DIVIDE & CONQUER ON HYBRID GPU-ACCELERATED MULTICORE SYSTEMS

CHRISTOF VÖMEL*, STANIMIRE TOMOV†, AND JACK DONGARRA‡

Abstract. With the raw compute power of GPUs being more widely available in commodity multicore systems, there is an imminent need to harness their power for important numerical libraries such as LAPACK. In this paper, we consider the solution of dense symmetric and Hermitian eigenproblems by LAPACK's Divide & Conquer algorithm on such modern heterogeneous systems.

We focus on how to make the best use of the individual strengths of the massively parallel manycore GPUs and multicore CPUs. The resulting algorithm overcomes performance bottlenecks that current implementations, optimized for a homogeneous multicore face. On a dual socket quad-core Intel Xeon 2.33 GHz with an NVIDIA GTX 280 GPU, we typically obtain up to about 10-fold improvement in performance for the complete dense problem.

The techniques described here thus represent an example on how to develop numerical software to efficiently use heterogeneous architectures. As heterogeneity becomes common in the architecture design, the significance and need of this work is expected to grow.

Key words. Symmetric eigenvalue problem, LAPACK, performance, GPU, Multicore, hybrid architecture, heterogeneous computing.

AMS subject classifications. 15A18, 15A23.

1. Introduction.

LAPACK [2, 12] is one of the fundamental numerical libraries. With a foundation of several decades of research in numerical linear algebra, today it faces the challenge of having to adapt to new hardware trends to maintain its edge from the high performance computing point of view.

As chip designers have to balance parallel performance and power consumption, heterogeneous hybrid GPU-based multicore platforms are setting out to become an important standard in contemporary platforms. The Matrix Algebra on GPU and Multicore Architectures (MAGMA) project [1, 27, 40, 41] aims at providing LAPACK functionalities on these new architectures. Recent efforts within this framework have focused on one-sided factorizations [26, 42] and reduction to Hessenberg form [39].

However, with eigenvalue problems being ubiquitous in computational science, availability of modern eigensolvers is a key component of the scientific software infrastructure and indeed crucial for application scientists from a large variety of disciplines. Note that dense eigenproblems do not only occur in their own right but also as a key subproblem in projection-based sparse eigensolvers, see the comments in a recent review paper on eigensolvers in electronic structure calculations [32]. Thus we address in this paper the solution of Hermitian and real symmetric eigenproblems by means of the Divide & Conquer algorithm [10, 20].

The core Divide & Conquer scheme in its original form applies to real symmetric tridiagonal systems. By standard practice [2, 30], one first transforms the dense real symmetric or complex Hermitian eigenproblem into a tridiagonal one via an orthogonal, respectively unitary, similarity transformation. Once the tridiagonal eigenproblem has been solved in a second step, the solution of the dense eigenproblem is

*Visitor at the Computer Science Division, University of California at Berkeley (voemel@eecs.berkeley.edu)

†Innovative Computing Laboratory Computer Science Department, University of Tennessee, Knoxville (tomov@cs.utk.edu)

‡Electrical Engineering and Computer Science Department, University of Tennessee Knoxville (dongarra@cs.utk.edu)

obtained by applying the appropriate orthogonal transformation to the eigenvectors from the tridiagonal part.

The organization of this paper is as follows. Section 2 reviews the ideas behind the Divide & Conquer algorithm from the theoretical and the high performance point of view. Section 3 describes the key points to address in order to obtain a high performance version of the algorithm on hybrid GPU-accelerated multicore systems. All three parts of the dense problem are covered: the reduction to tridiagonal form (Phase 1, Section 3.1), the solution of the tridiagonal problem (Phase 2, Section 3.2), and the final orthogonal transformation of the eigenvectors (Phase 3, Section 3.3). In Section 4, we study test cases and show performance results on our test system, a dual socket quad-core Intel Xeon 2.33 GHz with an NVIDIA GTX 280 GPU. In Section 5, we discuss perspectives and future work.

2. A brief overview of Divide & Conquer for tridiagonal eigenproblems. In his important review in 1973, Golub [19] first developed the theory of tridiagonal eigenproblems with a rank-1 modification. Bunch et al. [9] investigated this idea for the first time experimentally in 1978. Building on these works, Cuppen [10] then proposed a recursive scheme to solve the full tridiagonal eigenproblem in Divide & Conquer fashion.

To obtain the eigenvalues and -vectors of a real symmetric tridiagonal matrix T, three steps are performed:

1. The tridiagonal (sub-)eigenproblems $T_1 = Q_1 \Lambda_1 Q_1^T$ and $T_2 = Q_2 \Lambda_2 Q_2^T$ are solved for two matrices T_1, T_2 which are, up to a rank-1 modification, the diagonal blocks of the original system T.
2. Using the eigenpairs of T_1, T_2, all eigenvalues of T are computed by solution of a secular equation.
3. Once the eigenvalues of T are known, the eigenvectors of T are computed in two phases, consisting of an appropriate diagonal scaling followed by a matrix-matrix multiplication.

