Accelerating Path Tracking for Polynomial Homotopies

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

SIAM Conference on Parallel Processing for Scientific Computing
MS68 High Performance Symbolic Computation
Portland, 21 February 2014

Outline

1. Solving Polynomial Systems with Homotopies
   - compensating for the cost of quad double arithmetic
2. Orthogonalization and Delayed Normalization
   - accelerating the modified Gram-Schmidt method
   - overcoming the limits on the capacity of shared memory
   - solving random complex linear systems
3. The Chandrasekhar H-Equation
   - the discretization of an integral equation
   - accelerating Newton's method

solving polynomial systems

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.

Narrowing our focus:

1. The computationally intensive task when tracking solution paths defined by polynomial homotopies is Newton's method.
2. We focus on accelerating the modified Gram-Schmidt method to solve linear systems in the least squares sense.

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/`.

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

Solving $A\mathbf{x} = \mathbf{b}$ in the least squares sense:

$$\left[\begin{array}{cc} A & \mathbf{b} \end{array} \right] = \left[\begin{array}{cc} Q & \mathbf{q}_{n+1} \end{array} \right] \left[\begin{array}{cc} R & \mathbf{y} \\ 0 & z \end{array} \right].$$

some related work

- 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.

the capacity of shared memory

Shared memory is fast but limited in capacity.

For the computation of the complex conjugated inner product $\mathbf{a}_k^H \mathbf{a}_k$, we load the components of an m-dimensional vector into shared memory.

If shared memory can hold K components of a vector,
then let $L = \lceil m/K \rceil$ be the number of rounds it takes to compute

$$\mathbf{a}_k^H \mathbf{a}_k = \sum_{i=0}^{L-1} \sum_{j=0}^{K-1} \overline{a}_{k,i\star K+j} a_{k,i\star K+j}.$$

The value for K is typically a multiple of the warp size
and equals the number of threads in a block.

computing conjugated inner products

For capacity K of shared memory and $L = \lceil m/K \rceil$:

$$\mathbf{a}_k^H \mathbf{a}_k = \sum_{i=0}^{L-1} \sum_{j=0}^{K-1} \overline{a}_{k,i\star K+j} a_{k,i\star K+j}.$$

Executing a double loop:

1. The index j in the inner loop is the index of the thread in a block, so the inner loop is done simultaneously by all threads in the block.
2. The outside loop on i is a sum reduction and takes $\log_2(L)$ steps.

The computation of $\mathbf{a}_k^H \mathbf{a}_k$ for an n-dimensional vector $\mathbf{a}_k$ is reduced to

- m memory accesses,
- L steps to make all partial sums, and
- $\log_2(L)$ steps for the outer sum.

the reduction

For the reduction, we compute $r_{k,j} := \mathbf{q}_k^H \mathbf{a}_j$ of two m-vectors:

$$\begin{array}{ccc} \mathbf{q}_k & \mathbf{a}_j & \mathbf{q}_k^H \mathbf{a}_j \\ \left[\begin{array}{c} q_{k,0} \\ q_{k,1} \\ \vdots \\ q_{k,m-1} \end{array}\right] & \left[\begin{array}{c} a_{j,0} \\ a_{j,1} \\ \vdots \\ a_{j,m-1} \end{array}\right] & \left[\begin{array}{c} \bar{q}_{k,0} \star a_{j,0} \\ \bar{q}_{k,1} \star a_{j,1} \\ \vdots \\ \bar{q}_{k,K-1} \star a_{j,m-1} \end{array}\right] \end{array}$$

As we can keep K components of each m-vector in shared memory, thread t in a block computes $\bar{q}_{k,t} \star a_{j,t}$.

As we still need $\mathbf{q}_k$ for $\mathbf{a}_j := \mathbf{a}_j - r_{k,j}\mathbf{q}_k$,
$3m$ shared memory locations are used to perform the reductions.

The computation of $\mathbf{q}_k^H \mathbf{a}_j$ is performed in L rounds,
where $L = \lceil 3m/K \rceil$, for the capacity K of shared memory.

delayed normalization

When reducing vectors with the current column:

- each block of threads normalizes the current column,
- first block writes the normalized vector to global memory.

