2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Virtual Systolic Array for QR Decomposition

Jakub Kurzak, Piotr Luszczek,
Mark Gates, Ichitaro Yamazaki
University of Tennessee
Knoxville, TN 37996, USA
{*kurzak, luszczek, mgates3, iyamazak*}*@eecs.utk.edu*

Jack Dongarra
University of Tennessee, Knoxville, TN 37996, USA
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
University of Manchester, Manchester, M13 9PL, UK
dongarra@eecs.utk.edu

***Abstract*—Systolic arrays offer a very attractive, data-centric, execution model as an alternative to the von Neumann architecture. Hardware implementations of systolic arrays turned out not to be viable solutions in the past. This article shows how the systolic design principles can be applied to a software solution to deliver an algorithm with unprecedented strong scaling capabilities. Systolic array for the QR decomposition is developed and a virtualization layer is used for mapping of the algorithm to a large distributed memory system. Strong scaling properties are discovered, superior to existing solutions.**

***Keywords*-systolic array; QR decomposition; multi-core; message passing; dataflow programming; roofline model;**

I. Introduction

Systolic architectures targeting hardware implementations were haunted by an array of problems. They were constructed for a specific problem size, which in linear algebra meant the size of a dense matrix or the bandwidth of a band matrix. This put the feasibility of manufacturing them in silicon in question. They operated at the granularity of a single floating-point number, i.e., a single matrix element, which prevented them from achieving high efficiency. Finally, in many cases, they could offer high throughput, but suffered from high latency, e.g., they could maintain high utilization for a series of dense matrix factorizations, but were affected by high load imbalance for a single one. A software implementation of a systolic algorithm allows for solving virtually all of these problems.

In dense linear algebra, the level of granularity can easily be brought up by replacing operations on individual matrix elements with operations on matrix tiles, i.e., square submatrices of relatively small size compared to the size of the matrix. This mitigates the communication overhead by leveraging the *surface to volume* effect, i.e., the fact that a tile operation involves $O(n^3)$ floating point operations on $O(n^2)$ data.

Two simple mechanisms allow for resolving the problem of load imbalance. A virtualization layer can be used for flexible mapping of multiple systolic processing units to each physical hardware core. A *data bypass* mechanism can be used to speed up propagation of read-only data along one of the systolic array dimensions.

The following sections start with the motivation for seeking an algorithm with strong scaling properties, and move on to general background on systolic arrays and the QR decomposition. Then the systolic array for the QR decomposition is presented and its software implementation is described. Performance results are provided, along with comparisons against state-of-the-art software, which are followed with the discussion.

II. Motivation for Strong Scaling

Dense linear algebra software has traditionally been focused on *asymptotic scaling* or *weak scaling*. Asymptotic scaling describes how the solution time varies with the problem size for a fixed number of cores. Weak scaling describes how the solution time varies with the number of cores for a fixed problem size per core. Delivering good asymptotic scaling or weak scaling has been the main objective of legacy software packages, such as LAPACK [1] and ScaLAPACK [2]. Meeting this objective relies on the capability of growing the *total* problem size.

Current developments in microprocessor technology are marked by the continuation of Moore's law [3] and the demise of Dennard's scaling laws [4]. While the numbers of transistors are still increasing, the clock rates have been stagnant for almost a decade now. The result is an explosive growth in the level of on-chip parallelism, manifested both in the number of cores, i.e., *Thread Level Parallelism* (TLP), and the number of floating point units per core, i.e., *Instruction Level Parallelism* (ILP).

Recent reports from the *Defense Advanced Research Projects Agency* (DARPA) [5], [6] and the *International Exascale Software Project* (IESP) [7] paint the landscape of future *High Performance Computing* (HPC) systems. Floating point capabilities are expected to rise rapidly with increasing numbers of cores and floating point units. At the same time, memory capacity is expected to grow at a much slower pace, eventually falling behind arithmetic performance by an order of magnitude. Exascale systems are projected to have an order of magnitude lower ratio of memory capacity to floating point performance.

It is clear then that emphasis has to be shifted from asymptotic scaling and weak scaling to *strong scaling*, which

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.119

describes how the solution time varies with the number of cores for a fixed *total* problem size. Simply put, algorithms with no strong scaling properties are bound to run out of memory before reaching good performance levels on future large-scale systems. The solution presented in this article shows unprecedented strong scaling properties, i.e., the capability of utilizing very high numbers of cores to solve very small problems by today's dense linear algebra standards.

III. Background

A. Systolic Arrays

Systolic arrays are descendants of array-like architectures such as iterative arrays, cellular automata and processor arrays. A systolic array is a network of processors that rhythmically compute and pass data through the system. The seminal paper by Kung and Leiserson [8] defines systolic arrays as devices with "simple and regular geometries and data paths" with "pipelining as a general method for using these structures."

The systolic array paradigm is the counterpart of the von Neumann paradigm. While the von Neumann architecture is instruction-stream-driven by an instruction counter, the systolic array architecture is data-stream-driven by data counters. A systolic array is composed of matrix-like rows of processing units, each one connected to a small number of nearest neighbors in a mesh-like topology. The operation is transport-triggered, i.e., triggered by the arrival of a data object.

The term "systolic array" was coined in the paper by Kung and Leiserson [8], where they introduced basic systolic topologies and applied them to problems in dense linear algebra. Applications in signal processing were pointed out: convolution, *Finite Impulse Response* (FIR) filter, and the *Discrete Fourier Transform* (DFT). General discussion and motivation for systolic arrays is given in another publication by Kung [9] and also Fortes and Wah [10]. Systematic treatment of the topic is provided in books by Robert [11], [12] and Evans [13].

