On Massively Parallel Algorithms to Track One Path of a Polynomial Homotopy

Jan Verschelde
joint with Genady Yoffe and Xiangcheng Yu

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

2013 SIAM Conference on Applied Algebraic Geometry
Minisymposium on Algorithms in Numerical Algebraic Geometry
1-4 August 2013, Fort Collins, Colorado

Outline

1. Problem Statement
   - path tracking on a Graphics Processing Unit (GPU)
2. Massively Parallel Polynomial Evaluation and Differentiation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing the common factor of a monomial and its gradient
   - evaluating and differentiating products of variables
3. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - cost and accuracy of the modified Gram-Schmidt method
   - defining the kernels
   - occupancy of multiprocessors and resource usage
4. Computational Results
   - comparing speedup and quality up
   - simulating the tracking of one path

on Path Tracking on a GPU

1. Problem Statement
   - path tracking on a Graphics Processing Unit (GPU)
2. Massively Parallel Polynomial Evaluation and Differentiation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing the common factor of a monomial and its gradient
   - evaluating and differentiating products of variables
3. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - cost and accuracy of the modified Gram-Schmidt method
   - defining the kernels
   - occupancy of multiprocessors and resource usage
4. Computational Results
   - comparing speedup and quality up
   - simulating the tracking of one path

path tracking on a GPU

Tracking one path requires several thousands of Newton corrections.

Two computational tasks in Newton's method for $f(\mathbf{x}) = \mathbf{0}$:

1. evaluate the system f and its Jacobian matrix J_f at $\mathbf{z}$;
2. solve the linear system $J_f(\mathbf{z})\Delta\mathbf{z} = -f(\mathbf{z})$, and do $\mathbf{z} := \mathbf{z} + \Delta\mathbf{z}$.

Problem: high degrees lead to extremal values in J_f.

Double precision is insufficient to obtain accurate results.

Data parallelism in system evaluation and differentiation is achieved through many products of variables in the monomials.

Solving linear systems with least squares using a QR decomposition

- is more accurate than a LU factorization, and
- applies to overdetermined problems (Gauss-Newton).

Quality up: compensate cost of multiprecision with parallel algorithms.

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/`.

personal supercomputers

Computer with NVDIA Tesla C2050:

- HP Z800 workstation running Red Hat Enterprise Linux 6.4
  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.

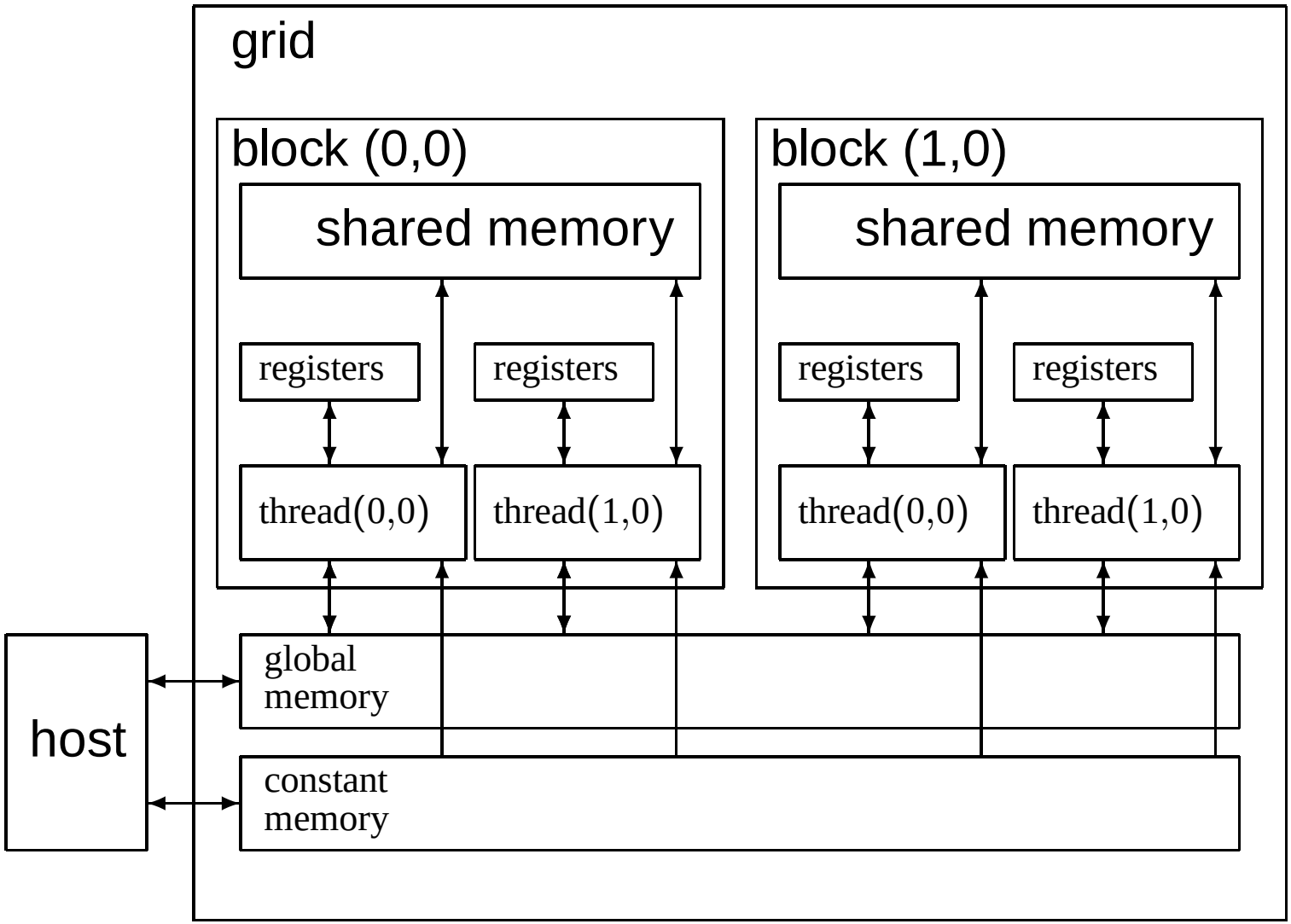
As the clock speed of the GPU is a third of the clock speed of the CPU, we hope to achieve a double digit speedup.