The problem with processing long vectors in rounds:

- vectors are read from global memory in stages,
- last block that reduces the last column is launched last ...

Solution: delay normalizations to the next iteration.

The first block

- writes the norm of the current vector to global memory,
- uses that norm in the next iteration on the previous vector.

Cost: one extra kernel launch at the end.

back substitution by many blocks

Solving $R\mathbf{x} = Q^H\mathbf{b}$, for capacity K of shared memory:

$$r_{\ell,\ell}x_\ell = y_\ell - \sum_{j=0}^{\ell-1} r_{\ell,j}x_j = y_\ell - \sum_{i=0}^{L-1}\sum_{j=0}^{K-1} r_{\ell,i\star K+j},$$

where $L = \lceil m/K \rceil$.

In first stage, L blocks can work simultaneously:

- pivot block computes last components of solution,
- other blocks write reductions of $\mathbf{b}$ to global memory.

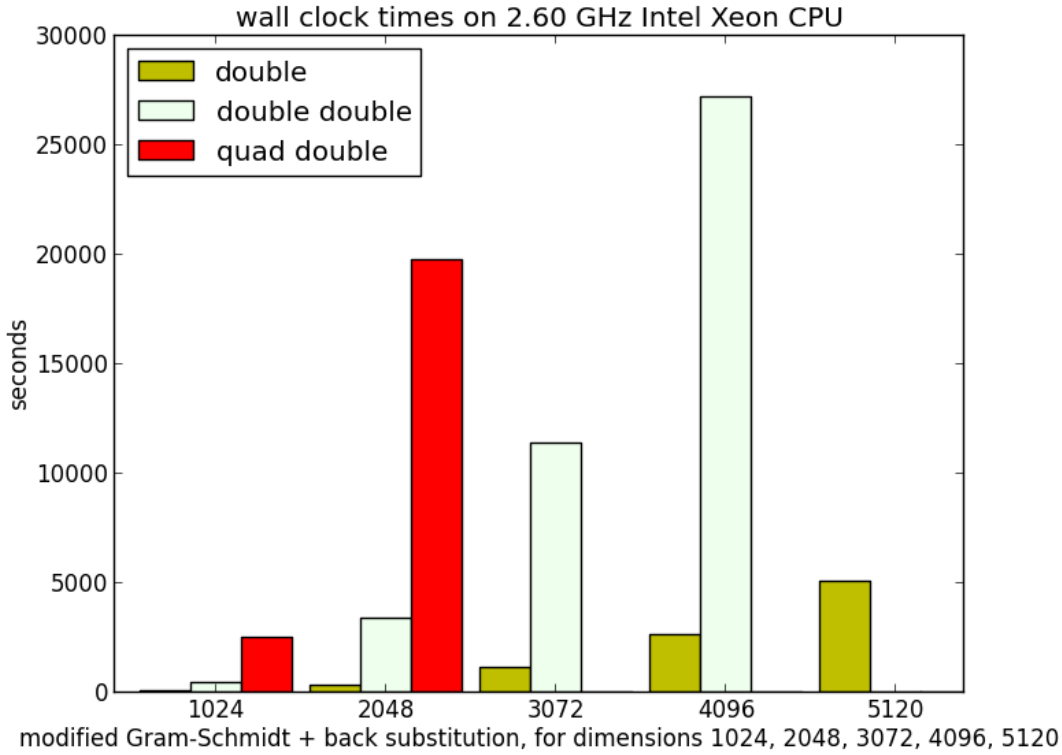
Second stage launches $L-1$ blocks,
third stage launches $L-2$ blocks, etc. ...

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 NVIDIA GPU Test Drive program of Microway gave access to two 10-core Xeon E5-2680v2 2.8GHz CPUs and one NVIDIA Tesla Atlas GPU, the K40: 2880 cores at 745MHz. Host runs CentOs Linux 6, used 4.4.6 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 without any optimizations. Our serial C++ code served mainly to verify correctness.