B. Tile QR Decomposition

The essence of the tile QR algorithm is the idea of applying Householder reflectors incrementally. Unlike the block algorithm of LAPACK and ScaLAPACK, which eliminate a full panel of the matrix at a time, the tile algorithm descends down the panel tile by tile, eliminating only one tile at a time. Such operation is much more cache friendly and much more suitable for pipelining, since elimination of each panel tile can be immediately followed by application of updates to the right.

The algorithm is derived from methods of modifying the factors of a matrix, following an update of a small rank. Gill et al. describe algorithms for modifying Cholesky and QR factors following a rank-one update, in their article from 1974 [14]. In 1994, Berry et al. combined Householder reflectors and Givens rotations to produce an algorithm which can be considered a precursor of the tile algorithms [15]. It combines Householder reflectors and Givens rotations to reduce the matrix to the block-Hessenberg form.

This approach was rediscovered a few years ago by Buttari et al. [16], [17] and was subsequently used to produce high performance codes for multicore processors [16], [17], the Cell processor [18], and systems with GPU accelerators [19]. The idea of incrementally applying Householder reflectors goes beyond the QR decomposition, though. It has also been successfully applied to block-bidiagonal reduction and block-tridiagonal reduction, leading to very fast singular value solvers and symmetric eigenvalue solvers.

for $k = 0$ **to** $N - 1$ **do**
  dgeqrt($_{inout}A_{kk}$)
  for $n = k + 1$ **to** N **do**
    dormqr($_{in}A_{kk}$, $_{inout}A_{kn}$)
  end for
  for $m = k + 1$ **to** N **do**
    dtsqrt($_{inout}A_{kk}$, $_{inout}A_{mk}$)
    for $n = k + 1$ **to** N **do**
      dtsmqr($_{in}A_{mk}$, $_{inout}A_{kn}$, $_{inout}A_{mn}$)
    end for
  end for
end for

Figure 1. Tile QR serial definition.

Figure 1 shows the serial definition of the tile QR algorithm. Parameters are matrix tiles, prefix indicates the direction, postfix indicates the position in the matrix. Figure 2 shows the tiles affected by each kernel in a 3×3 factorization. The kernels perform the following operations:

dgeqrt
: Performs QR factorization of a diagonal tile. Places the R factor in the upper triangle and Householder reflectors in the lower triangle.

dormqr
: Applies Householder reflectors computed by the *dgeqrt* kernel to one tile of the trailing submatrix.

dtsqrt
: Performs incremental QR factorization of a subdiagonal tile. Updates the R factor in the diagonal tile and places Householder reflector coefficients in the subdiagonal tile.

dtsmqr
: Applies Householder reflectors computed by the *dtsqrt* kernel to two tiles of the trailing submatrix.

Routines *dtsqrt* and *dtsmqr* might be considered precursors of *dtpmqrt* that was recently introduced in LAPACK.

Same as the canonical QR, the tile QR is numerically

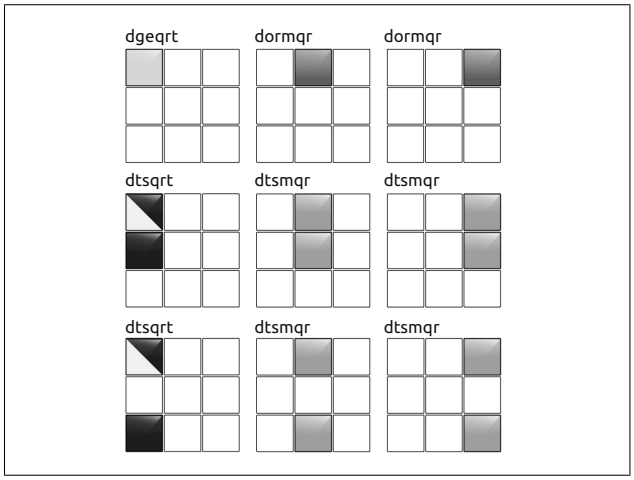

Figure 2. Kernel invocations in a 3×3 tile QR.

stable, because of the use of orthogonal transformations. The elements of the R factor are the same in absolute values, but different Householder reflectors are produced and a different procedure is required for their application to the right-hand side when solving a system of equations.

IV. Solution

The solution is built in three steps. First, a systolic array for the tile QR algorithm is developed and an extension of the systolic processing model is presented, referred to as *data bypass*. Then virtualization is applied, i.e., mapping of systolic array elements to physical cores. Finally, an abstraction layer is introduced for handling communication among cores through either intra-node (shared memory) or inter-node (message passing) mechanisms.

A. Systolic QR Algorithm

Canonical dense matrix factorizations, such as Gaussian elimination, Cholesky decomposition, or QR decomposition can be described with a set of nested loops with three levels of nesting, which is synonymous with $O(n^3)$ computational complexity. At the same time, systolic arrays traditionally target planar layouts, suitable for integrated circuits. Therefore, systolic arrays are built by applying a projection to the execution space of the algorithm. Usually, the projection is done along one of the dimensions, resulting in a square or triangular shape of the array. Projection can also be applied at an angle, producing a hexagonal array. The former approach is popular for dense matrices, the latter is popular for band matrices.

Here, projection along the m dimension (Figure 1) is applied, producing a traditionally shaped triangular systolic array (Figure 3). Each row of the array is responsible for one step of the factorization. Each diagonal processing unit is responsible for factoring one panel, by applying one *dgeqrt* operation and a sequence of *dtsqrt* operations. At each step a diagonal unit consumes one tile of the matrix and produces one tile of Householder coefficients, while retaining the R factor and updating it accordingly. The transformations are forwarded to the right, to the off-diagonal units. Each off-diagonal unit applies the transformations by invoking one *dormqr* operation and a sequence of *dtsmqr* operations. Householder reflections are received from the left and forwarded to the right. Matrix tiles are received from above and updated tiles are forwarded down. The matrix enters the array from the top, and the Householder reflections coefficients exit at the bottom. At the time of completion, the systolic units contain the final R factor.

