Least Squares on GPUs in Multiple Double Precision

Jan Verschelde*
University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
851 S. Morgan St. (m/c 249), Chicago, IL 60607-7045
Email: janv@uic.edu, URL: http://www.math.uic.edu/~jan.

***Abstract*—This paper describes the application of the code generated by the CAMPARY software to accelerate the solving of linear systems in the least squares sense on Graphics Processing Units (GPUs), in double double, quad double, and octo double precision. The goal is to use accelerators to offset the cost overhead caused by multiple double precision arithmetic. For the blocked Householder QR and the back substitution, of interest are those dimensions at which teraflop performance is attained. The other interesting question is the cost overhead factor that appears each time the precision is doubled.**

Experimental results are reported on five different NVIDIA GPUs, with a particular focus on the P100 and the V100, both capable of teraflop performance. Thanks to the high Compute to Global Memory Access (CGMA) ratios of multiple double arithmetic, teraflop performance is already attained running the double double QR on 1,024-by-1,024 matrices, both on the P100 and the V100. For the back substitution, the dimension of the upper triangular system must be as high as 17,920 to reach one teraflops on the V100, in quad double precision, and then taking only the times spent by the kernels into account. The lower performance of the back substitution in small dimensions does not prevent teraflop performance of the solver at dimension 1,024, as the time for the QR decomposition dominates.

In doubling the precision from double double to quad double and from quad double to octo double, the observed cost overhead factors are lower than the factors predicted by the arithmetical operation counts. This observation correlates with the increased performance for increased precision, which can again be explained by the high CGMA ratios.

Keywords and phrases. acceleration, back substitution, blocked Householder QR, Graphics Processing Unit (GPU), least squares, multiple double, multiprecision.

I. Introduction

Many applications in scientific computing may benefit from extended precision, e.g., [8] applies double doubles. However, the cost overhead caused by multiprecision arithmetic is a valid concern. This paper experimentally demonstrates that this cost overhead can be mitigated by the acceleration on a Graphics Processing Unit (GPU) capable of teraflop performance.

The least squares solution $\mathbf{x}$ of a linear system $A\mathbf{x} = \mathbf{b}$ minimizes the sum of the squares of $\mathbf{b}-A\mathbf{x}$, or $\|\mathbf{b}-A\mathbf{x}\|_2^2$. The decomposition of the matrix A into an orthogonal matrix Q and an upper triangular matrix R, $A = QR$ reduces $A\mathbf{x} = \mathbf{b}$ to $R\mathbf{x} = Q^T\mathbf{b}$, solved by back substitution. The Householder QR factorization is numerically stable [6, Theorem 3.5].

*Supported by the National Science Foundation under grant DMS 1854513.

TABLE I
Operational counts for double double, quad double, and octo double arithmetic. For example: one division (div) of two quad doubles requires 266 additions ($+$), 510 subtractions ($-$), 112 multiplications ($*$), and 5 divisions ($/$) in double precision arithmetic, which sums (Σ) up to 893 double precision floating-point operations and averages to 439.3.

	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

The blocked Householder QR factorization [4] is rich in matrix-matrix products [7], well suited for GPU acceleration, as demonstrated in [3] and [34], with further developments in [1], [2], [15], [25], and [26]. The development of the code for this paper benefited greatly from the exposition in [13]. To develop a GPU accelerated back substitution algorithm, ideas were taken from [21], based on formulas proposed in [9].

The suitability of double double and triple precision Basic Linear Algebra Subroutines (BLAS) was shown in [17], [18].

A. Multiple Double Arithmetic

A multiple double number is an unevaluated sum of multiple doubles. The arithmetical operations on multiple double numbers are defined by algorithms in double precision arithmetic. To extend the double precision m times, compute with m doubles. Table I tallies the cost overhead to multiply the precision with 2, 4, and 8, corresponding respectively to double double, quad double, and octo double precision, to about 32, 64, and 128 decimal places of precision. The averages (37.7, 439.3, 2379.0) predict the arithmetical cost overhead factors.

Parallel algorithms are applied to offset the cost overhead caused by multiple precision arithmetic. A specific question asks for the smallest dimension of the linear system for

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00139

which teraflop performance is obtained. Experiencing teraflop performance in quad double arithmetic on a GPU is similar to about 2.2 gigaflops performance in double arithmetic on a single threaded execution, as the average cost of quad double operations is 439. The 439 is obtained as the average of the Σ column under the quad double header in Table I.

Double double and quad double arithmetic are provided by QDlib [10], with its GPU version in [16]. The software CAMPARY [12] defines code generators for general multiple float and double arithmetical operations. The handbook [19, Chapter 14] describes multiple double arithmetic.

The Compute to Global Memory Access (CGMA) ratio [14] is the number of floating-point calculations performed by a kernel for each access to the global memory. Looking back at the counts in Table I, the division of two quad double numbers requires 893 double precision operations on a total of 8 doubles, naturally leading to a very high CGMA ratio. An alternative to the CGMA ratio is the roofline model [35]. This model is applied in Figure 5 to the tiled accelerated back substitution in quad double precision on the V100.

The specific motivation for this paper is the development of a scalable implementation of a new path tracker [23] to solve systems of polynomial equations in several variables. One component of the path tracker is the solution of a lower triangular block Toeplitz system [5], where the diagonal matrix is the evaluated Jacobian matrix at the current point on the path. An error analysis in [24] motivates the need for multiprecision arithmetic if a guaranteed accuracy is desired. Because of the propagation of roundoff errors, the leading coefficients in the power series must be computed most accurately, at a precision higher than the hardware double precision. Recently, PHCpack [29] was extended [30] with the code for the multiprecision arithmetic generated by the CAMPARY software, and applied to accelerate the polynomial evaluation and differentiation at power series [31].

The power series computation provides input to Padé approximations, applied in the holomorphic embedding load flow method [27], [28], to solve steady state equations of power systems, using complex analysis. As indicated in [22], multiprecision arithmetic adds significant value.

B. On Alternatives to CAMPARY

Compared to genuine multiprecision arithmetic, multiple double numbers have a limited number of precision levels, one cannot specify the precise number of bits in the precision. Another limitation is that the size of the exponents are the same as the exponent size of any double.

The authors of [11] compare CAMPARY and CUMP [20] to their GPU implementation of multiprecision arithmetic based on the multiple residue number system. The double double arithmetic of CAMPARY performs best for the problem of matrix-vector multiplication. Concerning quad double precision, the authors of [11] write "the CAMPARY library is faster than our implementation; however as the precision increases the execution time of CAMPARY also increases significantly."

