A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures

Mitch Horton*, Stanimire Tomov* and Jack Dongarra*†‡
*Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996
Email: {horton, tomov, dongarra}@eecs.utk.edu
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee
‡School of Mathematics & School of Computer Science, University of Manchester

***Abstract*—Three out of the top four supercomputers in the November 2010 TOP500 list of the world's most powerful supercomputers use NVIDIA GPUs to accelerate computations. Ninety-five systems from the list are using processors with six or more cores. Three-hundred-sixty-five systems use quad-core processor-based systems. Thirty-seven systems are using dual-core processors. The large-scale enabling of hybrid graphics processing unit (GPU)-based multicore platforms for computational science by developing fundamental numerical libraries (in particular, libraries in the area of dense linear algebra) for them has been underway for some time. We present a class of algorithms based largely on software infrastructures that have already been developed for homogeneous multicores and hybrid GPU-based computing. The algorithms extend what is currently available in the *Matrix Algebra for GPU and Multicore Architectures* (MAGMA) Library for performing Cholesky, QR, and LU factorizations using a single core or socket and a single GPU. The extensions occur in two areas. First, panels factored on the CPU using LAPACK are, instead, done in parallel using a highly optimized dynamic asynchronous scheduled algorithm on some number of CPU cores. Second, the remaining CPU cores are used to update the rightmost panels of the matrix in parallel.**

***Keywords*-GPU; multicore; QR; LU; Cholesky;**

I. Introduction

Until roughly 2004, microprocessor manufacturers were able to achieve higher performance by exploiting higher degrees of instruction level parallelism (ILP). Based on this approach, several generations of processors were built where clock frequencies were higher and higher and pipelines were deeper and deeper. As a result, applications could benefit from these innovations and achieve higher performance simply by relying on compilers that could efficiently exploit ILP. Due to a number of physical limitations (mostly power consumption and heat dissipation) this approach cannot be pushed any further. For this reason, chip designers have moved their focus from ILP to thread level parallelism (TLP) where higher performance can be achieved by replicating execution units (or cores) on the die while keeping the clock rates in a range where power consumption and heat dissipation do not represent a problem [1]–[3]. CPU designs have moved to multicores and are currently going through a renaissance due to the need for new approaches to manage the exponentially increasing (a) appetite for power of conventional system designs, and (b) gap between compute and communication speeds. Compute Unified Device Architecture (CUDA) [4] based multicore platforms stand out among a confluence of trends because of their low power consumption and, at the same time, high compute power and bandwidth [3]. Because of the prevalence of multicore and GPU architectures in the TOP500 list [5]; the existence of current conferences and workshops with emphasis on multicore and GPU technology [6]–[33]; the long list of GPU related success stories across academia, industry, and national research laboratories for specific applications and algorithms [34]–[46]; books related to general purpose GPU computing [47]–[49]; the emergence of compilers that understand GPU directives [50]–[53]; language in the current Exascale roadmap concerning heterogeneity in general and general purpose GPU programming in particular [54]; the fact that NVIDIA did $100 million in revenue from high performance computing last year, up from zero three years ago [55]; and relentless architectural advancements [56]–[63], it is clear that multicore processors and GPUs represent the future of high performance computing.

As multicore and GPU systems continue to gain ground in the high performance computing world, linear algebra algorithms have to be reformulated, or new algorithms have to be developed, in order to take advantage of the architectural features on these new architectures [64]. This work is a contribution to the development of these algorithms in the area of dense linear algebra, and will be included in the Matrix Algebra for GPU and Multicore Architectures (MAGMA) Library [38]. Designed to be similar to LAPACK [65] in functionality, data storage, and interface, the MAGMA library allows scientists to effortlessly port their LAPACK-relying software components and to take advantage of the new hybrid architectures.

The challenges in developing scalable high performance algorithms for multicore with GPU accelerators systems stem from their heterogeneity, massive parallelism, and the huge gap between the GPUs' compute power vs. the CPU-GPU communication speed. We show an approach that is largely based on software infrastructures that have already

been developed – namely, the QUeuing And Runtime for Kernels (QUARK) dynamic scheduler [66] and the MAGMA [38] library. The approach extends what is currently available in the MAGMA Library for performing Cholesky, QR, and LU factorizations using a single core or socket and a single GPU. The extensions occur in two areas. First, panels factored on the CPU using LAPACK are, instead, done in parallel using a highly optimized dynamic asynchronous QUARK scheduled algorithm on some number of CPU cores. Second, the remaining CPU cores are used to update the rightmost panels of the matrix in parallel. The approach aims to better utilize all available hardware.

The results of this work are communicated using the QR algorithm as a framework. The Cholesky and LU algorithms are similar in implementation. The paper is organized as follows. Section II provides an overview of the QR factorization. Section III illustrates how the QR factorization is performed by the MAGMA library using a single core or socket and a single GPU. Section IV describes the new approach, outlining how it differs from what is currently available in the MAGMA library. Section V briefly describes the QUARK dynamic scheduler. Section VI discusses autotuning. In particular, an explanation is given of how to — for a given matrix size, precision, architecture, and algorithm — choose the optimal number of cores for panel factorization, number of cores for panel updates, panel width, outer panel width, and inner panel width. Section VII describes algorithm optimization with respect to panel factorization. Section VIII presents results on two different architectures: a single NVIDIA GeForce GTX480 GPU with fifteen cores (streaming multiprocessors) @1.401 GHz connected to eight six-core Intel Xeon X5660 Westmere @2.8 GHz processors and a single NVIDIA Telsa M2070 GPU with fourteen cores (streaming mulitprocessors) @1.15 GHz connected to two six-core Intel Xeon X5660 Westmere @2.8 GHz processors. Finally, section IX discusses future work.