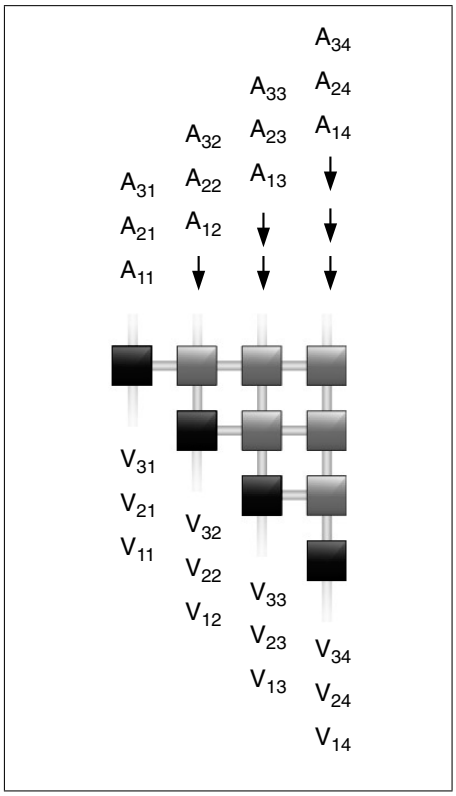

Figure 3. Systolic array for tile QR.

B. Data Bypass Extension

Traditionally, the systolic unit follows the cycle: *receive data, perform computation, send data*. As a result, the matrix enters the array at an angle, and propagates one step at a time in both the vertical and horizontal direction. This leads to high load imbalance, as many units are idle before the data reaches them. The slow propagation can easily be improved by introducing a simple *data bypass* mechanism, which allows for overlapping of communication and computation.

The data traveling in the vertical dimension is modified at every step, and processing has to follow the *receive, compute, send* cycle. At the same time, the data traveling in the horizontal dimension is only read at each step, and therefore

the original *receive, use, send* cycle can be replaced with a *receive, forward, compute* cycle. That is, upon reception from the left, the data is immediately forwarded to the right (Figure 4). This allows for overlapping communication and computation in the horizontal dimension and accelerating the feeding of the matrix into the array. It is natural to introduce such an extension if the target system is a distributed memory machine with dedicated communication hardware.

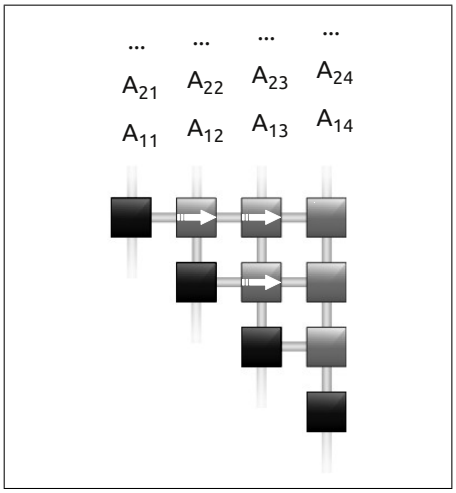

Figure 4. Data bypass in QR systolic array.

C. Virtualization Layer

The size of the systolic array is problem specific, i.e., the number of systolic units in the top row of the array equals the number of tiles in a row of the matrix. On top of that, the array has a triangular shape. Therefore, a one-to-one mapping of systolic units to physical cores would result in a rather odd number of cores being used. Plus, with the number of cores equal to the number of units, the load imbalance would be very high, due to the time required for the data to propagate to the units at the bottom of the array. In the general case, a much more flexible solution is desired.

To address this problem, a virtualization layer is introduced that allows for mapping of multiple systolic units to each physical core in the system. One simple assignment is a zigzag pattern, where consecutive cores are assigned along the rows of the array and "spill over" from row to row (Figure 5). If N is the dimension of the array, r is the row number, c is the column number, and P is the number of cores, then the unit in row r and column c belongs to the core $(Nr + c - r(r + 1)/2) \bmod P$.

Because the data shifts through the array from top to bottom and from left to right, such a mapping allows for quick propagation of work to cores. Many assignments are possible, e.g., block-cyclic or assignments relying on *space filling curves*, such as Morton and Hilbert curves. Different assignments will expose different tradeoffs between the load balance, the locality and the volume of communication. These alternative mappings are not investigated here.

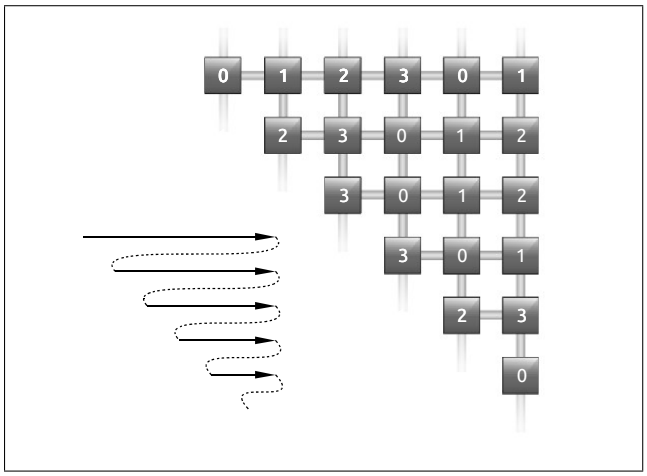

Figure 5. Mapping of systolic units to cores.

D. Communication Layer

In principle, communication proceeds through channels between pairs of systolic units. However, multiple units can be mapped to each core, and multiple cores reside in each node of the distributed memory system. Therefore, two levels of indirection are involved. Each core handles communication requests for its systolic units, and each node handles communication requests for all its cores.

