Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing

Stanimire Tomov[*,a], Rajib Nath[a], Jack Dongarra[a,b,c]

[a] *University of Tennessee (USA)*
Department of Electrical Engineering and Computer Science
1122 Volunteer Blvd
Knoxville TN 37996-3450
[b] *Oak Ridge National Laboratory (USA)*
[c] *University of Manchester (UK)*

Abstract

We present a Hessenberg reduction (HR) algorithm for hybrid systems of homogeneous multicore with GPU accelerators that can exceed 25× the performance of the corresponding LAPACK algorithm running on current homogeneous multicores. This enormous acceleration is due to proper matching of algorithmic requirements to architectural strengths of the system's hybrid components. The results described in this paper are significant because the HR has not been properly accelerated before on homogeneous multicore architectures, and it plays a significant role in solving nonsymmetric eigenvalue problems. Moreover, the ideas from the hybrid HR are used to develop a hybrid tridiagonal reduction algorithm (for symmetric eigenvalue problems) and a bidiagonal reduction algorithm (for singular value decomposition problems). Our approach demonstrates a methodology that streamlines the development of a large and important class of algorithms on modern computer architectures of multicore and GPUs. The new algorithms can be directly used in the software stack that relies on LAPACK.

Key words: Hessenberg reduction, tridiagonalization, bidiagonalization, two-sided factorizations, dense linear algebra, hybrid computing, GPUs.

*Corresponding author; phone (865) 974 - 8295
Email addresses: `tomov@eecs.utk.edu` (Stanimire Tomov), `rnath1@eecs.utk.edu` (Rajib Nath), `dongarra@eecs.utk.edu` (Jack Dongarra)

1. Introduction

Hardware trends. When processor clock speeds flat-lined in 2004, after more than fifteen years of exponential increases, CPU designs moved to homogeneous multicores. There is now widespread recognition that performance improvement on CPU-based systems in the near future will come from the use of multicore platforms. Along with multicores, the HPC community also started to use alternative hardware solutions that can overcome the shortcomings of standard homogeneous multicores on a number of applications. One important example is the use of Graphics Processing Units (or GPUs) for general purpose HPC. Graphics hardware, already a true many-core architecture, has substantially evolved over the years, exponentially outpacing CPUs in performance. Current GPUs have reached a theoretical peak performance of 1 TFlop/s in single precision, support the IEEE double precision arithmetic standard [18] (see Appendix A.2 [19] for exceptions; peak double precision performance though is currently an order of magnitude lower than the single precision performance), and have a programming model (e.g., see CUDA [19]) that may revive the *quest for a free lunch* [14]. These developments have pushed the use of GPUs to become pervasive [20, 28, 29]. Currently, major chip manufacturers, such as Intel, AMD, IBM and NVIDIA, make it more evident that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

1. Multi/many-cores, where the number of cores will continue to escalate;
2. Special purpose hardware and accelerators, especially GPUs.

These trends motivate our work because in order to efficiently use the emerging hybrid hardware, optimal software solutions will themselves have to hybridize, or in other words, to match algorithmic requirements to architectural strengths of the hybrid components. Indeed, in this paper we show that although there are algorithmic bottlenecks that prevent the reductions to upper Hessenberg, tridiagonal, and bidiagonal forms from efficiently using a multicore architecture, hybrid solutions that rely on proper task splitting and task scheduling over the multicore and GPU components can overcome these bottlenecks and as a result to yield enormous performance accelerations.

Two-sided factorizations. The reductions to upper Hessenberg, tridiagonal, and bidiagonal forms [13], also known as two-sided matrix factorizations, are important linear algebra problems, especially with their relevance to eigen/singular-value solvers. In particular, the Hessenberg reduction is the first step in computing the Schur decomposition of a non-symmetric square matrix, which in turn gives the solution for the non-symmetric eigenvalue problem. The operation count for the reduction of an $n \times n$ matrix is approximately $\frac{10}{3}n^3$ which, in addition to not running efficiently on current architectures, makes the reduction a very desirable target for acceleration. Furthermore, powering a Hessenberg matrix and solving a Hessenberg system of equations is cheap compared to corresponding algorithms for general matrices, which makes the factorization applicable in other areas as well [17].

The bottleneck. The problem in accelerating the two-sided factorizations stems from the fact that they are rich in Level 2 BLAS operations, which are bandwidth limited and therefore do not scale on multicore architectures and run only at a fraction of the machine's peak performance. There are dense linear algebra (DLA) techniques that can replace Level 2 BLAS operations with Level 3 BLAS. For example, in factorizations like LU, QR, and Cholesky, the application of consecutive Level 2 BLAS operations that occur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation be applied at once as a Level 3 BLAS (see LAPACK [1]). This approach totally removes Level 2 BLAS from Cholesky, and reduces its amount to $O(n^2)$ in LU, and QR, thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to HR [15], but in contrast to the one-sided factorizations, it still leaves about 20% of the total number of operations as Level 2 BLAS. We note that in practice 20% of Level 2 BLAS can take 70% of the total execution time on a single core, thus leaving the grim perspective that multicore use – no matter how many cores would be available – can ideally reduce only the 30% of the execution time that are spent on Level 3 BLAS. The amount of Level 2 BLAS operations in the other two-sided factorizations considered is even higher – 50% of the flops in both the bidiagonal and tridiagonal reductions are in Level 2 BLAS.

