Least Squares on GPUs in Multiple Double Precision

Jan Verschelde[†]

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

Online AriC Seminar (ENS Lyon, France)
18 November 2021

[†]Supported by the National Science Foundation, grant DMS 1854513.

Outline

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

problem statement

Given a linear system $A\mathbf{x} = \mathbf{b}$,
compute the least squares solution $\mathbf{x}$.

The least squares solution minimizes $\|\mathbf{b} - Ax\|_2^2$.

Two steps with the QR decomposition:

1. Compute the Householder QR factorization: $A = QR$.

   The blocked Householder QR [Bischof & Van Loan, 1987]
   is rich in matrix-matrix multiplications.

2. Solve the upper triangular system $R\mathbf{x} = Q^T\mathbf{b}$.

   Formulas in [Heller, 1978] are applied recursively in a parallel triangular matrix inversion [Nasri & Mahjoub, 2001].

Problem:
Can acceleration by Graphics Processing Units (GPUs) compensate the cost overhead caused by multiple double precision?

error analysis of a lower triangular block Toeplitz solver

joint with Simon Telen and Marc Van Barel, in the CASC 2020 proceedings

Solving $(A_0 + A_1 t + A_2 t^2 + \cdots + A_i t^i)(x_0 + x_1 t + x_2 t^2 + \cdots + x_i t^i)$
$= (b_0 + b_1 t + b_2 t^2 + \cdots + b_i t^i)$

leads to a lower triangular block system:

$$\begin{bmatrix} A_0 & & & & \\ A_1 & A_0 & & & \\ A_2 & A_1 & A_0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ A_i & A_{i-1} & A_{i-2} & \cdots & A_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_i \end{bmatrix}.$$

Let κ be the condition number of A_0. Let $\|A_0\| = \|x_0\| = 1$, $\|x_i\| \approx \rho^i$.
In our context, $\rho \approx 1/R$, where R is the convergence radius.

If $\|A_i\| \approx \rho^i$, then $\dfrac{\|\Delta x_i\|}{\|x_i\|} \approx \kappa^{i+1} \epsilon_{\text{mach}}$, and accuracy is lost.

With multiple double precision, a small ϵ_{mach} gives accurate results.

multiple double precision — error free transformations

```
Computing the 2-norm of a vector of dimension 64
of random complex numbers on the unit circle equals 8.
Observe the second double of the multiple double 2-norm.

double double : 8.00000000000000E+00 - 6.47112461314111E-32
  quad double : 8.00000000000000E+00 + 3.20475411419393E-65
  octo double : 8.00000000000000E+00 - 9.72609915198313E-129
```

- QDlib by Y. Hida, X. S. Li, and D. H. Bailey.
  Algorithms for quad-double precision floating point arithmetic.
  In the *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, 2001.
- CAMPARY by M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker.
  CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications.
  In *Mathematical Software – ICMS 2016, the 5th Internatical Conference on Mathematical Software*, pages 232–240, Springer-Verlag, 2016.

cost overhead of multiple double precision

	double double: 37.7x				
	$+$	$-$	$*$	$/$	Σ
add	8	12			20
mul	5	9	9		23
div	33	18	16	3	70

	quad double: 439.3x				
	$+$	$-$	$*$	$/$	Σ
add	35	54			89
mul	99	164	73		336
div	266	510	112	5	893

	octo double: 2379.0x				
	$+$	$-$	$*$	$/$	Σ
add	95	174			269
mul	529	954	259		1742
div	1599	3070	448	9	5126

alternatives and related work

- T. Nakayama and D. Takahashi. Implementation of multiple-precision floating-point arithmetic library for GPU computing. In *Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 343–349. ACTA Press, 2011.
- K. Isupov and V. Knyazkov. Multiple-precision matrix-vector multiplication on graphics processing units. *Program Systems: Theory and Applications* 11(3): 62–84, 2020.
  - The double double arithmetic of CAMPARY performs best for the problem of matrix-vector multiplication.
  - Concerning quad double precision, *"the CAMPARY library is faster than our implementation; however as the precision increases the execution time of CAMPARY also increases significantly."*

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

Graphics Processing Units (GPUs)

- Data parallel algorithms execute the same Single Instruction on Multiple Data elements (SIMD).
- Memory bandwidth of GPUs is typically ten times higher than the memory bandwidth of CPUs.

