# modified Gram-Schmidt Orthogonalization on a Graphics Processing Unit (GPU) with double double and quad double arithmetic

Jan Verschelde
joint 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

Graduate Computational Algebraic Geometry Seminar

# Outline

1. Problem Statement
   - solving linear systems accurately
   - the modified Gram-Schmidt method
   - cost and accuracy
2. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - defining the kernels
   - occupancy of multiprocessors and resource usage
3. Computational Results
   - experimental setup
   - running for increasing dimensions and precisions
   - simulating the tracking of one path

# Orthogonalization on a GPU

1. Problem Statement
   - solving linear systems accurately
   - the modified Gram-Schmidt method
   - cost and accuracy
2. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - defining the kernels
   - occupancy of multiprocessors and resource usage
3. Computational Results
   - experimental setup
   - running for increasing dimensions and precisions
   - simulating the tracking of one path

# solving linear systems accurately

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.

Solving linear systems with least squares using a QR decomposition

- is more accurate than a LU factorization, and
- applies to overdetermined problems.

Quality up: offset the extra cost with parallel algorithms.

# quad double precision

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

- 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/`.

# equipment

Current hardware:

- HP Z800 workstation running Red Hat Enterprise Linux 6.3
  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.

Next graphics compute processor: NVIDIA Tesla K20.

The K20 has 2,496 cores and delivers a peak double precision performance of 1.17 teraflops.

Problem: do we have algorithms and software?

# 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 |

# Orthogonalization on a GPU

1. Problem Statement
   - solving linear systems accurately
   - the modified Gram-Schmidt method
   - cost and accuracy
2. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - defining the kernels
   - occupancy of multiprocessors and resource usage
3. Computational Results
   - experimental setup
   - running for increasing dimensions and precisions
   - simulating the tracking of one path

# 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 $j$th block (for all $j : k < j \leq n$)
normalizes $\mathbf{a}_k$ and removes the component
of $\mathbf{a}_j$ in the direction of $\mathbf{a}_k$
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 $\ell$ 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:

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

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.

# Orthogonalization on a GPU

1. Problem Statement
   - solving linear systems accurately
   - the modified Gram-Schmidt method
   - cost and accuracy
2. Massively Parallel Modified Gram-Schmidt Orthogonalization
   - defining the kernels
   - occupancy of multiprocessors and resource usage
3. Computational Results
   - experimental setup
   - running for increasing dimensions and precisions
   - simulating the tracking of one path

# computers and compilers

Computations done on an HP Z800 workstation,
running Red Hat Enterprise Linux 6.3.

For speedups, we compare

- the run times on one core of an 3.47 Ghz Intel Xeon X5690;
- the run times on the NVIDIA Tesla C2050, with clock speed at 1147 Mhz, about three times slower than the CPU.

All times are wall clock times.

The C++ code is compiled with version 4.4.6 of gcc
and we use release 4.0 of the NVIDIA CUDA compiler driver.

The GPU has 448 cores, so we hope for double digit speedups.