The solution of the two eigenproblems in step 1 can be performed by any method. In particular, the algorithm can be applied recursively, leading to the Divide & Conquer algorithm.

However, in practice the theoretically elegant ideas incur some surprisingly hard numerically difficulties [16, 37]. The solution of the secular equation in step 2 is far from trivial [24]. Moreover, numerical instabilities in a naive implementation can ruin the numerical orthogonality of the eigenvectors computed in step 3. Only in 1995, Gu & Eisenstat finally cured the orthogonality issues in numerically backward stable fashion [20] by invoking an earlier result from perturbation theory [25]. Their ingenious modification made the Divide & Conquer algorithm sufficiently reliable to be one of LAPACK's workhorses for the symmetric eigenproblem [2, 13, 24, 31]. A parallel implementation for distributed memory architectures is also part of ScaLAPACK [38].

2.1. Theoretical aspects of Divide & Conquer. We review the algorithmic steps in the Divide & Conquer scheme. For a pedagogical treatment at textbook level, we refer the reader to the presentation in [11] on which the following description is based.

Consider the unreduced symmetric tridiagonal matrix T block-partitioned as

$$T = \left[\begin{array}{ccc|ccccc} a_1 & b_1 & & & & & & \\ b_1 & \ddots & \ddots & & & & & \\ & \ddots & \ddots & b_{m-1} & & & & \\ & & b_{m-1} & a_m & b_m & & & \\ \hline & & & b_m & a_{m+1} & b_{m+1} & & \\ & & & & b_{m+1} & \ddots & \ddots & \\ & & & & & \ddots & \ddots & b_{n-1} \\ & & & & & & b_{n-1} & a_n \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccccc} a_1 & b_1 & & & & & & \\ b_1 & \ddots & \ddots & & & & & \\ & \ddots & \ddots & b_{m-1} & & & & \\ & & b_{m-1} & a_m - b_m & & & & \\ \hline & & & & a_{m+1} - b_m & b_{m+1} & & \\ & & & & b_{m+1} & \ddots & \ddots & \\ & & & & & \ddots & \ddots & b_{n-1} \\ & & & & & & b_{n-1} & a_n \end{array}\right] + \left[\begin{array}{ccc|ccc} & & & & & \\ & & & & b_m & b_m \\ & & & & b_m & b_m \\ \hline & & & & & \\ & & & & & \\ & & & & & \end{array}\right],$$

where the classical choice is $m = \lfloor n/2 \rfloor$. [1] Hence T can be considered a rank-1 update of a block-diagonal matrix,

$$T = \left[\begin{array}{c|c} T_1 & 0 \\ \hline 0 & T_2 \end{array}\right] + b_m \cdot \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} [0, \ldots, 0, 1, 1, 0, \ldots, 0] =: \left[\begin{array}{c|c} T_1 & 0 \\ \hline 0 & T_2 \end{array}\right] + b_m v v^T ,$$

introducing the matrices T_1, T_2 from step 1 of the overall scheme. It follows that

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + b_m v v^T \tag{2.1}$$

$$= \begin{bmatrix} Q_1 \Lambda_1 Q_1^T & 0 \\ 0 & Q_2 \Lambda_2 Q_2^T \end{bmatrix} + b_m v v^T \tag{2.2}$$

$$= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \left(\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} + b_m u u^T \right) \begin{bmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{bmatrix}. \tag{2.3}$$

For the computation of the eigenvalues of T in step 2, it thus suffices to work with the similar matrix $D + \rho u u^T$. Here, $D := \text{blockdiag}(\Lambda_1, \Lambda_2)$ and $\rho := b_m$. If $D - \lambda I$ is nonsingular, then

$$\det(D + \rho u u^T - \lambda I) = \det\left((D - \lambda I)(I + \rho (D - \lambda)^{-1} u u^T)\right). \tag{2.4}$$

[1] Both LAPACK and ScaLAPACK [38] make this choice. In a hybrid heterogeneous computing environment, it can also make sense to consider an uneven partition, see the remarks in Section 3.2.

This yields for step 2 a secular equation whose roots are the eigenvalues of T:

$$0 = 1 + \rho \sum_{i=1}^{n} \frac{u_i^2}{d_i - \lambda}. \tag{2.5}$$

In step 3, for each (non-deflated) eigenvalue λ of $D + \rho u u^T$, one can compute the corresponding eigenvector from

$$(D - \lambda I)^{-1} u. \tag{2.6}$$

By (2.3), pre-multiplication of the vectors from (2.6) with blockdiag(Q_1, Q_2) finally yields the eigenvectors of T.[2]

2.2. Performance aspects of Divide & Conquer. For a performance analysis, we now return to the overall algorithm from the beginning of Section 2. If we assume that step 1 has been performed by a recursive call to the same algorithm, what remains is to look at steps 2 and 3. (In [11], one can find an evaluation of the recursive costs including step 1.)