Current work directions. A subject of current research in the field of DLA are efforts to design algorithms that will reach certain communication-optimal bounds [31, 34, 3]. In practice, e.g., in the context of one-sided matrix factorizations for homogeneous multicore architectures, this revolves

around developing algorithms that use blocked data structures and localized matrix transformations (e.g., not within the entire panel as in LAPACK, but within a data block or within two blocks when used for coupling the transformations) [8, 23, 2]. These ideas can be properly modified and applied in the context of GPUs as well. Direct application of the existing algorithms has not been successful so far, mostly because they lead to parallelism of small granularity which is good for homogeneous multicores, but not for current GPUs where large-granularity, data-parallel tasks are preferred [26, 35]. To account for this, current work, e.g., within the MAGMA project [36], is on MAGNUM-tile algorithms for multiGPUs where single GPUs are used for the computations within very large (magnum) data blocks (tiles) [21].

Ideas involving block data layouts and localized matrix transformations can also be used in the two-sided matrix factorizations. For example, similarity transformations based on the Householder transformation [13] can be used to annihilate matrix elements away from the diagonal of the matrix, leading to two-sided factorizations to band matrix forms [32, 33]. The band reduction can be done fast because it avoids certain data dependencies that lead to large Level 2 BLAS operations in the two-sided factorizations. In effect, it only delays the difficult to handle dependencies until a second stage reduction – to the full upper Hessenberg/bidiagonal/tridiagonal forms – that can totally eliminate the performance gains from the first stage. Indeed, there are no currently available results showing the computational feasibility of this two-stages approach for the reduction to Hessenberg and bidiagonal forms. For the case of tridiagonalization on multicore architectures though P. Bientinesi et al. [30] showed about two times performance improvement. We note that although the first stage was cast as Level 3 BLAS in their algorithm, its execution did not scale by increasing the number of cores used and the authors obtained better performance by using a GPU for that stage. A further drawback for the approach going through band form is that when the purpose of the factorization is the computation of eigenvectors, the orthogonal transformations used in the factorizations have to be accumulated into an orthogonal matrix, and that may be challenging to achieve in high performance because of the irregular nature and small granularity of the operations introduced during the second stage.

In contrast, the approach in this paper speeds up the two-sided factorizations and the results are in LAPACK data-compliant format, thus making the new algorithms directly usable in the software stack that relies on LAPACK.

The rest of the paper is organized as follows. In Section 2, we give background information on multicore and GPU-based computing in the area of DLA. Section 3 describes the standard HR algorithm, the proposed hybridization, and its extension to the tridiagonal and bidiagonal reductions. Next are performance results (Section 4) and finally conclusions (Section 5).

2. Hybrid GPU-based computing

The development of high performance DLA for new architectures, and in particular multicores, has been successful in some cases, like the one-sided factorizations, and difficult for others, like some two-sided factorizations. The situation is similar for GPUs - some algorithms map well, others do not. By combining these two architectures in a hybrid multicore + GPU system we seek to exploit the opportunity of developing high performance algorithms, as bottlenecks for one of the components (of this hybrid system) may not be for the other. Thus, proper work splitting and scheduling may lead to very efficient algorithms.

Previous work. This opportunity for acceleration has been noticed before in the context of one-sided factorizations. In particular, while developing algorithms for GPUs, several groups [27, 4, 2] observed that panel factorizations are often faster on the CPU than on the GPU, which led to the development of highly efficient one-sided hybrid factorizations for single CPU core + GPU [9, 26], multiple GPUs [26, 22, 21], and multicore+GPU systems [25]. M. Fatica [11] developed hybrid `DGEMM` and `DTRSM` for GPU-enhanced clusters, and used them to accelerate the Linpack benchmark. This approach, mostly based on BLAS level parallelism, results only in minor or no modifications to the original source code.

Further developments. The concept of representing algorithms and their execution flows as Directed Acyclic Graphs (DAGs) can be used to generalize and further develop the hybrid GPU-based computing approach. To accomplish this we split the computation into tasks and dependencies among them, and represent this information as a DAG, where DAG nodes are the tasks and DAG edges the dependencies [7]. Figure 1 shows an illustration. The nodes in red in this case represent the sequential parts of an algorithm (e.g., panel factorization) and the ones in green the tasks that can be done in parallel (e.g., the update of the trailing matrix). Proper scheduling can ensure very efficient execution. This is the case for the one-sided factorizations, where we

schedule the execution of the tasks from the critical path on the CPU (that are in general small, do not have enough parallelism, and therefore could not have been efficiently executed on the GPU) and the rest on the GPU (grouped in large task for single kernel invocation as shown; highly parallel).