II. Block QR Factorization

This section contains a high level explanation of the block QR factorization implemented by LAPACK. The explanation will facilitate an understanding of the description of the new approach given in Section IV. A detailed discussion of the block QR factorization can be found here [67]–[71].

Stewart refers to the QR factorization as, "the great success story of modern computation," [72]. Trefethen and Bau say, "One algorithmic idea in numerical linear algebra is more important than all the others: QR factorization [70]." It is used for solving linear systems of equations [64], [68], solving the linear least squares problem [65], [73], computing eigenvalues and eigenvectors [72], [74], computing the SVD [72], [74], and computing an orthonormal basis for a set of vectors [68]. Stewart says, "the underlying theory of the method continues to suggest new algorithms." [72] Golub and Van Loan give QR algorithms based on Householder, block Householder, Givens, and fast Givens transformations; Gram-Schmidt orthogonalization, and modified Gram-Schmidt orthogonalization [68]. The LAPACK QR factorization is a block Householder transformation implementation.

The QR factorization is a transformation that factorizes an $m \times n$ matrix A into its factors Q and R where Q is a unitary matrix of size $m \times m$ and R is a triangular matrix of size $m \times n$. The LAPACK version of this algorithm achieves higher performance on architectures with memory hierarchies by accumulating a number of Householder transformations in what is called a *panel factorization* which are, then, applied all at once by means of high performance Level 3 BLAS operations.

The LAPACK routine that performs the QR factorization is called `xGEQRF` where x can be `S`, `D`, `C`, or `Z` depending on the precision. Consider a matrix A of size $m \times n$ represented as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where A_{11} is of size $b \times b$, A_{12} is of size $b \times (n-b)$, A_{21} is of size $(m-b) \times b$, and A_{22} is of size $(m-b) \times (n-b)$.

The LAPACK algorithm for QR factorization can be described as a sequence of steps where, at each step, the transformation in Equation (1) is performed.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \Longrightarrow \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix}, \begin{pmatrix} R_{11} & R_{12} \\ 0 & \tilde{A}_{22} \end{pmatrix} \quad (1)$$

The transformation in Equation (1) is obtained in two steps:

1) **Panel factorization**. At this step a QR factorization of the panel (A_{*1}) is performed as in Equation (2).

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \Longrightarrow \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix}, (T_{11}), (R_{11}) \quad (2)$$

   This operation produces b Householder reflectors (V_{*1}) and an upper triangular matrix R_{11} of size $b \times b$, which is a portion of the final R factor, by means of the `xDGEQR2` LAPACK routine. At this step, a triangular matrix T_{11} of size $b \times b$ is produced by the `xLARFT` LAPACK routine. Note that V_{11} is a unit lower triangular matrix of size $b \times b$. The arrays V_{*1} and R_{11} overwrite A_{*1}. Temporary workspace is needed to store T_{11}.

2) **Trailing submatrix update.** At this step, the transformation that was computed in the panel factorization is applied to the trailing submatrix as shown in Equation (3).

$$\begin{pmatrix} R_{12} \\ \tilde{A}_{22} \end{pmatrix} = \left(I - \begin{pmatrix} V_{11} \\ V_{21} \end{pmatrix} (T_{11})(V_{11}^T V_{21}^T) \right) \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} \quad (3)$$

   This operation, performed by the `xLARFB` LAPACK routine, produces a portion R_{12} of the final R factor of size $b \times (n-b)$ and the matrix $\tilde{A}_{22}$.

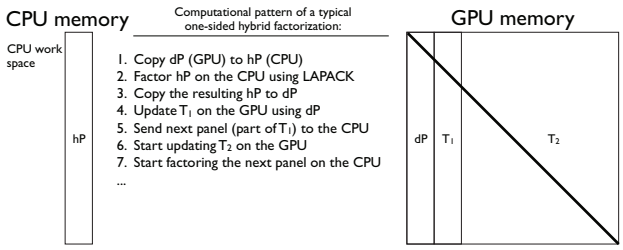

Figure 1. A typical hybrid pattern of computation and communication for the one-sided matrix factorizations in MAGMA 1.0

The QR factorization is continued by applying the transformation (1) to the submatrix $\tilde{A}_{22}$, and then, iteratively, until the end of the matrix A is reached.

Note that xGEQR2 and xLARFT are rich in Level 2 BLAS operations and cannot be efficiently parallelized on currently available shared memory machines. The speed of Level 2 BLAS computations is limited by the speed at which the memory bus can feed the cores. On current multicore architectures, because of the vast disproportion between the bus bandwidth and the speed of the cores, a single core can saturate the bus in double precision so there would be no advantage to using additional cores for a Level 2 BLAS operation. See [67] for more details. The LAPACK algorithm for QR factorization can use any flavor of parallel BLAS to exploit parallelism from the Level 3 BLAS xLARFB update on a multicore shared-memory architecture, but the panel update is considered a sequential operation.

