Evaluating polynomials in several variables and their derivatives on a GPU computing processor

Jan Verschelde
joint work with Genady Yoffe

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
jan@math.uic.edu

The 26th Parallel and Distributed Processing Symposium (IPDPS-12)
The 13th IEEE international Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12)
21-25 May 2012, Shanghai, China

Outline

1. Problem Statement
   - evaluating and differentiating polynomials in several variables
   - quad double arithmetic on a graphics compute processor
2. Massively Parallel Polynomial Evaluation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing monomial products from powers of variables
   - evaluating and differentiating products of variables
3. Computational Experiments
   - regularity assumptions on the input data
   - computational results with the Tesla C2050

problem statement

A polynomial in n variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ consists of a vector of nonzero complex coefficients with corresponding exponents in A:

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad c \in \mathbb{C} \setminus \{0\}, \quad \mathbf{x}^{\mathbf{a}} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}.$$

Given is a system $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ and some point $\mathbf{z} \in \mathbb{C}^n$.

The problem is to evaluate $\mathbf{f}$ and its Jacobian matrix $J_{\mathbf{f}}$ at $\mathbf{z}$,
i.e.: to compute the vector $\mathbf{f}(\mathbf{z})$ and the matrix $J_{\mathbf{f}}(\mathbf{z})$.

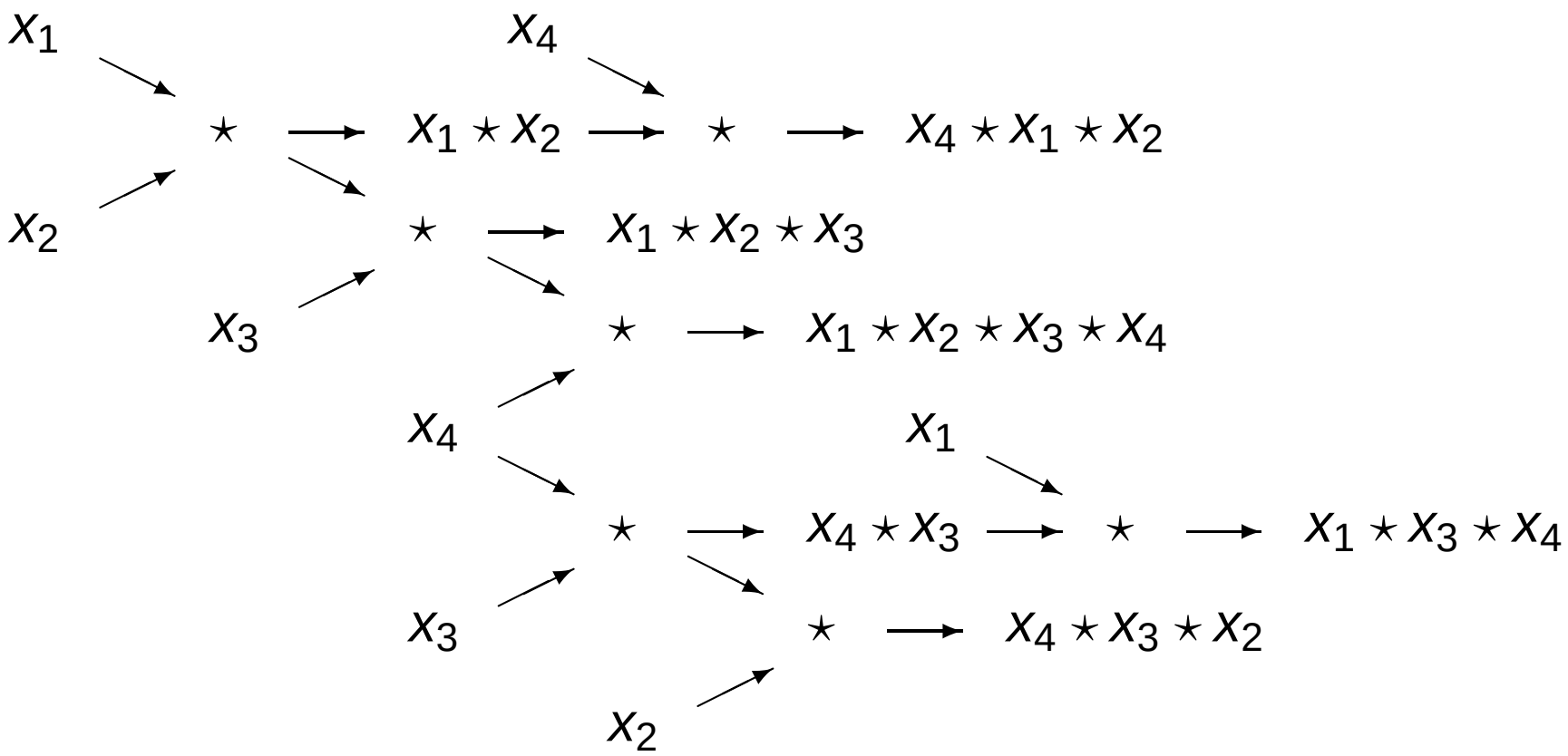
For large polynomial systems in many variables and high degrees:

- the cost of polynomial evaluation and differentiation often dominates the linear algebra of Newton's method; and
- the double precision as available in standard hardware is often insufficient to guarantee accurate results.

Goal: offset the extra cost of extended precision by parallel computing.

an arithmetic network for $x_1 \star x_2 \star x_3 \star x_4$

In *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation* by Griewank and Walther, 2nd edition, SIAM 2008, a product of variables is named Speelpenning's product.



Evaluating $x_1 \star x_2 \star \cdots \star x_n$ and its gradient takes $3n - 5$ multiplications.

quad double precision

A quad double is an unevaluated sum of 4 doubles,
improves working precision from 2.2×10^{-16} to 2.4×10^{-63}.

- Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In the *15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001. Software at `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.9.tar.gz`.

Predictable overhead: working with `double double` is of the same cost as working with complex numbers. Simple memory management.

The QD library has been ported to the GPU by

- M. Lu, B. He, and Q. Luo: **Supporting extended precision on graphics processors.** In the *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)*, pages 19–26, 2010.
  Software at `http://code.google.com/p/gpuprec/`.

computers and compilers

Hardware:

- HP Z800 workstation running Red Hat Enterprise Linux 6.1
  The CPU is an Intel Xeon X5690 at 3.47 Ghz.
- The processor clock of the NVIDIA Tesla C2050 Computing Processor runs at 1147 Mhz. The graphics card has 14 multiprocessors, each with 32 cores, for a total of 448 cores.

As the clock speed of the GPU is a third of the clock speed of the CPU, we hope to achieve a double digit speedup.

Compilers:

- Code written in C++ using `gcc` version 4.4.6.
- NVIDIA CUDA compiler driver `nvcc`, release 4.0, V0.2.1221.

monomial evaluation and differentiation

Polynomials are linear combinations of monomials $\mathbf{x}^{\mathbf{a}} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}$.

Separating monomial products from products of variables:

$$\mathbf{x}^{\mathbf{a}} = \left(x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1} \right) \star \left(x_{j_1} x_{j_2} \cdots x_{j_\ell} \right),$$

for $a_{i_m} \geq 1$, $m = 1, 2, \ldots, k$, $1 \leq i_1 < i_2 < \cdots < i_k \leq n$,
and $1 \leq j_1 < j_2 < \cdots < j_\ell \leq n$, with $\ell \geq k$.

Evaluating and differentiating $\mathbf{x}^{\mathbf{a}}$ in three steps:

1. compute the monomial products $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$
2. compute $x_{j_1} x_{j_2} \cdots x_{j_\ell}$ and its gradient
3. multiply the evaluated $x_{j_1} x_{j_2} \cdots x_{j_\ell}$ and its gradient with the evaluated monomial products

computing monomial products $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$

To evaluate $x_1^3 x_2^7 x_3^2$ and its derivatives, we first evaluate the factor $x_1^2 x_2^6 x_3$ and then multiply this factor with all derivatives of $x_1 x_2 x_3$.

Because $x_1^2 x_2^6 x_3$ is common to the evaluated monomial and all its derivatives, we call $x_1^2 x_2^6 x_3$ a *common factor*.