MGS + LS on one core 2.60GHz

p	n	real
D	1024	39.872s
	2048	5m23.402s
	3072	18m19.975s
	4096	43m18.139s
	5120	83m59.885s
DD	1024	7m 1.064s
	2048	56m 2.627s
	3072	189m22.950s
	4096	452m53.113s
QD	1024	41m 9.521s
	2048	329m 4.188s

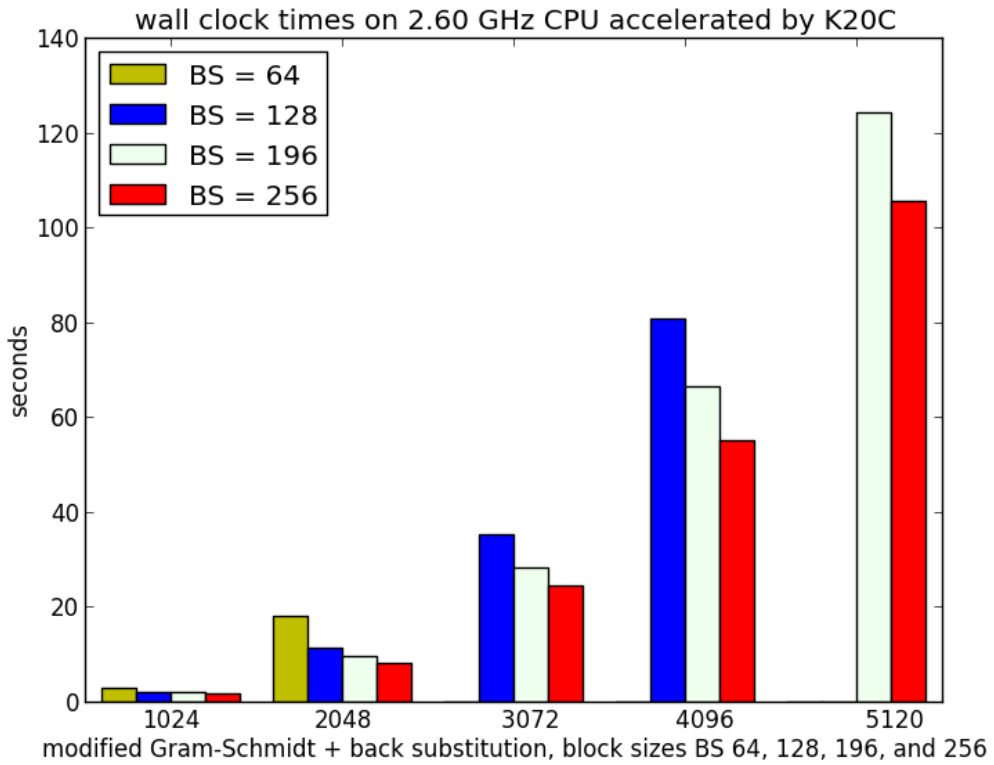


Cost overhead factor is about 8, observed in doubling the dimension.

1024 in QD (41 min) $\approx$ 2048 in DD (56 min) $\approx$ 4096 in D (43 min)

increasing the block size on the K20C

n	BS	real	user	sys	speedup
1024	32	5.067s	3.138s	1.501s	7.87
1024	64	3.045s	1.747s	1.099s	13.09
1024	96	2.544s	1.434s	0.966s	15.67
1024	128	2.159s	1.116s	0.801s	18.47
1024	160	2.140s	1.083s	0.859s	18.63
1024	192	1.971s	1.039s	0.782s	20.23
1024	224	1.862s	0.874s	0.794s	21.41
1024	256	1.712s	0.807s	0.744s	23.29
2048	64	18.112s	10.892s	6.950s	17.86
2048	96	14.568s	8.866s	5.516s	22.20
2048	128	11.414s	6.841s	4.342s	28.33
2048	160	10.610s	6.335s	4.042s	30.48
2048	192	9.595s	5.717s	3.634s	33.71
2048	224	9.244s	5.530s	3.461s	34.99
2048	256	8.098s	4.768s	3.147s	39.94
3072	96	44.692s	26.928s	17.518s	24.61
3072	128	35.435s	21.003s	14.198s	31.04
3072	160	33.395s	19.848s	13.251s	32.94
3072	192	28.310s	17.109s	11.002s	38.85
3072	224	26.342s	15.780s	10.390s	41.76
3072	256	24.411s	14.616s	9.542s	45.06
4096	128	1m20.761s	47.926s	32.490s	32.17
4096	160	1m14.422s	44.555s	29.596s	34.91
4096	192	1m 6.437s	43.248s	22.877s	39.11
4096	224	1m 1.094s	39.699s	21.001s	42.53
4096	256	55.140s	32.937s	21.898s	47.12
5120	160	2m20.774s	1m20.936s	59.490s	35.80
5120	192	2m 4.366s	1m13.106s	50.782s	40.52
5120	224	1m53.530s	1m 7.168s	45.975s	44.39
5120	256	1m45.579s	1m 3.446s	41.842s	47.74