III. MAGMA QR Factorization with a Single Core or Socket and a Single GPU

The MAGMA QR factorization with a single core or socket and single GPU (MAGMA 1.0) differs from the LAPACK QR factorization in 3 major respects. First, panels are factorized using xGEQRF as opposed to xGEQR2. Second, the xLARFB update is done on the GPU. Third, the xLARFB update is done in a lookahead fashion. Subtle differences include asynchronicity with respect to host and device execution and with respect to memory transfers where possible.

Figure (1) illustrates this typical pattern of hybridization. Several groups have observed that panel factorizations are often faster on the CPU than on the GPU [64]. Therefore, like the LAPACK QR factorization, the panel factorization is considered a sequential operation; there is not enough parallelism to keep the GPU busy. Unlike the LAPACK QR factorization, the panel factorization is done using the block QR factorization routine xGEQRF. If parallel BLAS is being used on the CPU, the panel factorizations are often faster when using several cores; depending on the architecture, a single socket or the entire host for the panel factorization will result in the optimal performance.

MAGMA 1.0 uses static scheduling and a right-looking version of the block QR factorization. The panel factorizations are scheduled on the CPU using calls to LAPACK, and the Level 3 BLAS updates on the trailing sub-matrices are scheduled on the GPU. The trailing matrix updates are split into 2 parts — one that updates just the next panel and a second one updating the rest, i.e., correspondingly submatrices T_1 and T_2 as given in Figure (1). The next panel update (i.e., T_1) is done first, sent to the CPU, and the panel factorization on the CPU is overlapped with the second part of the trailing matrix (i.e., T_2). This technique is known as *look-ahead* and has been used before [42], [75]–[77]. Its use enables the overlap of CPU and GPU work (and some communications).

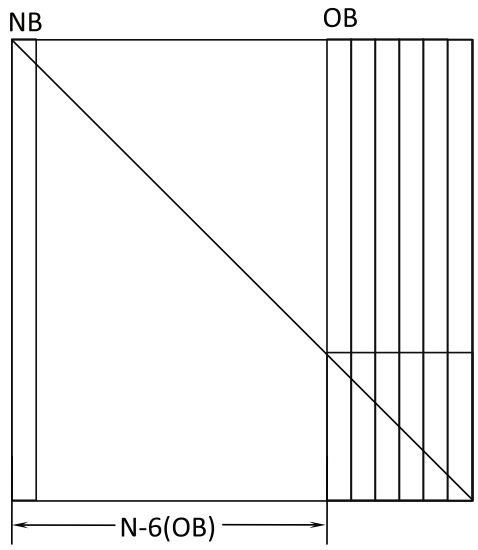

Figure 2. Computation splitting for new approach

IV. The New Approach

The new approach differs from MAGMA QR factorization with a single core or socket and a single GPU (MAGMA 1.0) in two areas. First, panels factored on the CPU using LAPACK are, instead, done in parallel using a highly optimized dynamic asynchronous scheduled algorithm on some number of CPU cores. Second, the remaining CPU cores are used to update the rightmost panels of the matrix in parallel. We think of the new approach as MAGMA QR factorization with all available cores and a single GPU. While it is true that if parallel BLAS is being used on the CPU, MAGMA 1.0 can use all available cores for the panel factorization, in practice, little, if any, additional speedup is attained past the number of cores on a single socket. It is a better use of the remaining cores to be tasked for other operations.

The computation is split as in Figure (2). Assuming an $N \times N$ matrix and a two socket, twelve core architecture, the first $N - 6(OB)$ columns are factorized as described in the previous section with two exceptions. First, the panels are factorized using an optimized dynamic asynchronous

scheduled algorithm using six cores. Second, whenever a panel factorization completes, one thread per core wakes from a busy wait from the remaining six cores, and the rightmost 6 panels are updated in parallel on the CPU. Note there is a final factorization to be done corresponding to the square in the lower right hand corner of Figure (2). The number of cores used for the panel factorization, and the number of cores used for the rightmost panel updates, depend on the architecture, matrix size, and precision. Tuning is discussed in detail in Section VI. Note that there are five parameters that are tunable for the new approach: the number of cores for panel factorization (Q), the number of cores for rightmost panel updates (P), the panel width for rightmost panel updates (OB), the panel width for the part of the matrix that does not include the rightmost panel updates (NB), and the inner panel width for panel factorizations (IB).

Figure (3) is a trace of the new approach on a single NVIDIA Tesla M2070 GPU with fourteen cores @1.15 GHz connected to two six-core Intel Xeon X5660 Westmere @2.8 GHz processors for single precision when the matrix size is 5920×5920 at the optimal parameter settings of $Q = 4$, $P = 8$, $NB = 128$, $OB = 172$, and $IB = 12$. We take *GPU core* to mean a single streaming multiprocessor. The y-axis can be thought of as core number; the bottom 14 green rows are GPU cores; the next 12 rows are CPU cores. The x-axis is time. White space is idle time. Black space indicates a busy wait. All other space represents useful work.