Definition (CGMA ratio)

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed by a kernel for each access to the global memory.

- The NVIDIA K20C, P100, and V100 are capable of teraflop performance in double precision.

five different NVIDIA GPUs

NVIDIA GPU	CUDA	#MP	#cores/MP	#cores	GHz
Tesla C2050	2.0	14	32	448	1.15
Kepler K20C	3.5	13	192	2496	0.71
Pascal P100	6.0	56	64	3584	1.33
Volta V100	7.0	80	64	5120	1.91
GeForce RTX 2080	7.5	46	64	2944	1.10

The double precision peak performance of the P100 is 4.7 TFLOPS. At 7.9 TFLOPS, the V100 is 1.68 times faster than the P100.

To evaluate the algorithms, compare the ratios of the wall clock times on the P100 over V100 with the factor 1.68.

For every kernel, the number of arithmetical operations is accumulated. The total number of double precision operations is computed using the cost overhead multipliers.

customized software

The code for the arithmetical operations generated by the CAMPARY software was customized for each precision.

- Instead of representing a quad double by an array of four doubles, all arithmetical operations work on four separate variables, one for each double.
  By this customization an array of quad doubles is stored as four separate arrays of doubles and a matrix of quad doubles is represented by four matrices of doubles.
- The `double2` and `double4` types of the CUDA SDK work for double double and quad double, but not for the more general multiple double arithmetic.
- QDlib provides definitions for the square roots and various other useful functions for double double and quad double arithmetic. Those definitions are extended to octo double precision.

computational algebraic geometry

The lower triangular block Toeplitz matrix originates
from the application of Newton's method
to compute power series developments of algebraic curves.

Power series are a symbolic-numeric way to solve polynomial systems.

With S. Telen, M. Van Barel: A Robust Numerical Path Tracking Algorithm for Polynomial Homotopy Continuation.
SIAM Journal on Scientific Computing 42(6):A3610–A3637, 2020.

Two computational tasks for polynomial homotopy continuation:

1. evaluation and differentiation of polynomials,
2. solving linear systems.

Both tasks are suitable for acceleration with graphics processing units.

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

partitioning an upper triangular system in tiles

Consider a 3-by-3-tiled upper triangular system $U\mathbf{x} = \mathbf{b}$

$$U = \begin{bmatrix} U_1 & A_{1,2} & A_{1,3} \\ & U_2 & A_{2,3} \\ & & U_3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix},$$

where U_1, U_2, U_3 are upper triangular, with nonzero diagonal elements.

Invert all diagonal tiles: $\begin{bmatrix} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{bmatrix}$.

- The inverse of an upper triangular matrix is upper triangular.
- Solve an upper triangular system for each column of the inverse.
- The columns of the inverse can be computed independently.

$\Rightarrow$ Solve many smaller upper triangular systems in parallel.

the second stage

Solve $U\mathbf{x} = \mathbf{b}$ for $U = \begin{bmatrix} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{bmatrix}$

in the following steps:

$$
\begin{array}{lllll}
1) & \mathbf{x}_3 & := & U_3^{-1}\mathbf{b}_3, & \\
2) & \mathbf{b}_2 & := & \mathbf{b}_2 - A_{2,3}\mathbf{x}_3, & \mathbf{b}_1 := \mathbf{b}_1 - A_{1,3}\mathbf{x}_3, \\
4) & \mathbf{x}_2 & := & U_2^{-1}\mathbf{b}_2, & \\
5) & \mathbf{b}_1 & := & \mathbf{b}_1 - A_{1,2}\mathbf{x}_2, & \\
6) & \mathbf{x}_1 & := & U_1^{-1}\mathbf{b}_1. &
\end{array}
$$

Statements on the same line can be executed in parallel.

