A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks

Azzam Haidar*, Stanimire Tomov*, and Jack Dongarra*†‡
*Electrical Engineering and Computer Science, University of Tennessee, Knoxville, Tennessee, U.S.A.
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA
‡School of Mathematics & School of Computer Science, University of Manchester, United Kingdom
{haidar, tomov, dongarra}@eecs.utk.edu

Raffaele Solcà§ and Thomas Schulthess§¶
§Institut for Theoretical Physics, ETH Zurich, Switzerland
¶Swiss National Supercomputer Center, Lugano, Switzerland
rasolca@itp.phys.ethz.ch, schultho@ethz.ch

***Abstract*—The adoption of hybrid GPU-CPU nodes in traditional supercomputing platforms such as the Cray-XK6 opens acceleration opportunities for electronic structure calculations in materials science and chemistry applications, where medium-sized generalized eigenvalue problems must be solved many times. These eigenvalue problems are too small to effectively solve on distributed systems, but can benefit from the massive computing power concentrated on a single node, hybrid CPU-GPU system. However, hybrid systems call for the development of new algorithms that efficiently exploit heterogeneity and massive parallelism of not just GPUs, but of multi/many-core CPUs as well. Addressing these demands, we developed a generalized eigensolver featuring novel algorithms of increased computational intensity (compared to the standard algorithms), decomposition of the computation into fine grained memory aware tasks, and their hybrid execution. The resulting eigensolvers are state-of-the-art in HPC, significantly outperforming existing libraries. We describe the algorithm and analyze its performance impact on applications of interest when different fractions of eigenvectors are needed by the host electronic structure code.**

I. Introduction

Scientific computing applications, ranging from computing frequencies that will propagate through a medium, to earthquake response of a bridge, or energy levels of electrons in nanostructure materials, require the solution of eigenvalue problems. There are many ways to mathematically formulate and numerically solve these problems [1]. In this work we are interested in dense eigensolvers for the generalized Hermitian-definite problems of the form

$$Ax = \lambda Bx, \tag{1}$$

where A is a Hermitian dense matrix and B is Hermitian positive definite. These solvers are needed for electronic structure calculations in materials science and chemistry [2], [3], [4], where (1) must be solved many times in the context of a parallel code. Modest matrix dimensions of a few thousand to ten or twenty thousands seem to pose a challenge for most practical purposes, because the matrices are too small for the eigensolver to effectively scale on a large distributed memory system. The alternative, which we target in this work, is to develop algorithms that will effectively scale on massively parallel shared memory systems, and in particular hybrid multicore CPU-GPU systems.

Solving (1) requires the development of a number of routines. First, the matrix B is decomposed using a Cholesky factorization $B = LL^H$, next the L factors are used to transform (1) to a standard eigenvalue problem ($\tilde{A} = L^{-1}AL^{-H}$, $\tilde{A}z = \lambda z$). Finally, the eigenvectors of $\tilde{A}$ must be transformed back to $x = L^{-H}z$. The standard eigenvalue problem by itself can also be divided into subproblems - first tridiagonalize $\tilde{A}$, next solve the tridiagonal eigenproblem, and finally transform back the eigenvectors.

Although all of these routines of interest are available in a number of standard libraries, hardware changes have motivated their redesign (see Section II) to more efficiently use the underlying hardware. In particular, hybrid architectures based on GPUs and multicore CPUs call for the development of new algorithms that can efficiently exploit the heterogeneity and the massive parallelism of not just the GPUs, but of the multi/many-core CPUs as well. Hybrid GPU-CPU nodes are already widely adopted in traditional supercomputing platforms such as the Cray XK6. This has opened many new acceleration opportunities in electronic structure calculations for materials science and chemistry applications, where the problems are too small to effectively scale on distributed memory systems with standard libraries like ScaLAPACK, but can benefit from the massive compute power concentrated on a single node of a hybrid GPU-CPU system.

The work that we present in this paper is concentrated on addressing the high demand for accelerated eigensolvers on a single node of a hybrid GPU-CPU system, as outlined above. To address the challenges related to the new architectures, we

designed a generalized eigensolver featuring the following:

- Novel algorithms of increased computational intensity (compared to the standard algorithms) based on the idea of successive band reduction factorizations;
- An innovative decomposition of the computation into fine grained memory aware tasks;
- An efficient scheduling mechanism for tasks' execution over the hybrid CPU-GPU system (aimed at reducing CPU-GPU data movements, overlapping CPU-GPU communications with computation, and mapping algorithmic task requirements to architectural strengths of the system's hardware components).