The top eight rows of the graph correspond to the CPU cores dedicated for the rightmost panel updates; again black corresponds to a busy wait. Cyan corresponds to the update. The middle four rows correspond to the CPU cores dedicated to the panel factorization. Additionally, the core corresponding to the ninth row from the top is responsible for computing a xLARFT; it can be seen in the trace as a blue rectangle. The core is also responsible for initiating a synchronous memory copy of the result of the panel factorization on the host to the GPU; that synchronous memory copy is seen as a yellow rectangle. Synchronous memory copies from the host to the GPU do not start until all kernels that have been launched on the GPU are finished; the call does not return until the memory copy is finished. The bottom fourteen rows correspond to the GPU cores dedicated to panel updates to the left of the eight rightmost panels. The white space in the middle four rows is when the cores dedicated for panel factorization are idle. The white space at the right of the top eight rows depicts the fact that the cores for rightmost panel updates are not used for the final factorization.

The flow of the new approach can be seen in the trace. Initially, four CPU cores are factorizing the first panel while eight CPU cores are in a busy wait and all GPU cores are idle. This is seen as the eight leftmost black rectangles in the top eight rows, the leftmost block of colored rectangles in rows nine through twelve, and the white space to the left of the bottom fourteen green rows. Next, eight CPU cores are woken from a busy wait and the rightmost eight panel updates begin; this can be seen as the leftmost eight cyan rectangles on the top eight rows. From Equation (3) it can be seen that $(V_{11}^T V_{21}^T) A_{*2}$ can be computed before the xLARFT is finished. Therefore the rightmost panel updates can be started early if they are split into two pieces, a piece that does not use the result from the xLARFT and a piece that does. While the computation of the first piece of the rightmost panel updates progresses, a xLARFT is done on a single CPU core; that can be seen as the leftmost large blue rectangle in row nine. Because the xLARFT finishes before the first update piece is done, both update steps are seen as a single cyan rectangle. Once the xLARFT is done, the result of the panel factorization is sent to the GPU; this can be seen as the sliver of yellow to the immediate right of the leftmost large blue rectangle in row nine. Next, the single panel to the right of the factorized panel is updated on the GPU; this can be seen as the leftmost thin column of green rectangles in the bottom eight rows. Finally, the remaining panels to the left of the eight rightmost panels, are updated on the GPU; this can be seen as the thick column of green rectangles to the immediate right of the leftmost thin column of green rectangles in the bottom eight rows. This process continues with the remaining panels to the left of the rightmost eight panels. The last grouping of rectangles in rows nine through twenty-six correspond to the final factorization depicted in the lower right hand square of Figure (2).

Note that the peak single precision CPU performance for the machine described above is 270 Gflop/s. The peak single precision GPU performance, however, is 1030 Gflop/s (The heights of the GPU cores in Figure (3) are scaled accordingly.). The new approach aims to start with a busy GPU since it is the more powerful of the two. After that, all that can be done to keep the CPU cores busy is done. This can be seen in Figure (3). The bottom fourteen rows show that the GPU has very little idle time. Our approach was to start with MAGMA 1.0 QR factorization code; this code has been optimized for the GPU over the course of several years. We enhanced the MAGMA 1.0 code to make better use of the CPU cores but the GPU code was unchanged. This approach differs from what others are currently doing to merge GPU and multicore code. Namely, to start with highly optimized multicore code and call GPU kernels where possible [78].

V. QUARK Dynamic Scheduler

Panels are factorized using a highly optimized version of the LAPACK block QR algorithm using the QUARK dynamic scheduler to schedule the subtasks on some number of CPU cores. We experimented with many different combinations of subtask granularities and scheduling policies. We use the version with the highest performance, and it will be described in detail.

