alpha_m)\to\mathbb{Z}_m[x]$
- lift the polynomial coefficients using Chinese remaindering (**partly** on the GPU): $\mathbb{Z}_m[x]\to\mathbb{Z}[x]$

Introduction to Displacement structure

Problem: the amount of parallelism exhibited by the modular algorithm is **far too low** to satisfy the needs of massively-threaded architecture like that of GPU

Solution: reduce the problem to computation with structured matrices because **matrix operations** typically map **very well** to the graphics hardware

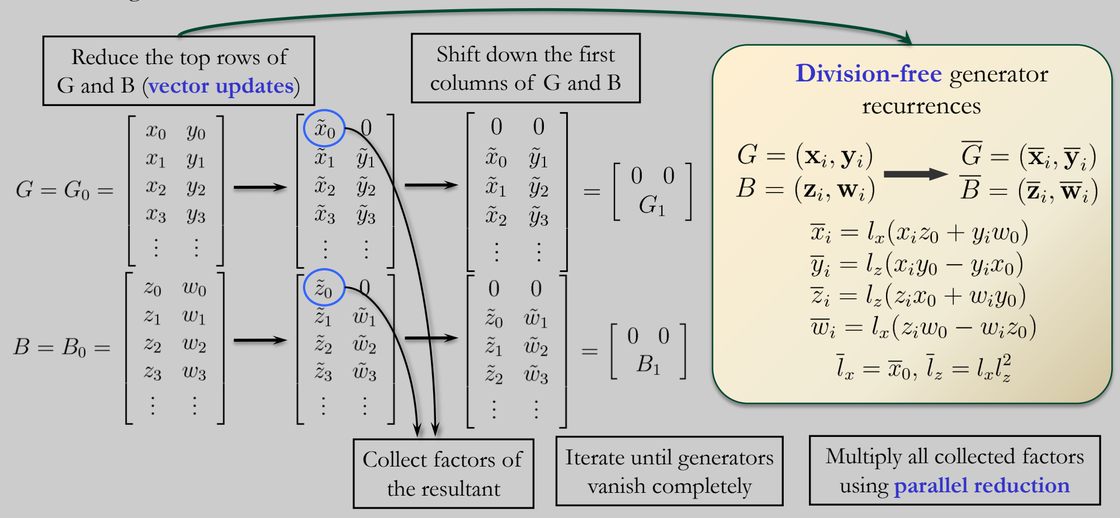
Computing univariate resultants over a prime field

Computation of the resultant reduces to triangular factorization of Sylvester matrix S which is shift-structured [2]:

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

$Z\in\mathbb{Z}^{n\times n}$ is a down-shift matrix $\qquad G,B\in\mathbb{Z}^{n\times 2}$ are generator matrices