The resulting eigensolvers are breakthrough state-of-the-art eigensolvers in high-performance computing, providing support for hybrid systems, and significantly exceeding the performance of currently available solutions.

The rest of the paper is organized as follows. In Section II we present related work. Section III summarizes this work's contributions. Next, Sections IV and V describe, correspondingly, the new generalized and standard eigensolver algorithms, giving detail on the overall design, as well as the design of their main components. Each of these two sections has a numerical results subsection. Finally, Section VII gives conclusions and future work directions.

II. Related Work

The LAPACK [5] and ScaLAPACK [6] libraries include a set of conventional eigensolver routines for shared-memory and distributed-memory systems, respectively. Vendor libraries like MKL [7] and ACML [8] provide highly tuned implementations for these libraries correspondingly for Intel and AMD processors.

Recent algorithmic work on eigenvalue problems has concentrated on accelerating the reduction to tridiagonal form, which is the most time consuming part of the algorithm (see the results sections). The standard approach from LAPACK [9] is to use a "single phase" (also referred to as a "one-stage") reduction. Alternatives are two- (or more) stage approaches where the matrices are first reduced to band form, and second, to the final tridiagonal form.

One of the first uses of a two-step reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [10], where a multi-stage method reduced a matrix to tridiagonal, bidiagonal, and Hessenberg forms [11]. With this approach, it was possible to recast the expensive memory-bound operations that occur during the panel factorization into a compute-bound procedure.

Consequently, a framework called Successive Band Reductions (SBR) was created [12], that integrated some of the multi-stage work. The SBR toolbox applies two-sided orthogonal transformations based on Householder reflectors and successively reduces the matrix bandwidth size until a suitable width is reached. The off-diagonal elements are then annihilated column-wise, which produces large fill-in blocks (or "bulges") that need to be chased down toward the bottom right corner of the matrix. The bulge chasing procedure may result in a substantial increase in the floating point operation count when compared with the standard single-phase approach from LAPACK. If eigenvectors are required in addition to eigenvalues, then the transformations may be efficiently accumulated using Level 3 BLAS operations to generate these vectors. SBR relies heavily on multithreaded BLAS that are optimized to achieve satisfactory parallel performance. However, such a parallelization paradigm incurs substantial overheads on multicore processors [13] as it fits the Bulk Synchronous Parallelism (BSP) model [14]. Communication bounds for such two-sided reductions have been established under the Communication Avoiding framework [15]. A multi-stage approach has also been applied to the Hessenberg reduction [16] as well as the QZ algorithm [17] for the generalized non-symmetric eigenvalue problem. These approaches, in contrast to our own, are not for hybrid GPU-CPU systems.

Tile algorithms have recently seen a rekindled interest when applied to the two-stage tridiagonal [13] and bidiagonal reductions [18]. Using high performance kernels combined with a data translation layer to execute on top of the tile data layout format, both implementations achieve a substantial improvement compared to the equivalent routines from the state-of-the-art numerical libraries. The off-diagonal elements are annihilated column-wise during the bulge chasing procedure, which engenders significant extra-flops due to the size of the bulges introduced. An element-wise annihilation has then been implemented based on cache-aware kernels, in the context of the two-stage tridiagonal reduction [19]. Using the coalescing task technique [19], the performance achieved is by far greater than any other available implementations.

With the emergence of GPUs, memory-bound and compute-bound operations can be accelerated by an order of magnitude or more. Tomov et al. [20] presented novel hybrid reduction algorithms for the two-sided factorizations, which in addition to the GPU's high-performance on compute-bound kernels, take advantage of the GPU's high bandwidth by offloading the expensive Level 2 BLAS operations of the panel factorizations to the GPU. Bientinesi et al. [21] accelerated the two-stage approach of the SBR toolbox by offloading the compute-bound kernels of the first stage to the GPU. The computation of the second stage (reduction to tridiagonal form), though, still remains on the host.

Vomel et al. [22] extended the tridiagonalization approach in [20] to the symmetric standard eigenvalue problem.

A recent distributed-memory eigensolver library, developed for electronic structure codes, is ELPA [3]. ELPA is based on ScaLAPACK and does not support GPUs. It includes new one-stage and two-stages tridiagonalizations, the corresponding eigenvectors transformation, and a modified divide and conquer routine that can compute the entire eigenspace or a part of it.

The algorithms presented in this article are implemented and included in MAGMA [23], a collection of next generation LAPACK/ScaLAPACK-compliant linear algebra libraries for hybrid GPU-CPU architectures.