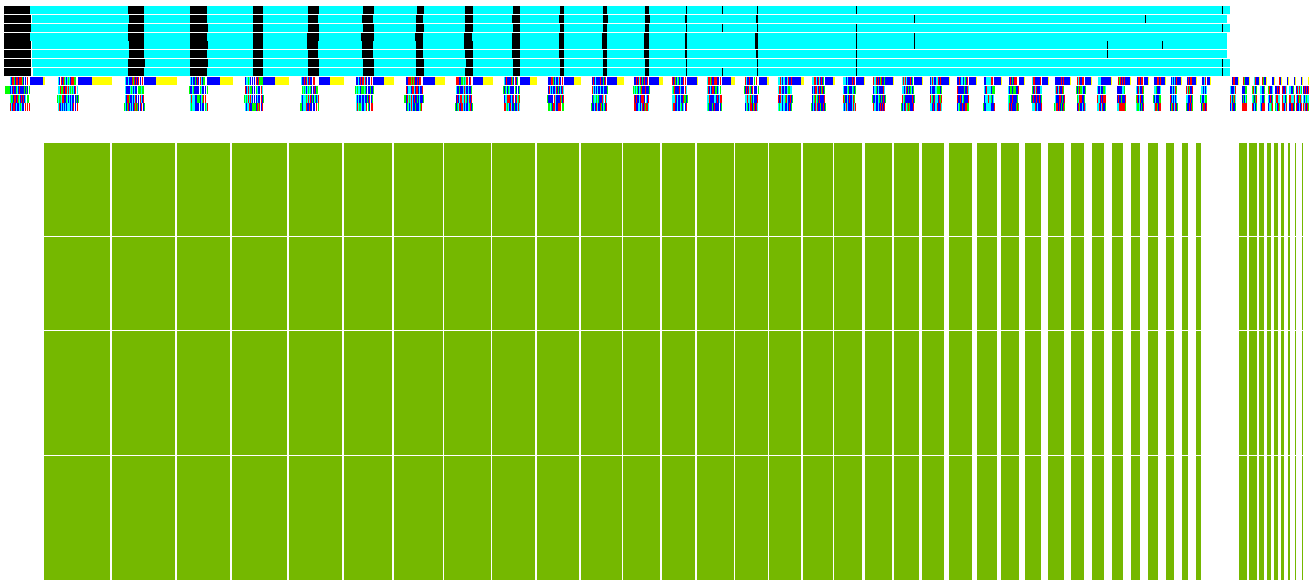

Figure 3. Trace of 5920×5920 Matrix

Individual subtasks, along with the dependencies between subtasks, are communicated to QUARK at subtask insertion time. QUARK is then free to schedule the subtasks among available cores in any order as long as the subtask dependencies are not violated. This concept of representing algorithms and their execution flows as Directed Acyclic Graphs (DAGs), where nodes represent the subtasks, and the edges represent the dependencies among them, is nothing new and is discussed in greater detail here [75], [76].

Optimal performance is observed when the subtasks are composed from the operations in Section II as follows. A single subtask is made from the `xGEQR2` and the `xLARFT` calls from the panel factorization step. The `xLARFB` call from the trailing submatrix step is split into three subtasks.

Optimal performance is observed when the subtasks are scheduled in a left looking fashion as opposed to the right looking approach described in Section II. Note that the DAG for QR block factorization with subtasks inserted in a left looking fashion will be identical to the DAG for QR block factorization with subtasks inserted in a right looking fashion. QUARK allows one to give subtasks priorities and that feature is exploited to achieve a left looking execution order.

VI. Autotuning

At first blush, it would appear that the tunable parameters are inextricably entwined such that adjusting one parameter can alter the effect of the other parameter settings. Therefore an autotuning approach suggests itself whereby all possible values of all five parameters are tried, noting what combination of parameters results in the best performance. However, even after careful pruning, autotuning a single matrix size for a single precision on a reasonably fast architecture will take more than one calendar year. This is clearly unacceptable.

It turns out that autotuning burden can be greatly mitigated by the combination of assuming orthogonality and noticing some rules of thumb. The first rule of thumb is that the optimal panel width, for that part of the matrix that does not include the rightmost panel updates, is the very same panel width already recorded for MAGMA 1.0. This panel width exists in a lookup table at runtime and is a function of matrix size, precision, and algorithm. The orthogonality assumption allows one to fix this optimal panel width regardless of the value of the other parameter settings. The second rule of thumb is that the optimal number of cores for the panel factorization is the number of cores on a single socket. The third rule of thumb is that the number of cores for the rightmost panel updates is the number of remaining cores for large enough matrices and a linear function of the matrix size otherwise. Experimenting with the number of cores for panel factorizations and rightmost panel updates on different architectures results in slight modifications to the second and third rules of thumb, depending on the architecture, but the end result is that the information exists in a lookup table and is available at runtime. The fourth rule of thumb is that IB is always 12. This number was observed over the course of much hand tuning.

Thus the only parameter that needs to be tuned is OB, the panel width for the rightmost panel updates. An autotuner was written that tests a number of values for OB for every matrix size at every precision and takes two hours to complete on a reasonably fast architecture. All tuning results are