Computer with NVIDIA Tesla K20C:

- Microway RHEL workstation with Intel Xeon E5-2670 at 2.6 Ghz.
- The NVIDIA Tesla K20C has 2,496 cores (13×192) at a clock speed of 706 Mhz. The peak double precision performance of 1.17 teraflops is twice of that of the C2050.

Massively parallel means: launch *ten thousands threads*.

CUDA device memory types



preliminary results

papers with Genady Yoffe:

- *Evaluating polynomials in several variables and their derivatives on a GPU computing processor.*
  In the Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2012), pages 1391-1399. IEEE Computer Society, 2012.
- *Orthogonalization on a General Purpose Graphics Processing Unit with Double Double and Quad Double Arithmetic.*
  In the Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2013), pages 1373-1380. IEEE Computer Society, 2013.

With regularity assumptions on the input and for large enough dimensions, GPU acceleration can compensate for the overhead of one extra level of precision.

on Path Tracking on a GPU

1. Problem Statement
   - path tracking on a Graphics Processing Unit (GPU)
2. Massively Parallel Polynomial Evaluation and Differentiation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing the common factor of a monomial and its gradient
   - evaluating and differentiating products of variables
3. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - cost and accuracy of the modified Gram-Schmidt method
   - defining the kernels
   - occupancy of multiprocessors and resource usage
4. Computational Results
   - comparing speedup and quality up
   - simulating the tracking of one path

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 the product of variables from the monomial:

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

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 common factor $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$
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 common factor

computing common factors $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$

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*.