It turns out that there is a both elegant and efficient algorithm for computing the roots of the secular equation (2.5) in step 2. The details are given in [11, 24], we just sketch the ideas here. As a consequence of eigenvalue interlacing [30], the roots are located between poles of the secular function. Thus, if one replaces the linear model in Newton's root finder by a hyperbolic one determined by those poles, convergence is fast and within $O(n^2)$ operations for all n eigenvalues.

The computations of $\hat{u}$ and of the eigenvectors of $D+\rho\hat{u}\hat{u}^T$ in step 3 have the same complexity, $O(n^2)$ operations. The last part of step 3 however, the pre-multiplication with blockdiag(Q_1, Q_2), has $O(n^3)$ complexity when the matrices Q_1, Q_2 are dense, making it the cost-dominating part of the overall scheme.

This is demonstrated by actual data displayed in Figure 2.1. We consider the tridiagonal arising from a shifted 1D Laplace problem. Shown is the fraction of the floating point instructions (using PAPI [7, 8] and TAU [36]) for the main components of the algorithm. Function `slaed3` holds exclusively the dense matrix multiplications.

In practical experiments, it turns out that the complexity of the Divide & Conquer algorithm varies with the matrix at hand [10, 13, 38]. There is a phenomenon called deflation when the two submatrices T_1, T_2 have a (numerically) identical eigenvalue, or if one entry of $\hat{u}$ is numerically negligible. Deflation allows to omit the eigenvector computation by explicit matrix-matrix multiply for those vectors where deflation occurs. When substantial deflation occurs, the computational complexity of tridiagonal Divide & Conquer becomes closer to quadratic rather than cubic. In contrast to that, the example from Figure 2.1 shows one of the most difficult matrix classes for the Divide & Conquer algorithm, see the remarks in [13, 28], and is thus a good benchmark problem for our solver.

3. Achieving High Performance on Hybrid GPU-accelerated Multicore Systems. The development of high performance linear algebra for homogeneous multicore architectures has been successful in some cases, and difficult for others. The

[2] While (2.6) holds in theory, one can see that in finite precision this formula can easily cause complications for close eigenvalues. The computed vectors lose their orthogonality. However, as a cure one can replace u by a numerically near-by $\hat{u}$ in the eigenvector formula and obtain, with careful evaluation of the associated expression, numerically orthogonal vectors, see [11, 20] for more details.

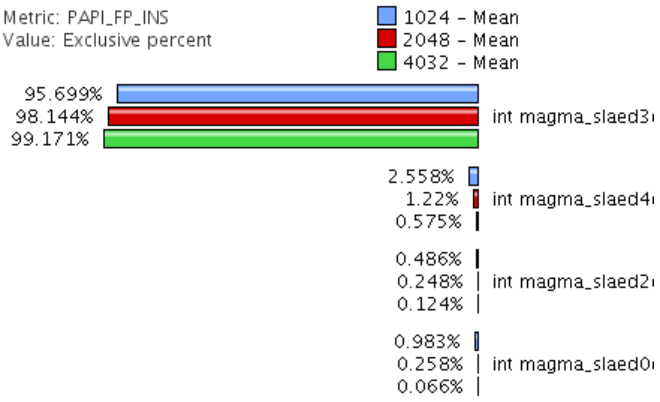

FIG. 2.1. *Fraction of the floating point instructions for the main components of the tridiagonal Divide & Conquer algorithm on three tridiagonal Toeplitz matrices of increasing size – the bulk is in* `slaed3`*, the matrix-matrix multiplications of the algorithm. This establishes the importance of this kernel for high performance.*

situation is similar for GPUs – some algorithms map well while others do not. A combination of multicore and GPUs though can be beneficial if the respective individual strengths of the two components can be leveraged successfully. In this way, a hybrid algorithm that properly splits and schedules the tasks (so that expensive communications are either avoided or overlapped with computations as much as possible) can be very efficient [40].

Indeed, hybrid algorithms have been successfully used in a number of one and two-sided matrix factorizations [39, 42]. A key ingredient for the one-sided matrix factorizations is the splitting of the computations into a so-called panel-factorization on the CPU, followed by updates of the Schur complement, done on the GPU. This particular task scheduling is mandated by the fact that the panel factorization is inefficient on the GPU while updates on the other hand can be performed very efficiently there.

Our eigensolver requires a two-sided reduction to tridiagonal form but similar considerations apply in this context. The essential details are given in Section 3.1. The performance-relevant aspects of the tridiagonal Divide & Conquer part have been described in the previous Section 2.2, their influence on the technical execution is described in Section 3.2. The back-transformation of the eigenvectors is conceptually the easiest and briefly reviewed in Section 3.3.