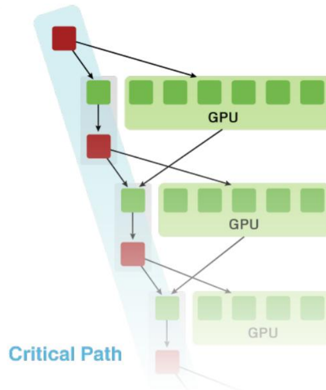

Figure 1: Algorithms as DAGs for hybrid GPU-based computing

The hybrid approaches mentioned so far have used GPUs for Level 3 BLAS parts of their computation. We note that the introduction of GPU memory hierarchies, e.g., in NVIDIA's CUDA-enabled GPUs [29], provided the opportunity for an incredible boost of Level 3 BLAS [26, 16], because memory could be reused rather than having performance relying exclusively on high bandwidth as in earlier GPUs. Indeed, one can see that early attempts to port DLA on GPUs have failed to demonstrate speedup compared to CPUs [10, 12]. Nevertheless, high bandwidth has always been characteristic for GPUs, and can be instrumental in overcoming bandwidth bottlenecks in a number of very important DLA algorithms, as shown in this paper. We design a hybrid HR algorithm that exploits the strength of multicore and GPU architectures, where related to GPUs, we use their high performance on both Level 3 and Level 2 BLAS.

3. Hessenberg reduction

The HR algorithm reduces a general $n \times n$ matrix A to upper Hessenberg form H by an orthogonal similarity transformation $Q^T AQ = H$. The matrix

Q is represented as a product of $n-1$ elementary reflectors

$$Q = H_1\ H_2\ \ldots\ H_{n-1}, \quad H_i = I - \tau_i\ v_i v_i^T,$$

where τ_i is scalar, and v_i is a vector. In the block HR algorithm a set of nb reflectors, where nb is referred to as block size, can be grouped together

$$H_1\ H_2\ \ldots\ H_{nb} \equiv I - V\ T\ V^T,$$

where $V = (v_1|\ldots|v_{nb})$, and T is $nb \times nb$ upper triangular matrix. This transformation, known as *compact WY transform* [5, 24], is the basis for the delayed update idea mentioned above, where instead of applying nb Level 2 BLAS transformations (that are inefficient on current architectures), one can apply the accumulated transformation as a Level 3 BLAS. The resulting algorithm is known as block HR.

3.1. Block Hessenberg reduction

Algorithm 1 gives (in pseudo-code) the block HR, as currently implemented in LAPACK (function `DGEHRD`). Function `DGEHD2` on line 6 uses

Algorithm 1 `DGEHRD`(n, A)

1: **for** $i = 1$ to $n - nb$ step nb **do**
2: `DLAHR2`$(i,\ A(1:n,\ i:n),\ V,\ T,\ Y)$
3: $A(1:n,\ i+nb:n)\ -=\ Y\ V(nb+1:n-i+1,\ :\ )^T$
4: $A(1:i,\ i:i+nb)\ -=\ Y(1:i,\ :\ )V(1:nb,\ :\ )^T$
5: $A(i+1:n,\ i+nb:n)\ =\ (I - V\ T\ V^T)\ A(i+1:n,\ i+nb:n)$
6: **end for**
7: `DGEHD2`(...)

unblocked code to reduce the rest of the matrix. Algorithm 2 gives the pseudo-code for `DLAHR2`. `DLAHR2` performs the two-sided reduction for the current panel and accumulates matrices V and T for the *compact WY transform* $(I - V\ T\ V^T)$, and matrix $Y \equiv A(1:n,\ i:n)\ V\ T$. We denote by $Y_j \equiv (y_1|\ldots|y_j)$ the first j columns of Y, by T_j the submatrix $T(1:j,\ 1:j)$, and by $V_j \equiv (v_1|\ldots|v_j)$ the first j columns of V. Householder(j, x) returns a vector v and a scalar $\tau = v^T v/2$ where

$$v(1:j) = 0, \quad v(j+1) = 1, \quad v(j+2:\ ) = x(2:\ )/(x(1) + sign(x(1))||x||_2).$$

Algorithm 2 DLAHR2(i, A, V, T, Y)

1: **for** $j = 1$ to nb **do**
2: $A(i+1:n,\ j) -= Y_{j-1}\ A(i+j-1,\ 1:j-1)$
3: $A(i+1:n,\ j) = (I - V_{j-1}\ T_{j-1}^T\ V_{j-1}^T)\ A(i+1:n,\ j)$
4: $[v_j,\ \tau_j] =$ Householder$(j,\ A(i+j+1:n,\ j))$
5: $y_j = A(i+1:n,\ j+1:n)\ v_j$
6: $T_j(1:j-1,\ j) = -\tau_j\ T_{j-1}\ V_{j-1}^T\ v_j; \quad T_j(j,j) = \tau_j$
7: **end for**
8: $Y(1:i,\ 1:nb) = A(1:i,\ i:n)\ V\ T$

3.2. On designing the hybrid algorithm

The design consists of identifying the bottlenecks and properly splitting the computation into tasks and scheduling their execution over the multicore host and the GPU. Clearly, the bottleneck in the HR algorithm is in the panel factorization – line 5 of Algorithm 2, also illustrated on Figure 2.

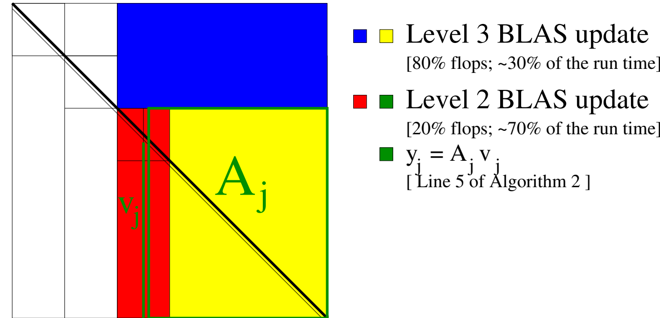

Figure 2: Current computational bottleneck: the Level 2 BLAS $y_j = A_j v_j$

3.2.1. Task splitting

Every iteration of the HR Algorithm 1 is split into three coarse-level, data-parallel tasks. Each of these tasks is done in parallel (nested parallelism) on the GPU or the multicore. The tasks and denoted by P_i, M_i, and G_i and update the three matrices correspondingly denoted by P_i, M_i, and G_i on Figure 3, Left, and described as follows:

- The panel factorization task P_i
  P_i accounts for 20% of the flops and updates the current panel, i.e., line 2 of Algorithm 1, accumulating matrices V_i, T_i and Y_i.