# running at different precisions

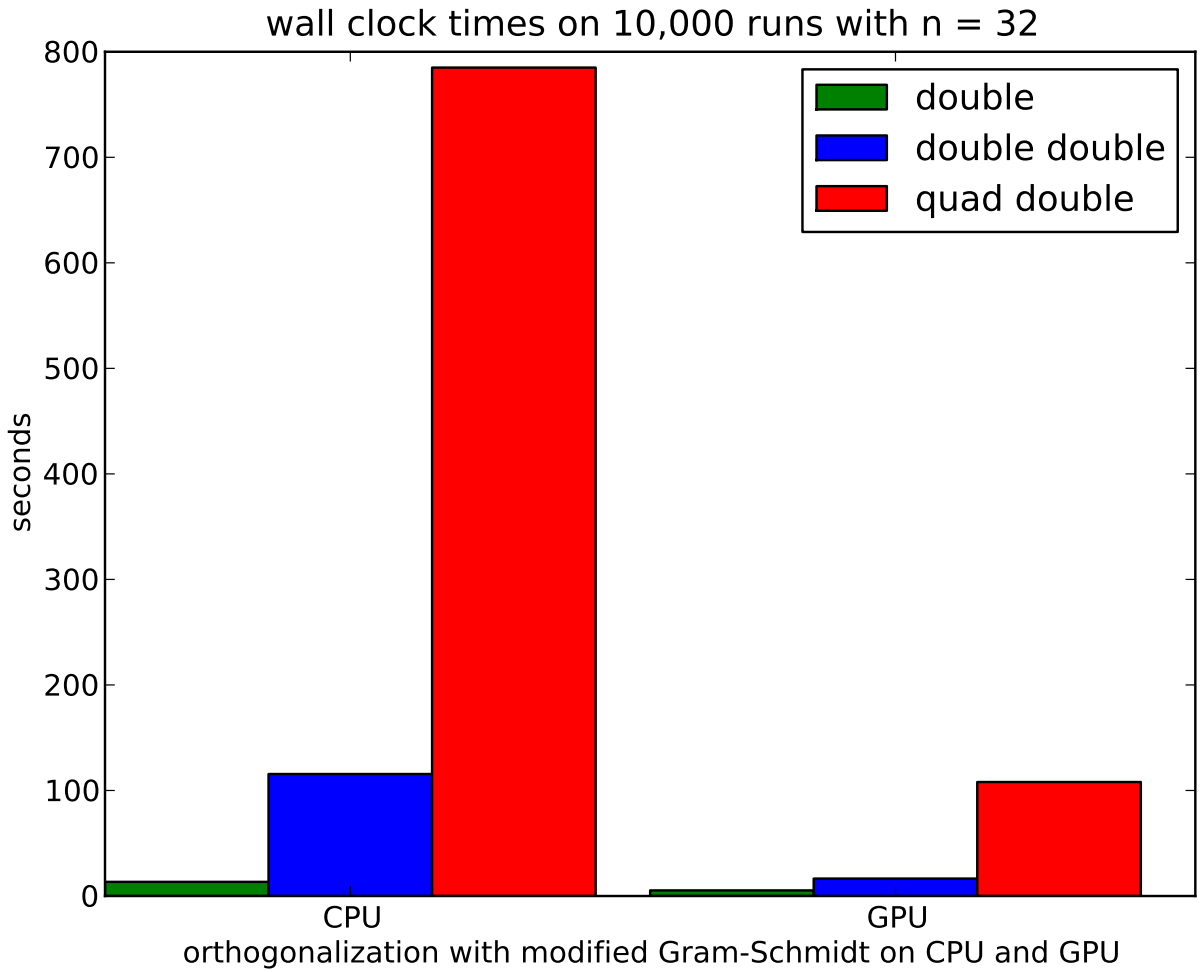
Wall clock times and speedups for 10,000 orthogonalizations, on 32 random complex vectors of dimension 32:

| precision | 1 CPU core | Tesla C2050 | speedup |
|---|---|---|---|
| complex double | 13.4 sec | 5.3 sec | 2.5 |
| complex double double | 115.6 sec | 16.5 sec | 7.0 |
| complex quad double | 785.0 sec | 108.0 sec | 7.3 |

For quality up, compare

- the 115.6 seconds with complex double doubles on CPU;
- the 108.0 seconds with complex quad doubles on GPU.