In multiple double precision, several blocks of threads collaborate in the computation of one matrix-vector product.

two stages, three kernels

Algorithm 1: TILED ACCELERATED BACK SUBSTITUTION.

Input : N is the number of tiles,
n is the size of each tile,
U is an upper triangular Nn-by-Nn matrix,
$\mathbf{b}$ is a vector of size Nn.

Output : $\mathbf{x}$ is a vector of size Nn: $U\mathbf{x} = \mathbf{b}$.

1. Let U_1, U_2, ..., U_N be the diagonal tiles.
   The kth thread solves $U_i\mathbf{v}_k = \mathbf{e}_k$, computing the kth column U_i^{-1}.
2. For $i = N, N-1, \ldots, 1$ do
   1. n threads compute $\mathbf{x}_i = U^{-1}\mathbf{b}_i$;
   2. simultaneously update $\mathbf{b}_j$ with $\mathbf{b}_j - A_{j,i}\mathbf{x}_i$,
      $j \in \{1, 2, \ldots, i-1\}$ with $i-1$ blocks of n threads.

A parallel execution could run in time proportional to Nn.

data staging

A matrix U of multiple doubles is stored as $[U_1, U_2, \ldots, U_m]$,

- U_1 holds the most significant doubles of U,
- U_m holds the least significant doubles of U.

Similarly, $\mathbf{b}$ is an array of m arrays $[\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m]$,
sorted in the order of significance.

In complex data, real and imaginary parts are stored separately.

The main advantages of this representation are twofold:

1. facilitates staggered application of multiple double arithmetic,
2. benefits efficient memory coalescing,
   as adjacent threads in one block of threads read/write adjacent data in memory, avoiding bank conflicts.

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

experimental setup

About the input matrices:

- Random numbers are generated for the input matrices.
- Condition numbers of random triangular matrices almost surely grow exponentially [Viswanath & Trefethen, 1998].
- In the standalone tests, the upper triangular matrices are the Us of an LU factorization of a random matrix, computed by the host.

Two input parameters are set for every run:

- The size of each tile is the number of threads in a block.
  The tile size is a multiple of 32.
- The number of tiles equals the number of blocks.
  As the V100 has 80 streaming multiprocessors,
  the number of tiles is at least 80.

back substitution on the V100, milliseconds, Gigaflops

double double precision

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	1.2	9.3	46.3
multiply with inverses	1.7	3.3	8.9
back substitution	7.9	4.7	12.2
time spent by kernels	5.0	17.3	67.4
wall clock time	82.0	286.0	966.0
kernel time flops	190.6	318.7	525.1
wall clock flops	11.7	19.2	36.7