C. Contributions and Organization

The main result is the teraflop performance obtained for the multiple double precision least squares solver, obtained already for relatively modest dimensions. The code generated by the CAMPARY software is applied to solving linear systems in the least squares sense in double double, quad double, and octo double precision, for real and complex matrices. The resulting programs are self contained, available in a github repository, under the GPL-v3.0 License, thus promoting reproducibility.

The next two sections on accelerating the back substitution and the blocked Householder QR are meant to provide self-contained introductions to the parallel implementations and to explain the legends in the tables in the computational experiments section. Multiple double arithmetic allows for a finer granularity level as more blocks of threads can collaborate in one matrix-vector product. The computational experiments start in the fourth section.

II. Accelerated Back Substitution

Data parallel algorithms execute the same instructions on different data. On graphics processing units, this execution is performed by blocks of threads, scheduled in multiples of 32. These blocks reside on a number of streaming multiprocessors, for a total of several thousands of cores. In order to fully occupy the device, the parallelism must be sufficiently fine involving tens of thousands of threads.

In accelerating the back substitution to solve an upper triangular linear system, the coefficient matrix is divided up into tiles. The ideas will be illustrated on a 3-by-3 tiled system:

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

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix}, \tag{2}$$

where U_1, U_2, U_3 are upper triangular matrices, with nonzero elements on their diagonal, and $A_{1,2}$, $A_{1,3}$, and $A_{2,3}$ are general matrices. All matrices have the same dimensions. The length of $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$ equals the number of rows in each matrix.

In the traditional version of the back substitution algorithm, the last instruction to compute x_i is the division by the element on the diagonal. To introduce more parallelism, the tiles on the diagonal are first inverted. The parallel back substitution happens in two stages:

1) Invert all tiles on the diagonal:

$$V = \begin{bmatrix} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{bmatrix}. \tag{3}$$

The inverse of an upper triangular matrix is again upper triangular. Each column of the inverse is the solution of an upper triangular system. The columns of the inverse can be computed independently from each other.

2) The back substitution alternates between multiplying with the inverses and updating the right hand side vectors. The statements on the same line below are executed in parallel.

$$\mathbf{x}_3 := U_3^{-1}\mathbf{b}_3, \quad (4)$$

$$\mathbf{b}_2 := \mathbf{b}_2 - A_{2,3}\mathbf{x}_3, \quad \mathbf{b}_1 := \mathbf{b}_1 - A_{1,3}\mathbf{x}_3, \quad (5)$$

$$\mathbf{x}_2 := U_2^{-1}\mathbf{b}_2, \quad (6)$$

$$\mathbf{b}_1 := \mathbf{b}_1 - A_{1,2}\mathbf{x}_2, \quad (7)$$

$$\mathbf{x}_1 := U_1^{-1}\mathbf{b}_1. \quad (8)$$

In each step, at least one matrix-vector multiplication is executed. Each back substitution step requires less work. With multiple double arithmetic, the back substitution steps are executed at a finer level: multiple blocks of threads cooperate to compute one matrix-vector product.

One could be concerned that the matrix inverse would lead to numerical instabilities. However, the tiles are of a much smaller size than the entire matrix, typically by a factor of at least the number of multiprocessors. Smaller upper triangular matrices have smaller condition numbers than larger ones.

The example suffices to introduce the main ideas in the algorithm. To describe the parallelism better, the accelerated algorithm is presented next in a more formal manner.

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 n-by-n tiles on the diagonal of U. Replace each U_i with its inverse U_i^{-1}, with N blocks of n threads. Labeling threads starting the count at 1, the k-th thread in each block solves the upper triangular system $U\mathbf{v} = \mathbf{e}_k$, where k is the k-th n-dimensional unit vector.
2) For $i = N, N-1, \ldots, 1$ do
   a) Compute $\mathbf{x}_i := U_i^{-1}\mathbf{b}_i$ by one block of n threads.
   b) Simultaneously update $\mathbf{b}_j := \mathbf{b}_j - A_{j,i}\mathbf{x}_i$, for $j \in \{1, 2, \ldots, i-1\}$, with $i-1$ blocks of n threads.

Algorithm 1 executes $1 + N(N+1)/2$ kernel launches.

If N streaming multiprocessors are available and n is a good fit to keep the device fully occupied, then the computation of all inverses in the first stage can happen in time proportional to n^2, which is the cost of solving one upper triangular linear system of dimension n.

Each step in the second stage of Algorithm 1 involves a matrix-vector multiplication executed in time proportional to n, done by one block of threads. There are N steps and if sufficiently many multiprocessors are available, then the total cost of the second stage is proportional to Nn. If $n \approx N$, the cost of the first stage can be viewed as proportional to Nn, so a good parallel execution of Algorithm 1 can be done in time proportional to Nn, which corresponds to the dimension of the upper triangular linear system $U\mathbf{x} = \mathbf{b}$.

The formulation of Algorithm 1 does not specify the staging of the data. In particular, the matrix U of multiple doubles is *not* stored as $U = [u_{i,j}]$, where $u_{i,j}$ is a multiple double, but as an array $U = [U_1, U_2, \ldots, U_m]$ of m matrices, where U_1 holds the most significant doubles and U_m holds the least significant doubles. Similarly, the $\mathbf{b}$ in the input of Algorithm 1 is an array of m arrays $[b_1, b_2, \ldots, b_m]$, ordered in the order of significance. This facilitates the staggered application of multiprecision arithmetic and benefits the efficient memory coalescing: adjacent threads in one block of threads read adjacent data in memory, avoiding bank conflicts. This representation naturally extends to complex arrays, where the real and imaginary parts are kept separately.

The two questions which will be answered experimentally are the following. What is the smallest dimension for which teraflop performance is obtained? Obviously, the lower threshold for N should be the number of streaming multiprocessors, and n should be a multiple of 32. The second question asks for the cost overhead factor in the three times the precision is doubled, from double to double double, from double double to quad double, and from quad double to octo double.