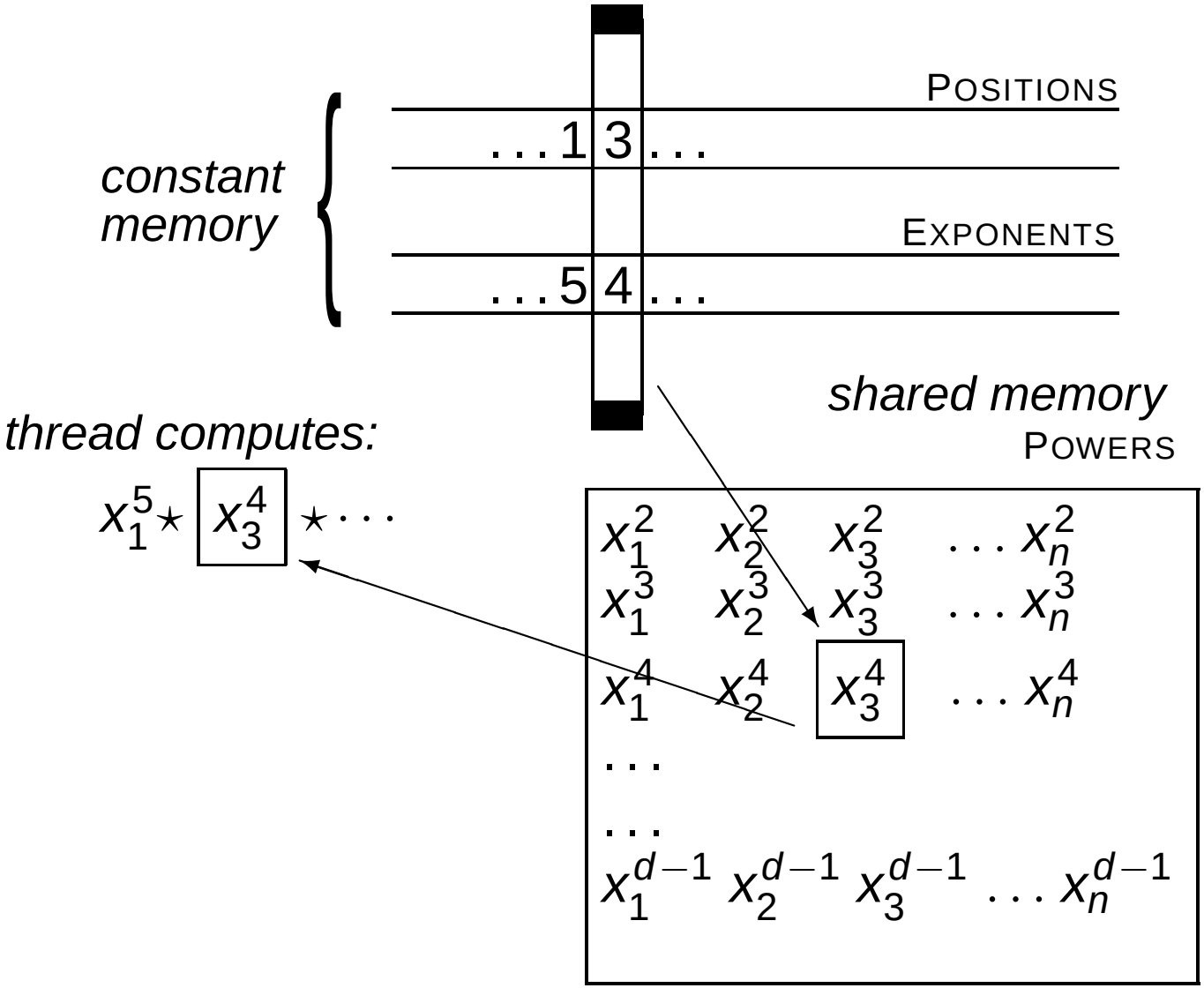
The kernel to compute common factors operates in two stages:

1. Each of the first n threads of a thread block computes sequentially powers from the 2nd to the $(d-1)$th of one of the n variables.
2. Each of the threads of a block computes a common factor for one of the monomials of the system, as a product of k quantities computed at the first stage of the kernel.

The precomputed powers of variables are stored in shared memory: the (i,j)th element stores x_j^i, minimizing bank conflicts.

The positions and exponents of variables in monomials are stored in two one dimensional arrays in constant memory.

common factor calculation



memory locations

we illustrate the work done by *one* thread

To compute the derivatives of $s = x_1x_2x_3x_4$,

- Q stores the backward product, and
- the ith partial derivative of S is stored in memory location L_i.

L_1	L_2	L_3	L_4	Q
	x_1			
	x_1	$x_1 \star x_2$		
	x_1	x_1x_2	$(x_1x_2) \star x_3$	
	x_1	$(x_1x_2) \star x_4$	$x_1x_2x_3$	x_4
	$x_1 \star (x_3x_4)$	$x_1x_2x_4$	$x_1x_2x_3$	$x_4 \star x_3$
$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$	$(x_4x_3) \star x_2$
$\frac{\partial s}{\partial x_1}$	$\frac{\partial s}{\partial x_2}$	$\frac{\partial s}{\partial x_3}$	$\frac{\partial s}{\partial x_4}$	

Only explicitly performed multiplications are marked by a star $\star$.

the example continued

Given $s = x_1x_2x_3x_4$ and its gradient, with $\alpha = x_1^2x_2^6x_3^3x_4^4$ we evaluate $\beta = c\,x_1^3x_2^7x_3^4x_4^5$ and its derivatives, denoting $\gamma = \frac{1}{c}\beta = x_1^3x_2^7x_3^4x_4^5$.