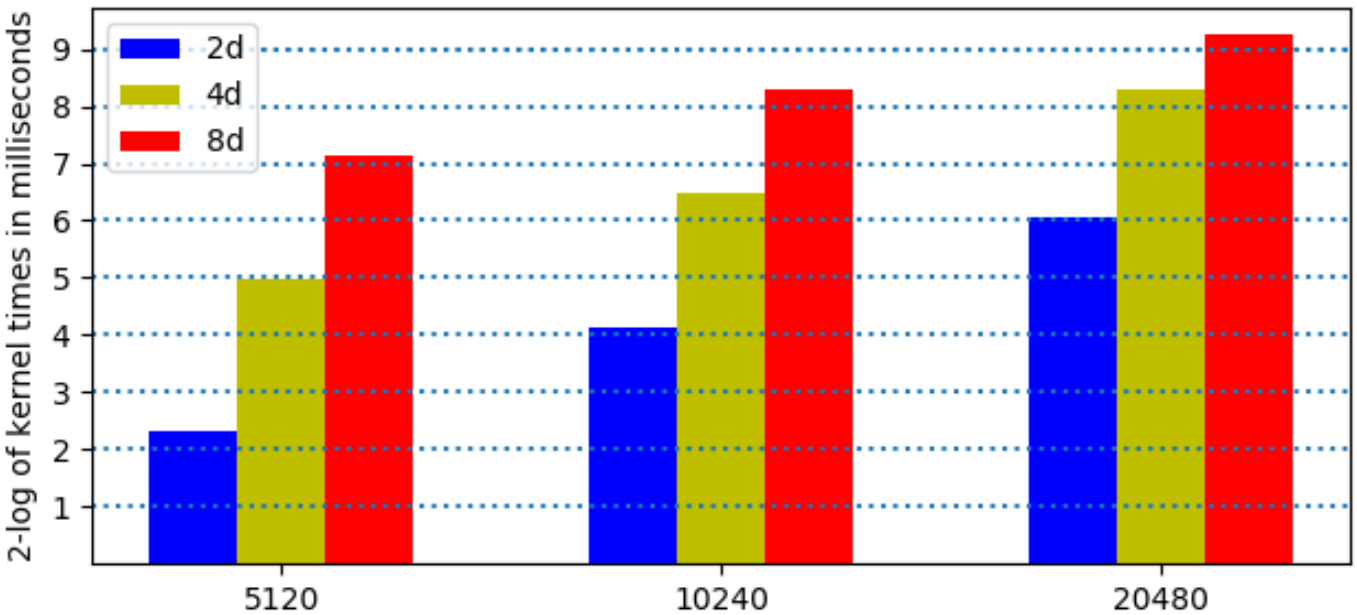
quad double precision

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	6.2	38.3	137.4
multiply with inverses	12.2	23.8	63.1
back substitution	13.3	26.7	112.2
time spent by kernels	31.7	88.8	312.7
wall clock time	187.0	619.0	2268.0
kernel time flops	299.4	614.2	1122.3
wall clock flops	50.8	88.1	154.8

2-logarithms of times on the V100 in 3 precisions

Consider the doubling of the dimension and the precision.

1. Double the dimension, expect the time to quadruple.
2. From double double to quad double: 11.7 is multiplier, from quad double to octo double: 5.4 times longer.