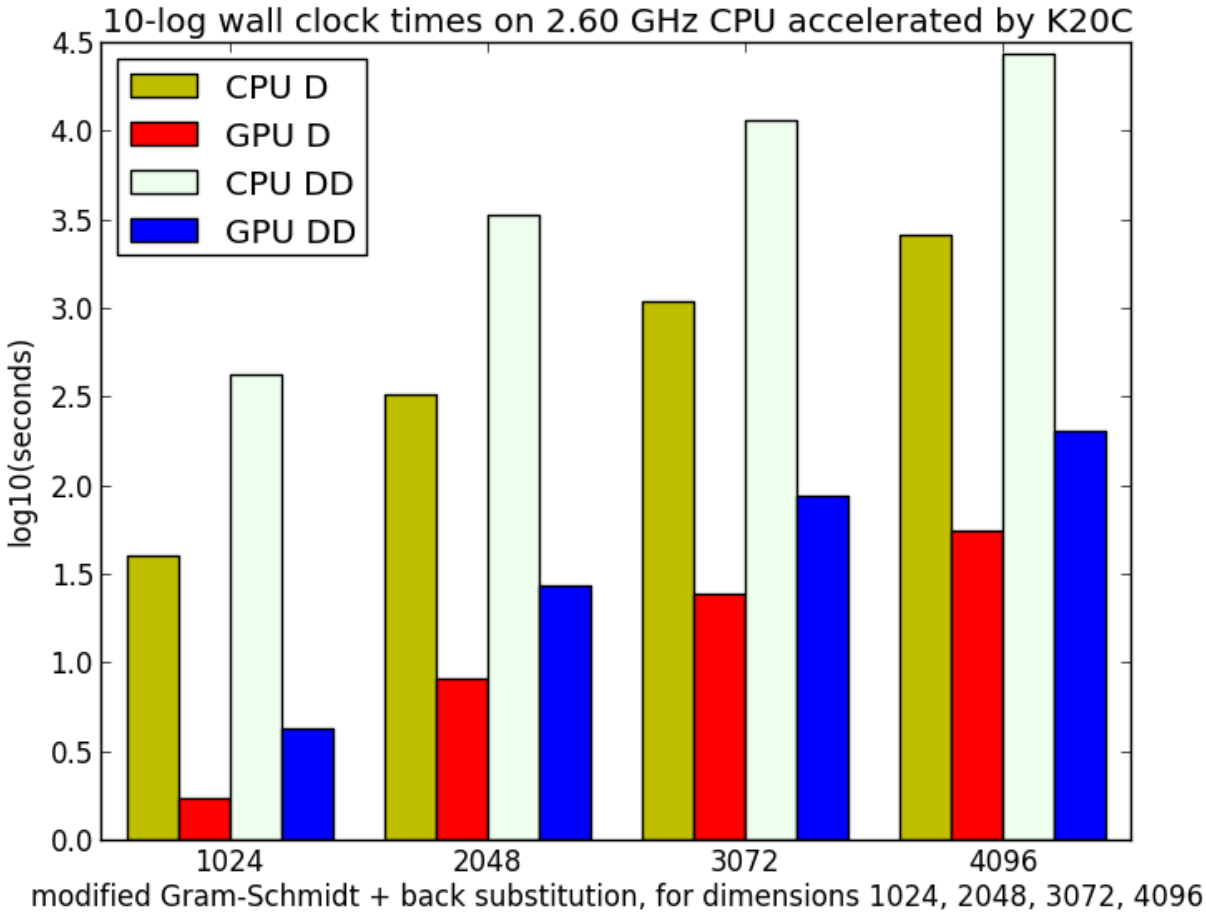


$\Rightarrow$ Better speedups
for larger block sizes.