# corresponding bar plot



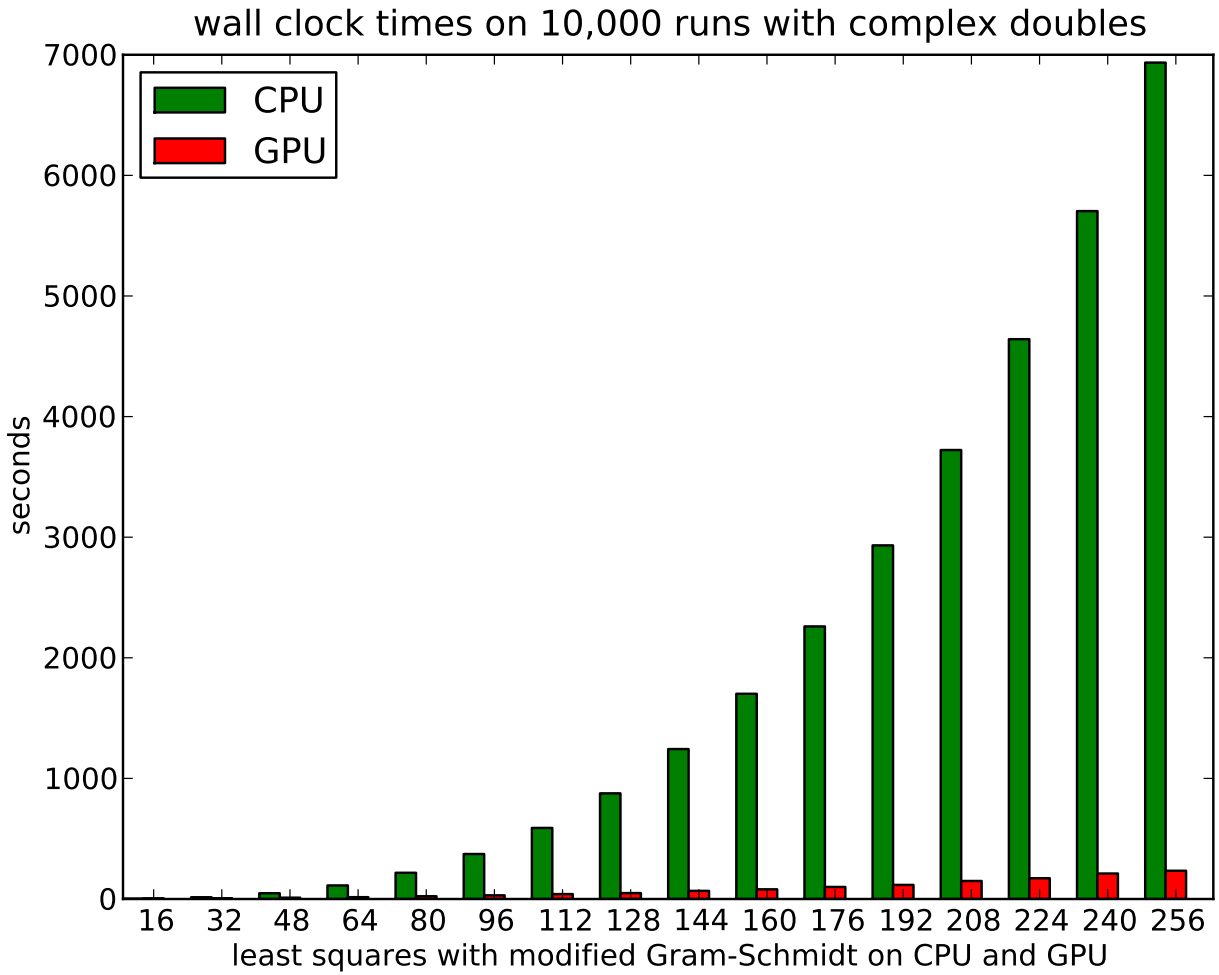
# for increasing dimensions with complex doubles

Wall clock times for 10,000 runs, each followed by one backsubstitution:

complex double arithmetic

| $n$ | CPU | GPU | speedup |
|---|---|---|---|
| 16 | 2.01 | 4.11 | 0.49 |
| 32 | 14.61 | 6.52 | 2.24 |
| 48 | 47.80 | 11.11 | 4.30 |
| 64 | 112.60 | 15.38 | 7.32 |
| 80 | 217.52 | 22.89 | 9.50 |
| 96 | 373.06 | 30.43 | 12.26 |
| 112 | 589.35 | 40.82 | 14.44 |
| 128 | 876.11 | 49.10 | 17.84 |
| 144 | 1243.26 | 67.41 | 18.44 |
| 160 | 1701.57 | 80.42 | 21.16 |
| 176 | 2260.07 | 99.94 | 22.61 |
| 192 | 2932.15 | 116.90 | 25.08 |
| 208 | 3722.77 | 149.45 | 24.91 |
| 224 | 4641.71 | 172.30 | 26.94 |
| 240 | 5703.77 | 211.30 | 26.99 |
| 256 | 6935.10 | 234.29 | 29.60 |