III. Main Contributions

We developed new algorithms for the generalized eigenvalue problem as well as their highly efficient implementations for hybrid CPU-GPU systems. Some components of the solver were leveraged (when sufficiently efficient) or extended (when possible) from the MAGMA, PLASMA [24], LAPACK, and vendor-optimized BLAS libraries, which will be specified through the presentation when applicable.

The main contributions of this work are the algorithmic and computing innovations to solve (1) in an entirely compute-bound computation that is task-parallelized and efficiently executed on hybrid CPU-GPU systems. The design highlights are described as follows.

A. Fine grained memory aware tasks

Our approach to parallelism and efficient hardware use relies on splitting the computation into tasks (of certain granularity) as well as the task's proper execution over the available hardware components. Our particular choices, as described in the algorithms of interest, are an essential component for obtaining high performance.

B. Hybrid CPU-GPU execution/scheduling

A hybrid CPU-GPU task scheduling is another indispensable ingredient for obtaining high-performance. Indeed, task scheduling may require CPU-GPU data movements – a slow operation that can be minimized and possibly overlaped with CPU and GPU computations. Our scheduling mechanisms, as described in the algorithms of interest, reduce CPU-GPU communication, overlap CPU and GPU computations, and moreover, map computational tasks to the strengths of the heterogeneous hardware components of the system.

C. Increased computational intensity

Finally, we extended the ideas of the two-stage reductions to design a new hybrid tridiagonalization. The new algorithm is compute-bound, vs. the memory-bound conventional algorithm, and efficiently uses both the multicore host and the GPU.

IV. Generalized Eigensolver for hybrid CPU-GPU architectures

The first step in solving (1), as previously described, is to compute the Cholesky factorization of $B = LL^H$. Then, the generalized eigenvalue problem is transformed to a standard eigenvalue problem according to:

$$\tilde{A}\mathbf{z} = \lambda\mathbf{z}, \tag{2}$$

where

$$\tilde{A} = L^{-1}AL^{-H}. \tag{3}$$

This transformation step can be computed by LAPACK's xHEGST routine. Our hybrid design and implementation of this transformation step is described below in Section IV-A. After solving the standard Hermitian eigenproblem (2), as described in Section V, the eigenvectors X of the generalized problem (1) are computed by back-solving $L^H X = Z$. This operation can be performed easily on the GPU by applying L^{-H} to Z using the xTRSM routine from BLAS (CUBLAS or MAGMA).

A. Hybrid algorithm transforming the generalized to standard eigenvalue problem

We developed a "`magma_xhegst`" hybrid routine to perform (3). To describe it, we start by outlining LAPACK's xHEGST in Algorithm 1. Here the matrices A and B are correspondingly split into nt-by-nt submatrices $A_{i,j}$ and $B_{i,j}$ of size $nb \times nb$.

Algorithm 1 The LAPACK zhegst routine.

1: **for** i = 1, 2, ... nt **do**
2: xHEGS2($A_{i,i}$, $B_{i,i}$)
3: $A_{i+1:nt,i} = A_{i+1:nt,i}B_{i,i}^{-1}$ (xTRSM)
4: $A_{i+1:nt,i} \mathrel{-}= \frac{1}{2}B_{i+1:nt,i}A_{i,i}$ (xHEMM)
5: $A_{i+1:nt,i+1:nt} \mathrel{-}=$ $A_{i+1:nt,i}B_{i+1:nt,i}^H + B_{i+1:nt,i}A_{i+1:nt,i}^H$ (xHER2K)
6: $A_{i+1:nt,i} \mathrel{-}= \frac{1}{2}B_{i+1:nt,i}A_{i,i}$ (xHEMM)
7: $A_{i+1:nt,i} = B_{i+1:nt,i+1:nt}^{-1}A_{i+1:nt,i}$ (xTRSM)
8: **end for**

Algorithm 2 The magma_zhegst routine.