MGS + LS in DD and QD on the K20C

p	n	BS	real	user	sys	speedup
DD	1024	32	11.846s	8.563s	3.050s	35.54
	1024	64	6.608s	4.437s	1.967s	63.72
	1024	96	5.359s	3.440s	1.665s	78.57
	1024	128	4.270s	2.749s	1.320s	98.61
	2048	64	44.798s	27.813s	16.720	75.06
	2048	96	34.929s	21.765s	12.860s	96.27
	2048	128	27.039s	16.838s	9.922s	124.36
	3072	96	1m49.947s	1m 6.248s	43.375s	103.35
	3072	128	1m26.581s	51.433s	34.724s	131.24
	4096	128	3m21.074s	1m57.756s	1m22.789s	135.14
QD	1024	32	4m12.664s	3m22.920s	49.078s	9.77
	1024	64	2m30.463s	1m53.848s	36.221s	16.41
	2048	64	19m20.893s	11m48.509s	7m30.749s	17.01

comparing wall clock times at *logarithmic scale*



Observe that the rightmost bar is shorter than fifth bar from the left: solving a linear system for $n = 4,096$ with the GPU (DD) takes less time than solving a linear system for $n = 2,048$ on the CPU (D).

comparing with the new NVIDIA K40

The speedup is computed from the system times.

p	n	real	user	sys	speedup
D	1024	1.303s	0.697s	0.496s	1.5
	2048	6.673s	4.136s	2.428s	1.23
	3072	20.342s	12.577s	7.628s	1.25
	4096	46.284s	28.287s	17.823s	1.23
	5120	1m29.107s	54.447s	34.407s	1.22
DD	1024	3.264s	2.206s	0.950s	1.39
	2048	21.214s	13.458s	7.650s	1.30
	3072	1m 8.450s	41.942s	26.285s	1.32
	4096	2m39.031s	1m37.148s	1m 1.397s	1.35
QD	1024	2m 0.069s	1m32.061s	27.672s	1.31
	2048	15m24.871s	9m44.882s	5m37.872s	1.33

Speedups confirm the theoretical peak performance of the K40.

For $n = 4,096$ (DD), the time of 452m53.113s (more than 7.5 hours) on one core is reduced to 2m39.031s, a speedup of 170.87.

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 only the linear system solving

For $n = 1,024$, when only the linear system solving was accelerated:

precision	wall clock time	speedup
double double	41.193s	62.18
quad double	15m34.527s	16.30

The speedups match closely the speedups of 63.72 and 16.41 for solving one linear system.

Performing the entire Newton's method on the device did not give better speedups for quad double complex arithmetic.

Accelerating of evaluation, differentiation, and linear system solving improved speedups for complex double double arithmetic:

- from 62.18 to 72.72, for block size 64; and
- to 127.42, for block size 128.

running six iterations with Newton's method

On one core of the host (CPU), and with GPU acceleration (GPU):