The implementation relies on launching one software thread per hardware core in the system, and dedicating one thread (one core) to serve as a *communication proxy* to handle all inter-node (message passing) communication. The actual system of choice, the Kraken supercomputer at the National Institute for Computational Sciences, has 12 cores per node (two six-core sockets).

From the standpoint of a core, the communication model is flat (core to core). However, all communication requests are "hijacked" by the communication proxy. (A rudimentary protocol connects the *worker* cores to the *proxy* core). The proxy core uses shared memory mechanisms to handle local, intra-node communication, and message passing to handle non-local, inter-node communication. Intra-node communication is handled through memory aliasing and involves no copies.

E. Data Distribution

Unlike in traditional systolic arrays, here data has an initial and final location within the system. It is important to mention that the matrix is laid out in the memory of each node by tiles, where each tile is stored in a continuous memory region, which is beneficial both for kernel performance and the performance of communication.

At the beginning, the tiles of the matrix are assigned to the entry points of the systolic array, i.e., the cores where the top row of the array is placed. This means that initially the matrix follows a 1D block-cyclic distribution. If the number of cores is smaller than the number of units in the top row, then the matrix is spread across all cores. Otherwise it is spread across a subset of cores. Core placement is synonymous with node placement, so there is only one copy of a given tile in a node.

When processing is finished, the R factor is distributed across all the cores, and the Householder reflectors' coefficients are distributed across the cores where the diagonal of the array resides. If desired, the final distribution can easily be reshuffled to the initial distribution. The cost would be negligible, considering the overall volume of communication in the course of the factorization. The reshuffling is not done here.

V. Experimental Setup

A. Target Hardware

All runs were done on the Kraken supercomputer at the National Institute for Computational Sciences. The Kraken machine is a Cray system operated by the University of Tennessee and located in Oak Ridge, Tennessee. The entire system consists of 9408 compute nodes. The experiments presented here used up to 1984 nodes, which is about one fifth of the machine. Each node contains two 2.6 GHz six-core AMD Opteron (Istanbul) processors, 16 GB of memory and the Cray SeaStar2+ connection.

B. Software Stack

The code was compiled with the default compiler on the Kraken system, which is *The Portland Group, Inc.* (PGI) compiler. The *core_blas* kernels from the PLASMA package [20] were used as the serial building blocks. The code was linked against the *Basic Linear Algebra Subprograms* (BLAS) provided by Cray (LibSci) and Cray's *Message Passing Interface* (MPI).

VI. Performance Results

A. Systolic QR

Three fixed problem sizes of approximately $10K$, $20K$, and $40K$ (exactly $N = 10,368$, $N = 20,736$, and $N = 41,472$) were tested for a varying number of cores, as shown in Figures 6, 7, and 8. For each test, two tile sizes of $nb = 192$ and $nb = 256$ were tested. The smaller $nb = 192$ tile size achieved higher peak performance, and is shown here. For the largest problem size, $N = 40K$, the larger $nb = 256$ block size had better performance for a smaller number of cores, less than 7500 cores, but for a larger number of cores, achieved a peak performance of 18.2 Gflop/s, compared to 22.9 Gflop/s peak for $nb = 192$. In all three instances, the performance initially increases as more cores are used, and then eventually plateaus as the maximum amount of parallelism, determined by the critical path, is exploited. For all three problem sizes, systolic QR achieved a significantly higher peak performance than the Cray LibSci QR and HPL tests.

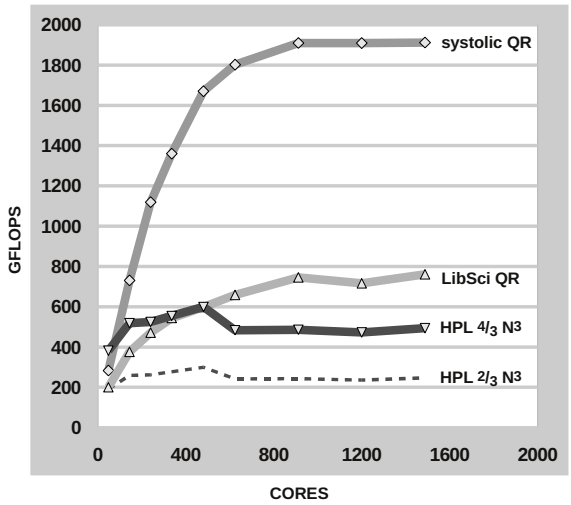

Figure 6. Systolic QR performance for a problem of size $10K \times 10K$.

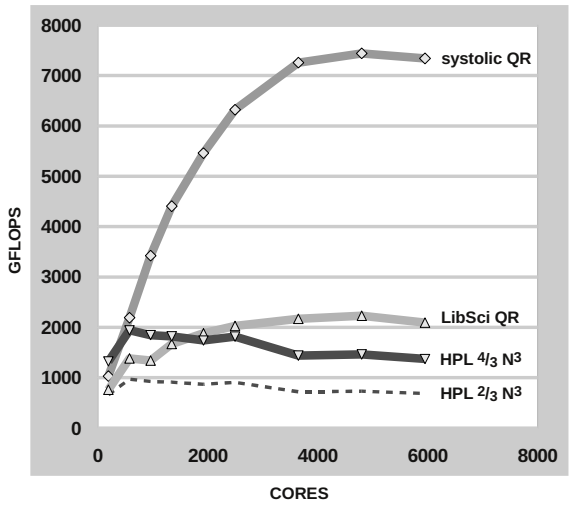

Figure 7. Systolic QR performance for a problem of size $20K \times 20K$.