1: copy $B \rightarrow dB$
2: copy $A \rightarrow dA$
3: **for** i = 1, 2, ... nt **do**
4: xHEGS2($A_{i,i}$, $B_{i,i}$)
5: overlapped copy $A_{i,i} \rightarrow dA_{i,i}$
6: $dA_{i+1:nt,i} = dA_{i+1:nt,i}dB_{i,i}^{-1}$
7: $dA_{i+1:nt,i} \mathrel{-}= \frac{1}{2}dB_{i+1:nt,i}dA_{i,i}$
8: $dA_{i+1:nt,i+1:nt} \mathrel{-}=$ $dA_{i+1:nt,i}dB_{i+1:nt,i}^H + dB_{i+1:nt,i}dA_{i+1:nt,i}^H$
9: overlapped copy $dA_{i+1,i+1} \rightarrow A_{i+1,i+1}$
10: $dA_{i+1:nt,i} \mathrel{-}= \frac{1}{2}dB_{i+1:nt,i}dA_{i,i}$
11: $dA_{i+1:nt,i} = dB_{i+1:nt,i+1:nt}^{-1}dA_{i+1:nt,i}$
12: **end for**
13: copy $dA \rightarrow A$

The hybrid version of this routine uses the GPU to perform the Level 3 BLAS operations. The resulting code is illustrated in Algorithm 2, where dA and dB are the corresponding A and B matrices stored into the device (GPU) memory. The CPUs are responsible for computing the $A_{i,i}$ blocks using xHEGS2 (line 4 of Algorithm 2) while the GPU is responsible for updating the rest of the matrix. To optimize the code, we split this operation into two phases: (a) partially compute a panel $dA_{i+1:nt,i}$ (lines 6 and 7), then use it to update the trailing matrix $dA_{i+1:nt,i+1:nt}$ (line 8) by a xHER2K, and (b) continue the computation of the panel $dA_{i+1:nt,i}$ (lines 10 and 11). Since the result of phase (b) is not needed by any of the subsequent steps $i+1, i+2, \ldots$, once phase (a)

is finished we copy the updated $dA_{i+1,i+1}$ to the CPU to allow its computation by xHEGS2 while the GPU performs phase (b). This provides an overlap between the CPU and GPU computation, and allows for hideing the CPU-GPU communication.

V. Hybrid GPU-CPU standard eigensolver

To solve a Hermitian (symmetric) eigenproblem of the form $\tilde{A}\mathbf{z} = \lambda\mathbf{z}$, finding its eigenvalues Λ and eigenvectors Z so that $\tilde{A} = Z\Lambda Z^H$, where ^{H} denotes conjugate-transpose, the standard algorithm follows three steps [25], [26], [27]. First, reduce the matrix to tridiagonal form, called the "reduction phase", using an orthogonal transformation Q such that $\tilde{A} = QTQ^H$, where T is a tridiagonal matrix. Note that the eigenvalues of the tridiagonal matrix are the same as those of the original matrix. Second, compute eigenpairs (Λ, E) of the tridiagonal matrix (called the "solution phase"). Third, back transform eigenvectors of the tridiagonal matrix to eigenvectors of the original matrix, $Z = QE$, called the "back transformation phase".

Due to the computational complexity and data access patterns, it is well known that the reduction phase is considerably more time consuming than the other two phases combined. Several approaches exist to compute the tridiagonalization of a dense matrix.

A. Standard One-Stage Tridiagonalization

There are two algorithmic approaches — the standard one-stage approach from LAPACK [28], where block Householder transformations are used to directly reduce the dense matrix to tridiagonal form, and a second, two-stage (or more) approach [12], [29], where block Householder transformations are used to first reduce the matrix to band form, and a second, bulge chasing stage is used to reduce the band matrix to tridiagonal [19].

The one-stage tridiagonalization is characterized by iterating two computational phases: panel factorization and update of the trailing submatrix. The parallelism in this approach resides primarily within the trailing submatrix update phase. Synchronization points are required between each panel factorization and trailing submatrix update, preventing overlap of the two computational phases. The panel factorization step is actually the critical phase because it relies on symmetric matrix-vector multiplications involving the trailing submatrix (50% of the flops). For that this approach suffers from a lack of efficiency and is well known to be memory bound. Standard software for reducing a symmetric dense matrix to tridiagonal form is available in LAPACK [5] and in MAGMA [20] through the xHETRD routine.

B. Two-Stage Tridiagonalization

The two-stage approach permits us to cast expensive memory operations occurring during the panel factorization into faster, compute intensive ones. This is done by splitting the original one-stage approach into a compute-intensive phase (first stage) and a memory-bound phase (second stage or bulge chasing stage). The first stage reduces the original Hermetian (symmetric) dense matrix to a Hermetian band form. The second stage applies the bulge chasing procedure, where all the extra entries are annihilated column-wise.

