GPU acceleration of Newton's method for large systems of polynomial equations in double double and quad double arithmetic

Jan Verschelde
joint work with Xiangcheng Yu

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

the 16th IEEE International Conference on High Performance Computing and Communications, 20-22 August 2014, Paris, France

Outline

GPU accelerated Newton's method

1. Solving Polynomial Systems Numerically
   - compensating for the cost of double doubles and quad doubles
   - how large? sizing up the problem dimensions
2. Polynomial System Evaluation and Differentiation
   - the reverse mode of algorithmic differentiation
   - arithmetic circuits
   - computational results
3. A Massively Parallel Modified Gram-Schmidt Method
   - implementing a right-looking algorithm
4. Computational Results
   - the Chandrasekhar H-Equation
   - the cyclic *n*-roots problem

solving polynomial systems numerically

Problem statement: when solving large polynomial systems, the hardware double precision may not be sufficient for accurate solutions.

Our goal: accelerate computations with general purpose Graphics Processing Units (GPUs) to compensate for the overhead caused by double double and quad double arithmetic.

Our first results (jointly with Genady Yoffe) on pursuing this goal with multicore computers are in the PASCO 2010 proceedings.

Our focus is on Newton's method:

1. To evaluate and differentiate the polynomial systems, we apply the reverse method of algorithmic differentiation.
2. To solve linear systems in the least squares sense, we apply the modified Gram-Schmidt method.

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/`.

GPU accelerated Newton's method

1. Solving Polynomial Systems Numerically
   - compensating for the cost of double doubles and quad doubles
   - how large? sizing up the problem dimensions
2. Polynomial System Evaluation and Differentiation
   - the reverse mode of algorithmic differentiation
   - arithmetic circuits
   - computational results
3. A Massively Parallel Modified Gram-Schmidt Method
   - implementing a right-looking algorithm
4. Computational Results
   - the Chandrasekhar H-Equation
   - the cyclic n-roots problem

about *large* polynomial systems: how large?

The number of isolated solutions and/or the degrees of positive dimensional solution sets may grow exponentially in the dimensions of the input: degrees of the equations and number of variables.

Because of the exponential growth of the output (e.g. 20 quadrics $\Rightarrow 2^{20}$ solutions), *global* solving is limited to rather small dimensions.

In ***local*** solving, we focus on ***one*** solution and may increase the dimension to several hundreds and even thousands.

One limitation is the storage needed for all exponent vectors and coefficients of the monomials.

Most systems arising in applications are *sparse*, the number of monomials grows modestly in the number of variables.

some related work in algorithmic differentiation

- M. Grabner, T. Pock, T. Gross, and B. Kainz. Automatic differentiation for GPU-accelerated 2D/3D registration. In *Advances in Automatic Differentiation*, pages 259–269. Springer, 2008.
- G. Kozikowski and B.J. Kubica. Interval arithmetic and automatic differentiation on GPU using OpenCL. In *PARA 2012*, LNCS 7782, pages 489-503, Springer 2013.

some related work in polynomial system solving

- R.A. Klopotek and J. Porter-Sobieraj. Solving systems of polynomial equations on a GPU. In *Preprints of the Federated Conference on Computer Science and Information Systems, September 9-12, 2012, Wroclaw, Poland*, pages 567–572, 2012.
- M.M. Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *ACM Communications in Computer Algebra*, 45(2):127–128, 2011.

some related work in numerical linear algebra

- M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the IPDPS 2011*, pages 48–58. IEEE Computer Society, 2011.
- T. Bartkewitz and T. Güneysu. Full lattice basis reduction on graphics cards. In *WEWoRC'11 Proceedings*, LNCS vol. 7242, pages 30–44, Springer, 2012.
- J. Demmel, Y. Hida, X.S. Li, and E.J. Riedy. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Trans. Math. Softw.*, 35(4):28:1–28:32, 2009.
- D. Mukunoki and D. Takashashi. Implementation and evaluation of triple precision BLAS subroutines on GPUs. In *Proceedings of PDSEC 2012*, pages 1372–1380. IEEE Computer Society, 2012.
- V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