Figure 9 shows an execution trace for a problem of size $10K \times 10K$ running on 240 cores. The critical path of the algorithm is clearly visible at the end of the factorization but is well hidden throughout the earlier stages of execution by the updates performed by the **dtsmqr** function.

B. Cray Scientific Libraries (LibSci) package

Performance results were compared against the Cray LibSci package, which includes an implementation of the QR factorization from ScaLAPACK. LibSci distributes the matrix in a 2D block-cyclic fashion, with a $p \times q$ processor grid. The performance is sensitive to the ratio of p to q,

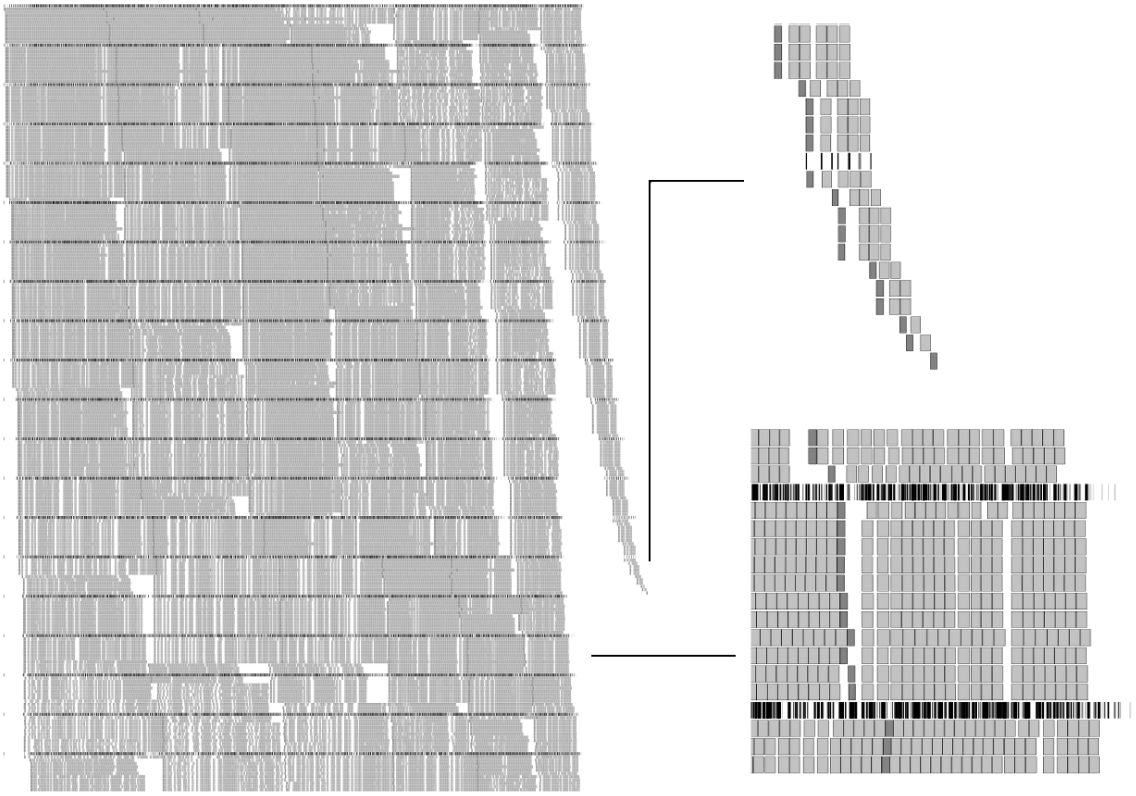

Figure 9. Execution trace for systolic QR tile factorization for a problem of size $10K \times 10K$ on 240 cores.

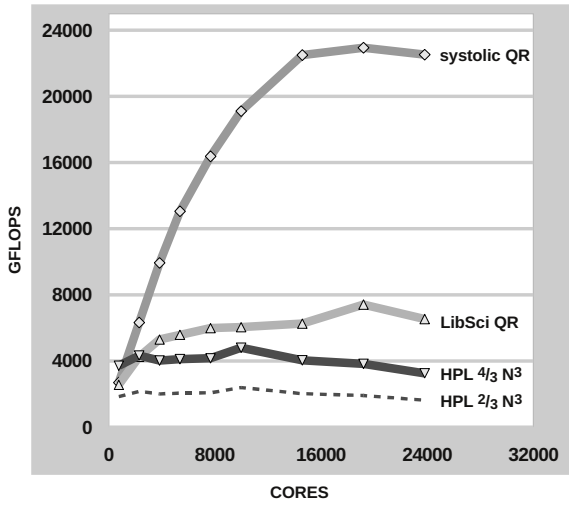

Figure 8. Systolic QR performance for a problem of size $40K \times 40K$.

and to the block size nb. For LibSci, we found that on a large number of cores, a wide processor grid with a small block size is preferred. We ran LibSci with the block sizes of $nb = 8, 16, 32, 64, 128, 256$ on the processor grids with $p = 2, 3, 4, 6, \ldots$ for $p < q$. Figures 6, 7, and 8 show the best performance of LibSci from test runs on different numbers of cores.