3.1. Phase 1: reduction to tridiagonal form. For the sake of simplicity in presentation, we assume henceforth the dense system at hand to be real symmetric but the following discussion applies equally to the complex Hermitian case.

The first phase of the overall procedure consists of reducing the given matrix A to real symmetric tridiagonal form,

$$T = Q^T A Q. \tag{3.1}$$

The orthogonal similarity transformations used in this step form Q as a product of elementary Householder reflectors, see for example LAPACK's subroutine `sytrd`. A single Householder reflector is designed to annihilate all elements below or above the first off-diagonal of A, see [11].

In terms of work in the overall algorithm, Phase 1 generally is the most expensive one of the three phases [13, 43]. Achieving high performance is thus especially important here. In a related context, namely the orthogonal reduction of a non-symmetric matrix to Hessenberg form [39], it was noted that the traditional reduction from dense to tridiagonal form does not maintain high performance when migrated from the traditional single-core LAPACK environment to multicore architectures. It was found that with up to 20% of the work consisting of Level 2 BLAS operations, the overall algorithm would run at only a small fraction of the machine's theoretical peak performance. This bottleneck is aggravated in the two-sided reduction to tridiagonal form where the Level 2 BLAS can make up to 50% of the total number of floating point operations.

One important approach to overcoming this dilemma consists of a reduction to intermediate banded and subsequent tridiagonal form [5, 6, 21, 22, 23]. Reduction to banded form can be done using BLAS 3 and avoids complicated data dependencies that would otherwise impose Level 2 BLAS operations. However, it is still hard to achieve high performance in the bulge-chasing technique used to go from narrow-banded to tridiagonal form and GPU-acceleration may mainly benefit the reduction to banded form [3].

Alternatively to the aforementioned strategy, we adapt the panel-factorization techniques outlined in [39] to our case. The data for A is copied to the GPU and all subsequent Level 3 BLAS updates are performed directly on the GPU. The panels are factored in a hybrid fashion, using both the CPU and GPU. From the technical point of view, the algorithm mirrors the LAPACK implementation of the compact WY block Householder transformation [4, 35]. Communications occur only for the computation of the Householder data while processing the panels. Once it is available on the GPU, a panel is sent back to the CPU. There, all factorization operations with memory-footprint restricted to the panel are performed using LAPACK code. The time-consuming (Level 2 BLAS) symmetric matrix-vector products of the trailing submatrix with the Householder transformation vectors are not performed on the CPU but on the GPU where this can be done more efficiently. Since the trailing matrix is already on the GPU, we only have to send the corresponding Householder vector, perform the matrix-vector product on the GPU and, finally, return the result back to the CPU.

In summary, communications concern panel data (GPU to CPU), the computed Householder data (CPU to GPU), and the result of block Householder vector times the trailing submatrix (GPU to CPU). The intermediate Householder vectors and their products with the trailing sub-matrices are kept on the GPU as well and used in the subsequent trailing matrix update. Note that every panel communication of $\mathrm{O}(n \times nb)$ data involves $\mathrm{O}(n^2 \times nb)$ flops in updates, and that every $\mathrm{O}(n)$ Householder data transfer can be amortized by $\mathrm{O}(n^2)$ flops in the subsequent matrix-vector product. This $O(n)$ ratio between floating point operations and data communication is key for efficiency, the same consideration that motivated the original development of blocked algorithms in the later 1980s [17].

3.2. Phase 2: Divide & Conquer for the tridiagonal eigenproblem. Key to achieving high performance in the second phase of the algorithm is again the amortization of expensive communication between CPU and GPU through algorithmic parts with a favorable 'volume/surface' ratio between floating point operations and data traffic.

In Divide & Conquer, these considerations concern to a large extent the efficient

combination of (possibly multi-threaded) Level 3 BLAS on the (multicore) CPU, and of Level 3 BLAS that has been optimized for the GPU. Indeed, Figure 2.1 showed that in the hardest cases, when deflation is not possible, the bulk of the computation lies in matrix-matrix multiplications to obtain eigenvectors from those of the two submatrices (see the last part of Step 3 in Section 2.1).

In order to achieve high performance, it is necessary to involve both the CPU and GPU in the computation, with the workload balanced according to communication costs and computational performance. One previous example of such an approach for GPU-acceleration was investigated in [18] in the context of the *LU* factorization.

As a starting point, one can compute the Level 3 BLAS matrix-matrix multiply (GEMM) performance ratio of multicore CPU *vs* GPU as an indicator of how to split between the CPU and the GPU. Subsequently, the ratio of the splitting can be adjusted to account for the data transfer time and other factors.