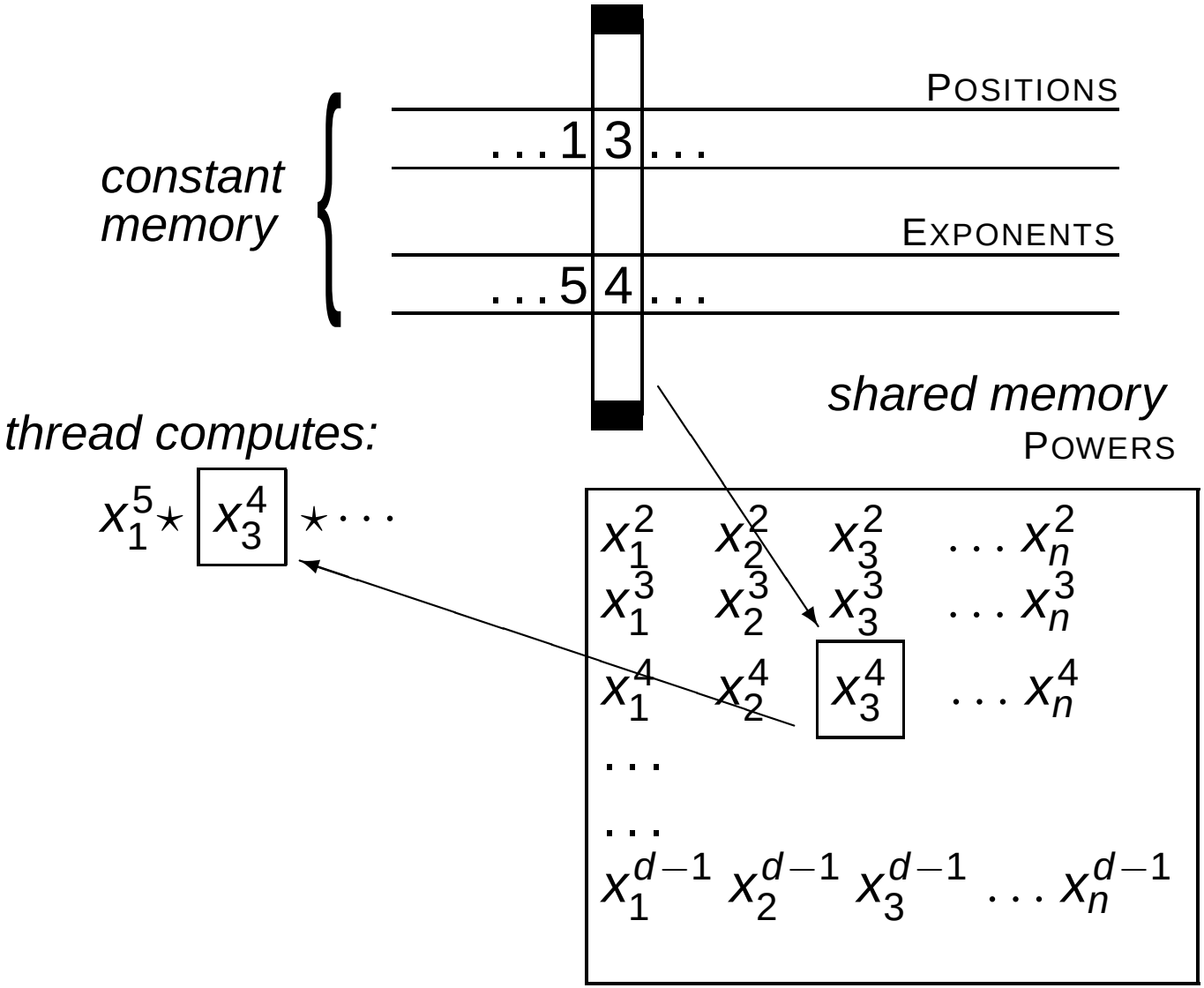
The kernel to compute common factors operates in two stages:

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 $\star$.

relaxing regularity assumptions

Separate threads evaluate and differentiate products of variables:

$$\overbrace{x_0x_1x_2x_3}^{f_0} \cdot \overbrace{x_4x_5x_6x_7}^{f_1} \cdot \overbrace{x_8x_9x_Ax_B}^{f_2} \cdot \overbrace{x_Cx_Dx_Ex_F}^{f_3}$$

$x_1x_2x_3$	$x_5x_6x_7$	$x_9x_Ax_B$	$x_Dx_Ex_F$
$x_0x_2x_3$	$x_4x_6x_7$	$x_8x_Ax_B$	$x_Cx_Ex_F$
$x_0x_1x_3$	$x_4x_5x_7$	$x_8x_9x_B$	$x_Cx_Dx_F$
$x_0x_1x_2$	$x_4x_5x_6$	$x_8x_9x_A$	$x_Cx_Dx_E$

A variable occurs in at most one factor, e.g.: $\dfrac{\partial}{\partial x_0}(f_0f_1f_2f_3) = \dfrac{\partial f_0}{\partial x_0} f_1f_2f_3.$

Multiply all derivatives

of f_0 by $f_1f_2f_3$
of f_1 by $f_0f_2f_3$
of f_2 by $f_0f_1f_3$
of f_3 by $f_0f_1f_2$