C. High Performance Linpack (HPL)

The LU factorization using the High Performance Linpack (HPL) benchmark is also compared. LU is a different algorithm, with $O(\frac{2}{3}N^3)$ flops compared to the $O(\frac{4}{3}N^3)$ flops for QR. Results here plot both the raw performance result, based on $\frac{2}{3}N^3$ flops, and for better comparison with QR, a virtual Gflop/s rate using $\frac{4}{3}N^3$ flops. This provides a more meaningful comparison of problems per second or time-to-solution. That is, which algorithm can solve a system of linear equations fastest, regardless of theoretical operation count. Similar size problems over the same range of cores were tested, with the problem size set to be a multiple of the block sizes. HPL also uses a 2D block-cyclic distribution, with performance sensitive to the ratio of p to q, and to the block size nb. For each number of cores, several different processor grids were tried. Nearly square processor grids, with $p \approx q$, performed poorly. Wide processor grids, with $p \leq \frac{1}{2}q$, achieved good performance. The best performing grid size tested for each number of cores is shown in Figures 6, 7, and 8. Different block sizes

were also tested for each grid size. Generally, a moderate block size of $nb = 120$ achieved the best performance. A larger block size of $nb = 220$ achieved best performance for larger numbers of cores, over 2500 cores for $n = 21120$, and over 9900 cores for $n = 42240$. Smaller block sizes of $nb = 20$ and $nb = 40$ were tested on some grids and found to have worse performance.

VII. Discussion

A. Analysis of Scalability Bounds with Amdahl's Law

for $k = 0$ **to** $N - 1$ **do**
  dgeqrt($_{inout}A_{kk}$)
  dormqr($_{in}A_{kk}$, $_{inout}A_{k,k+1}$)
  dtsqrt($_{inout}A_{kk}$, $_{inout}A_{k+1,k}$)
  dtsmqr($_{in}A_{k+1,k}$, $_{inout}A_{k,k+1}$, $_{inout}A_{k+1,k+1}$)
end for

Figure 10. Operations on critical path of the tile QR.

The attention is next shifted to application of Amdahl's Law [21] because it allows analysis of strong scaling properties of the systolic QR factorization.

The serial portion of the QR factorization is the computation performed on the diagonal of the matrix because the off-diagonal updates may be performed in a parallel and scalable fashion [22], [23], [2]. In fact, the diagonal computation constitutes the critical section of the algorithm. Figure 10 shows the four essential operations that are performed on the critical path, but the **dormqr** and **dtsqrt** may proceed in parallel with each other. Based on this observation, the sequential computation time may be formulated as follows:

$$t_{\text{comp}} = 3\frac{\frac{N}{B}\frac{4}{3}B^3}{\alpha} = 3N\,B^2/\alpha. \tag{1}$$

where B is the tile size and α is the average Gflop/s rate of the four tile operations from Figure 10. Trivially, N/B yields the number of tiles in a single column or row, while $\frac{4}{3}B^3$ is the total number of floating point operations performed for the standard QR algorithm. t_{comp} has been obtained experimentally by performing a sequential run and measuring time spent in execution of the critical path. Based on this experiment, the execution rate α was determined to be 3.2 Gflop/s and 3.3 Gflop/s for tile sizes B of 192 and 256, respectively. Not surprisingly, this is much lower than the measured execution rate of **dtsmqr** which was barely below 7.5 Gflop/s. In addition to computation, there is a need to account for communication time that has data transmission and latency components. The transmission of data occurs by utilizing bandwidth β for $4N/B$ tiles (overlap of **dormqr** and **dtsqrt** is not possible because there is only a single network interface shared between all cores) of B^2 elements total after λ latency delay, and the communication happens only for every Cth tile because there are C cores per node:

$$t_{\text{comm}} = 4\left(\frac{N}{B}B^2\frac{1}{\beta} + \frac{N}{B}\lambda\right)\frac{1}{C}. \tag{2}$$

Parameters β and λ (bandwidth and latency) are usually provided by a vendor and are often measured with micro-benchmarks [24]. Just as it was the case with computation time, the experimental method of measuring these parameters was chosen.

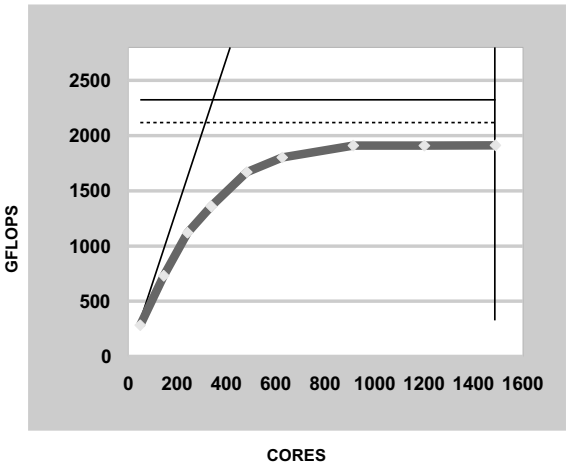

Figure 11. Systolic QR performance for a problem of size $10K \times 10K$ for tile size 192 and its theoretical performance bounds.

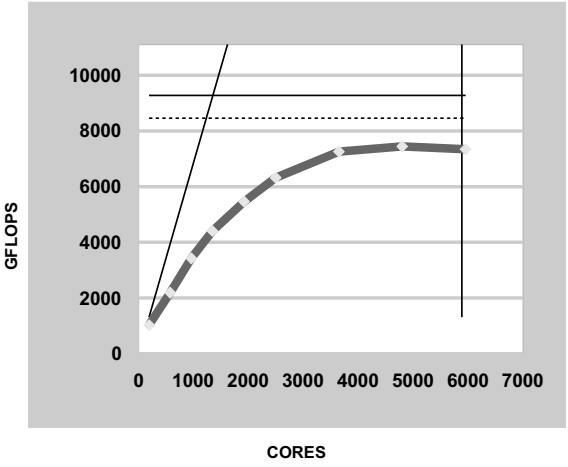

Figure 12. Systolic QR performance for a problem of size $20K \times 20K$ for tile size 192 and its theoretical performance bounds.