In the architecture used in the experiments shown in Section 4, a hybrid dual socket quad-core Intel Xeon 2.33 GHz with NVIDIA GTX 280 GPU, it turned out as optimal to off-load about 75% to the GPU and leave 25% to the CPU, corresponding to a performance ratio of 3 to 1. In general, the work-balance between CPU and GPU is an important parameter that is to be tuned, even automatically, to account for the heterogeneity. It is interesting that the work distribution could both be done on the scheduling level or, equivalently, in the mathematical formulation by allowing uneven partition of the tridiagonal matrix, see the remarks in Section 2.1.

It is possible to even further economize on the amount of data movement. Recall that one of the factors in the cost-dominating matrix-matrix product is generated from (2.6). Thus, instead of sending the eigenvectors of $D + \rho uu^T$ as a dense matrix, it is better to just send the data necessary for invoking (2.6) and perform the computations locally. In this context, note that the calculation of different eigenvectors of $D + \rho uu^T$ by (2.6) is data-parallel and thus well suited for the GPU.

3.3. Phase 3: eigenvector computation. From the algorithmic point of view, Phase 3 is the most straightforward. We have to perform the matrix-matrix multiplication between Q from Phase 1 and the matrix of eigenvectors of the tridiagonal matrix from Phase 2. When Phase 1 is executed as described in Section 3.1, all the computations of Phase 3 are Level 3 BLAS and can be efficiently applied both on the CPU and the GPU.

Since Q is stored as product of block Householder reflectors in WY form [4, 35], nothing has to be recomputed during the application of Q. The multiplication is performed simultaneously for different subsets of eigenvectors on GPU and CPU. Similar to Phase 2, the workload is balanced to account for the different computational capacities and communication costs.

4. Test Problems and Results. This Section shows performance data for all three phases of the eigensolver described in Section 3. The architecture used for this study is a dual socket quad-core Intel Xeon CPU E5410 @2.33 GHz with an NVIDIA GTX 280 GPU attached through a PCI-e x16 slot. The GTX280 GPU has 30 multiprocessors, each multiprocessor having eight SIMD functional units @1.33 GHz. The theoretical performance peaks in single precision are 936 GFlop/s for the GTX280 and 149 GFlop/s for the eight core CPU host; the theoretical bus bandwidth peaks are correspondingly 141.7 GB/s for the GPU and 10.64 GB/s for the CPU. Achievable bandwidth peaks (through the PCI-e x16) are 5.7 GB/s host to device and 5.5 GB/s device to host using pinned memory. The latency for transfers in either direction is about 11 microseconds. We use CPU BLAS and LAPACK from MKL 10.3

and GPU BLAS from CUBLAS 4.0 RC and MAGMA 1.0 RC4. All computations are performed in single precision. Achievable SGEMM performances using these libraries are 375 GFlop/s for the GPU and 128 GFlop/s for the CPU. The GPU driver used is 270.27 from NVIDIA.

The presentation of the following results is analogous to the layout of Section 3. The performance of the reduction to tridiagonal form is shown in Section 4.1; Section 4.2 gives details about the tridiagonal part, and Section 4.3 presents the transformation of the eigenvectors.

In order to interpret the results, it is important to keep in mind that the complexity of Phases 1 and 3 is determined by the dimension of the original dense matrix. The complexity of Phase 2 however, the tridiagonal problem, depends on both the matrix dimension and the numerical data of the tridiagonal at hand, see the remarks in Section 2.2. Further, performance depends on the type of the kernels used and the amount and frequency of data transfers between the CPU and the GPU. This is discussed separately below for each of the three phases.

4.1. Phase 1: reduction to tridiagonal form. As pointed out in the introductory remarks, the reduction of a dense matrix to tridiagonal form is oblivious to the numerical data. It thus suffices to study performance for just one class of sample matrices.

Figure 4.1 shows the performance of the dominating part, Phase 1 from Section 3.1, the reduction to tridiagonal form.

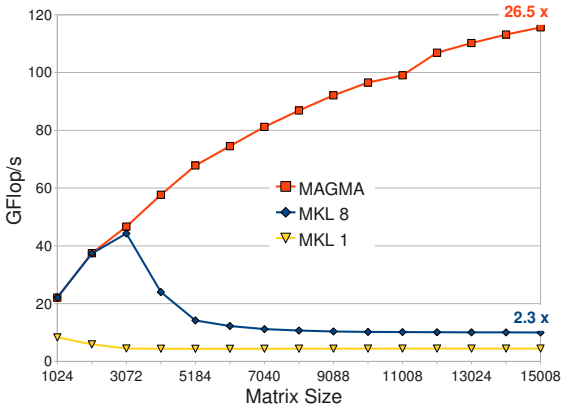

FIG. 4.1. *Performance of the reduction to tridiagonal form on a single and eight Intel Xeon 2.33 GHz cores using MKL, and a hybrid system of eight Intel Xeon cores and a GTX280 GPU. MAGMA denotes our algorithm.*

The gains of the reduction to tridiagonal form mirror those reported for reduction to Hessenberg form in [39]. The performance of up to about 120 GFlops demonstrates the usefulness of GPU acceleration of this phase.