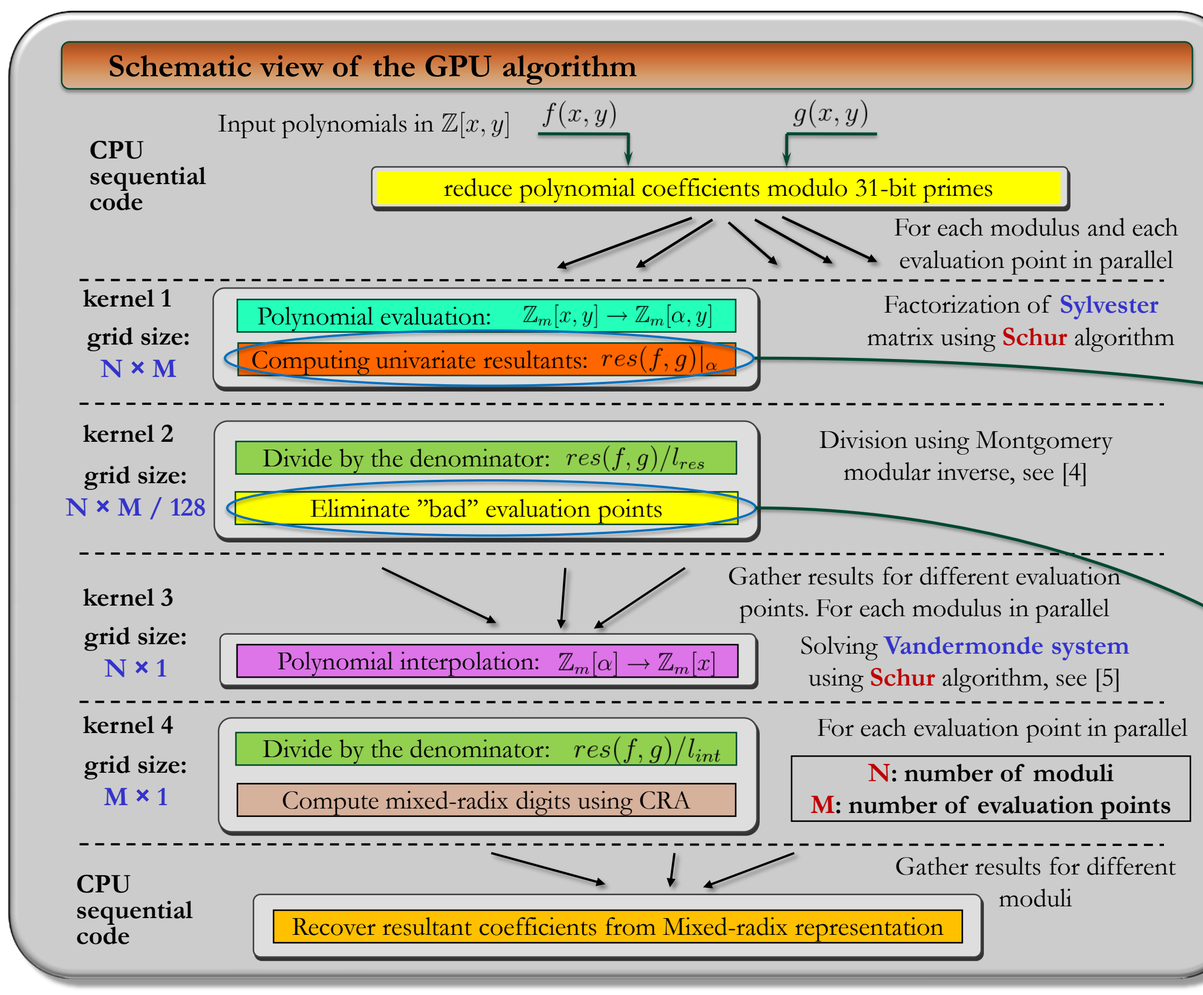
The **Generalized Schur Algorithm** computes matrix factorization in **O(n²)** time by operating solely on matrix generators:

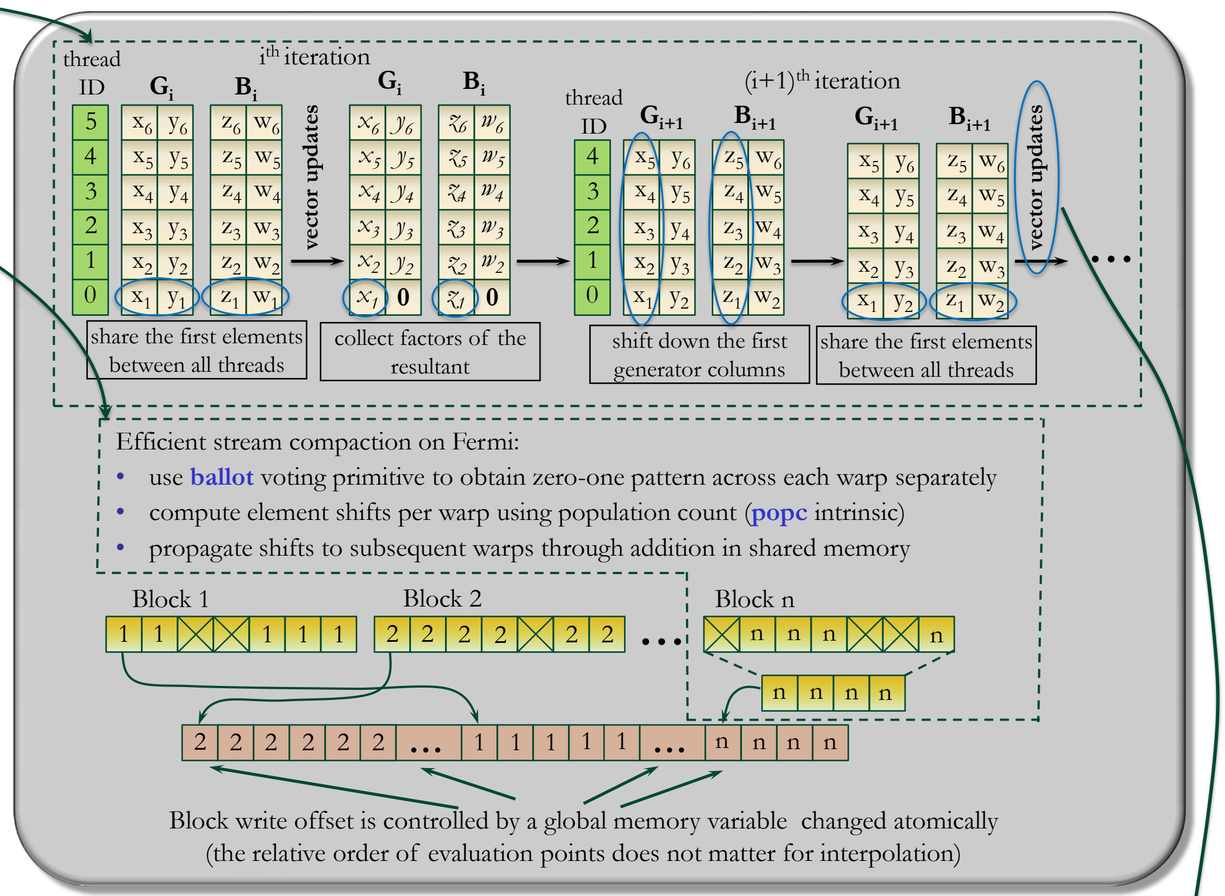


Polynomial interpolation over a prime field

Reduce the problem to solving the Vandermonde system using the generalized Schur algorithm in **O(n²)** time (see [5]): $V\mathbf{a}=\mathbf{y}\quad V_{ij}=x_i^j$

Schematic view of the GPU algorithm





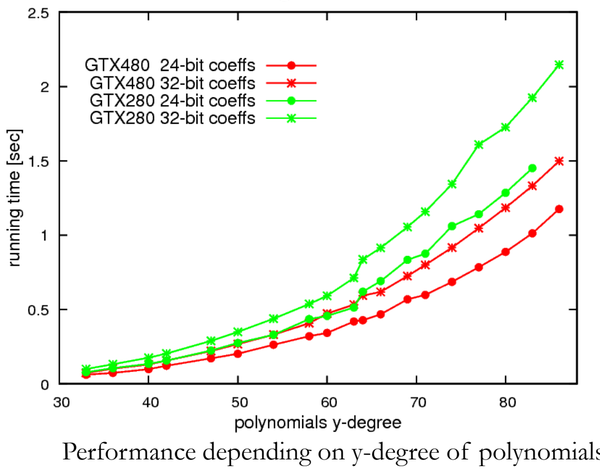
Performance comparison with the resultant algorithm from 32-bit Maple 13 (deterministic)

Target graphics card: **GeForce GTX480** Host machine: **Dual-Core AMD Opteron 2220SE**, Linux platform

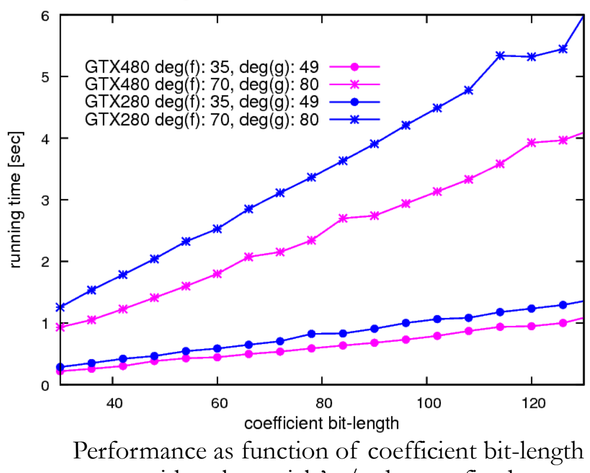
Instance	Maple time	GPU time	CUDA blocks executed	Instance	Maple time	GPU time	CUDA blocks executed
$\deg_x$f: **40** $\deg_x$g: **39** $\deg_y$f: **19** $\deg_y$g: **17** bits: **32** **dense**	**12.2 s**	**0.057 s**	**56 × 1372** **32×2 threads**	$\deg_x$f: **42** $\deg_x$g: **33** $\deg_y$f: **31** $\deg_y$g: **20** bits: **32** **dense**	**101.2 s**	**0.48 s**	**223 × 1874** **64 threads**
$\deg_x$f: **36** $\deg_x$g: **42** $\deg_y$f: **19** $\deg_y$g: **17** bits: **320** **dense**	**114.4 s**	**0.781 s**	**488× 1353** **32×2 threads**	$\deg_x$f: **10** $\deg_x$g: **7** $\deg_y$f: **95** $\deg_y$g: **93** bits: **16** **sparse**	**157.8 s**	**1.24 s**	**206 × 1604** **96 threads**
$\deg_x$f: **40** $\deg_x$g: **30** $\deg_y$f: **31** $\deg_y$g: **20** bits: **100** **sparse**	**56.7 s**	**0.4 s**	**215 × 1740** **32×2 threads**	$\deg_x$f: **10** $\deg_x$g: **7** $\deg_y$f: **95** $\deg_y$g: **93** bits: **120** **dense**	**timed out (> 15 min)**	**6.35 s**	**951 × 1604** **96 threads**