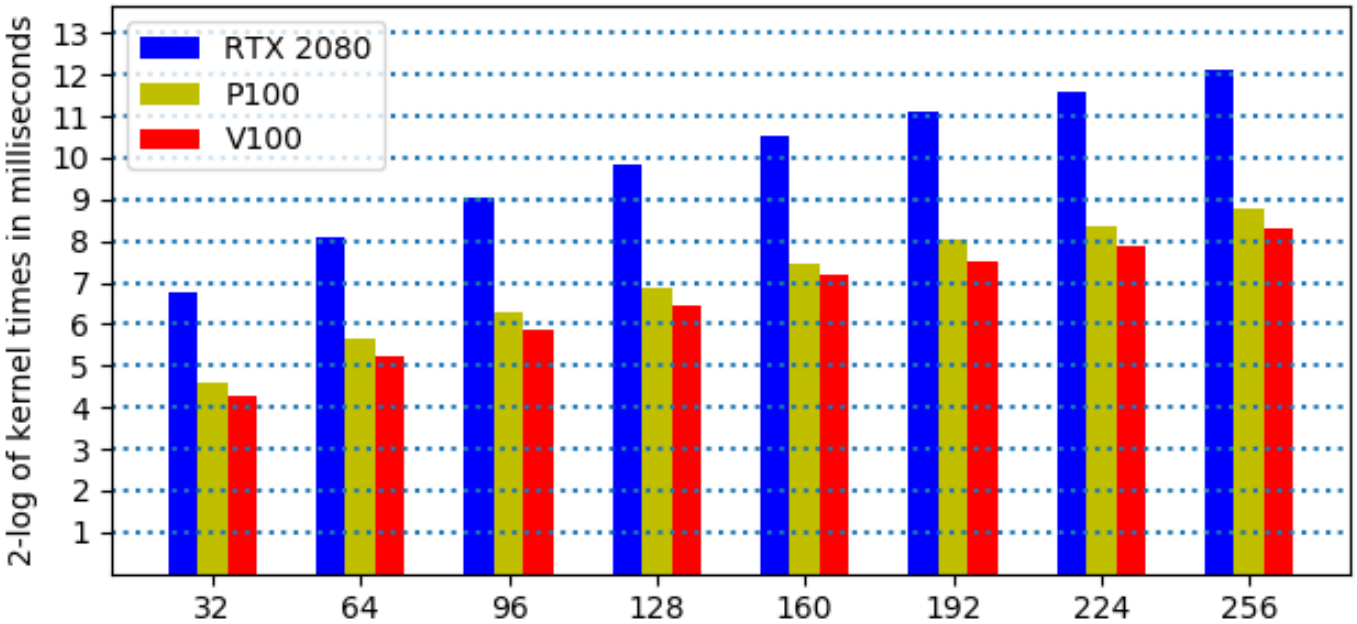


The heights of the bars are closer to each other in higher dimensions.

kernel times in quad double precision on 3 GPUs

The V100 has 80 multiprocessors,
its theoretical peak performance is 1.68 times that of the P100.

The value for N is fixed at 80, n runs from 32 to 256:



Observe the heights of the bars as the dimensions double
and the relative performance of the three different GPUs.

$20480 = 320 \times 64 = 160 \times 128 = 80 \times 256$

Back substitution in quad double precision, for $20480 = N \times n$, for three different combinations of N and n, on the V100.

stage in Algorithm 1	320×64	160×128	80×256
invert diagonal tiles	13.5	35.8	132.3
multiply with inverses	49.0	47.5	64.3
back substitution	84.6	91.7	112.3
time spent by kernels	147.1	175.0	308.9
wall clock time	2620.0	2265.0	2071.0
kernel time flops	683.0	861.1	1136.1
wall clock flops	38.3	66.5	169.5

The units of all times are milliseconds, flops unit is Gigaflops.

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

the blocked Householder QR

Consider a $3m$-by-$3n$ tiled matrix, $m \geq n$:

$$A = \left[\begin{array}{c|c|c} & A_{1,2} & \\ \hline A_{1,1} & A_{2,2} & A_{2,3} \\ \hline & & A_{3,3} \end{array}\right], \qquad \begin{array}{l} A_{1,1} \text{ is } 3m\text{-by-}n, \\ A_{1,2} \text{ is } m\text{-by-}2n, \ A_{2,3} \text{ is } m\text{-by-}n, \\ A_{2,2} \text{ is } 2m\text{-by-}n, \ A_{3,3} \text{ is } m\text{-by-}n. \end{array}$$

The Householder transformations are accumulated in an orthogonal $3m$-by-$3m$ matrix Q.

The upper triangular reduction R of A is written in the matrix A.

Early GPU implementations:

- Baboulin, Dongarra, and Tomov, TR UT-CS-08-200, 2008
- Kerr, Campbell, and Richards, GPGPU'09 conference
- Volkov and Demmel, Conference on Supercomputing, 2008

evolution of Q and R, I is the identity matrix

$$
\begin{aligned}
A,\ Q &= \left[\begin{array}{c|c|c} A_{1,1} & \begin{array}{c|c} A_{1,2} & \\ \hline A_{2,2} & \begin{array}{c} A_{2,3} \\ \hline A_{3,3} \end{array} \end{array} & \end{array}\right], \quad \left[\begin{array}{c|c|c} I & & \\ \hline & I & \\ \hline & & I \end{array}\right] \\
&\rightarrow \left[\begin{array}{c|c|c} R_{1,1} & \begin{array}{c|c} R_{1,2} & \\ \hline A_{2,2} & \begin{array}{c} A_{2,3} \\ \hline A_{3,3} \end{array} \end{array} & \end{array}\right], \quad \left[\begin{array}{c|c|c} Q_1 & I & \\ \hline & & I \end{array}\right] \\
&\rightarrow \left[\begin{array}{c|c|c} R_{1,1} & \begin{array}{c|c} R_{1,2} & \\ \hline R_{2,2} & \begin{array}{c} R_{2,3} \\ \hline A_{3,3} \end{array} \end{array} & \end{array}\right], \quad \left[\begin{array}{c|c|c} Q_1 & Q_2 & \\ \hline & & I \end{array}\right] \\
&\rightarrow \left[\begin{array}{c|c|c} R_{1,1} & \begin{array}{c|c} R_{1,2} & \\ \hline R_{2,2} & \begin{array}{c} R_{2,3} \\ \hline R_{3,3} \end{array} \end{array} & \end{array}\right], \quad \left[\begin{array}{c|c|c} Q_1 & Q_2 & Q_3 \end{array}\right].
\end{aligned}
$$