- The trailing matrix update task G_i
  Task G_i accounts for 60% of the flops and updates submatrix

$$G_i = (I - V_i\ T_i\ V_i^T)\ G_i\ (I - V_i\ T_i\ V_i(nb+1:\ ,\ :\ )^T)$$

- The "top" matrix update task M_i
  Task M_i accounts for 20% of the flops and updating the submatrix

$$M_i = M_i\ (I - V_i\ T_i\ V_i^T).$$

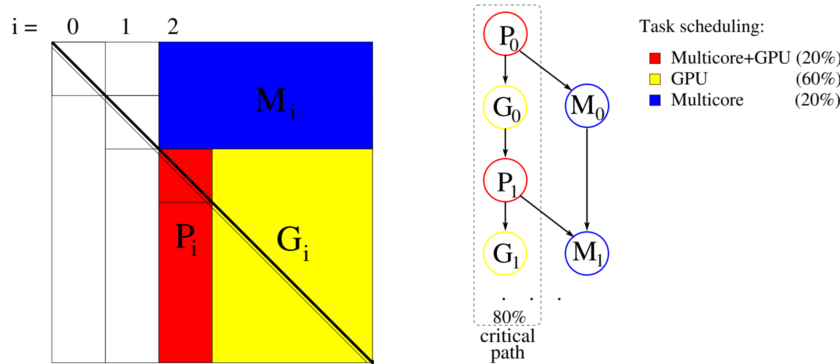

Figure 3: Main tasks and their scheduling

We note that splitting line 3 of Algorithm 1 and merging it into tasks G_i and M_i is motivated by a memory footprint analysis. Indeed, using this splitting task M_i becomes independent of G_i and falls off the critical path of the algorithm (see Figure 3, Right). This is an important contribution to the design of a parallel HR algorithm as it removes dependencies that in turn enable overlapping task M_i with that of the P_i.

3.2.2. Scheduling

The coarse-level scheduling (over the system's hybrid components) is given on Figure 3, Right. The tasks on the critical path must be done as fast as possible – and are scheduled in a hybrid fashion on both the Multicore and GPU. The memory footprint of task P_i, with 'P' standing for panel, is both P_i and G_i but G_i is accessed only for the time consuming computation of $y_j = A_j v_j$ (see Figure 2). Therefore, the part of P_i that is constrained to the panel (not rich in parallelism, with flow control statements) is scheduled

on the multicore, and the time consuming $y_j = A_j v_j$ (highly parallel but requiring high bandwidth) is scheduled on the GPU. G_i, with 'G' standing for GPU, is scheduled on the GPU. This is Level 3 BLAS update and can be done very efficiently on the GPU. Moreover, note that G_{i-1} contains the matrix A_j needed for task P_i, so for the computation of $A_j v_j$ we have to only send v_j to the GPU and the resulting y_j back from the GPU to the multicore. The scheduling so far heavily uses the GPU, so in order to make the critical path execution faster and at the same time to make a better use of the multicore, task M_i, with 'M' standing for multicore, is scheduled on the multicore.

3.3. Hybrid Hessenberg reduction

Algorithm 3 gives in pseudo-code the hybrid HR algorithm. Prefix 'd', standing for device, before a matrix denotes that the matrix resides on the GPU memory. The algorithm name is prefixed by MAGMA, standing for *Matrix Algebra for GPU and Multicore Architectures*, and denoting a project[1] on the development of a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current Multicore+GPU systems [36].

Algorithm 3 `MAGMA_DGEHRD`(n, A)

1: Send matrix A from the CPU to matrix dA on the GPU
2: **for** $i = 1$ to $n - nb$ step nb **do**
3: `MAGMA_DLAHR2`(i, V, T, dP_i, dV, dT, dY)
4: Send d$G_{i-1}(1 : nb, \ :\ )$ to the multicore (asynchronously)
5: Schedule G_i on the GPU (asynchronously; using dV, dT, and dY)
6: Schedule M_i on the multicore (asynchronously; using V and T)
7: **end for**
8: `MAGMA_DGEHD2`(...)

Algorithm 4 gives the pseudo-code for `MAGMA_DLAHR2`.