# corresponding bar plot



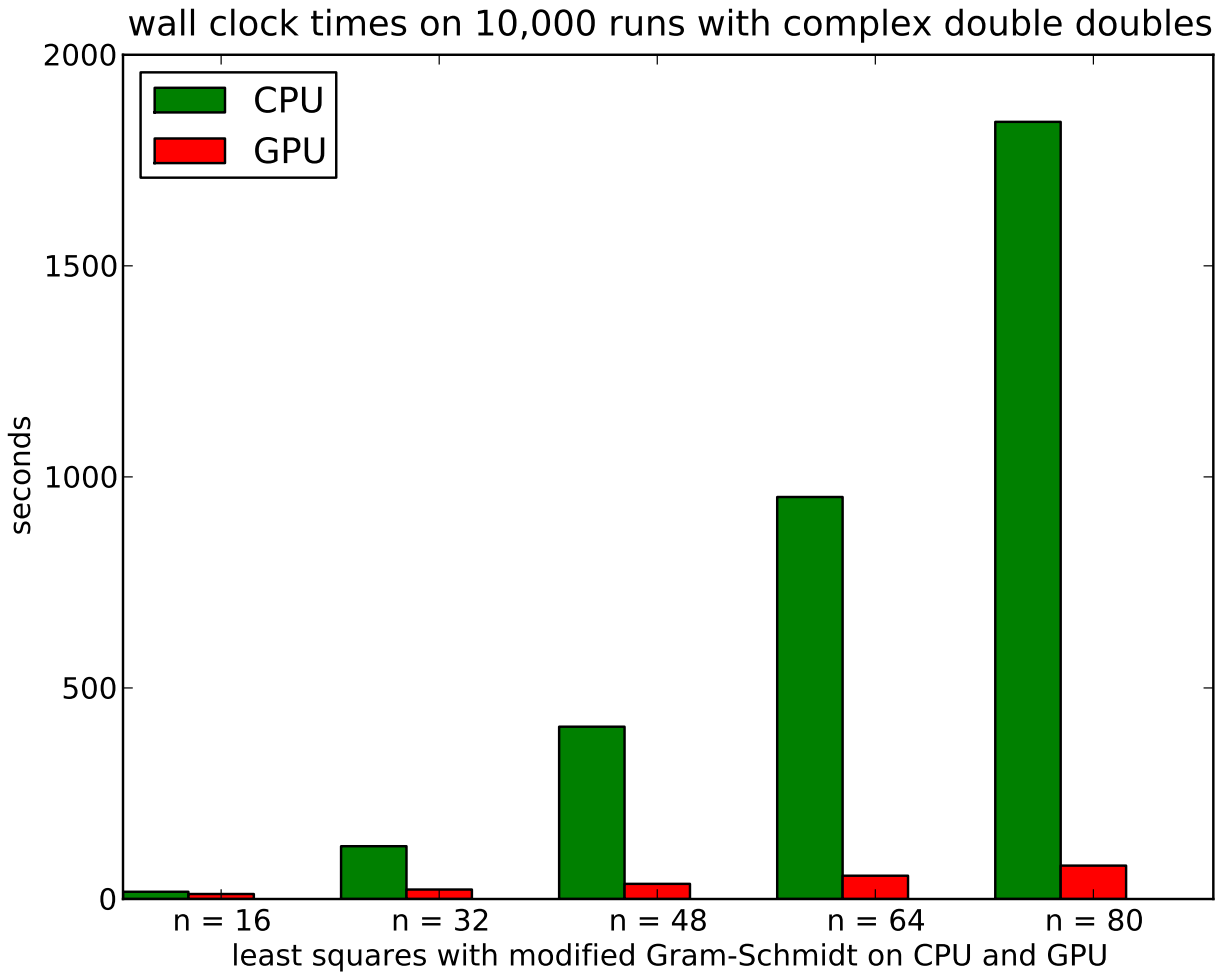
# running with double doubles and quad doubles

Wall clock times for 10,000 runs of the modified Gram-Schmidt method (each followed by one backsubstitution) in complex double double and complex quad double arithmetic:

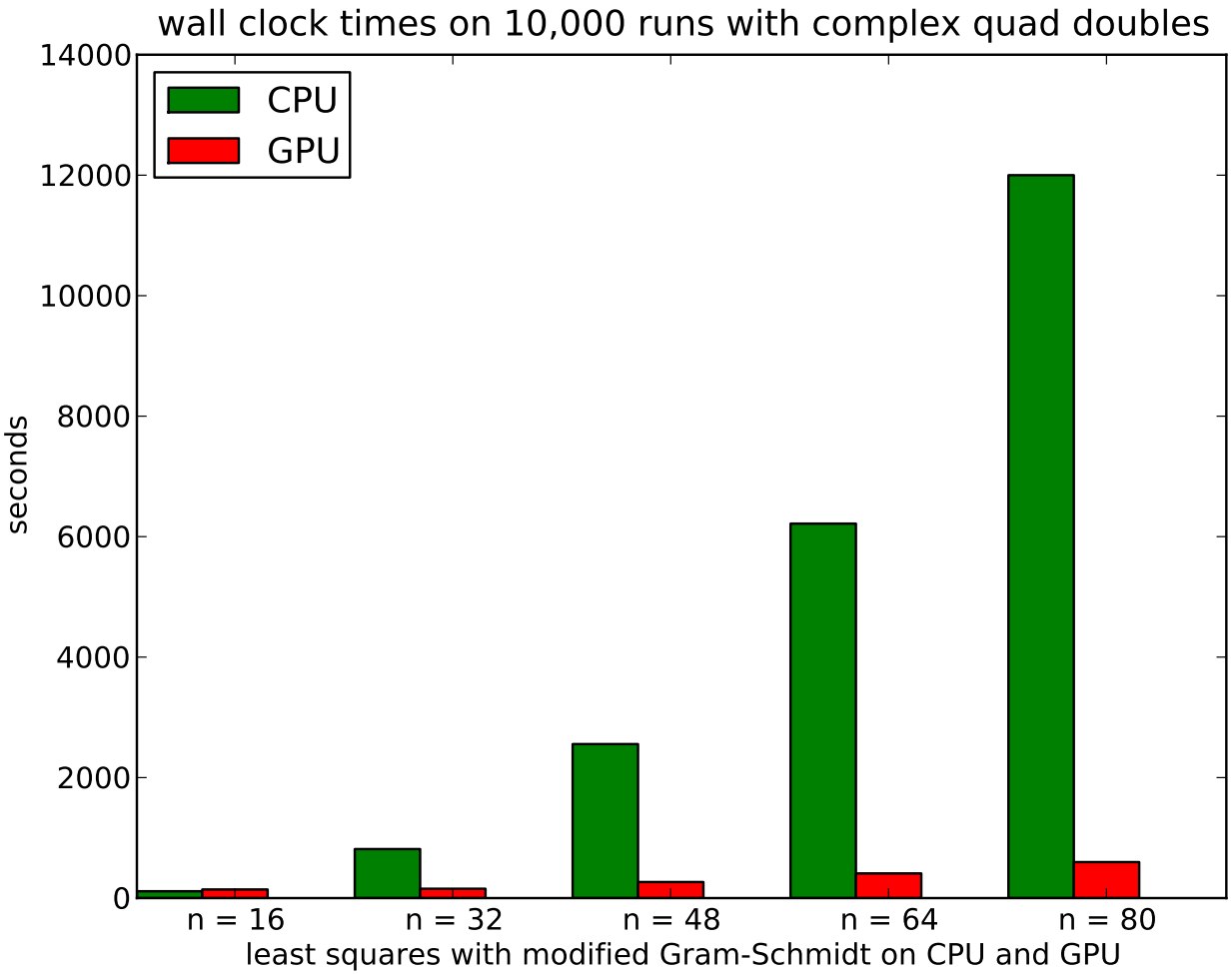
| | complex double double | | | complex quad double | | |
|---|---|---|---|---|---|---|
| $n$ | CPU | GPU | speedup | CPU | GPU | speedup |
| 16 | 17.17 | 11.85 | 1.45 | 113.51 | 143.07 | 0.79 |
| 32 | 125.06 | 22.44 | 5.57 | 813.65 | 155.32 | 5.24 |
| 48 | 408.20 | 35.88 | 11.38 | 2556.36 | 266.55 |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

Double digit speedups occur for $n \geq 48$.
For quality up, compare CPU time for double doubles with GPU time for quad doubles.

# corresponding bar plot for double doubles



# corresponding bar plot for quad doubles



# simulating the tracking of one path

Along a path we may do 10,000 Newton corrections.