Householder Reflectors

The Householder reflector P is represented by a vector $\mathbf{v}$:

$$P = I - \beta \mathbf{v}\mathbf{v}^T, \quad \beta = 2/\mathbf{v}^T\mathbf{v}, \quad P\mathbf{x} = \|\mathbf{x}\|_2 \mathbf{e}_1,$$

where $\mathbf{x}$ is the current column and $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$.

The Householder matrices are aggregated in an orthogonal matrix

$$P_{WY} = I + WY^T,$$

where Y stores the Householder vectors in a trapezoidal shape.
W is defined by the Householder vectors and the corresponding βs.

With this WY representation, the updates to Q and R are

$$\begin{aligned} Q &= Q + Q \star W \star Y^T, \\ R &= R + Y \star W^T \star C, \end{aligned}$$

which involve many matrix-matrix products.

four stages, several kernels

Algorithm 2: BLOCKED ACCELERATED HOUSEHOLDER QR.

Input : N is the number of tiles,
n is the size of each tile,
M is the number of rows, $M \geq Nn$,
A is an M-by-Nn matrix.

Output : Q is an orthogonal M-by-M matrix,
R is an M-by-Nn matrix, $A = QR$.

For k from 1 to N do

1. Compute Householder vectors for one tile, reduce $R_{k,k}$.
2. Define Y, compute W and $Y \star W^T$.
3. Add $Q \star YW^T$ to update Q.
4. If $k < N$, add $YW^T \star C$ to update R.

Householder QR is cubic in the dimension

The code is written for multiple double precision.

Thanks to the high Compute to Global Memory Access ratios of multiple double precision, entries of a matrix can be loaded directly into the registers of a kernel (bypassing shared memory).

The computation cost is proportional to M^3, for $M = Nn$.
The hope is to reduce the cost by a factor of M.

Least Squares on GPUs in Multiple Double Precision

1. Problem Statement
   - least squares solving
   - graphics processing units
2. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results
3. Accelerated Blocked Householder QR
   - accumulating the Householder reflectors
   - experimental results