Figure 4 illustrates the communications between the multicore and GPU for inner/outer iteration j/i. Copies 1..4 are correspondingly steps/lines 1, 6, and 9 from Algorithm 4 and line 4 from Algorithm 3. Note that this pattern of communication allows us to overlap the CPU and GPU work as desired –

[1]see http://icl.cs.utk.edu/magma/

Algorithm 4 `MAGMA_DLAHR2`$(i,\ V,\ T,\ dP_i,\ dV,\ dT,\ dY)$

1: Send dP_i from the GPU to P on the multicore
2: **for** $j = 1$ to nb **do**
3: $\quad P(\ :\ ,\ j) -= \ Y_{j-1}\ T_{j-1}\ P(j-1,\ 1:j-1)$
4: $\quad P(\ :\ ,\ j)\ =\ (I - V_{j-1}\ T_{j-1}^T\ V_{j-1}^T)\ \ P(\ :\ ,\ j)$
5: $\quad [v_j,\ \tau_j]\ =\ $ Householder$(j,\ P(j+1:\ ,\ j)\ )$
6: $\quad$ Send v_j from the multicore to dv_j on the GPU
7: $\quad dy_j = \ dA(i+1:n,\ j+1:n)\ \ dv_j$
8: $\quad T_j(1:j-1,\ j)\ =\ -\tau_j\ T_{j-1}v_j;\quad T_j(j,j) = \tau_j$
9: $\quad$ Send dy_j from the GPU back to y_j on the CPU
10: **end for**
11: Send T from the multicore to dT on the GPU

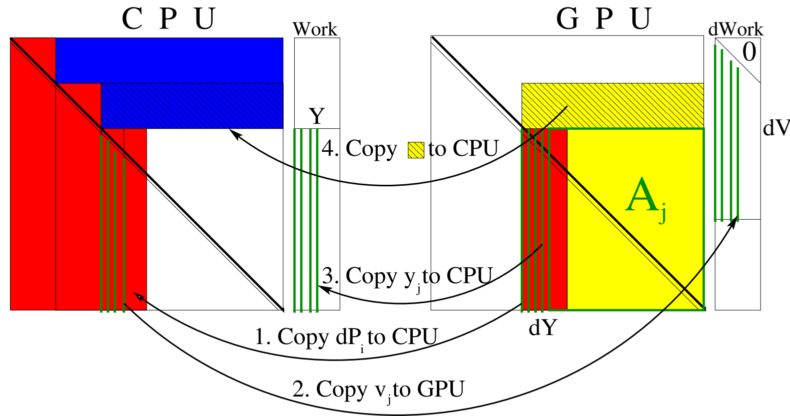

Figure 4: CPU/GPU communications for inner/outer iteration j/i.

in the outer loop (Algorithm 3) step 6 on the multicore is overlapped with steps 3, 4, and 5 on the GPU, and in the inner loop step 8 on the multicore is overlapped with step 7 on the GPU. Note that the zeroes in the upper triangular part of dV can be (and are) reused in all outer steps.

3.4. Differences with LAPACK

Our user interface is exactly as LAPACK's `DGEHRD`. The user gives and receives the factored matrix in the same format. The result is the same up to round-off errors related to a slightly different order of applying certain computations. In particular, LAPACK's matrix-matrix multiplications involving V are split into 2 multiplications: a `DTRMM` with the lower triangular

sub-matrix $V(1:nb,\ 1:nb)$ and a DGEMM with the rest of V. As nb is usually small, multiplications on the GPU with triangular $nb \times nb$ matrices is slow. Therefore, we keep zeroes in the upper triangular part of $V(1:nb,\ 1:nb)$ and perform multiplications with V using just one kernel call. For the same reason, multiplications with T are performed as DGEMMs. In LAPACK, matrix $Y = A\ V\ T$ is accumulated during the panel factorization. We accumulate $A\ V$ during the panel factorization and T is applied at once as a DGEMM during the matrix update part of the algorithm. Our work space is twice larger than LAPACK's work space on both the multicore and the GPU. This means we need work space of size $2 \times n \times nb$. On the multicore the additional space is used to enable processing tasks P and M in parallel (as each of them needs $n \times nb$ work space). On the GPU the additional space is used to separately store V from the matrix so that we can put zeroes only once in its upper triangular part, and use V as mentioned above. These modifications, in addition to providing higher performance, make also the code very readable, and shorter than LAPACK's.

3.5. Extension to other two-sided factorizations

The methodology from the hybrid HR algorithm can be used to develop other two-sided factorizations, e.g., tridiagonalization for symmetric matrices and bidiagonalization for general matrices:

Tridiagonalization. This is the reduction of a symmetric matrix to symmetric tridiagonal form by orthogonal similarity transformations. Using directly the HR algorithm on a symmetric matrix yields a tridiagonal matrix reduction in $\frac{10}{3}n^3 + O(n^2)$ flops, but exploiting the symmetry reduced the flops count to $\frac{4}{3}n^3 + O(n^2)$ (function SYTRD from LAPACK). Thus, compared to the HR algorithm there are no G_i tasks and therefore the multicore can not be used in a similar way. The rest of the methodology developed for the HR algorithm can be applied directly. The only difference is that the trailing matrix updates G_i are SYR2Ks *vs* GEMMs in HR and the bottleneck matrix-vector products in the panels P_i are SYMV *vs* GEMV. The tridiagonalization has 50% of its flops into the panel factorization, making the performance of the symmetric matrix-vector product even more important for the overall performance of the algorithm.