$\Rightarrow$ Run same algorithm with variables $x_0 = f_0, x_1 = f_1, x_2 = f_2,$ and $x_3 = f_3$.

on Path Tracking on a GPU

1. Problem Statement
   - path tracking on a Graphics Processing Unit (GPU)
2. Massively Parallel Polynomial Evaluation and Differentiation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing the common factor of a monomial and its gradient
   - evaluating and differentiating products of variables
3. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - cost and accuracy of the modified Gram-Schmidt method
   - defining the kernels
   - occupancy of multiprocessors and resource usage
4. Computational Results
   - comparing speedup and quality up
   - simulating the tracking of one path

the modified Gram-Schmidt method

Input: $A \in \mathbb{C}^{m \times n}$.
Output: $Q \in \mathbb{C}^{m \times n}$, $R \in \mathbb{C}^{n \times n}$: $Q^H Q = I$,
R is upper triangular, and $A = QR$.

let $\mathbf{a}_k$ be column k of A
for k from 1 to n do
  $r_{kk} := \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$
  $\mathbf{q}_k := \mathbf{a}_k / r_{kk}$, $\mathbf{q}_k$ is column k of Q
  for j from $k+1$ to n do
    $r_{kj} := \mathbf{q}_k^H \mathbf{a}_j$
    $\mathbf{a}_j := \mathbf{a}_j - r_{kj} \mathbf{q}_k$

Number of arithmetical operations: $2mn^2$.
With $A = QR$, we solve $A\mathbf{x} = \mathbf{b}$ as $R\mathbf{x} = Q^H \mathbf{b}$, minimizing $||\mathbf{b} - A\mathbf{x}||_2^2$.

the cost of multiprecision arithmetic

User CPU times for 10,000 QR decompositions with $n = m = 32$:

precision	CPU time	factor
double	3.7 sec	1.0
complex double	26.8 sec	7.2
complex double double	291.5 sec	78.8
complex quad double	2916.8 sec	788.3

Taking the cubed roots of the factors $7.2^{1/3} \approx 1.931$, $78.8^{1/3} \approx 4.287$, $788.3^{1/3} \approx 9.238$, the cost of using multiprecision is equivalent to using double arithmetic, after multiplying the dimension 32 of the problem respectively by the factors 1.931, 4.287, and 9.238, which then yields respectively 62, 134, and 296.

Orthogonalizing 32 vectors in $\mathbb{C}^{32}$ in quad double arithmetic has the same cost as orthogonalizing 296 vectors in $\mathbb{R}^{296}$ with doubles.

measuring the accuracy

$$\text{Consider } e = ||A - QR||_1 = \max_{\substack{i=1,2,\ldots,m \\ j=1,2,\ldots,n}} \left| a_{ij} - \sum_{\ell=1}^{n} q_{i\ell} r_{\ell j} \right|.$$

For numbers in $[10^{-g}, 10^{+g}]$, let $m_e = \min(\log_{10}(e))$, $M_e = \max(\log_{10}(e))$, and $D_e = m_e - M_e$.

	complex double			complex double double		
g	m_e	M_e	D_e	m_e	M_e	D_e
1	-14.5	-14.0	0.5	-30.6	-30.1	0.5
4	-11.7	-11.0	0.7	-27.8	-27.1	0.7
8	-7.8	-7.0	0.8	-24.0	-23.1	1.0
12	-3.9	-3.1	0.8	-20.1	-19.2	0.9
16	-0.2	1.0	1.2	-16.4	-15.1	1.3

	complex double double			complex quad double		
g	m_e	M_e	D_e	m_e	M_e	D_e
17	-15.5	-14.1	1.3	-48.1	-47.1	1.0
20	-12.6	-11.1	1.5	-45.1	-44.2	0.9
24	-8.8	-7.2	1.6	-41.3	-40.2	1.2
28	-4.7	-3.2	1.5	-37.7	-36.1	1.6
32	-1.0	0.8	1.9	-33.9	-32.2	1.8