III. BLOCKED ACCELERATED HOUSEHOLDER QR

The blocked Householder QR is introduced on a $3m$-by-$3n$ tiled matrix, $m \geq n$:

$$A = \left[\begin{array}{c|c|c} & A_{1,2} & \\ \hline A_{1,1} & A_{2,2} & A_{2,3} \\ \hline & & A_{3,3} \end{array}\right], \quad \begin{array}{l} A_{1,1} \text{ is } 3m\text{-by-}n, \\ A_{1,2} \text{ is } m\text{-by-}2n, \\ A_{2,2} \text{ is } 2m\text{-by-}n, \end{array} \quad (9)$$

$A_{2,3}$ and $A_{3,3}$ are m-by-n. 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. The m-by-m identity matrix is represented by I in the sequence of the evolution of A, Q below:

$$\left[\begin{array}{c|c|c} & A_{1,2} & \\ \hline A_{1,1} & A_{2,2} & A_{2,3} \\ \hline & & A_{3,3} \end{array}\right], \left[\begin{array}{c|c|c} & I & \\ \hline I & & \\ \hline & & I \end{array}\right] \quad (10)$$

$$\rightarrow \left[\begin{array}{c|c|c} & R_{1,2} & \\ \hline R_{1,1} & A_{2,2} & A_{2,3} \\ \hline & & A_{3,3} \end{array}\right], \left[\begin{array}{c|c|c} & I & \\ \hline Q_1 & & \\ \hline & & I \end{array}\right] \quad (11)$$

$$\rightarrow \left[\begin{array}{c|c|c} & R_{1,2} & \\ \hline R_{1,1} & R_{2,2} & R_{2,3} \\ \hline & & A_{3,3} \end{array}\right], \left[\begin{array}{c|c|c} & I & \\ \hline Q_1 & Q_2 & \\ \hline & & I \end{array}\right] \quad (12)$$

$$\rightarrow \left[\begin{array}{c|c|c} & R_{1,2} & \\ \hline R_{1,1} & R_{2,2} & R_{2,3} \\ \hline & & R_{3,3} \end{array}\right], \left[\begin{array}{c|c|c} & I & \\ \hline Q_1 & Q_2 & Q_3 \end{array}\right]. \quad (13)$$

One tile $R_{k,k}$ is computed column by column. For each column, a Householder vector $\mathbf{v}$ and corresponding $\beta = 2/\mathbf{v}^T\mathbf{v}$ value is computed. The Householder reflector $P = I - \beta\mathbf{v}\mathbf{v}^T$ with the $\mathbf{v}$ determined so $P\mathbf{x} = \|\mathbf{x}\|_2\mathbf{e}_1$ (with a sign computation as in [7, Algorithm 5.1.1]), where $\mathbf{x}$ contains the numbers in the current column starting at the diagonal, and where $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. The Householder matrices are aggregated in an orthogonal matrix of the form

$$P_{WY} = I + WY^T, \quad (14)$$

where Y stores the Householder vectors and has a trapezoidal shape. The matrix W is computed from the Householder vectors and their corresponding β values. With this WY representation of the Householder matrices, the updates to Q and R can then written as

$$Q = Q + Q \star W \star Y^T, \quad (15)$$
$$R = R + Y \star W^T \star C, \quad (16)$$

where C is the current matrix to be updated. The above formulas are rich in matrix-matrix products which are very suitable for GPU acceleration. The columns $\mathbf{z}$ of the matrix W follow the formulas

$$\mathbf{z} = -\beta(\mathbf{v} + WY^T\mathbf{v}), \quad (17)$$

which require matrix-vector products. On complex data, the transpose ^{T} is replaced by the Hermitian transpose ^{H}.

As stated in [13], the computation of W is expected to be the bottleneck.

The structured description of the accelerated version of the blocked Householder QR algorithm below serves as an explanation of the legend of the tables in the next section.

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 $k = 1, 2, \ldots, N$ do

1) For $\ell = 1, 2, \ldots n$ do
   a) compute $\mathbf{v}$ and β,
   b) update $R_{k,k}$.

   If the size of the current column is less than n, then only one block of threads computes. Otherwise, several blocks compute $\mathbf{v}$, collaborate to update of $R_{k,k}$, and there is one separate kernel to compute $\beta R^T \star \mathbf{v}$, which also involves a sum reduction with multiple blocks.
2) Given n pairs $(\mathbf{v}, \beta)$ computed in the previous stage, the matrices W, Y, and their product $Y \star W^T$ are computed.
3) Update Q in two stages:
   a) $QWY := Q \star WY^T$, where $WY^T = (YW^T)^T$,
   b) $Q := Q + QWY$.

   Separating the matrix-matrix multiplication from the addition clearly shows the cost differences. In multiple precision arithmetic, the cost of the addition is however not negligible.
4) If $k < N$, then update R, in two stages:
   a) $YWTC := YWT \star C$,
   b) $R := R + YWTC$.

As the code is geared towards multiple double arithmetic, the implementation of the matrix-matrix products differs from the double precision implementations recommended in the literature. When defining kernels for the matrix-matrix multiplication in double precision, tiles of matrices are loaded into shared memory to obtain better CGMA ratios, as explained in [14, Chapter 5]. Thanks to the high CGMA ratios of multiple double precision, the entries of the matrix can be loaded directly into the registers of the kernel that computes one number of the product.

The staging of the data applies the same representation of multiple double vectors and matrices via multiple arrays of doubles, as explained at the end of Algorithm 1.

The computational cost of Algorithm 2 is proportional to M^3, if $M = Nn$, for notational simplicity. If the device is fully occupied, then the hope is to reduce the cost by a factor of M and to observe a time proportional to M^2. In the experiments, as the dimension then doubles, the hope is to observe the total time multiplied by a factor closer to four than to eight.

As before, with the back substitution, the first question is to ask for which dimensions teraflop performance is attained. The second is to experimentally compute the actual cost overhead factors of doubling the precision.

IV. COMPUTATIONAL EXPERIMENTS

In designing the experiments, the first concern is to find sufficiently large dimensions at which teraflop performance is attained, mainly in quad double precision. In the runs at different precisions, timings on the double precision version are listed, but are not used in the comparisons as the implementation was made for multiple double precision arithmetic. Another reason for not comparing the timings of runs in double precision is that the dimensions are not yet large enough to fully occupy the device.

In a first comparison of runs at different precisions, the tile size and the corresponding number of threads per block are fixed to the same number for all precisions. However, in double double precision, the number of threads per block should be higher than in octo double precision.

In all tables, all time units are milliseconds. The units of flops (floating-point operations per second) are gigaflops.

A. Notes on the Implementation

The code for the multiple double arithmetical operations generated by the CAMPARY software [12] was customized for each precision in the following manner. Instead of representing a quad double number 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. If one would be only interested in double double and quad double, then the `double2` and `double4` types of the CUDA SDK will work just as well (as we did in [32]), but then performance drops are to be expected with complex quad doubles already and then also for the more general multiple double arithmetic.

QDlib [10] provides definitions for the square roots and various other useful functions for double double and quad double arithmetic. Those definitions are extended to octo

TABLE II
THE COLUMNS LIST THE CUDA CAPABILITY, THE NUMBER OF MULTIPROCESSORS, THE NUMBER OF CORES PER MULTIPROCESSOR, THE TOTAL NUMBER OF CORES, AND THE GPU CLOCK RATE. FOR EVERY GPU, ITS HOST CPU IS LISTED WITH ITS CLOCK RATE.