5 GPUs on 8×128, double double, milliseconds, Gigaflops

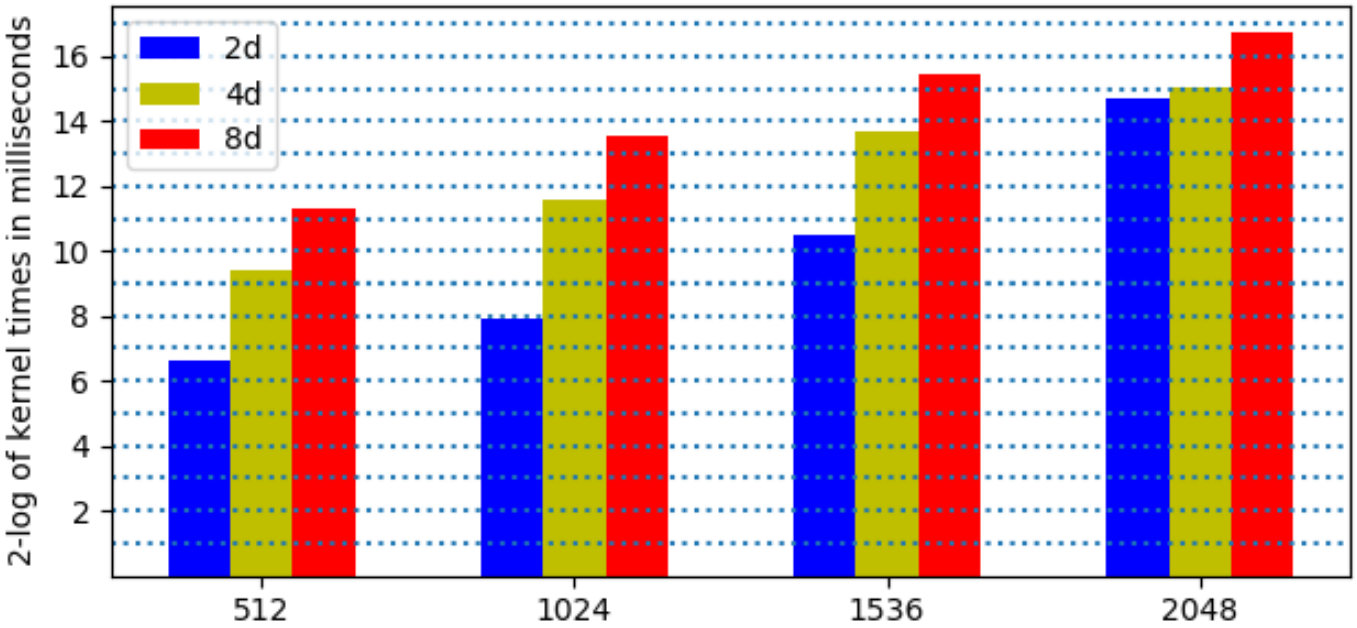
stage in Algorithm 2	Linux on the host				Windows
	C2050	K20C	P100	V100	RTX 2080
β, v	35.5	43.8	21.4	16.2	26.2
$\beta R^T \star v$	418.8	897.8	89.6	76.6	389.7
update R	107.0	107.6	23.0	15.2	47.5
compute W	1357.8	1631.8	349.2	222.4	1298.4
$Y \star W^T$	100.0	50.3	9.7	6.6	153.5
$Q \star WY^T$	790.9	423.9	77.2	52.1	1228.8
$YWT \star C$	6068.5	2345.2	141.2	61.6	822.6
$Q + QWY$	2.4	1.6	0.4	0.4	0.7
$R + YWTC$	7.4	4.2	0.7	0.5	0.8
all kernels	8888.3	5506.1	712.4	451.5	3968.2
wall clock	9083.0	5682.0	826.0	568.0	4700.0
kernel flops	115.8	187.0	1445.3	2280.4	259.5
wall flops	113.4	181.2	1247.2	1812.7	219.1

2-logarithms of times on the V100 in 3 precisions

For increasing dimensions,
we expect the time to be multiplied by 8 when the dimension doubles.

Increasing precisions, multipliers are 11.7 (2d to 4d) and 5.4 (4d to 8d).

$512 = 4 \times 128$, $1024 = 8 \times 128$, $1536 = 12 \times 128$, $2048 = 16 \times 128$