parallel modified Gram-Schmidt orthogonalization

Input: $A \in \mathbb{C}^{m \times n}$, $A = [\mathbf{a}_1\ \mathbf{a}_2\ \ldots\ \mathbf{a}_n]$,
$\mathbf{a}_k \in \mathbb{C}^m$, $k = 1, 2, \ldots, n$.
Output: $A \in \mathbb{C}^{m \times n}$, $A^H A = I$ (i.e.: $A = Q$),
$R \in \mathbb{C}^{n \times n}$: $R = [r_{ij}]$, $r_{ij} \in \mathbb{C}$,
$i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$.
for k from 1 to $n - 1$ do
launch kernel `Normalize_Remove`(k)
with $(n - k)$ blocks of threads,
as the jth block (for all $j : k < j \leq n$)
normalizes $\mathbf{a}_k$ as $\mathbf{q}_k := \mathbf{a}_k / \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$
and removes the component of $\mathbf{a}_j$ as $\mathbf{a}_j := \mathbf{a}_j - (\mathbf{q}_k^H \mathbf{a}_j)\mathbf{q}_k$
launch kernel `Normalize`(n) with one
thread block to normalize $\mathbf{a}_n$.

occupancy of the multiprocessors

The Tesla C2050 has 448 cores, with $448 = 14 \times 32$:
14 multiprocessors with each 32 cores.

For dimension 32, the orthogonalization launches the kernel `Normalize_Remove()` 31 times:

- while first 7 of these launches employ 4 multiprocessors,
- launches from 8 to 15 employ 3 multiprocessors,
- launches 16 to 23 employ 2 multiprocessors,
- and finally launches 24 to 31 employ only one multiprocessor.

Earlier stages of the algorithm are responsible for the speedups.

computing inner products

In computing $\mathbf{x}^H\mathbf{y}$ the products $\overline{x}_\ell \star y_\ell$ are independent of each other.

The inner product $\mathbf{x}^H\mathbf{y}$ is computed in two stages:

1. All threads work independently in parallel: thread ℓ calculates $\overline{x}_\ell \star y_\ell$ where the operation $\star$ is a complex double, a complex double double, or a complex quad double multiplication.

   Afterwards, all threads in the block are synchronized.

2. The application of a reduction to sum the elements in $(\overline{x}_1 y_1, \overline{x}_2 y_2, \ldots, \overline{x}_m y_m)$ and compute $\overline{x}_1 y_1 + \overline{x}_2 y_2 + \cdots + \overline{x}_m y_m$.

   The $+$ in the sum above corresponds to the $\star$ in the item above and is a complex double, a complex double double, or a complex quad double addition. There are $\log_2(m)$ steps but if m equals the warp size, there is thread divergence in every step.

shared memory locations

Shared memory is fast memory shared by all threads in one block.

$r_{kj} := \mathbf{q}_k^H \mathbf{a}_j$ is inner product of two m-vectors:

$$\begin{matrix} \mathbf{q}_k \\ \begin{bmatrix} q_{k1} \\ q_{k2} \\ \vdots \\ q_{km} \end{bmatrix} \end{matrix} \quad \begin{matrix} \mathbf{a}_j \\ \begin{bmatrix} a_{j1} \\ a_{j2} \\ \vdots \\ a_{jm} \end{bmatrix} \end{matrix} \quad \begin{bmatrix} \bar{q}_{k1} \star a_{j1} \\ \bar{q}_{k2} \star a_{j2} \\ \vdots \\ \bar{q}_{km} \star a_{jm} \end{bmatrix}$$

Thread t computes $\bar{q}_{kt} \star a_{jt}$.

If we may override $\mathbf{q}_k$, then $2m$ shared memory locations suffice, but we still need $\mathbf{q}_k$ for $\mathbf{a}_j := \mathbf{a}_j - r_{kj}\mathbf{q}_k$.

We need $3m$ shared memory locations to perform the reductions.

the orthonormalization stage

After computing $\mathbf{a}_k^H \mathbf{a}_k$, the orthonormalization stage consists of

- one square root computation,
- followed by m division operations.

The first thread of a block performs $r_{kk} := \sqrt{\mathbf{a}_k^H \mathbf{a}_k}$.

After a synchronization, the m threads independently perform in-place divisions $a_{k\ell} := a_{k\ell}/r_{kk}$, for $\ell = 1, 2, \ldots, m$ to compute $\mathbf{q}_k$.

Increasing the precision,

- we expect an increased parallelism as the cost for the arithmetic increased and each thread does more work independently.
- Unfortunately, also the cost for the square root calculation — executed in isolation by the first thread in each block — also increases.

parallel back substitution

Solving $R\mathbf{x} = Q^H\mathbf{b}$:

Input: $R \in \mathbb{C}^{n \times n}$, an upper triangular matrix,
$\mathbf{y} \in \mathbb{C}^n$, the right hand side vector.
Output: $\mathbf{x}$ is the solution of $R\mathbf{x} = \mathbf{y}$.
for k from n down to 1 do
  thread k does $x_k := y_k / r_{kk}$
  for j from 1 to $k-1$ do
    thread j does $y_j := y_j - r_{jk} \star x_k$

Only one block of threads executes this code.