hardware and software

- Our main target: the NVIDIA Tesla K20C, with 2496 cores at 706 MHz, hosted by a RHEL workstation of Microway, with Intel Xeon E5-2670 at 2.6 GHz. Used 4.4.7 of gcc and 5.5 of nvcc.
- Our other computer is an HP Z800 RHEL workstation with 3.47 GHz Intel Xeon X5690, hosting the NVIDIA Tesla C2050 has 448 cores at 1147 Mhz. Used 4.4.7 of gcc and 5.5 of nvcc.
- The C++ code for the Gram-Schmidt method to run on the host is directly based on the pseudo code, compiled with `-O2` flag. Our serial C++ code served mainly to verify correctness.

GPU accelerated Newton's method

polynomial evaluation and differentiation

We distinguish three stages:

1. Common factors and tables of power products:

   $$x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n} = x_{i_1} x_{i_2} \cdots x_{i_k} \times x_{j_1}^{e_{j_1}} x_{j_2}^{e_{j_2}} \cdots x_{j_\ell}^{e_{j_\ell}}$$

   The factor $x_{j_1}^{e_{j_1}} x_{j_2}^{e_{j_2}} \cdots x_{j_\ell}^{e_{j_\ell}}$ is common to all partial derivatives.
   The factors are evaluated as products of pure powers of the variables, computed in shared memory by each block of threads.

2. Evaluation and differentiation of products of variables:
   Computing the gradient of $x_1 x_2 \cdots x_n$ with the reverse mode of algorithmic differentiation requires $3n - 5$ multiplications.

3. Coefficient multiplication and term summation.
   Summation jobs are ordered by the number of terms so each warp has the same amount of terms to sum.

GPU accelerated Newton's method

arithmetic circuits

First to evaluate the product:

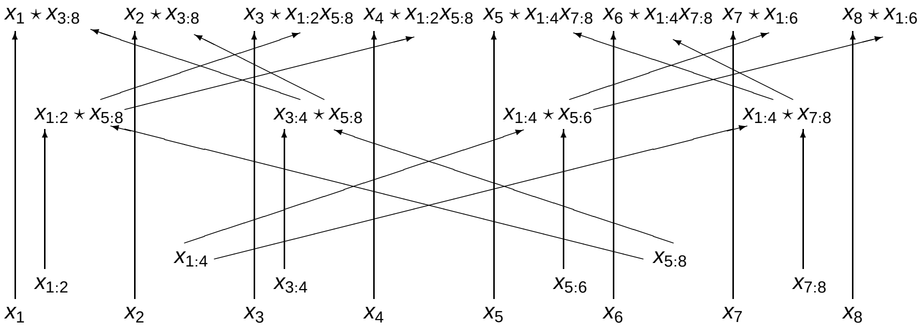
$$\begin{tikzcd}
 & & & x_1 x_2 \star x_3 x_4 & & & \\
 & x_1 \star x_2 \arrow[urr] & & & & x_3 \star x_4 \arrow[ull] & \\
x_1 \arrow[ur] & & x_2 \arrow[ul] & & x_3 \arrow[ur] & & x_4 \arrow[ul]
\end{tikzcd}$$

and then to compute the gradient:

$$\begin{tikzcd}
x_1 \star x_3 x_4 & & x_2 \star x_3 x_4 & & x_1 x_2 \star x_3 & & x_1 x_2 \star x_4 \\
 & x_1 x_2 \arrow[urrr] \arrow[urrrrr] & & & & x_3 x_4 \arrow[ulllll] \arrow[ulll] & \\
x_1 \arrow[uu] & & x_2 \arrow[uu] & & x_3 \arrow[uu] & & x_4 \arrow[uu]
\end{tikzcd}$$

computing the gradient of $x_1 x_2 \cdots x_8$

Denote by $x_{i:j}$ the product $x_i \star \cdots \star x_k \star \cdots \star x_j$ for all k between i and j.



The computation of the gradient of $x_1 x_2 \cdots x_n$ requires

- $2n - 4$ multiplications, and
- $n - 1$ extra memory locations.