complex double double arithmetic

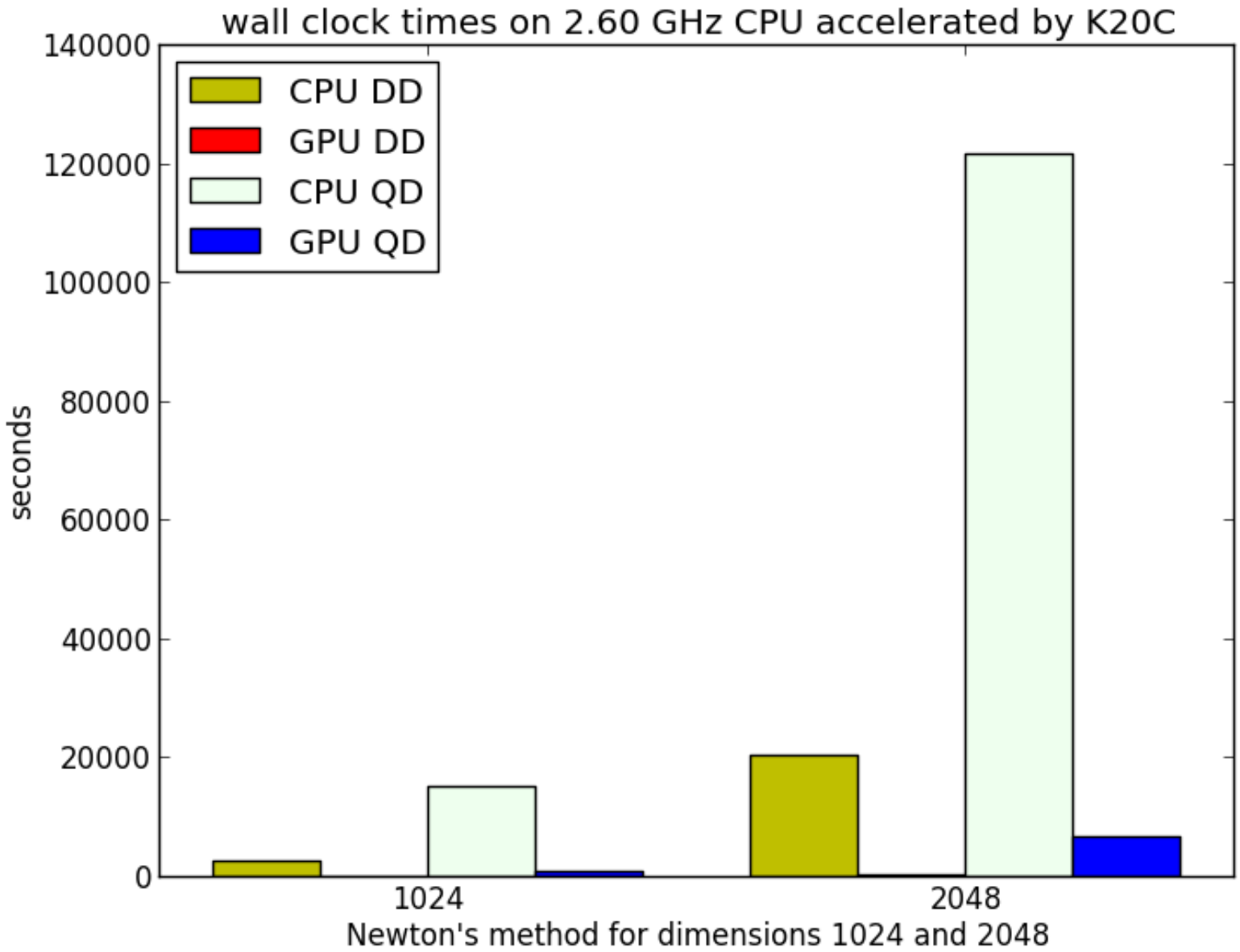
n	real	user	sys	speedup
CPU 1024	42m41.480s	42m37.692s	0.038s	
GPU 1024	20.102s	11.664s	8.236s	127.42
CPU 2048	341m47.998s	341m18.009s	0.362s	
GPU 2048	2m29.770s	1m26.373s	1m03.014s	136.92

complex quad double arithmetic

n	real	user	sys	speedup
CPU 1024	253m51.126s	253m24.170s	4.802s	
GPU 1024	15m11.362s	9m28.399s	5m41.532s	16.71
CPU 2048	2027m40.726s	2024m38.715s	3.055s	
GPU 2048	110m51.042s	63m21.470s	47m21.105s	18.29

The acceleration of Newton's method is done entirely by the GPU.

observe what happens when the dimension doubles ...



conclusions

- Solving linear systems in the least squares sense after the modified Gram-Schmidt method on problems of dimensions 2,048, 3,072, and 4,096 in complex double double arithmetic gives speedups of 124, 131, and 135 when accelerated by the K20C. Even better speedups with the newest K40.
- The good speedups on linear system solving compensate for the memory transfers between host and device.
  The evaluation and differentation stage of Newton's method can thus be done on the host with speedups resulting from accelerated linear system solving.
- For applications where the cost of evaluation and differentiation grows only linearly in the dimension, the orders of the speedups of several iterations of Newton's method match those of the linear systems solving.