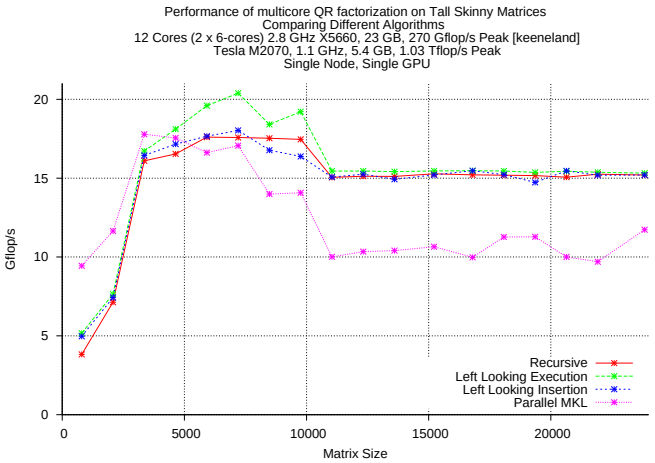

Figure 4. Panel Factorization Technique Comparison

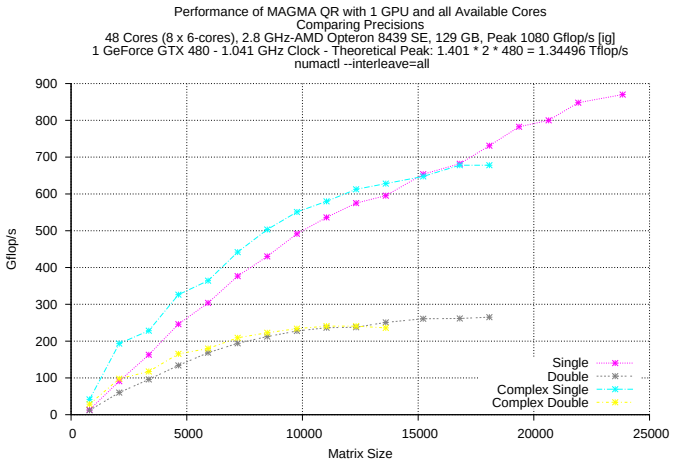

Figure 5. New Approach Performance on 48 Core Machine, Comparing Precisions

stored in a lookup table and are available at runtime. Hence the user is not required to provide parameter values in order to get a good first approximation at optimal performance. The user *can* provide parameter values at runtime but it is not a requirement.

VII. Panel Factorization Optimization

Four different panel factorization flavors were compared: left looking subtask insertion, left looking subtask execution, recursive, and parallel MKL. The difference between left looking subtask insertion and left looking subtask execution lies only in that for left looking subtask execution the subtasks are prioritized in order to enforce left looking execution. The recursive technique uses one level of recursion for the right half of the panel in place of the call to `xGEQR2`.

Figure (4) shows the results for two six-core Intel Xeon X5660 Westmere @2.8 GHz processors. Note that the number of cores used for the panel factorization is the optimal number of cores when the panel factorization is used as part of the new approach for QR factorization: 6 cores on the left side of the graph, 2 cores on the right side of the graph, and 3 or 4 cores in the middle.

VIII. Results

Figure (5) shows how the new approach performs at four precisions using a single NVIDIA GeForce GTX480 GPU with fifteen cores @1.401 GHz connected to eight six-core Intel Xeon X5660 Westmere @2.8 GHz processors. Figure (6) shows how the new approach performs at four precisions using the machine described in Section (IV). Figure (7) shows how the new approach performs compared to MAGMA 1.0 and MKL for the same machine.

IX. Summary and Future Work

This paper shows how to redesign the QR factorization to enhance it's performance in the context of a single GPU and

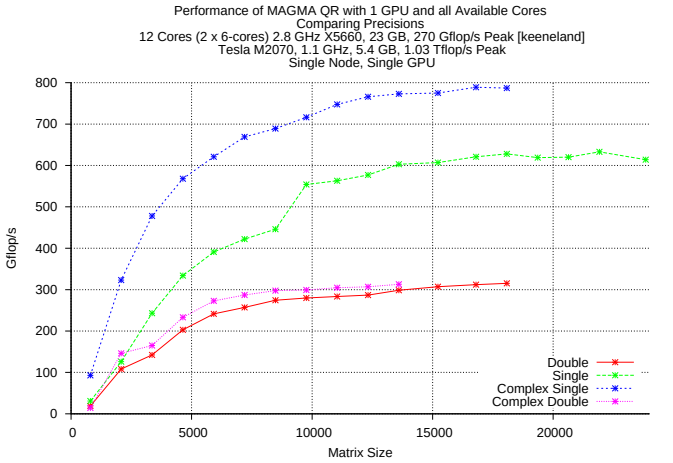

Figure 6. New Approach Performance on 12 Core Machine, Comparing Precisions

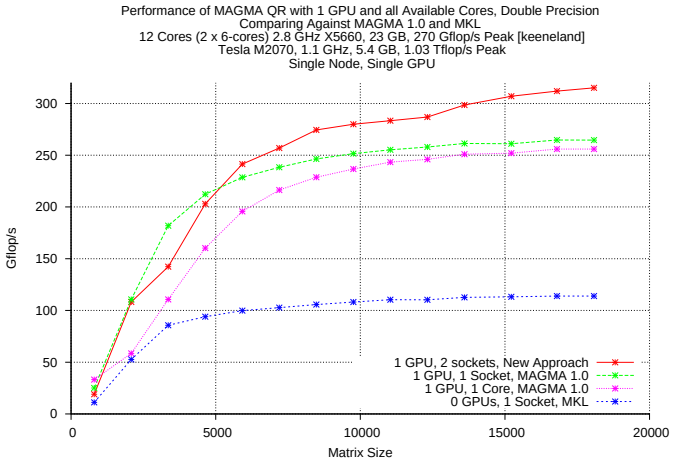

Figure 7. New Approach Performance on 12 Core Machine, Compared to MAGMA 1.0 and MKL

many CPU cores. Using a single NVIDIA GPU (fourteen cores @1.15 GHz) connected to two six-core Intel Xeon X5660 Westmere @2.8 GHz processors, the new approach achieves 315 GFlop/s in double precision. This is an increase of 19% over MAGMA 1.0. The increase is due to a better use of all available CPU cores and reuses concepts developed for the MAGMA library and the QUARK dynamic scheduler. Although this paper focused on the QR factorization, the framework is in place to extend the algorithm to LU and Cholesky factorizations. The new approach will eventuallly be included in a future release of MAGMA and can be used for a full high-performance linear solver.

From Figure (3) it can be seen that only one core is used to compute the xLARFT and it is often the case that all the other CPU cores are waiting on that core to finish. Speedup could be obtained by using all the cores dedicated to panel factorizations to compute the xLARFT in parallel.