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

NVIDIA GPU	host CPU	GHz
Tesla C2050	Intel X5690	3.47
Kepler K20C	Intel E5-2670	2.60
Pascal P100	Intel E5-2699	2.20
Volta V100	Intel W2123	3.60
GeForce RTX 2080	Intel i9-9880H	2.30

TABLE III
BLOCKED HOUSEHOLDER QR IN DOUBLE DOUBLE PRECISION, ON A 1,024-BY-1,024 MATRIX, WITH 8 TILES OF SIZE 128.

stage in Algorithm 2	Linux on the host C2050	K20C	P100	V100	Windows 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

double precision, also with the customization of representing an octo double number as eight different variables.

The `__forceinline__` directive was added to all the device functions that define the multiple double arithmetic. All `.cu` files are compiled with `nvcc -O3`.

For every kernel in the implementation of Algorithms 1 and 2, a small function accumulates the number of arithmetical operations. Then the total number of floating-point operations is computed at the end of the run, using the numbers in Table I as multipliers. In the application of the roofline model, the number of bytes in each computation is obtained from the dimensions of the problem, multiplied by the size of each multiple double number.

Random numbers were generated for the input matrices. In the standalone tests on the back substitution solver, the random upper triangular matrices were computed on the host as the output of an LU factorization, as the condition numbers of random triangular matrices almost surely grow exponentially [33]. All tests were run on well conditioned problems, so the residuals $\|\mathbf{b} - A\mathbf{x}\|_2^2$ of the computed solution $\mathbf{x}$ to the linear system $A\mathbf{x} = \mathbf{b}$ is of the expected accuracy, corresponding to the level of the multiple double precision.

The same code runs on five different NVIDIA GPUs. The C2050, K20C, P100, V100 are housed in CentOS workstations and the gcc compiler is used to compile the code on the host. The RTX 2080 resides in a Windows laptop, and the community edition of Microsoft Visual Studio is used.

The code is free and open source, released under the GPU GPL license, in the PHCpack source available on `https://github.com/janverschelde/PHCpack`.

B. Equipment

Using the same setup as in [31], Table II lists the main characteristics of five GPUs used to develop the code.

While running the same software on different GPUs is convenient, the obvious disadvantage is that the more advanced features of the newer devices are not utilized. Important in the investigation of the scalability is the attention to teraflop performance and the ratios of the theoretical peak performances of the V100 over the P100.

In each run, the elapsed times of the kernel launches are measured by `cudaEventElapsedTime` and are expressed in milliseconds. The wall clock times include the sum of times spent by the kernels, with the added memory transfers. The kernel flops in the tables below are the totals of the counts of the double precision operations over the sum of the times spent by the kernels. The total wall clock time is used in the wall flops.

C. Blocked Householder QR on Five Different GPUs

The theoretical double peak performance of the P100 and the V100 are 4.7 TFLOPS and 7.9 TFLOPS respectively. Therefore, if the code scales well, one may expect the V100 to be about 1.68 times faster than the P100.

For the total kernel time in Table III, compare the scaled observed time on the V100: $451.5 \times 1.68 \approx 758.5$, with the observed 712.4 of the P100. Comparing wall clock times is harder, because of different clock speeds of the host processor and the workstation that hosts the P100 has 256GB of RAM, whereas the RAM of the host of the V100 holds 32GB.

For historical perspective, the oldest C2050 was purchased in 2011 and the V100 in 2019. The ratio of the sum of the times spent on all kernels of the C2050 over V100: $8888.3/451.5 \approx 19.6$, indicating about a double speedup every two years. The tile size of 128 (and the number of threads per block) is most likely not the best choice for the K20C, which has 192 cores per streaming multiprocessor. We used the K20C in [32]. In the single experiment comparison in Table III, the GeForce RTX 2080 Max-Q outperforms the K20C.

D. Blocked Householder QR in Four Different Precisions

Based on the operational counts in Table I, one could predict the overhead factors from the averages in the Σ column, which are 37.7 for double double, 439.3 for quad double, and 2379.0 for octo double. Based on those averages, going from double double to quad double would cause all times to be multiplied by 11.7. Similarly, the predicted overhead factor is 5.4 when going from quad double to octo double.

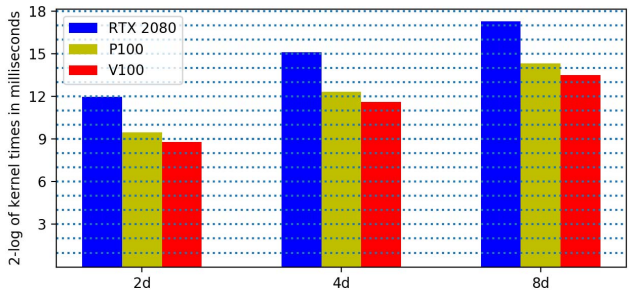

Fig. 1. 2-logarithms of the times spent by all kernels of QR on the RTX 2080, the P100, and the V100 in double double (2d), quad double (4d), and octo double (8d) precision, with data in Table IV.

Table IV illustrates the cost overhead of doubling the precision three times and is summarized by Figure 1. From double double to quad double, taking ratios of kernels times $5187.0/712.7 \approx 7.3$ on the P100, and the similar ratio on the V100 is $3167.0/446.8 \approx 7.1$. Both ratios are consistent and less than the predicted factor of 11.7. On the RTX 2080, the ratio is $35826.7/3999.5 \approx 9.0 < 11.7$. From quad double to octo double, the kernels time ratio on the P100 is $20547.5/5187.0 \approx 4.0$ and $11754.6/3167.0 \approx 3.7$. In both cases, the observed factors are less than the predicted 5.4, as is also the case on the RTX 2080: $160802.8/35826.7 \approx 4.5$. That the observed cost overhead factors are more favorable than the predicted ones correlates with the increased performance for increased precisions.

E. Real and Complex Double Double QR

Working with complex arithmetic requires about four times as many arithmetical operations than on real data. Table V lists times on 512-by-512 matrices, of real and complex double double numbers. Keeping the dimension 512 constant, fewer tiles but larger tiles are selected with each execution.

Looking at the flops in Table V, teraflop performance is reached for both real and complex matrices, for 128 as the tile size. At 4×128, the device is best occupied. But if interested in total wall clock times, then 16×32 is best.

In a dimension as small as 512, the computation of W dominates. Would this still be the case if the dimensions increase?