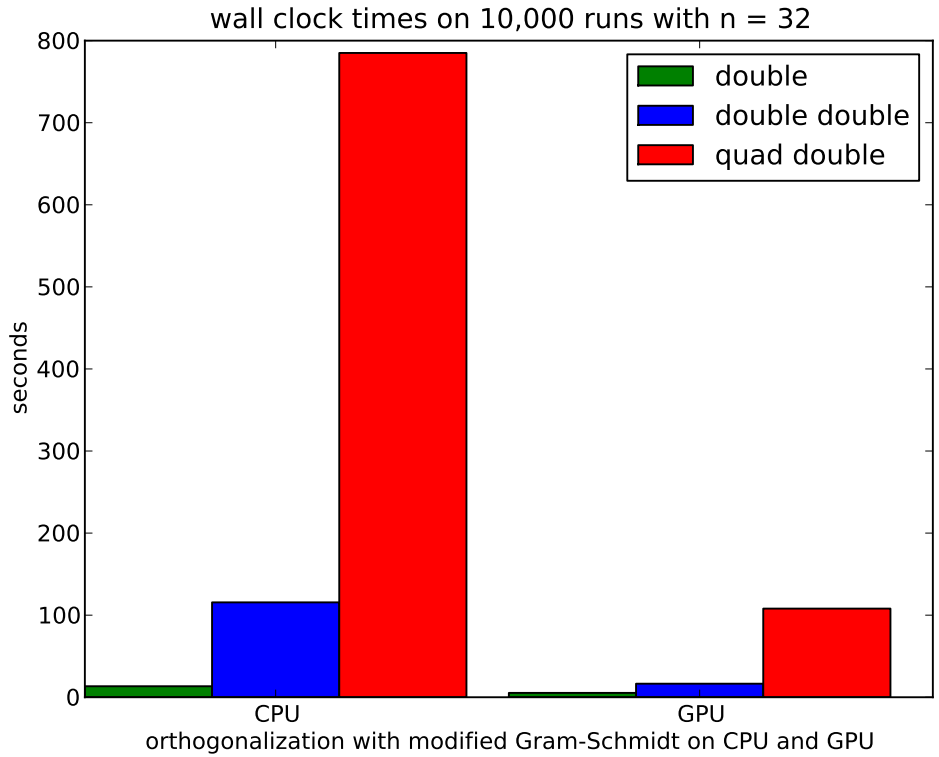
on Path Tracking on a GPU

1. Problem Statement
   - path tracking on a Graphics Processing Unit (GPU)
2. Massively Parallel Polynomial Evaluation and Differentiation
   - stages in the evaluation of a system and its Jacobian matrix
   - computing the common factor of a monomial and its gradient
   - evaluating and differentiating products of variables
3. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - cost and accuracy of the modified Gram-Schmidt method
   - defining the kernels
   - occupancy of multiprocessors and resource usage
4. Computational Results
   - comparing speedup and quality up
   - simulating the tracking of one path

running at different precisions

Wall clock times and speedups for 10,000 orthogonalizations,
on 32 random complex vectors of dimension 32, times in seconds:

	3.47GHz CPU & C2050		
p	CPU	C2050	speedup
D	14.43	5.34	2.70
DD	122.34	14.29	8.56
QD	799.75	125.95	6.35
	2.60GHz CPU & K20C		
p	CPU	K20C	speedup
D	16.19	5.75	2.82
DD	149.69	17.10	8.75
QD	850.55	119.10	7.14