Bidiagonalization. This is the reduction of a general matrix to bidiagonal form by orthogonal transformations, e.g., $Q^T AP$ is bidiagonal where Q and P are orthogonal and A a general m-by-n matrix. The bidiagonalization

is function GEBRD in LAPACK and the implementation is asymptotically in $4mn^2 - 4n^3/3$, $m \geq n$ flops. Compared to the HR algorithm there are two panels being factored at each step – a block of columns as in the HR algorithm and a corresponding block of rows. The methodology developed for the HR algorithm again can be applied directly. The difference is that similar to the bidiagonalization there are no G_i tasks (as these are the block of rows panels). Both panels are factored on the CPU and the two large matrix-vector products (needed in the panels) are offloaded to the GPU. Similarly to the bidiagonalization, the tridiagonalization has 50% of its flops into the panel factorizations, making the performance of the general matrix-vector product even more important for the overall performance of the algorithm.

4. Performance Results

The performance results in this section use NVIDIA's GeForce GTX 280 GPU and its multicore host, a dual socket quad-core Intel(R) Xeon(R) E5410 operating at 2.33 GHz (i.e., peak is 149 GFlop/s in single and 74.5 GFlop/s in double precision arithmetic). The GTX 280 has 30 multiprocessors, each multiprocessor having 8 SIMD functional units operating at 1.30 GHz, each unit capable of executing up to three (single floating point) operations per cycle. The GTX 280 is connected to the host via PCI Express 16x adapter card (5.7 GB/s of CPU-to-GPU and 5.5 GB/s GPU-to-CPU bandwidth for pinned memory; latency is approximately 11 μs in both directions). The theoretical bandwidth peak is 141 GB/s. The CPU FSB is 1333 MHz and the theoretical bandwidth peak is 10.41 GB/s. On the multicore we use LAPACK and BLAS from MKL 10.0 and on the GPU CUBLAS 2.3, unless otherwise noted.

Performance. Figure 5 shows the performance of 2 hybrid HR algorithms, and the block HR on single core and multicore in double precision arithmetic. The basic hybrid HR is for 1 core + GPU, and uses CUBLAS 2.3. The Multicore+GPU hybrid algorithm is the one described in the paper plus various kernels' optimizations, described as follows. The result shows that we achieve an enormous 16× speedup compared to the current block HR running on multicore. We see that the basic implementation brings most of the acceleration.

Note that we get asymptotically within 90% of the "upper bound" performance, as shown on Figure 5. Here upper bound denotes the performance

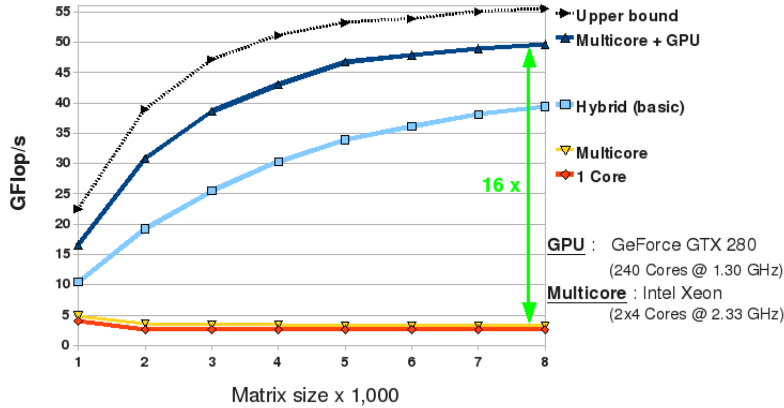

Figure 5: Performance (in double precision) for the hybrid HR.

of the *critical path* (only tasks P_i and G_i) of our algorithm when we do not count synchronization and data transfer times.

Figure 6 shows the performance of the HR, bidiabonalization, and tridiagonalization algorithms using one GPU (left) and multicore (right) in single precision arithmetic. Compared asymptotically to the multicore algorithms, the speedup for the hybrid HR is 25×, for the tridiagonalization 8×, and for the bidiagonalization 20×.

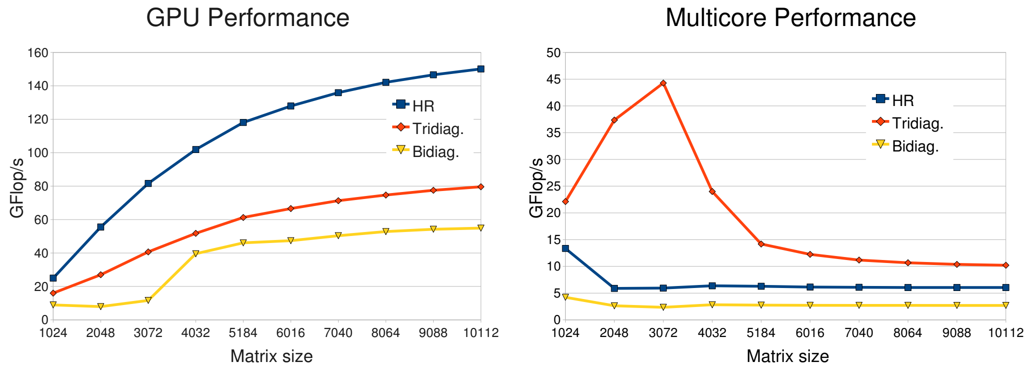

Figure 6: Performance of the two-sided factorizations using one CPU core and one GPU (left) and multicore (right) in single precision arithmetic.