F. Quad Double QR for Increasing Dimensions

How do the execution times of the QR decomposition evolve for increasing dimensions? Table VI lists the times for dimensions 512, 1024, 1536, and 2048. Figure 2 shows the evolution of all kernel times.

At dimension 512, the computation of W dominates in all precisions. The accumulated times of all kernels devoted to computing W drops to the thirdmost largest time in dimension 2048. The two most time consuming kernels are those that do the matrix-matrix multiplications.

Doubling the dimension, from 512 to 1024, the ratios of the wall clock times in double double, quad double, and octo double precision are respectively $321.0/155.0 \approx 2.1$, $3366.0/777.0 \approx 4.3$, and $12735.0/2681.0 \approx 4.8$, corresponding to significant increases in performance.

TABLE IV
BLOCKED HOUSEHOLDER QR IN DOUBLE (1D), DOUBLE DOUBLE (2D), QUAD DOUBLE (4D), AND OCTO DOUBLE (8D) PRECISION, ON A 1,024-BY-1,024 MATRIX, WITH 8 TILES OF SIZE 128, ON THE RTX 2080, THE P100, AND THE V100.

stage in Algorithm 2	times on the RTX 2080			
	1d	2d	4d	8d
β, v	13.0	26.3	108.1	451.8
$\beta R^T \star v$	46.3	338.0	1740.9	4994.3
update R	11.3	47.7	376.9	1669.6
compute W	111.7	1309.4	12346.8	56484.2
$Y \star W^T$	5.1	154.7	1476.3	6746.7
$Q \star WY^T$	40.2	1238.7	11815.5	54008.9
$YWT \star C$	110.1	833.3	7957.3	36430.5
$Q + QWY$	0.3	0.7	3.1	9.4
$R + YWTC$	0.5	0.8	1.9	7.3
all kernels	338.6	3999.5	35826.7	160802.8
wall clock	562.0	4708.0	37087.0	163219.0
kernel flops	141.5	257.4	284.1	299.7
wall flops	85.2	218.7	274.5	295.3
stage in Algorithm 2	times on the P100			
	1d	2d	4d	8d
β, v	12.6	21.6	58.3	412.4
$\beta R^T \star v$	44.4	89.7	760.7	2998.5
update R	14.2	23.0	96.3	359.9
compute W	98.3	349.3	2752.3	9857.5
$Y \star W^T$	3.0	9.7	96.8	484.7
$Q \star WY^T$	25.0	77.0	747.0	3745.4
$YWT \star C$	67.1	141.3	672.8	2681.7
$Q + QWY$	0.2	0.4	0.9	2.0
$R + YWTC$	0.4	0.7	1.8	5.5
all kernels	256.2	712.7	5187.0	20547.5
wall clock	311.0	827.0	5381.0	20870.0
kernel flops	180.6	1444.6	1962.4	2345.4
wall flops	154.0	1244.8	1891.5	2309.2
stage in Algorithm 2	times on the V100			
	1d	2d	4d	8d
β, v	7.3	15.8	37.4	180.2
$\beta R^T \star v$	28.8	77.2	470.4	1304.0
update R	9.4	15.1	58.9	197.9
compute W	79.5	223.2	1551.0	5700.9
$Y \star W^T$	0.8	6.5	66.3	281.3
$Q \star WY^T$	8.2	56.7	516.8	2249.2
$YWT \star C$	24.0	51.4	464.4	1834.5
$Q + QWY$	0.2	0.4	0.8	1.6
$R + YWTC$	0.2	0.4	1.0	4.8
all kernels	158.4	446.8	3167.0	11754.6
wall clock	206.0	560.0	3356.0	12059.0
kernel flops	302.5	2304.3	3214.0	4099.9
wall flops	232.8	1837.3	3033.0	3996.3

While teraflop performance is maintained, notice in Table VI the drop in performance at dimension 2048 in double double arithmetic. This drop is most likely due to the kernels for the matrix-matrix multiplication that do not take advantage of the shared memory, as the double double arithmetic has not yet a high enough CGMA ratio, compared to the higher multiple double arithmetic. Although not as much as in double double precision, there is also a drop in performance in the other precisions for the two largest dimensions.

TABLE V
BLOCKED HOUSEHOLDER QR IN DOUBLE DOUBLE PRECISION, ON REAL AND COMPLEX MATRICES OF DIMENSION 512, FOR INCREASING TILE SIZES, $512 = 16 \times 32 = 8 \times 64 = 4 \times 128 = 2 \times 256$, ON THE V100.

stage in Algorithm 2	on real matrices			
	16×32	8×64	4×128	2×256
β, v	6.5	10.7	7.8	7.7
$\beta R^T \star v$	12.4	22.0	20.2	20.0
update R	2.3	4.9	9.9	46.6
compute W	22.9	41.9	54.1	81.7
$Y \star W^T$	0.5	0.9	1.0	1.1
$Q \star WY^T$	4.3	7.0	3.9	2.8
$YWT \star C$	4.0	6.3	3.6	1.5
$Q + QWY$	0.1	0.1	0.1	0.1
$R + YWTC$	0.1	0.1	0.1	0.1
all kernels	53.2	94.0	100.5	161.6
wall clock	101.0	170.0	155.0	208.0
kernel flops	428.4	785.9	1089.8	777.3
wall flops	226.6	434.5	707.4	603.3
stage in Algorithm 2	on complex matrices			
	16×32	8×64	4×128	2×256
β, v	8.5	8.4	8.3	8.9
$\beta R^T \star v$	20.6	36.8	36.7	37.3
update R	3.0	6.8	20.5	204.7
compute W	38.9	126.6	144.3	248.9
$Y \star W^T$	0.9	3.3	3.7	4.5
$Q \star WY^T$	12.7	26.4	15.1	11.3
$YWT \star C$	12.4	18.6	9.8	5.1
$Q + QWY$	0.2	0.2	0.1	0.1
$R + YWTC$	0.3	0.2	0.1	0.1
all kernels	97.4	227.4	238.5	420.8
wall clock	158.0	306.0	311.0	479.0
kernel flops	628.9	1299.8	1836.7	1194.8
wall flops	387.2	967.3	1407.8	1050.5

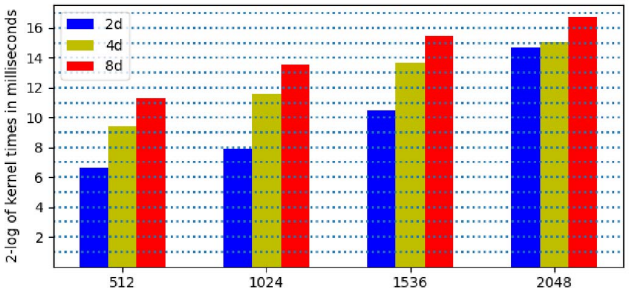

Fig. 2. 2-logarithms of the times spent by all kernels of QR on the V100 in double double (2d), quad double (4d), and octo double (8d) precision, for increasing dimensions, for the data in Table VI.