Another thing that can be seen is that replacing the synchronous memory copy corresponding to the yellow rectangles in the trace with an asynchronous memory copy would free up one CPU core for useful work.

REFERENCES

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[2] M. Baboulin, J. Dongarra, and S. Tomov, "Some issues in dense linear algebra for mulitcore and special purpose architectures," University of Tennessee, Tech. Rep., May 6, 2008, technical Report UT-CS-08-200 (also LAPACK Working Note 200).

[3] H. Ltaif, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A scalable high performant Cholesky factorization for multicore with GPU accelerators," University of Tennessee, Tech. Rep., Nov. 25, 2009, technical Report UT-CS-09-646 (also LAPACK Working Note 223).

[4] Apr. 2011, NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[5] Apr. 2011, http://www.top500.org/lists/2010/11/highlights.

[6] Jul. 2011, the 2011 International Conference on High Performance Computing & Simulation. http://hpcs11.cisedu.info/.

[7] Apr. 2011, Many-Core and Reconfigurable Supercomputing Conference. http://www.mrsc2011.eu/.

[8] May 2011, 25th International Conference on Supercomputing. http://ics11.cs.arizona.edu/.

[9] Apr. 2012, GPU Technology Conference 2012. http://www.nvidia.com/object/gpu_technology_conference.html.

[10] Jun. 2011, the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing. http://www.hpdc.org/2011/.

[11] Aug. 2011, Euro-Par 2011. http://europar2011.bordeaux.inria.fr/.

[12] May 2011, 25th IEEE International Parallel & Distributed Processing Symposium. http://www.ipdps.org/.

[13] Dec. 2011, the 9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2011). http://aiccsa2011.hpcl.gwu.edu.

[14] Jul. 2011, 2011 Symposium on Application Accelerators in High Performance Computing (SAAHPC'11). http://saahpc.ncsa.illinois.edu.

[15] Sep. 2011, 9th International Conference on Parallel Processing and Applied Mathematics. http://www.ppam.pl/.

[16] Apr. 2011, 9th annual workshop on Charm++ and its applications. http://charm.cs.illinois.edu/charmWorkshop/program.php.

[17] May 2011, parallel CFD 2011. http://parcfd2011.bsc.es/.

[18] May 2011, NAFEMS World Congress 2011. http://www.nafems.org/congress/.

[19] May 2011, LS-DYNA 8th European Users Conference. http://www.lsdynaeuc.alyotech.fr/web/guest;jsessionid=e43b16332e9d80fee2f6d17a57ce.

[20] Oct. 2011, GPU Technology Summit. http://sagivtech.com/gpu-technology-summit.htm.

[21] May 2011, International Symposium: Computer Simulations on GPU. www.cond-mat.physik.uni-mainz.de/~weigel/GPU2011/home/.

[22] Jun. 2011, International Supercomputing Conference. http://www.supercomp.de/isc11_ap/php/HTML/event_detail.php?evid=157&request_day_id=11.

[23] Jul. 2011, The 2011 International Conference of Parallel and Distributed Computing. http://www.iaeng.org/WCE2011/ICPDC2011.html.

[24] Sep. 2011, ENUMATH Conference 2011. http://www2.le.ac.uk/departments/mathematics/research/enumath2011.

[25] Apr. 2012, InPar 2012 - Innovative Parallel Computing. http://innovativeparallel.org/.

[26] Oct. 2011, Accelerated HPC Symposium 2011. http://www.lanl.gov/conferences/AHPCS/.

[27] Feb. 2012, 20th Euromicro International Conference on Parallel, Distrubuted and Network-based Processing. http://conf.laas.fr/GPU/.

[28] May 2011, GPGPU Computing for Scientific Applications. http://www.famaf.unc.edu.ar/grupos/GPGPU/EscuelaGPGPU2011/.

[29] Jun. 2011, 2011 IEEE Intelligent Vehicles Symposium. .

[30] Jul. 2011, Genetic and Evolutionary Computation Conference 2011. http://www.sigevo.org/gecco-2011/index.html.

[31] Jul. 2011, The 2011 World Congress in Computer Science Computer Engineering and Applied Computing. http://www.world-academy-of-science.org/worldcomp11/ws.

[32] Sep. 2011, SPIE Conference on High-Performance Computing in Remote Sensing. http://spie.org/app/program/index.cfm?fuseaction=conferencedetail&export_id=x12522&ID=x6267&redir=x6267.xml&conference_id=952872&event_id=948125.

[33] Feb. 2012, SIAM Conference on Parallel Processing for Scientific Computing. http://www.siam.org/meetings/pp12/.

[34] Apr. 2011, NVIDIA CUDA ZONE. http://www.nvidia.com/object/cuda_home.html.

[35] Apr. 2011, General-purpose computation using graphics hardware. http://www.gpgpu.org/.

[36] Apr. 2011, CUDA Toolkit 4.0 CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUBLAS_Library.pdf.

[37] V. Volkov and J. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC 08: Proceedings of the 2008 ACM/IEEE*, title=Conference on Supercomputing. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[38] S. Tomov, R. Nath, P. Du, and J. Dongarra, "MAGMA version 0.2 users's guide," Tech. Rep., Nov. 2009, http://icl.eecs.utk.edu/magma/.