Take dimension 32 and compare the wall clock times for

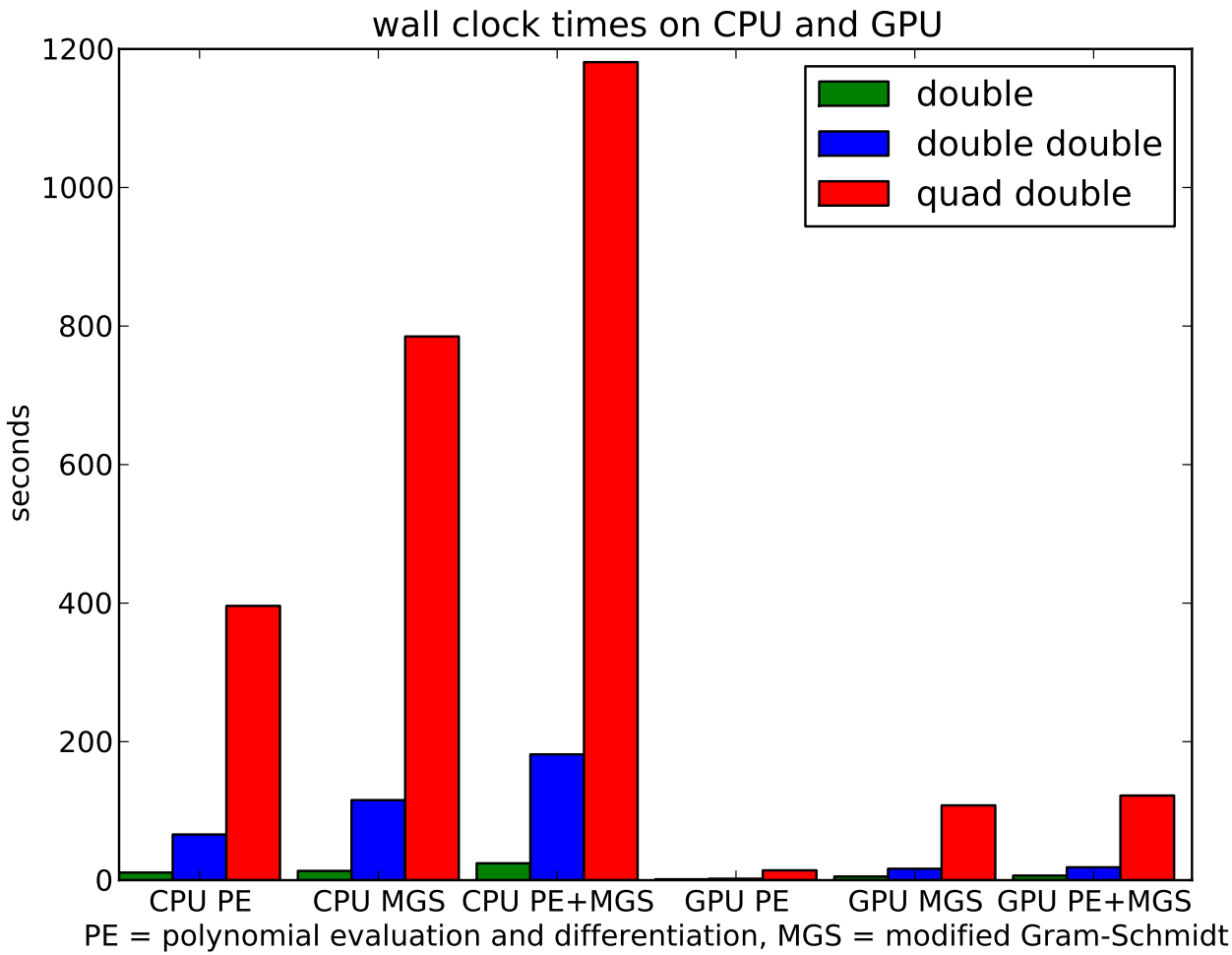
- 10,000 orthogonalizations; and
- 10,000 polynomial evaluations and differentiations of polynomial system of 32 equations with 32 variables, with 32 monomials per polynomial, with 5 variables in each monomial, with variable degrees uniformly taken from $\{1, 2, 3, 4, 5\}$,

for increasing levels of precision.

# wall clock times

| precision | CPU PE | GPU PE | speedup |
|---|---|---|---|
| complex double | 11.0 sec | 1.3 sec | 8.5 |
| complex double double | 66.0 sec | 2.1 sec | 31.4 |
| complex quad double | 396.0 sec | 14.2 sec | 27.9 |
| precision | CPU MGS | GPU MGS | speedup |
| complex double | 13.4 sec | 5.3 sec | 2.5 |
| complex double double | 115.6 sec | 16.5 sec | 7.0 |
| complex quad double | 785.0 sec | 108.0 sec | 7.0 |
| precision | CPU PE+MGS | GPU PE+MGS | speedup |
| complex double | 24.4 sec | 6.6 sec | 3.7 |
| complex double double | 181.6 sec | 18.6 sec | 9.8 |
| complex quad double | 1181.0 sec | 122.2 sec | 9.7 |

# corresponding bar plot



# conclusions

The fine granularity gives speedups because of the increased cost of the multiprecision arithmetic.

With a graphics compute processor we can compensate for the cost overhead of multiprecision and achieve quality up.

Future plans:

- Combine into Newton's method and path tracker.
- Integrate into the PHCpack solver.