The CPUs with MKL perform well as long as a matrix is small enough to fit into the fast local caches in the memory hierarchy. For larger problems however, we observe a performance drop on the CPUs that is to be expected for memory-bound computations. In contrast, the hybrid code can well amortize latency costs of CPU-GPU data transfers for larger matrices through available parallelism. This accounts for the greater performance seen on the larger problems.

The time for the CPU-GPU data transfers are included in the measurements. At the beginning of this phase the matrix is on the CPU memory. It is copied once to the GPU memory at the beginning of the computation. Further, throughput the computation panels (blocks of 32 columns below the diagonal) are transferred to the CPU and panel results of the same size are copied back to the GPU. This amounts to communicating about $2n^2/2$ elements between the CPU and GPU in $2n/32$ transfers. In addition, to process each column of a panel, data of size of the column is sent to the GPU, multiplied by the trailing matrix, and the result sent back to the CPU. This amounts to a total of $2n^2/2$ elements transferred in about $2n$ transactions. To summarize, the communication for this phase amounts to $3n^2$ elements transferred in $2n + n/16 + 1$ transactions.

4.2. Phase 2: Divide & Conquer for the tridiagonal eigenproblem. We first return to the benchmark problem in Figure 2.1 from Section 2.2. Toeplitz-like eigenvalue distributions represent one of the most difficult matrix classes for the Divide & Conquer algorithm [13, 28]. In fact, for these kind of problems, LAPACK's tridiagonal MRRR algorithm [14, 15], its biggest competitor, runs an order of magnitude faster, see [13, 28].

The difficulties of LAPACK's Divide & Conquer can be directly attributed to its $O(n^3)$ complexity. Figure 4.2 demonstrates this on a single core.

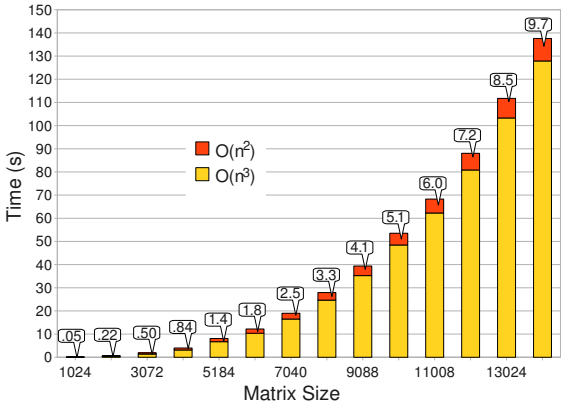

FIG. 4.2. *Sequential execution time for the Divide & Conquer using one single Intel Xeon 2.33 GHz core and MKL. Shown are the times for the $O(n^3)$ flops in matrix-matrix multiply (GEMM) and for the rest $O(n^2)$ flops (in the bubbles) for various matrix sizes.*

However, using the techniques outlined in Section 3.2, the algorithm can be accelerated substantially. The left part of Figure 4.3 shows the speedup of the parts with $O(n^3)$ complexity; the right part displays the total speedup of the overall tridiagonal problem.

The results show that the performance of the hybrid matrix-matrix multiply (GEMM) scales both with increasing the number of CPU cores and when adding the GPU. Operations are split in a ratio of one to three between the eight CPU cores and the GPU, as described in Section 3.2. The computations are overlapped with the transfers to and from the GPU. With the strong scaling of the $O(n^3)$ part, the smaller $O(n^2)$ part of the overall algorithm then becomes a bottleneck, resulting in a total speedup of 9.

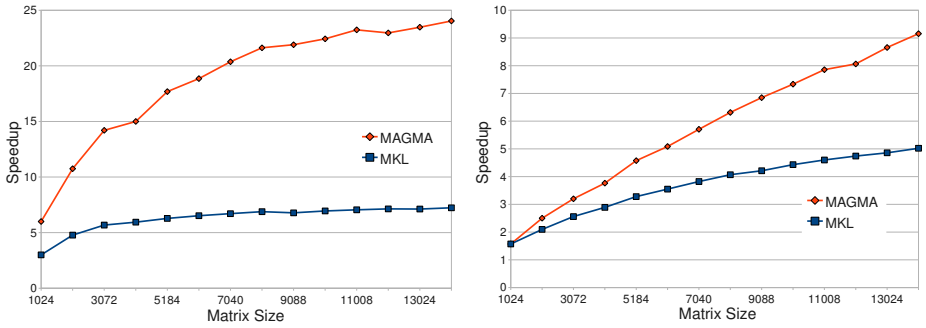

FIG. 4.3. *Speedup in the tridiagonal Divide & Conquer due to Level 3 BLAS (left) and the resulting speedup for the overall tridiagonal algorithm (right). The results are for two systems – a multicore system with eight Intel Xeon 2.33 GHz cores (using MKL), and a hybrid system of the same multicore but enhanced with a GTX280 GPU. MAGMA denotes our algorithm*