L_1	L_2	L_3	L_4	L_5
$\frac{\partial s}{\partial x_1} \star \alpha$	$\frac{\partial s}{\partial x_2} \star \alpha$	$\frac{\partial s}{\partial x_3} \star \alpha$	$\frac{\partial s}{\partial x_4} \star \alpha$	
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4} \star x_4$
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	γ
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1} \star (3c)$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2} \star (7c)$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3} \star (4c)$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4} \star (5c)$	$\gamma \star c$
$\frac{\partial \beta}{\partial x_1}$	$\frac{\partial \beta}{\partial x_2}$	$\frac{\partial \beta}{\partial x_3}$	$\frac{\partial \beta}{\partial x_4}$	β

Note that the coefficients $(3c), (7c), (4c), (5c)$ are precomputed.
Only explicitly performed multiplications are marked by a star $\star$.

regularity assumptions on the input data

Graphics compute processors exploit data parallelism.

Every thread evaluates and differentiates one monomial.

- On the one hand, to keep all 14 multiprocessors occupied about 1,000 monomials are needed.
- On the other hand, as monomials are stored as positions and exponents in constant memory, the 65,536 bytes of constant memory impose an upper bound on the number of monomials.

Let n be the number of polynomials in the system,
m be the number of monomials per polynomial,
k be the number of variables per monomial,
using one byte for a position and one byte for an exponent,
then we need $n \times m \times k \times 2$ bytes.

As examples, we take $n = m$ between 30 and 40, and $k = n/2$.

limits of shared memory capacity

With double double precision coefficients, dimension 70 is okay.

1. $(n/2+1) \times 2 \times \text{sizeof(double double)} \leq (70/2+1) \times 2 \times 16 = 1,152$ bytes in shared memory. To handle 32 momomials by a block of 32 threads we would need then at most

$$32 \times 1,152 = 36,864 \text{ bytes of shared memory.}$$

2. For storing values of the variable we would need

$$n \times \text{sizeof(complex double double)} \leq$$

$$70 \times 2 \times \text{ sizeof(double double)} = 70 \times 2 \times 16 = 2,240.$$

3. Allocation both spaces in shared memory leaves $(49,152 - (36,864 + 2,240)) > 10,000$ bytes of shared memory.

computational experiments

We generate a system with random complex coefficients:

- a system of 32 polynomials,
- each monomial has 9 variables with nonzero power of at most 2,
- a varying number of monomials per polynomial: 22, 32, and 48 lead to 704, 1024, and 1536 monomials in the system.

Wall clock times and speedups for 100,000 evaluations:

#monomials	Tesla C2050	1 CPU core	speedup
704	14.514 sec	1min 50.9 sec	7.60
1024	15.265 sec	2min 39.3 sec	10.44
1536	17.000 sec	3min 58.7 sec	14.04

At least 1000 monomials are needed for a modest speedup.

monomials of higher degrees

We generate a system with random complex coefficients:

- a system of 32 polynomials,
- each monomial has 16 variables with nonzero power ≤ 10,
- a varying number of monomials per polynomial: 22, 32, and 48 lead to 704, 1024, and 1536 monomials in the system.

Wall clock times and speedups for 100,000 evaluations:

#monomials	Tesla C2050	1 CPU core	speedup
704	19.068 sec	3min 16.9 sec	10.33
1024	20.800 sec	4min 43.3 sec	13.62
1536	21.763 sec	7min 05.8 sec	19.56

With higher degrees, we obtain higher speedups.

conclusions

We obtained modest speedups with our first code for the evaluation and differentiation of a polynomial system and its Jacobian matrix.

On randomly generated systems, preliminary experiments show that

- for good occupancy at least 1000 monomials are needed,
- the size of constant memory limits more than 2000 monomials,
- speedups increase with higher degrees.

Ongoing and future work includes

- quality up factors with double double and quad double precision,
- adding a linear solver on the GPU implements Newton's method,
- integration in the polynomial system solver of PHCpack.