GPU accelerated Newton's method

optimized parallel reduction

Evaluation and differentiation of 65,024 monomials in 1,024 doubles:

- Times on the K20C obtained with `nvprof` (the NVIDIA profiler) are in milliseconds (ms).
- Dividing the number of bytes read and written by the time gives the bandwidth.
- Times on the CPU are on one 2.6GHz Intel Xeon E5-2670, with code optimized with the `-O2` flag.

	method	time	bandwidth	speedup
CPU		330.24ms		
GPU	one thread per monomial	86.43ms		3.82
	one block per monomial	15.54ms	79.81GB/s	21.25
	sequential addressing	14.08ms	88.08GB/s	23.45
	unroll last wrap	10.19ms	121.71GB/s	32.40
	complete unroll	9.10ms	136.28GB/s	36.29

different sizes of monomials

Evaluation and differentiation of *m* monomials of different size *n*:

- by 65,024 blocks with 512 threads per block for 1,024 doubles in shared memory,
- accelerated by the K20C with timings in milliseconds obtained by the NVIDIA profiler.
- Times on the CPU are on one 2.6GHz Intel Xeon E5-2670, with code optimized with the `-O2` flag.

n	*m*	CPU	GPU	speedup
1024	1	330.24ms	9.12ms	36.20
512	2	328.92ms	8.73ms	37.66
256	4	320.78ms	8.84ms	36.29
128	8	309.02ms	8.15ms	37.89
64	16	289.30ms	7.27ms	39.77
32	32	256.07ms	9.51ms	26.94
16	64	230.34ms	8.86ms	25.99
8	128	218.74ms	7.79ms	28.07
4	256	202.20ms	7.05ms	28.69

GPU accelerated Newton's method

modified Gram-Schmidt on the GPU

With Gram-Schmidt orthonormalization we solve linear systems in the least squares sense, capable of handling overdetermined problems.

In [Volkov-Demmel, 2008], the left-looking scheme is dismissed because of its limited inherent parallelism.

For thread-level parallelism, we implemented the right-looking version of the modified Gram-Schmidt method.

- Although left-looking is preferable for memory transfers,
- our computations are compute bound for complex double double, quad double, and complex quad double arithmetic.

GPU accelerated Newton's method

the discretization of an integral equation

Formulating a polynomial system for any dimension:

$$f_i(H_1, H_2, \ldots, H_n) = 2nH_i - cH_i \left(\sum_{j=0}^{n-1} \frac{i}{i+j} H_j \right) - 2n = 0, \quad i = 1, 2, \ldots, n,$$

where c is some real nonzero constant, $0 < c \leq 1$.

The cost to evaluate and differentiate grows linear in $n \ldots$

$\Rightarrow$ the cost of Newton's method is dominated
by the cost of solving a linear system which grows as n^3.

The value for the parameter c we used in our experiments is $33/64$.

Starting at $H_i = 1$ for all i leads to a quadratically convergent process.

a benchmark problem

The problem was treated with Newton's method in [1]
and added to a collection of benchmark problems in [2].
In [3], the system was studied with methods in computer algebra.

1. C.T. Kelley. Solution of the Chandrasekhar *h*-equation by Newton's method. *J. Math. Phys.*, 21(7):1625–1628, 1980.
2. J.J. Moré. A collection of nonlinear model problems. In *Computational Solution of Nonlinear Systems of Equations*, volume 26 of *Lectures in Applied Mathematics*, pages 723–762. AMS, 1990.
3. L. Gonzalez-Vega. Some examples of problem solving by using the symbolic viewpoint when dealing with polynomial systems of equations. In J. Fleischer, J. Grabmeier, F.W. Hehl, and W. Küchlin, editors, *Computer Algebra in Science and Engineering*, pages 102–116. World Scientific, 1995.

experimental setup

for a number of iterations :

1. The host evaluates and differentiates the system at the current approximation, stored in an n-by-$(n+1)$ matrix $[A\ \mathbf{b}]$, with $\mathbf{b} = -f(H_1, H_2, \ldots, H_n)$; print b_1.
2. $A\Delta\mathbf{x} = \mathbf{b}$ is solved in the least squares sense,
   either entirely by the host; or
   if accelerated, then
   2.1 the matrix $[A\ \mathbf{b}]$ is transferred
       from the host to the device;
   2.2 the device does a QR decomposition on $[A\ \mathbf{b}]$
       and back substitution on the system $R\Delta\mathbf{x} = \mathbf{y}$;
   2.3 the matrices Q, R, and the solution $\Delta\mathbf{x}$
       are transferred from the device to the host.
3. The host performs the update $\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$
   to compute the new approximation.
   The first component of $\Delta\mathbf{x}$ and $\mathbf{x}$ are printed.