In order to understand what fluctuations in performance a user could encounter in practice on the tridiagonal part, we also ran our code for a variety of application test matrices from electronic structure calculations. As extremes we encountered on one end a matrix class from LAPW [33] which behaved just like the benchmark problem above. On the negative end, we found a matrix class from SIESTA [34] where only 50% performance gains were achievable. On matrices from the test collection [28] where some deflation happens, the tridiagonal Divide & Conquer algorithm tended to behave more like an $O(n^{2.8})$ rather than an $O(n^3)$ algorithm [13]. Thus, on average the practitioner can expect about 80% of the speedup reported for the model problem. A heuristic argument due to Cuppen [10] suggests that the less diagonally dominant a tridiagonal at hand, the closer it resembles the above benchmark problem. There seems no precise, formal proof of this assertion but it can well serve a practitioner to make a first educated guess.

4.3. Phase 3: eigenvector computation. Figure 4.4 shows the performance of matrix-matrix multiply needed for Phase 3 of the overall algorithm. Just like Phase 1, this operation is oblivious to the numerical data and one class of sample matrices suffices to show what performance can be expected.

Like in Phase 2, a good load balance ratio is about one to three between the eight CPU cores and the GPU. Besides participating in the update of the orthogonal matrix, one of the CPU cores is responsible for the triangular factors of the block Householder reflectors. In addition to the eigenvector matrix data, the CPU-GPU communications for this phase involve the columns of the block reflectors plus their associated triangular factors as computed in Phase 1.

5. Conclusive and perspective. Despite the importance of numerical eigensolvers for many applications and the great performance promised by GPU-accelerated multicore architectures, it is hard to keep up the pace with algorithmic developments. To the best of our knowledge, this paper describes the first dense eigensolver for such modern architectures in scientific literature.

We provided in-depth theoretical considerations for the algorithmic design of all three phases of the dense algorithm and studied performance on practical examples. In conclusion, since the performance of the tridiagonal part is matrix dependent, so is the performance of the overall dense algorithm. Realistically, on a test architecture

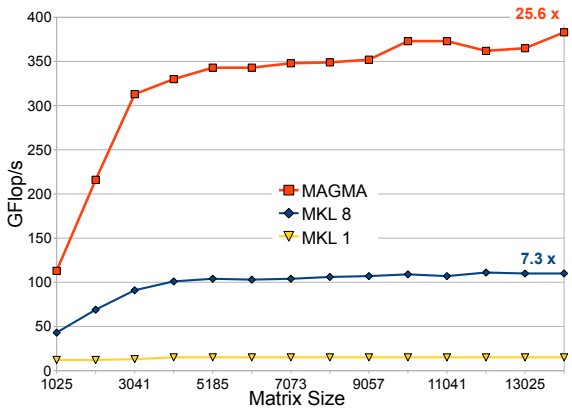

FIG. 4.4. *Performance of matrix-matrix multiply on a single and eight Intel Xeon 2.33 GHz cores using MKL, and a hybrid system of eight Intel Xeon cores and a GTX280 GPU. MAGMA denotes our algorithm.*

like the one used in this paper, a user can expect an overall speedup of about one order of magnitude over LAPACK in a practical application if the matrix is large enough and the bulk of computations arises from non-deflated eigenvalues.

With further optimized BLAS kernels such as [29] becoming available in MAGMA and CUBLAS, we plan to extend our work to other GPU architectures and other precisions. The use of multiple GPUs on a single node or in a distributed environment is another area of current research, leading for example to the development of hybrid distributed algorithms in the style of ScaLAPACK. With data distributed according to a ScaLAPACK block-cyclic layout, panels could be sent between CPU and processed by ScaLAPACK code that is enhanced with GPU computations for trailing matrices outside the current panel. In this regard, new features introduced in CUDA 4.0 do offer a unified view of CPU/GPU memory to simplify coding and mitigate the latency for CPU-GPU data transfers.

The current MAGMA 1.0 release [27] contains a basic version of the Divide & Conquer algorithm that does not yet implement the ideas outlined here. The algorithm described in this paper will be part of a future MAGMA release.

Acknowledgments. Implementation and computations were done at the Innovative Computing Laboratory at the University of Tennessee. The authors would like to thank the National Science Foundation, Microsoft Research, and NVIDIA for supporting this research effort. We are also grateful to M. Petschow for remarks on an earlier draft, and to the referees for their comments on how to improve the original submission.

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180:012037, 2009.

[2] 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, Philadelphia, 3. edition, 1999.