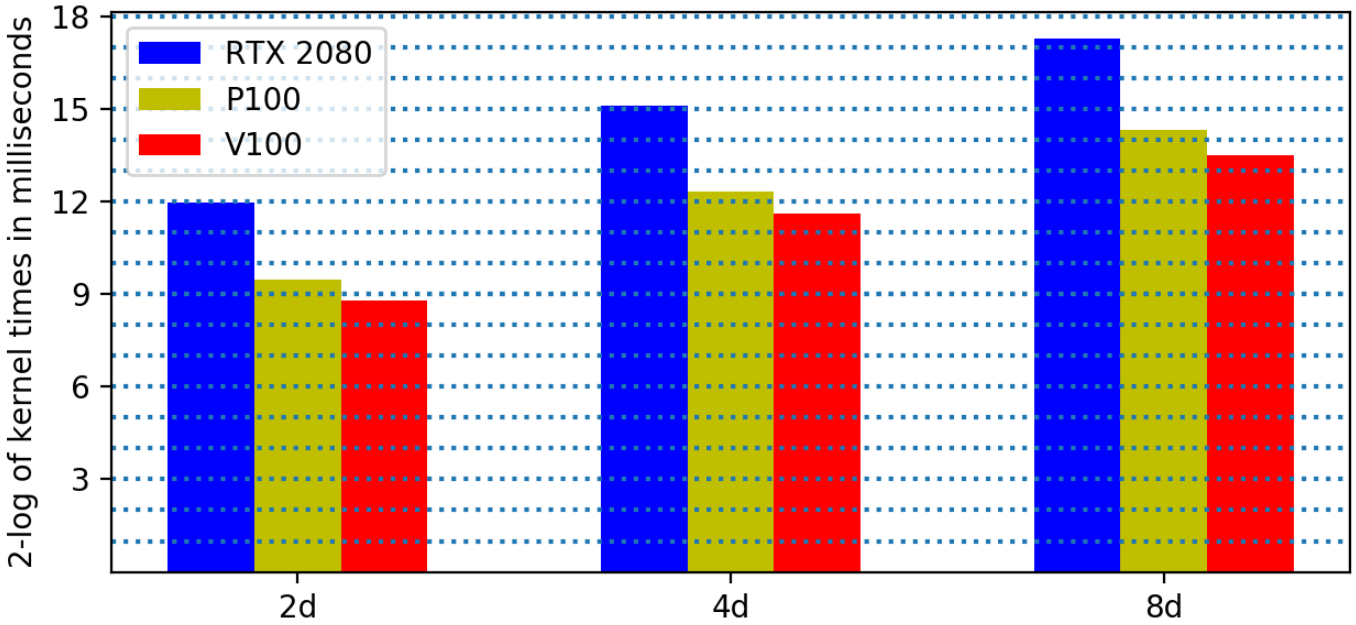


The performance drops at 2048 in double double precision.

2-logarithms of times on 3 GPUs in 3 precisions

Increasing precisions, multipliers are 11.7 (2d to 4d) and 5.4 (4d to 8d).

On 8 tiles for size 128:



The observed cost overhead factors are less than the multipliers.
The heights of the bars follow a regular pattern.

least squares on the V100 in 4 precisions, 8×128

BS = Back Substitution, times in milliseconds, flops unit is Gigaflops.

stage	1d	2d	4d	8d
QR kernel time	157.9	451.1	3020.6	11924.5
QR wall time	204.0	566.0	3203.0	12244.0
BS kernel time	2.0	4.0	28.0	114.5
BS wall time	4.0	7.0	35.0	127.0
QR kernel flops	303.4	2282.2	3369.8	4041.4
QR wall flops	235.1	1819.6	3177.8	3936.1
BS kernel flops	8.1	89.8	127.9	149.1
BS wall flops	4.2	49.8	102.9	134.5
total kernel flops	299.6	2262.9	3340.0	4004.4
total wall flops	230.8	1797.3	3144.7	3897.0

Teraflop performance is attained already in double double precision.

conclusions

Taking 439, the average number of double operations in the tallies of the operational counts for quad double arithmetic, as the scaling factor, teraflop performance on a GPU can be viewed as 2.2 gigaflops on a single threaded computation.

Using this interpretation, the experiments show that GPU acceleration does compensate the cost overhead of quad double arithmetic.

In any case, the observed cost overhead ratios in going from double double to quad double are less than the ratios predicted by the operational count tallies, thanks to the high CGMA ratios.

The good performance on problems of dimension 1,024 in quad double precision observed on many GPUs should be encouraging to consider the use of multiple double arithmetic in scientific applications.

All code is available under the GPL-3.0 license at

`https://github.com/janverschelde/PHCpack/src/GPU/Matrices`