The constructed model may be used to estimate how close the presented implementation reaches the theoretical scalability limits. In Figures 11, 12, 13, 14, 15, and 16 the solid line indicates a model based on computation only, and the dashed line represents a model based on computation and

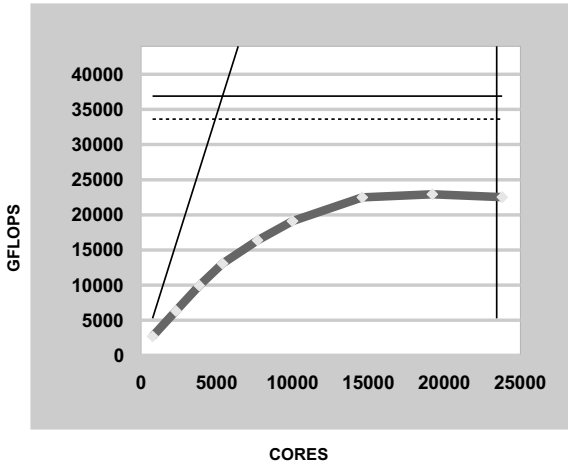

Figure 13. Systolic QR performance for a problem of size $40K \times 40K$ for tile size 192 and its theoretical performance bounds.

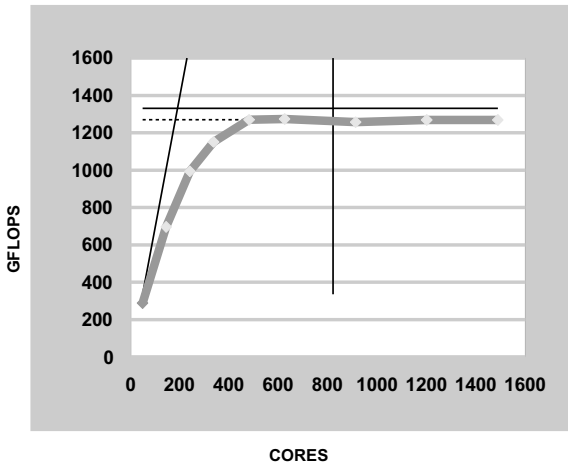

Figure 14. Systolic QR performance for a problem of size $10K \times 10K$ for tile size 256 and its theoretical performance bounds.

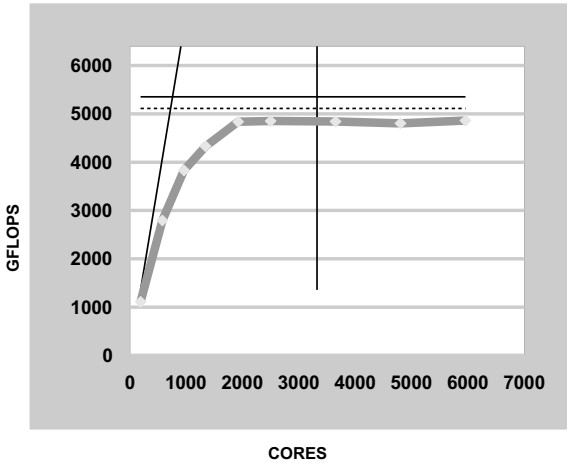

Figure 15. Systolic QR performance for a problem of size $20K \times 20K$ for tile size 256 and its theoretical performance bounds.

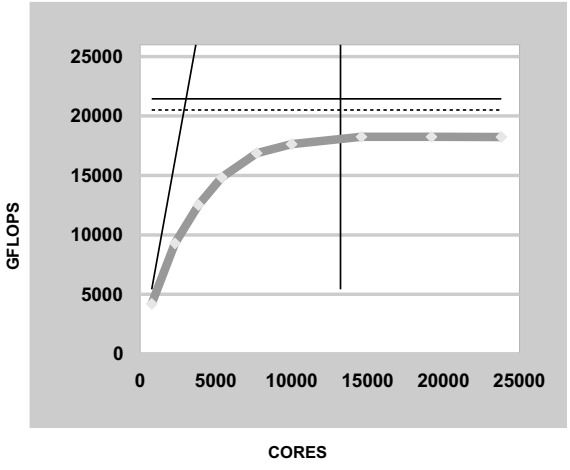

Figure 16. Systolic QR performance for a problem of size $40K \times 40K$ for tile size 256 and its theoretical performance bounds.

communication. On average, the latter model is within 13% of the theoretical limit for $B = 192$ and 5% for $B = 256$.

The model can also be used to obtain the sensitivity of scalability with respect to computational speed, bandwidth, and latency. Table I shows this sensitivity as improvement factors one can obtain by using both computation and communication times in our model. Using computation-only time for the critical path reduces accuracy of the model by mostly a single digit factor. Only for $N = 10K$ and $B = 256$ do we see $33\times$ improvement, but for this configuration, the modeling error is already low: below 1%. On the other hand, using communication-only time is very inaccurate, orders of magnitude in fact, and hence should be avoided. It is then possible to conclude that the presented implementation is still computation-bound and improving performance of a single core would benefit the scalability the most.

B. Further Observations

The systolic approach allowed for testing the scalability of the algorithms under extreme conditions, meaning extremely large number of cores for relatively small matrix sizes. Notably, the performance charts include a run where 1,488 cores were used to factor a matrix consisting of 1,600 tiles (number of cores approaching the number of tiles in the matrix). They also include a run where 23,808 cores were used to factor a matrix of size 41,472 (number of cores approaching the size of the matrix).

Another interesting observation can be made. The largest run involved a matrix of size 41,472, which occupies $41{,}472^2 \times 8$ bytes $<$ 13 GB of memory. The code maintained parallel efficiency of 57% when ran on 192 nodes.