TABLE VI
BLOCKED HOUSEHOLDER QR IN DOUBLE DOUBLE, QUAD DOUBLE, AND OCTO DOUBLE PRECISION, ON REAL MATRICES OF INCREASING DIMENSIONS, FOR INCREASING NUMBER OF TILES, $512 = 4 \times 128$, $1024 = 8 \times 128$, $1536 = 12 \times 128$, $2048 = 16 \times 128$, ON THE V100.

stage in Algorithm 2	double double precision			
	512 4×128	1024 8×128	1536 12×128	2048 16×128
β, v	7.9	8.2	16.5	34.5
$\beta R^T \star v$	20.3	36.7	144.6	652.0
update R	9.9	20.5	28.9	58.8
compute W	53.9	144.2	556.2	2278.7
$Y \star W^T$	1.0	3.7	24.9	194.2
$Q \star WY^T$	4.0	15.1	201.3	3048.9
$YWT \star C$	3.4	9.7	481.6	20534.4
$Q + QWY$	0.1	0.1	0.7	5.6
$R + YWTC$	0.1	0.1	0.9	7.8
all kernels	100.5	238.2	1455.8	26815.0
wall clock	155.0	321.0	1627.0	27230.0
kernel flops	1089.7	1839.0	2475.1	1087.8
wall flops	706.5	1364.9	2214.4	1071.2
stage in Algorithm 2	quad double precision			
	512 4×128	1024 8×128	1536 12×128	2048 16×128
β, v	21.0	37.4	54.1	71.6
$\beta R^T \star v$	115.5	470.9	1073.8	1939.9
update R	49.0	59.0	74.6	91.2
compute W	412.6	1553.5	3438.2	6104.3
$Y \star W^T$	9.6	66.3	214.2	517.6
$Q \star WY^T$	41.5	538.3	2511.0	7643.9
$YWT \star C$	24.9	409.3	6057.1	17991.2
$Q + QWY$	0.1	0.8	2.5	5.7
$R + YWTC$	0.1	0.9	5.6	7.0
all kernels	674.3	3136.5	13431.2	34372.5
wall clock	777.0	3366.0	13835.0	34960.0
kernel flops	1605.7	3245.3	2366.8	2097.0
wall flops	1392.6	3024.4	2297.7	2061.7
stage in Algorithm 2	octo double precision			
	512 4×128	1024 8×128	1536 12×128	2048 16×128
β, v	94.7	188.3	282.8	385.1
$\beta R^T \star v$	309.5	1309.1	3009.7	5416.5
update R	167.5	199.0	245.1	300.4
compute W	1568.2	5828.6	12908.6	22944.3
$Y \star W^T$	48.3	308.8	957.5	2082.8
$Q \star WY^T$	187.2	2334.8	11259.2	34508.6
$YWT \star C$	114.9	2104.0	15982.0	42044.8
$Q + QWY$	0.3	1.6	5.1	11.6
$R + YWTC$	0.2	5.9	29.8	75.2
all kernels	2490.8	12280.1	44679.8	107769.2
wall clock	2681.0	12735.0	45419.0	108763.0
kernel flops	2058.2	3924.4	3368.5	3166.4
wall flops	1912.0	3784.2	3313.6	3137.5

G. Back Substitution in Four Different Precisions

The high CGMA ratios makes that the overhead cost of doubling the precisions is less than the predicted overhead factors. Would this also be the case for the back substitution?

Consider the doubling of both the dimension and the precision. Table VII records the times of the back substitution on upper triangular matrices of sizes $5120 = 64 \times 80$, $10240 = 128 \times 80$, and $20480 = 256 \times 80$, where the first factors in the dimensions are the size of each tile and the second factors are the number of tiles. In octo double precision, shared memory capacities limit the tile size to 128, so then $20480 = 128 \times 160$. The high wall clock time in octo double precision for 20480 is due to the limited 32 GB of RAM at the host. Despite this anomaly, the times spent by all kernels appear regular enough to reliably measure the cost overhead factors from doubling the precisions.

In double precision, the times spent by the kernels are not large enough to attain a good performance. At the largest dimension, half a teraflop is reached in double double precision; in quad and octo double precision, 1.1 teraflop is observed.

Figure 3 shows the 2-logarithms of the times spent by all kernels. As the cost of the back substitution is quadratic in the dimension, the times are expected to quadruple when the

TABLE VII
BACK SUBSTITUTION IN FOUR DIFFERENT PRECISIONS ON PROBLEMS OF INCREASING SIZE, ON THE V100.

double precision			
stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	0.4	5.2	30.8
multiply with inverses	0.8	1.5	4.3
back substitution	1.8	2.2	5.9
time spent by kernels	3.0	8.9	41.0
wall clock time	47.0	147.0	526.0
kernel time flops	14.5	28.5	39.9
wall clock flops	0.9	1.7	3.1
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
octo double precision			
stage in Algorithm 1	64×80	128×80	128×160
invert diagonal tiles	43.8	110.6	133.3
multiply with inverses	47.7	97.5	196.0
back substitution	49.2	108.0	283.7
time spent by kernels	140.7	316.2	613.1
wall clock time	465.0	1400.0	84448.0
kernel time flops	321.3	820.1	1166.7
wall clock flops	97.1	185.2	8.5

TABLE VIII
BACK SUBSTITUTION IN QUAD DOUBLE PRECISION IN DIMENSION $20480 = N \times n$, 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

dimension is doubled. This quadrupling is observed in double double precision, but then becomes closer to doubling in octo double precision, thanks to the higher performance in higher precision. Observe that the heights of the quad double bar is closer to the octo double bar than to the double double bar. This is consistent with the predicted cost overhead ratios, which are higher when going from double double to quad double compared to the ratios from quad double to octo double.

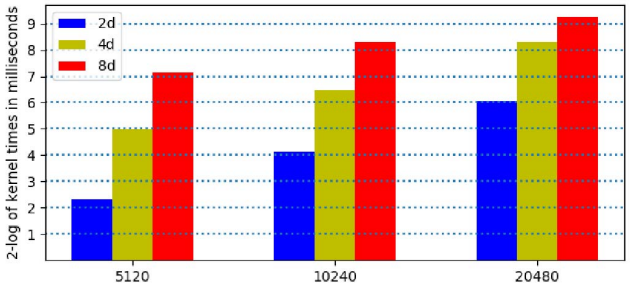

Fig. 3. 2-logarithms of the times in Table VII spent by all kernels of back substitution on the V100, for dimension 5120, 10240, 20480, in double (1d), double double (2d), quad double (4d), and octo double (8d) precision.