$\deg_{x/y}$ – degrees in x/y of polynomials **f** and **g**; **bits** – coefficient bit-length; **sparse/dense** – varying density of polynomials; **CUDA blocks executed:** # of blocks run by **1st** resultant kernel (**N × M**) and # of threads per block

GTX280 using 24-bit modular arithmetic **vs** **GTX480** using 31-bit modular arithmetic



Performance depending on y-degree of polynomials with coefficients bit-length fixed



Performance as function of coefficient bit-length with polynomials' x/y-degrees fixed

Realization of 31-bit modular arithmetic on the GPU

NVidia Fermi architectural features:

- native 32-bit integer multiplication support (instead of 24-bit multiplication on GT200)
- full-speed double-precision arithmetic (**8x** faster than that of GT200)
- modulo operation ('%') is costly: implement modular reduction in floating-point
- new set of video instructions: can do several arithmetic operations at a time (PTX assembly [3])

Modular multiply-add: $(a+b\cdot c) \bmod m$

```
// D2I_TRUNC = (double)3^51 (fast mantissa truncation)
// inv_m = (double)1 / m
double f = (double)b * (double)c * inv_m + D2I_TRUNC;
unsigned s = b * c - __double2loint(f) * m;
// equivalent to min(s, s + m)
asm volatile("vadd.u32.u32.u32.min %0,%1,%2,%3;" :
             "=r"(s) : "r"(s), "r"(m), "r"(s));
s += a;
// equivalent to min(s, s - m)
asm volatile("vsub.u32.u32.u32.min %0,%1,%2,%3;" :
             "=r"(s) : "r"(s), "r"(m), "r"(s));
return s;
```

Vector updates: $(a\cdot b-c\cdot d) \bmod m$

```
// inv_m = (double)1 / m
double f1 = (double)a * (double)b;
double f2 = (double)c * (double)d;
double f = (f1 - f2) * inv_m;
unsigned r = (unsigned)__double2int_rd(f);
unsigned s = a * b - c * d - r * m;
// equivalent to min(s, s + m)
asm volatile("vadd.u32.u32.u32.min %0,%1,%2,%3;" :
             "=r"(s) : "r"(s), "r"(m), "r"(s));
// equivalent to min(s, s - m)
asm volatile("vsub.u32.u32.u32.min %0,%1,%2,%3;" :
             "=r"(s) : "r"(s), "r"(m), "r"(s));
return s;
```

Raw performance: up to **154 GMad/s** on the GTX480 graphics processor.

GMad/s = 10^9 modular multiply-adds per second.

For comparison: 2.5GHz Quad-Core Xeon E5420 can do about **1 GMad/s** per core.

References:

[1] Collins G.E.: "The calculation of multivariate polynomial resultants", SYMSAC'71, 1971, 212-2

[2] Kailath T. and Sayed A.: "Displacement structure: theory and applications", SIAM review, 1995, 297–386

[3] PTX: Parallel Thread Execution. ISA Version 2.1. NVIDIA Corp., 2010

[4] Emeliyanenko P.: "Modular Resultant Algorithm for Graphics Processors", ICA3PP'10, 2010, 427-440

[5] Emeliyanenko P.: "A complete modular resultant algorithm targeted for realization on graphics hardware", PASCO'10, 2010, 35-43

MAX-PLANCK-GESELLSCHAFT

max planck institut informatik