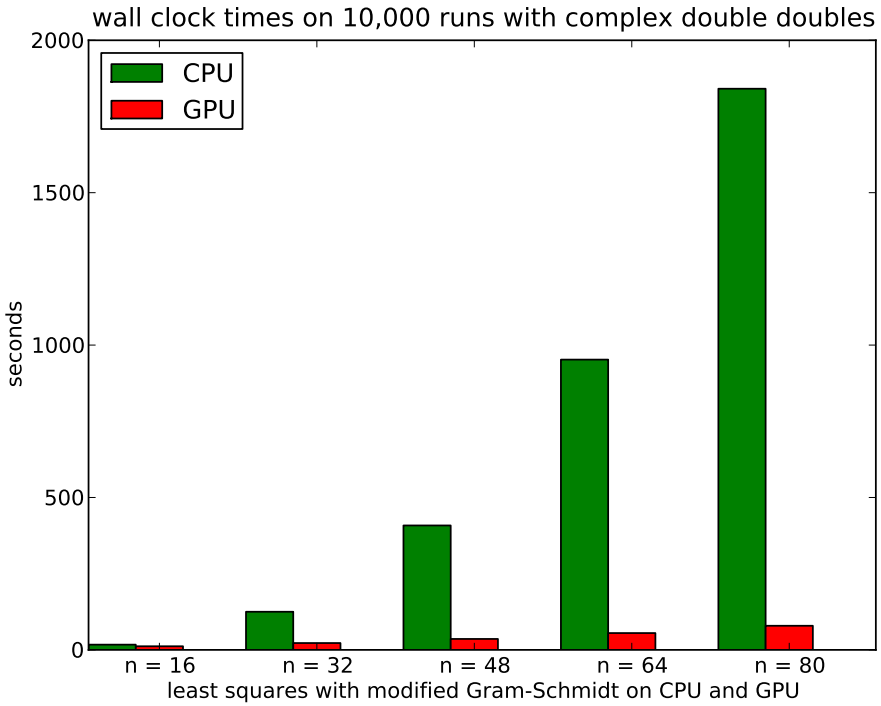
For quality up, compare

- the 122.34 seconds with complex double doubles on CPU;
- the 125.95 seconds with complex quad doubles on C2050.

running with double doubles and quad doubles

Wall clock times for 10,000 runs of the modified Gram-Schmidt method (each followed by one backsubstitution), on 3.47GHz CPU and C2050:

	complex double double		
n	CPU	GPU	speedup
16	17.17	11.85	1.45
32	125.06	22.44	5.57
48	408.20	35.88	11.38
64	952.35	55.18	17.26
80	1841.07	79.11	23.27
	complex quad double		
n	CPU	GPU	speedup
16	113.51	143.07	0.79
32	813.65	155.32	5.24
48	2556.36	266.55	9.59
64	6216.06	409.57	15.18
80	12000.15	597.47	20.08

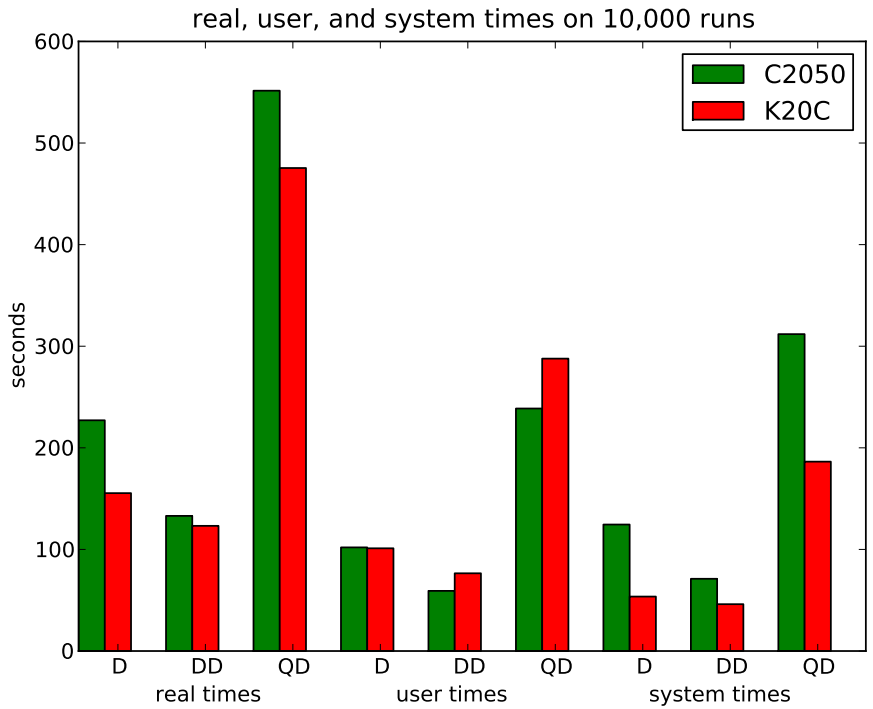


Double digit speedups for $n \geq 48$.