Optimizations. We optimized the GPU matrix-vector product as it is critical for the performance. Figure 7 shows the GEMV performances from

MAGMA, CUBLAS, and MKL. The theoretically optimal implementation

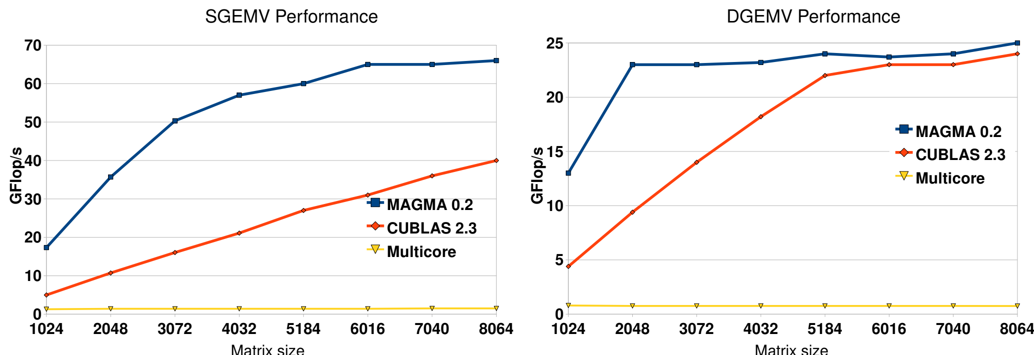

Figure 7: Performance of CPU *vs* GPU matrix-vector product.

in single precision would have a performance of 70 GFlop/s (the theoretical maximum bandwidth of 141 GB/s over 2). This would assume 100% bus utilization and 100% overlap of the computation with the communication needed. MAGMA achieves 66 GFlop/s which is 94% of the theoretical `SGEMV` peak on the GPU. MKL gets up to 1.4 GFlop/s which is 28% of the theoretical `SGEMV` peak on the multicore.

All algorithms use block size $nb = 32$. Testing with larger nb gives slower performance results. For $nb = 32$ we used `MAGMA_DGEMM` kernels that outperform CUBLAS 2.3 by 10 GFlop/s on average. These kernels are based on the auto-tuning approach described in [16].

We also optimized the multicore implementation of tasks M_i in the HR algorithm. Our original implementation used MKL's parallel BLAS to get an average performance of about 17 GFlop/s for matrix of size $8,000$ (the averages for P_i and G_i are correspondingly 23 GFlop/s and 64 GFlop/s), and about 10 GFlop/s towards the end of the computation. We changed this to a 1-D block row partitioning of M_i and assigned the update for single block of rows to a single core. This is a trivial splitting and was easy to code using OpenMP. The performance improved to an average of 30 GFlop/s and up to 45 GFlop/s towards the end of the computation. High performance towards the end of the computation is important (especially for large matrices) because this is when M_i becomes larger and larger compared to P_i and G_i. Using the optimized code on a matrix of size $8,000$, the execution of tasks M_i is totally overlapped with the execution of P_i and G_i for the first 97% of

the computation, and becomes dominant in the last 3% of the computation. In our case this was not a bottleneck because of the high performance that we achieve at the end. Another solution is if the GPU is scheduled to do part of M_i near the end of the computation.

The tridiagonalization and the bidiagonalization are implemented only in single precision as a proof of concept. We optimized a `SYMV` GPU kernel to achieve up to 45 GFlop/s (included in MAGMA 0.2 [36]). Although this is 10 to 15× faster than CUBLAS's `SYMV`, it is still far away from the 66 GFlop/s achieved for the `GEMV` kernel. This motivated another optimization, namely, a GPU implementation of `SYR2K` that explicitly generates the entire symmetric matrix resulting from the operation, so that we can use `GEMV` in the panels instead of the slower `SYMV`. This approach does not need extra memory. The kernel does not perform extra operations, just the extra copy needed, and reaches up to 256 GFlop/s *vs* 149 GFlop/s in CUBLAS's `SYR2K` and 291 GFlop/s in MAGMA BLAS's `SYR2K` (to be included in MAGMA 0.3). Note that using a 45 GFlop/s `SYMV` kernel (for 50% of the flops) and a 291 GFlop/s `SYR2K` kernel (for the rest 50%), the optimal performance for the tridiagonalization will be

$$\frac{45 * 291}{0.5 * 291 + 0.5 * 45} \approx 78 \text{ GFlop/s.}$$

Using the 66 GFlop/s `GEMV` kernel (for 50% of the flops) and the 256 GFlop/s modified `SYR2K` kernel (for the rest 50%), the optimal performance for the tridiagonalization will be

$$\frac{66 * 256}{0.5 * 256 + 0.5 * 66} \approx 105 \text{ GFlop/s.}$$

We achieve 76% of his peak for a matrix of size $10,000$.

The current tridiagonalization and the bidiagonalization implementations are not fully optimized as further improvements are possible in the CUDA BLAS kernels needed.

5. Conclusions

We presented a hybrid HR algorithm that can exceed 25× the performance of the current LAPACK algorithm running just on current homogeneous multicore architectures. Moreover, we showed how to extend the ideas

from the HR algorithm to the bidiagonalization and tridiagonalization algorithms (to achieve acceleration of correspondingly 20× and 8×). The results are significant because the reductions presented have not been properly accelerated before on homogeneous multicore architectures, and they play a significant role in solving eigenvalue and singular value decomposition problems. Moreover, our approach demonstrates a methodology that streamlines the development of a large and important class of DLA algorithms on modern computer architectures of multicores and GPUs.