[39] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[40] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[41] S. Tomov, M. Baboulin, J. Dongarra, S. Moore, V. Natoli, G. Peterson, and D. Richie, "Special-purpose hardware and algorithms for accelerating dense linear algebra," Parallel Processing for Scientific Computing, Tech. Rep., Mar. 12, 2008, http://www.cs.utk.edu/~tomov/PP08_Tomov.pdf.

[42] V. Volkov and J. W. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of gpus," EECS Department, University of California, Berkeley,, Tech. Rep., may 2008, tech. Report UCB/EECS-2008-49.

[43] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie, "Exploring new architectures in accelerating CFD for Air Force applicatons," in *Proceedings of HPCMP Users Group Conference 2008*, Jul. 14, 2008, http://www.cs.utk.edu/ tomov/ugc2008 final.pdf.

[44] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," Oct. 2008, LAPACK Working Note 210.

[45] M. Fatica, "Acclerating Linpack with CUDA on heterogenous clusters," *GPGPU-2*, pp. 46–51, 2009, washington, DC.

[46] S. Tomov and J. Dongarra, "Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing," May 2009, technical Report 219, LAPACK Working Note 219.

[47] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: a Hands-on Approach*. Morgan Kaufmann, 2010.

[48] J. Sanders and E. Kandrot, *CUDA by Example: an Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[49] W.-m. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[50] Apr. 2011, PGI CUDA Fortran Compiler. http://www.pgroup.com/resources/cudafortran.htm.

[51] Apr. 2011, HMPP Workbench. http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36.

[52] Apr. 2011, R-Stream - high level compiler. https://www.reservoir.com/rstream.

[53] Apr. 2011, OpenMP. http://openmp.org/wp/openmp-compilers/.

[54] Apr. 2011, Exascale Roadmap. http://www.exascale.org/mediawiki/images/2/20/IESP-roadmap.pdf.

[55] May 2011, Supercomputing and High Performance See Growing GPU Adoption. http://blogs.forbes.com/tomgroenfeldt/2011/05/06/supercomputing-and-high-performance-see-growing-gpu-adoption/.

[56] D. Patterson, "NVIDIA's next-generation CUDA compute and graphics architecture, code-named Fermi, adds muscle for parallel processing," Tech. Rep., Apr. 12, 2011, http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf.

[57] T. Halfhill, "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges," Tech. Rep., Apr. 12, 2011, http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf.

[58] P. Glaskowsky, "NVIDIA's Fermi: the first complete GPU computing architecture," Tech. Rep., Apr. 12, 2011, http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf.

[59] N. Brookwood, "NVIDIA solves the GPU computing puzzle," Tech. Rep., Apr. 12, 2011, http://www.nvidia.com/content/PDF/fermi_white_papers/N.Brookwood_NVIDIA_Solves_the_GPU_Computing_Puzzle1.pdf.

[60] Apr. 2011, NVIDIA Announces Project Denver to Build Custom CPU Cores Based on ARM Architecture, Targeting Personal Computers to Supercomputers. http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&releasejsp=release_157&xhtml=true&prid=705184.

[61] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[62] S. Dighe, S. Vangal, N. Borkar, and S. Borkar, "Lessons learned from the 80-core tera-scale research processor," *Intel Technology Journal*, vol. 13, no. 4, pp. 118–129, 2009.

[63] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, dec 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[64] S. Tomov and J. Dongarra, "Dense linear algebra for hybrid gpu-based systems," in *Scientific computing with multicore and acclerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, 2011, pp. 37–55.

[65] Z. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK users' guide." SIAM, Philadelphia, PA, Tech. Rep., 1992, http://www.netlib.org/lapack/lug/.

[66] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK users' guide: QUeueing And Runtime for Kernels," Tech. Rep., University of Tennessee Innovative Computing Laboratory Technical Report, ICL-UT-11-02.

[67] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "Parallel tiled QR factorization for mulitcore architectures," Tech. Rep., Jul. 2007, LAPACK Working Note 190, UT-CS-07-598.

[68] G. Golub and C. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

[69] R. Schreiber and C. van Loan, "A storage-efficient WY representation for products of Householder transformations," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 53–57, 1989.

[70] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997.

[71] C. Bischof and C. Van Loan, "The WY representation for products of Householder matrices," *SIAM J. of Sci. Stat. Computing*, vol. 8, pp. s2–s13, 1987.

[72] G. W. Stewart, *Matrix Algorithms Volume II: Eigensystems*. SIAM, 2001.

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

[74] G. W. Stewart, *Introduction to Matrix Computations*. Academic Press, 1973.

[75] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 820, pp. 1–18, 2003.

[76] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *PARA 2006*, Umea Sweden, 2006.

[77] J. Kurzak and J. Dongarra, "Implementing linear algebra routines on multicore processors with pipelining and a look ahead," Tech. Rep., Sep. 2006, LAPACK Working Note 178. Also available as UT-CS-06-581.

[78] J. Kurzak, R. Nath, P. Du, and J. Dongarra, "An implementation of the tile QR factorization for a GPU and multiple CPUs," Sep. 2010, LAPACK Working Note 229.