[3] P. Bientinesi, F. Igual, D. Kressner, and E. Quintana-Orti. Reduction to Condensed Forms

for Symmetric Eigenvalue Problems on Multi-core Architectures. In *Eighth International Conference on Parallel Processing and Applied Mathematics*, 2009.
[4] C. Bischof and C. van Loan. The *WY* Representation for Products of Householder Matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):2–13, 1987.
[5] C. H. Bischof, B. Lang, and X. Sun. A framework for symmetric band reduction. *ACM Trans. Math. Software*, 26(4):581–601, 2000.
[6] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox software for successive band reduction. *ACM Trans. Math. Software*, 26(4):602–616, 2000.
[7] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of DoD HPCMP Users Group Conference*, 1999.
[8] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perf. Comp. Appl.*, 14(3):189–204, 2000.
[9] J. Bunch, P. Nielsen, and D. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.
[10] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36(2):177–195, 1981.
[11] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.
[12] J. W. Demmel and J. J. Dongarra. LAPACK 2005 Prospectus: Reliable and Scalable Software for Linear Algebra Computations on High End Computers. Technical Report UT-CS-05-546, University of Tennessee, Knoxville, TN, USA, 2005. (LAPACK Working Note #164).
[13] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel. Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. *SIAM J. Sci. Comput.*, 30(3):1508–1526, 2008.
[14] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, University of California, Berkeley, California, 1997.
[15] I. S. Dhillon, B. N. Parlett, and C. Vömel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Software*, 32(4):533–560, 2006.
[16] J. J. Dongarra and D. C. Sorensen. A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, 1987.
[17] J. J. Dongarra, D. C. Sorensen, and S. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comp. Appl. Math.*, 27(1-2):215–227, 1989.
[18] M. Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *ACM International Conference Proceeding Series; Vol. 383. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51, 2009.
[19] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15:318–334, 1973.
[20] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16(1):172–191, 1995.
[21] B. Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14(6):1320–1338, 1993.
[22] B. Lang. Using Level 3 BLAS in Rotation-Based Algorithms. *SIAM J. Sci. Comput.*, 19(2):626–634, 1998.
[23] B. Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25:845–860, 1999.
[24] R.-C. Li. Solving secular equations stably and efficiently. Computer Science Dept. Technical Report CS-94-260, University of Tennessee, Knoxville, TN, November 1994. (LAPACK Working Note #89).
[25] K. Löwner. Über monotone Matrixfunctionen. *Math. Z.*, 38:177–216, 1934.
[26] H. Ltaief, S. Tomov, R. Nath, P. Du, , and J. Dongarra. A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators. Technical Report UT-CS-09-646, LAPACK Working Note 223, 2009.
[27] MAGMA 1.0 RC4. `http://icl.cs.utk.edu/magma/software/index.html`, 2011.
[28] O. A. Marques, C. Vömel, J. W. Demmel, and B. N. Parlett. Algorithm 880: A Testing Infrastructure for Symmetric Tridiagonal Eigensolvers. *ACM Trans. Math. Software*, 35(1):8:1–8:13, 2008.
[29] R. Nath, S. Tomov, and J. Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24:511–515, 2010.
[30] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM Press, Philadelphia, PA, 1998.
[31] J. Rutter. A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem. Computer Science Dept. Technical Report UCB/CSD 94/799, UC Berkeley, Berkeley, CA, 1994. (also LAPACK Working Note #69).
[32] Y. Saad, J. R. Chelikowsky, and S. M. Shontz. Numerical Methods for Electronic Structure

Calculations of Materials. *SIAM Review*, 52(1):3–54, 2010.

[33] D. Sanchez-Portal, P. Ordejon, Artacho E., and J. M. Soler. Density functional method for very large systems with LCAO basis sets. *Int. J. Quant. Chem.*, 65:453–461, 1997.

[34] D. Sanchez-Portal, P. Ordejon, Artacho E., and J. M. Soler. Density functional method for very large systems with LCAO basis sets. *Int. J. Quant. Chem.*, 65:453–461, 1997.

[35] R. Schreiber and C. Charles van Loan. A Storage-Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[36] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[37] D. C. Sorensen and P. T. P. Tang. On the Orthogonality of Eigenvectors computed by Divide-and-Conquer Techniques. *SIAM J. Numer. Anal.*, 28(6):1752–1775, 1991.

[38] F. Tisseur and J. Dongarra. A Parallel Divide and Conquer algorithm for the Symmetric Eigenvalue Problem. *SIAM J. Sci. Comput.*, 6(20):2223–2236, 1999.

[39] S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report UT-CS-09-642, LAPACK Working Note 219, 2009.

[40] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[41] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. Technical report, University of Tennessee, Knoxville, TN, USA, 2009.

[42] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. Technical Report UT-CS-09-649, LAPACK Working Note 225, 2010.

[43] C. Vömel. ScaLAPACK's MRRR algorithm. *ACM Trans. Math. Software*, 37(1):1:1–1:35, 2010.