N	B	Improvement over comp.-only	Improvement over comm.-only
$10K$	**192**	$2\times$	$106\times$
$20K$	**192**	$2\times$	$87\times$
$40K$	**192**	$1\times$	$32\times$
$10K$	**256**	$33\times$	$14383\times$
$20K$	**256**	$2\times$	$439\times$
$40K$	**256**	$1\times$	$197\times$

Table I
IMPROVEMENT FACTORS FOR ESTIMATING ACHIEVED PERFORMANCE BASED ON THE COMPLETE MODEL VERSUS COMPUTATION-ONLY MODEL AND COMMUNICATION-ONLY MODEL.

Interestingly, this matrix can still fit entirely in the memory of one node, which is 16 GB. In other words, nearly 200 nodes were used efficiently to solve a problem which fits in the memory of a single node.

VIII. CONCLUSION

This article showed how systolic design principles can be applied to a software solution to deliver an algorithm with strong scaling capabilities never seen before. In fact, the observed scaling closely approaches the limit dictated by the Amdahl's law. The achieved capabilities outperform, by a many-fold margin, current state-of-the-art software packages and vendor-tuned libraries, neither of which have been designed with a systolic architecture in mind, but whose design blueprints are aimed at high performance levels and good scalability properties. It is posited that virtual systolic architecture offers a simple, yet effective, computational model which makes conceptualization of large scale dense linear algebra algorithms possible, and in addition, makes extreme cases of strong scaling feasible at thousand core-count regimes.

ACKNOWLEDGMENT

This work is supported by grant #SHF-1117062: "Parallel Unified Linear algebra with Systolic ARrays (PULSAR)" from the *National Science Foundation* (NSF).

The authors would like to thank the *National Institute for Computational Sciences* (NICS) for a generous time allocation on the Kraken supercomputer.

The authors would also like to thank Yves Robert for sharing his expertise on systolic arrays in many stimulating conversations.

REFERENCES

[1] E. 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*. Philadelphia, PA: SIAM, 1992, http://www.netlib.org/lapack/lug/.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997, http://www.netlib.org/scalapack/slug/.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[4] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999, DOI:10.1109/40.782564.

[5] P. Kogge (Editor & Study Lead), "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office, Tech. Rep. 278, 2008, http://www.er.doe.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf.

[6] V. Sarkar (Editor & Study Lead), "Exascale software study: Software challenges in extreme scale systems," DARPA Information Processing Techniques Office, Tech. Rep. 159, 2008, http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSSreport101909.pdf.

[7] J. Dongarra, P. Beckman *et al.*, "The international exascale software roadmap," *Int. J. High Perf. Comput. Applic.*, vol. 25, no. 1, 2011, ISSN:1094-3420 (to appear).

[8] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings*. Society for Industrial and Applied Mathematics, 1978, pp. 256–282, ISBN:0898711606.

[9] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982, DOI:10.1109/MC.1982.1653825.

[10] J. A. B. Fortes and B. W. Wah, "Systolic arrays-from concept to implementation," *Computer*, vol. 20, no. 7, pp. 12–17, 1987, DOI:10.1109/MC.1987.1663616.

[11] Y. Robert, *Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Manchester University Press, 1991, ISBN:0470217030.

[12] P. Quinton and Y. Robert, *Systolic Algorithms & Architectures*. Prentice Hall, 1991, ISBN:0138807906.

[13] D. J. Evans, *Systolic Algorithms (Topics in Computer Mathematics)*. Routledge, 1991, ISBN:2881248047.

[14] P. E. Gill, G. H. Golub, W. A. Murray, and M. A. Saunders, "Methods for modifying matrix factorizations," *Mathematics of Computation*, vol. 28, no. 126, pp. 505–535, 1974.

[15] M. W. Berry, J. J. Dongarra, and Y. Kim, "LAPACK working note 68: A highly parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form," Computer Science Department, University of Tennessee, Tech. Rep. UT-CS-94-221, 1994, http://www.netlib.org/lapack/lawnspdf/lawn68.pdf.

[16] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency Computat.: Pract. Exper.*, vol. 20, no. 13, pp. 1573–1590, 2008, DOI:10.1002/cpe.1301.

[17] ——, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput. Syst. Appl.*, vol. 35, pp. 38–53, 2009, DOI:10.1016/j.parco.2008.10.002.

[18] J. Kurzak and J. J. Dongarra, "QR factorization for the Cell Broadband Engine," *Scientific Programming*, vol. 17, no. 1-2, pp. 31–42, 2009, DOI:10.3233/SPR-2009-0268.

[19] J. Kurzak, R. Nath, P. Du, and J. J. Dongarra, "An implementation of the tile QR factorization for a GPU and multiple CPUs," in *Proceedings of the State of the Art in Scientific and Parallel Computing Conference, PARA'10*. Reykjavík: Lecture Notes in Computer Science 7134, June 6-9 2010, pp. 248–257, DOI:10.1007/978-3-642-28145-7.

[20] E. Agullo, A. Buttari, J. Dongarra, M. Faverge, B. Hadri, A. Haidar, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "PLASMA users' guide," Electrical Engineering and Computer Science Department, University of Tennessee, Tech. Rep., http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf.

[21] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of the AFIPS Conference*. Atlantic City, NJ: Academic Press, April 18-20 1967, pp. 483–485.

[22] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: a portable linear algebra library for distributed memory computers design issues and performance," *Comp. Phys. Comm*, vol. 97, no. 1-2, p. 1996, 1-15, DOI:10.1016/0010-4655(96)00017-3.

[23] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, no. 3, pp. 173–184, 1996.

[24] P. Luszczek, J. Dongarra, and J. Kepner, "Design and implementation of the HPC Challenge benchmark suite," *CT Watch Quarterly*, vol. 2, no. 4A, 2006, http://www.ctwatch.org/quarterly/articles/2006/11/design-and-implementation-of-the-hpc-challenge-benchmark-suite/index.html.