Acknowledgments

This work is supported by Microsoft, NVIDIA, the U.S. National Science Foundation, and the U.S. Department of Energy. We thank Julien Langou (UC, Denver) and Hatem Ltaief (UT, Knoxville) for their valuable suggestions and discussions on the topic.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.

[2] M. Baboulin, J. Dongarra, and S. Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures*, Proc. International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Trondheim, Norway, 2008.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in linear algebra*, Tech. report, LAPACK Working Note 218, May 2009.

[4] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Ortí, *Solving dense linear systems on graphics processors*, Technical Report ICC 02-02-2008, Universidad Jaime I, February, 2008.

[5] C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM J. Sci. Stat. Comp. **8** (1987), no. 1, S2–S13, Parallel processing for scientific computing (Norfolk, Va., 1985). MR 88f:65070

[6] S. Browne, C. Deane, G. Ho, and P. Mucci, *PAPI: A portable interface to hardware performance counters*, (June 1999).

[7] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, and S. Tomov, *The impact of multicore on math software*, In PARA 2006, Umea Sweden, 2006.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Technical Report UT-CS-07-600, University of Tennessee, 2007, LAPACK Working Note 191.

[9] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie, *Exploring new architectures in accelerating CFD for Air Force applications*, Proc. of HPCMP UGC08, July 14-17, 2008.

[10] K. Fatahalian, J. Sugerman, and P. Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA), ACM, 2004, pp. 133–137.

[11] M. Fatica, *Accelerating Linpack with CUDA on heterogenous clusters*, GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), ACM, 2009, pp. 46–51.

[12] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, *LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 3.

[13] G. H. Golub and C. F. Van Loan, *Matrix computations*, second ed., Baltimore, MD, USA, 1989.

[14] W. Gruener, *Larrabee, CUDA and the quest for the free lunch*, http://www.tgdaily.com/content/view/38750/113/, 08/2008, TGDaily.

[15] S. Hammarling, D. Sorensen, and J. Dongarra, *Block reduction of matrices to condensed forms for eigenvalue computations*, J. Comput. Appl. Math **27** (1987), 215–227.

[16] Y. Li, J. Dongarra, and S. Tomov, *A note on auto-tuning GEMM for GPUs.*, Proc. of 9th ICCS '09 (Baton Rouge, LA), Springer-Verlag, 2009, vol. 5544, pp. 884 - 892.

[17] C. F. Van Loan, *Using the Hessenberg decomposition in control theory*, North-Holland, Amsterdam, 1982.

[18] NVIDIA, *Nvidia Tesla doubles the performance for CUDA developers*, Computer Graphics World (06/30/2008).

[19] NVIDIA, *NVIDIA CUDA Programming Guide*, 6/07/2008, Version 2.0.

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

[21] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, *A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators*, Tech. report, LAPACK Working Note 223, November 2009.

[22] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, *An Extension of the StarSs Programming Model for Platforms with Multiple GPUs*, In Proc. of *Euro-Par '09*, pages 851–862, Delft, The Netherlands, 2009.

[23] G. Quintana-Ortí, E. S. Quintana-Ortí, R. van de Geijn, F. G. Van Zee, and E. Chan, *Programming matrix algorithms-by-blocks for thread-level parallelism*, ACM Trans. Math. Softw., Vol. 36, no. 3, pp. 1–26, 2009.

[24] R. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Stat. Comp. **10** (1989), no. 1, 53–57. MR 90b:65076

[25] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems.*, Parallel Computing (In Press), DOI: 10.1016/j.parco.2009.12.005.

[26] V. Volkov and J. Demmel, *Benchmarking GPUs to tune dense linear algebra*, Proc. of SC '08, November 15-21, 2008, Austin, Texas.

[27] V. Volkov and J. W. Demmel, *Using GPUs to accelerate linear algebra routines*, Poster at PAR lab winter retreat, January 9, 2008, http://www.eecs.berkeley.edu/~volkov/volkov08-parlab.pdf.

[28] *General-purpose computation using graphics hardware*, http://www.gpgpu.org.

[29] *NVIDIA CUDA ZONE*, http://www.nvidia.com/object/cuda_home.html.

[30] P. Bientinesi, F. Igual, D. Kressner, and E. Quintana-Orti, *Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures*, Aachen Institute for Computational Engineering Science, RWTH Aachen, AICES-2009-11, March 2009.

[31] James W. Demmel, Laura Grigori, Mark Frederick Hoemmen and Julien Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, Tech. report, LAPACK Working Note 204, August 2008.

[32] H. Ltaief, J. Kurzak, and J. Dongarra, *Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited*, Tech. report, LAPACK Working Note 208, August 2009.

[33] H. Ltaief, J. Kurzak, and J. Dongarra, *Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures*, Tech. report, LAPACK Working Note 209, October 2009.

[34] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Communication-optimal Parallel and Sequential Cholesky decomposition* Tech. report, LAPACK Working Note 215, February 2009.

[35] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, *Dense Linear Algebra Solvers for Multicore with GPU Accelerators*, Proc. of IPDPS 2010, Atlanta, GA, April 2010.

[36] S. Tomov, R. Nath, P. Du, and J. Dongarra, *MAGMA version 0.2 Users' Guide*, http://icl.cs.utk.edu/magma, November 2009.