accelerating on the K20C

Running six iterations of Newton's method in complex double double arithmetic on one core on the CPU and accelerated by the K20C with block size equal to 128, once with the evaluation and differentiation done by the CPU (GPU1) and once with all computations on the GPU (GPU2).

n	mode	real	user	sys	speedup
1024	CPU	5m22.360s	5m21.680s	0.139s	
	GPU1	24.074s	18.667s	5.203s	13.39
	GPU2	20.083s	11.564s	8.268s	16.05
2048	CPU	42m41.597s	42m37.236s	0.302s	
	GPU1	2m45.084s	1m48.502s	56.175s	15.52
	GPU2	2m29.770s	1m26.373s	1m03.014s	17.10
3072	CPU	144m13.978s	144m00.880s	0.216s	
	GPU1	8m50.933s	5m34.427s	3m15.608s	16.30
	GPU2	8m15.565s	4m43.333s	3m31.362s	17.46
4096	CPU	340m00.724s	339m27.019s	0.929s	
	GPU1	20m26.989s	13m39.416s	6m45.799s	16.63
	GPU2	19m24.243s	11m01.558s	8m20.698s	17.52

GPU accelerated Newton's method

the cyclic n-roots problem

A system relevant to operator algebras is:

$$\begin{cases} x_0 + x_1 + \cdots + x_{n-1} = 0 \\ x_0x_1 + x_1x_2 + \cdots + x_{n-2}x_{n-1} + x_{n-1}x_0 = 0 \\ i = 3, 4, \ldots, n-1 : \displaystyle\sum_{j=0}^{n-1} \prod_{k=j}^{j+i-1} x_{k \bmod n} = 0 \\ x_0x_1x_2 \cdots x_{n-1} - 1 = 0. \end{cases}$$

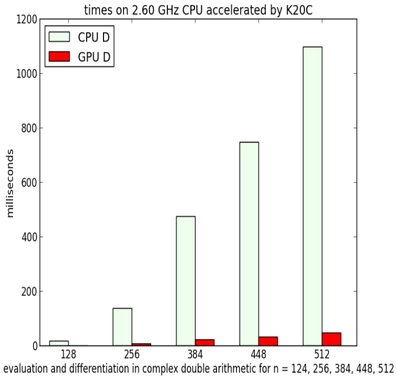
The system is a benchmark problem in computer algebra.
The cost to evaluate and differentiate the system is $O(n^3)$.

accelerating on the K20C

Evaluation and differentiation of the cyclic *n*-roots problem in complex double (D), complex double double (DD), and complex quad double (QD) arithmetic.

Times in milliseconds are obtained with the NVIDIA profiler.

n	CPU	GPU	speedup
D 128	16.39ms	1.13ms	14.87
256	136.26ms	6.97ms	19.55
384	475.94ms	21.59ms	22.05
448	747.44ms	32.68ms	22.87
512	1097.20ms	46.43ms	23.63
DD 128	144.27ms	6.45ms	22.36
256	1169.07ms	37.15ms	31.47
384	3981.07ms	120.48ms	33.04
448	6323.17ms	182.19ms	34.52
512	9411.55ms	267.39ms	35.20
QD 128	1349.55ms	29.45ms	45.82
256	10987.87ms	152.82ms	71.90
384	37323.08ms	513.78ms	72.64
448	59247.04ms	809.15ms	73.22



conclusions

On a massively parallel implementation of Newton's method for large polynomial systems:

- The cost of linear algebra dominates in case the cost to evaluate and differentiate the system is linear in the dimension.
- For systems where the cost to evaluate and differentiate is cubic in the dimension, the computations
    - are memory bound for double and double double arithmetic; and
    - become compute bound for complex double double, quad double, and complex quad double arithmetic.