H. Tiled Back Substitution in Quad Double Precision

Table VIII lists times for different choices of N and n. The V100 has 80 streaming multiprocessors, so in Table IX, $N = 80$ and multiples of 32 are taken for n, in runs on matrices of dimension 2,560, 5,120, 7,680, 10,240, 12,800, 15,360, 17,920, and 20,480. Teraflop performance on the V100 is attained for $n = 224$, at dimension $80 \times n = 17,920$. Reading the first two lines of Table IX, observe that the time to invert the diagonal tiles increases from a tiny 1.9 to 11.4 milliseconds as the dimension doubles, and from $n = 96$ on, the time spent on inverting the diagonal tiles dominates the times of the other two stages. The difference between the wall clock time and the time spent by all kernels is significant.

Figure 4 shows the evolution of the times spent by all kernels listed in Table IX. In the 2-logarithm plot, an increase of one unit in the height of a bar equals a doubling of the time. For which dimensions is the cost of Algorithm 1 proportional to Nm? If the dimension doubles from 2,560 to 5,120 and from 5,120 to 10,240 (corresponding to the bars for 32, 64, and 128 in Figure 4), the doubling of the time is observed for the P100 and the V100, for the RTX 2080, the increase from dimension 5,120 to 10,240 is more than three times.

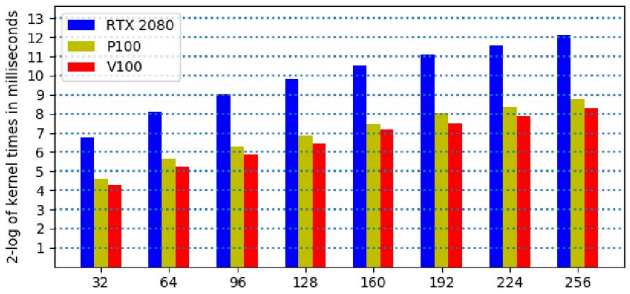

Fig. 4. 2-logarithms of the times spent by all kernels for the back substitution on the RTX 2080, the P100, and the V100 in quad double precision.

Computing the ratios of the times spent by all kernels on the P100 over the V100 gives $732.2/237.1 \approx 3.1$ for dimension 17,920 and $813.1/314.5 \approx 2.6$ for dimension 20,480. That those ratios are still far above the expected 1.68 ratio is most likely because the number 80 (of blocks and tiles) coincides

TABLE IX
TILED ACCELERATED BACK SUBSTITUTION IN QUAD DOUBLE PRECISION ON THE V100. THE DIMENSION OF THE MATRICES ARE MULTIPLES OF 80, THAT IS: $80 \times n$, WHERE $n = 32, 64, 96, 128, 160, 192, 224,$ AND 256.

stage in Algorithm 1	32	64	96	128	160	192	224	256
invert diagonal tiles	1.9	11.4	21.2	36.3	61.8	78.9	103.3	138.2
multiply with inverses	6.4	12.7	18.2	23.9	38.9	47.1	55.2	63.1
back substitution	11.3	13.8	19.8	26.2	44.2	58.6	78.6	113.2
time spent by kernels	19.6	37.8	59.2	86.4	145.0	184.6	237.1	314.5
wall clock time	90.0	251.0	482.0	776.0	1181.0	1577.0	2150.0	2886.0
kernel time flops	94.9	250.9	439.6	631.7	677.4	867.0	1025.9	1115.9
wall clock flops	20.7	37.8	54.0	70.3	83.1	101.5	113.2	121.6

with the number of streaming multiprocessors of the V100, whereas the P100 has 64 streaming multiprocessors.

What is the best choice of N and n for a matrix of dimension 20,480? For the parallelism in GPU acceleration, fixing N at 80 gives the best performance as illustrated in Table VIII. Doubling n from 64 to 128, and to 256 increases the time spent by all kernels, but decreases the total wall clock time from 2.620 seconds to 2.071 seconds, as the performance then nearly doubles.

The roofline model [35] is applied to visualize the performance. The *arithmetic intensity* of a computation is the ratio of the number of floating point operations over the number of bytes in the computation. For the V100, the ridge point is computed as 7900/870 = 9.08, as the ratio of the theoretical peak performance and the memory bandwidth. Problems with an arithmetic intensity larger than 9 are compute bound, as their performance is bounded by the theoretical peak performance of 7.9 TFLOPS. A problem with an arithmetic intensity less than 9 is memory bound, as its performance is bounded by the memory bandwidth of 870 GB/second. Table X lists the arithmetic intensities for the back substitution in quad double precision, for dimensions that are multiples of 80. Figure 5 shows the roofline model for this experiment.

The leftmost dot in Figure 5 is an outlier because at $n = 32$, the V100 is only half occupied, as the V100 has 64 cores per streaming multiprocessor. For an increasing number of threads per block, the arithmetic intensity increases.

I. Least Squares Solving in Four Different Precisions

Table XI summarizes the times and the flops of solving a linear system in the least squares sense, in four different precisions. The blocked accelerated Householder QR of Algorithm 2 is followed by Algorithm 1, the tiled accelerated back substitution.

TABLE X
ARITHMETIC INTENSITY (1) AND THE KERNEL TIME FLOPS (2) FOR THE TILED ACCELERATED BACK SUBSTITUTION IN QUAD DOUBLE PRECISION ON THE V100. THE DIMENSIONS ARE MULTIPLES OF 80, THAT IS: $80 \times n$, WHERE $n = 32, 64, 96, 128, 160, 192, 224,$ AND 256.

	32	64	96	128	160	192	224	256
(1)	58.71	1500	2740	4308	6203	8427	10980	13860
(2)	119.1	263.9	440.7	633.8	679.0	852.9	1036.0	1113.6

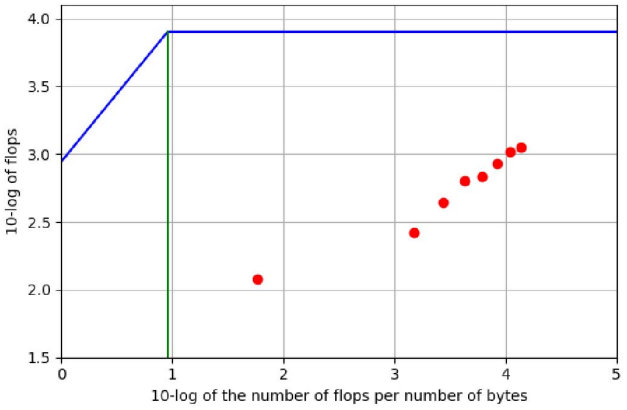

Fig. 5. Roofline plot for the data in Table X. The first coordinate of each dot is the 10-log of the arithmetic intensity and the 10-log of the flops is the second coordinate of each dot. As n increases, the dots move upwards to the right, illustrating that the problem becomes more compute bound.