Comparing C2050 and K20C

Real, user, and system times (in seconds) for 10,000 orthogonalizations on dimensions $n = 256$, $n = 128$, and $n = 85$.

	D	DD	QD
n	256	128	85
real C2050	227.1	133.0	551.5
real K20C	155.4	123.2	475.4
speedup	1.461	1.079	1.160
user C2050	102.0	59.2	238.7
user K20C	101.1	76.5	287.8
sys C2050	124.5	71.1	311.9
sys K20C	53.6	46.1	186.4
speedup	2.323	1.542	1.673

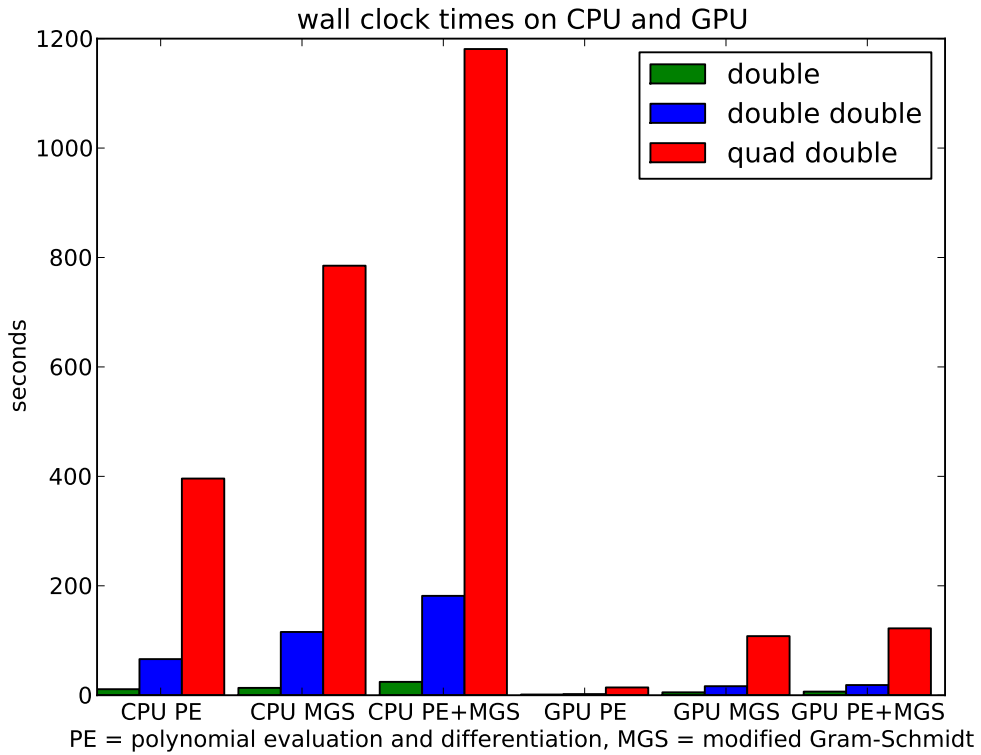


Because the host of the K20C runs at 2.60GHz, the frequency of the host of the C2050 was set to range between 2.60GHz and 2.66GHz.

simulating the tracking of one path

Consider 10,000 Newton corrections in dimension 32 on a system with 32 monomials per polynomial, 5 variables per monomial, degrees uniformly taken from $\{1, 2, 3, 4, 5\}$, for precision p.

p	CPU PE	GPU PE	speed up
D	11.0	1.3	8.5
DD	66.0	2.1	31.4
QD	396.0	14.2	27.9
p	MGS	MGS	
D	13.4	5.3	2.5
DD	115.6	16.5	7.0
QD	785.0	108.0	7.0
p	SUM	SUM	
D	24.4	6.6	3.7
DD	181.6	18.6	9.8
QD	1181.0	122.2	9.7



Quality up: GPU acceleration compensates for extra precision.

Summary

Some conclusions:

- While GPUs offer a theoretical teraflop peak performance, the problems must contain enough data parallelism to perform well.
- The fine granularity in algorithms to
  - evaluate and differentiate poynomials; and
  - orthogonalize and solve in the least squares sense

  leads to a massively parallel Gauss-Newton method.
- Already for modest dimensions we achieve quality up:
  GPU acceleration compensates for one level of extra precision.