TABLE XI
LEAST SQUARES SOLVING IN DOUBLE (1D), DOUBLE DOUBLE (2D), QUAD DOUBLE (4D), AND OCTO DOUBLE (8D) PRECISION, ON A 1,024-BY-1,024 LINEAR SYSTEM, WITH 8 TILES OF SIZE 128, ON THE V100. BS = BACK SUBSTITUTION.

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

Comparing the kernel times in Table XI in all precisions shows that the time for the back substitution is about 100 times less than the time for the QR decomposition. Consequently, the lower performance of the back substitution in small dimensions does not lead to a significant reduction in the overall performance of the solver.

As the QR decomposition has a cost that is cubic in the dimension, versus the quadratic cost of the back substitution,

one could have expected at dimension 1,024 to see a factor of one thousand in the ratios between the QR and the back substitution. Or equivalently, the times for the QR would have been one thousand times longer than for the back substitution. Instead, the observed factor is closer to one hundred than one thousand, thanks to the well performing GPU accelerated QR.

As a final observation, times on the QR decomposition of a random 1,024-by-1,024 matrix in quad double precision, on the V100 appear in Table IV, Table VI, Table XI, with respective kernel times 3167.0, 3136.5, 3020.6, and respective wall clock times 3356.0, 3366.0, 3203.0, illustrating the fluctuations of the measured milliseconds.

V. 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 overhead cost 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.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, and M. Faverge. QR factorization on a multicore node enhanced with mulitple GPU accelerators. In *2011 IEEE International Parallel and Distributed Processing Symposium*, pages 932–943. IEEE, 2011.

[2] M. Anderson, G. Ballard, J. Demmel, and K. Kreutzer. Communication-avoiding QR decomposition for GPUs. In *2011 IEEE International Parallel and Distributed Processing Symposium*, pages 48–58. IEEE, 2011.

[3] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. Technical Report UT-CS-08-200, University of Tennessee, 2008.

[4] C. Bischof and C. F. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s1–s13, 1987.

[5] N. Bliss and J. Verschelde. The method of Gauss-Newton to compute power series solutions of polynomial homotopies. *Linear Algebra and its Applications*, 542:569–588, 2018.

[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[8] Y. He and C. H. Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *The Journal of Supercomputing*, 18:259–277, 2001.

[9] D. H. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, 1978.

[10] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001)*, pages 155–162. IEEE Computer Society, 2001.

[11] K. Isupov and V. Knyazkov. Multiple-precision matrix-vector multiplication on graphics processing units. *Program Systems: Theory and Applications*, 11(3):62–84, 2020.

[12] M. Joldes, J.-M. Muller, V. Popescu, and Tucker. W. CAMPARY: Cuda Multiple precision arithmetic library and applications. In *Mathematical Software – ICMS 2016, the 5th International Conference on Mathematical Software*, pages 232–240. Springer-Verlag, 2016.

[13] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In D. Kaeli and M. Leeser, editors, *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'09)*, pages 71–78. ACM, 2009.

[14] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann, 2010.

[15] J. Kurzak, R. Nath, P. Du, and J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple GPUs. In *Applied Parallel and Scientific Computing, 10th International Conference, PARA 2010*, volume 7134 of *Lecture Notes in Computer Science*, pages 248–257. Springer-Verlag, 2012.

[16] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)*, pages 19–26, 2010.

[17] D. Mukunoki and D. Takashashi. Implementation and evaluation of quadruple precision BLAS functions on GPUs. In *Applied Parallel and Scientific Computing, 10th International Conference, PARA 2010*, volume 7133 of *Lecture Notes in Computer Science*, pages 249–259. Springer-Verlag, 2012.

[18] D. Mukunoki and D. Takashashi. Implementation and evaluation of triple precision BLAS subroutines on GPUs. In *The 2012 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1372–1380. IEEE, 2012.

[19] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Springer-Verlag, second edition, 2018.

[20] 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.

[21] W. Nasri and Z. Mahjoub. Optimal parallelization of a recursive algorithm for triangular matrix inversion on MIMD computers. *Parallel Computing*, 27:1767–1782, 2001.

[22] S. D. Rao and D. J. Tylavsky. Theoretical convergence guarantees versus numerical convergence behavior of the holomorphically embedded power flow method. *Electrical Power and Energy Systems*, 95:166–176, 2018.

[23] S. Telen, M. Van Barel, and J. Verschelde. A robust numerical path tracking algorithm for polynomial homotopy continuation. *SIAM J. Sci. Comput.*, 42(6):A3610–A3637, 2020.

[24] S. Telen, M. Van Barel, and J. Verschelde. Robust numerical tracking of one path of a polynomial homotopy on parallel shared memory computers. In *Proceedings of the 22nd International Workshop on Computer Algebra in Scientific Computing (CASC 2020)*, volume 12291 of *Lecture Notes in Computer Science*, pages 563–582. Springer-Verlag, 2020.

[25] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[26] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *The 2010 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–8. IEEE, 2010.

[27] A. Trias. The holomorphic embedding load flow method. In *2012 IEEE Power and Energy Society General Meeting*, pages 1–8. IEEE, 2012.

[28] A. Trias and J. L. Martin. The holomorphic embedding loadflow method for DC power systems and nonlinear DC circuits. *IEEE Transactions on Circuits and Systems*, 63(2):322–333, 2016.

[29] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999.

[30] J. Verschelde. Parallel software to offset the cost of higher precision. *ACM SIGAda Ada Letters*, 40(2):59–64, 2020.

[31] J. Verschelde. Accelerated polynomial evaluation and differentiation at power series in multiple double precision. In *The 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 740–749. IEEE, 2021.

[32] J. Verschelde and G. Yoffe. Orthogonalization on a general purpose graphics processing unit with double double and quad double arithmetic. In *The 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1373–1380. IEEE, 2013.

[33] D. Viswanath and L. N. Trefethen. Condition numbers of random triangular matrices. *SIAM J. Matrix Anal. Appl.*, 19(2):564–581, 1998.

[34] 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. Article No. 31.

[35] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.