1) First stage – the reduction to band: The first stage applies a sequence of block Householder transformations to reduce a Hermetian (symmetric) dense matrix to a Hermetian band matrix. This stage uses compute-intensive matrix-matrix multiplications and eliminates the memory-bound matrix-vector products from the one-stage panel factorization. The reduction to band has been shown to have a good data access pattern and large portion of Level 3 BLAS operations [12], [30], [31]. A hybrid version can be made very efficient as well since the GPU is efficient on the Level 3 BLAS, and moreover, CPU-GPU communications can be overlapped with computation. The technique used here is similar to the one developed for multicore [19]. However, instead of tile techniques, we use block techniques since they are better suited for GPU operations. Given a dense $n \times n$ symmetric matrix $\tilde{A}$, the matrix is divided into $nt = n/nb$ block-columns of size nb. The reduction to band proceeds panel by panel, where for each panel a QR decomposition is performed to generate the Householder reflectors V required to zero out the elements below the nb^{th} subdiagonal, as shown in Figure 1). The panel factorization is followed by applying these reflectors from the left and the right to the trailing symmetric matrix (the red triangular of Figure 1), according to

$$\tilde{A} = \tilde{A} - WV^H - VW^H, \tag{4}$$

where V and T define the block of Householder reflectors and W is computed as

$$\begin{aligned} W &= X - \tfrac{1}{2}VT^HV^HX, \text{ where} \\ X &= \tilde{A}VT. \end{aligned} \tag{5}$$

The hybrid CPU-GPU algorithm is illustrated in Figure 1. Since the matrix is symmetric, only the lower part is referenced, and the upper part (gray color) stays untouched. We first run the QR decomposition (xGEQRT kernel) of a panel on the CPUs. Once the panel factorization of step i is finished,

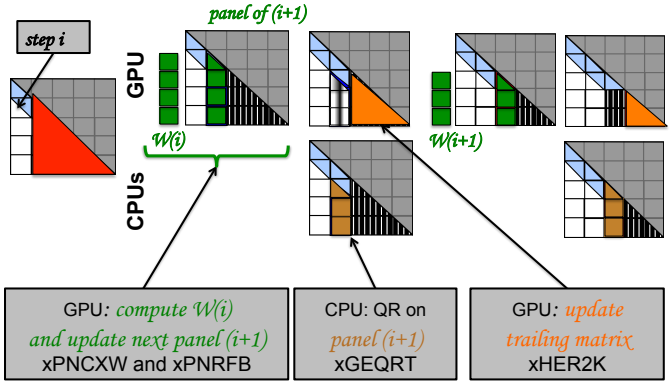


Fig. 1. Description of the reduction to band form, stage 1.

we compute W on the GPU, as defined by equation (5). This kernel is called xPNCXW. Its computation involves a matrix-matrix multiplication (xGEMM) to compute VT, then a Hermitian matrix-matrix multiplication to compute $X = AVT$ (xHEMM), which is the dominant cost of computing W, and finally another inexpensive xGEMM. Once W is computed, the trailing matrix update (applying transformations on the left and right) defined by equation (4) can be performed. However, to allow overlap of CPU and GPU computation, this computation is split into two. The first part, named xPNRFB, applies on the GPU the left and the right updates to the next panel (i.e., panel of step $i+1$, colored in green in Figure 1). Next, the remainder of the trailing submatrix (represented by the orange triangle in Figure 1) is updated using a xHER2K. While the xHER2K is executing, the panel for step $i+1$ is sent to the CPUs, the CPUs perform the next panel factorization (xGEQRT), and the resulting Householder reflectors V_{i+1} are sent back to the GPU. In this way, the factorization of panels $i = 2, \ldots, nt$ and the associated communications are hidden by overlapping them with the GPU computations, as demonstrated in Figure 2. Figure 2 shows a snapshot of the execution trace of the reduction to band form, where we can easily identify the overlap between CPU and GPU computation. Note that the high-performance GPU is continuously busy, either computing W or updating the trailing matrix (xHER2K), while the lower performance CPUs wait for the GPU as necessary.

The reduction to symmetric band tridiagonal form can be easily derived for the upper case. All the operations will then be based on the LQ factorization numerical kernels, as described in Ltaief et al. [32]. The first stage algorithm is a compute intensive phase, based on the GPU kernel which is a Level 3 BLAS computation. Therefore, it is critical to supply a large enough block-panel size (e.g., $nb \geq 64$) so that the Level 3 BLAS can run close to the theoretical peak performance of the GPU.

2) Second stage – the reduction from band to tridiagonal using fine grained memory aware tasks: The band form is further reduced to the final condensed form using the bulge chasing technique. This procedure annihilates the extra off-diagonal elements by chasing the created fill-in elements

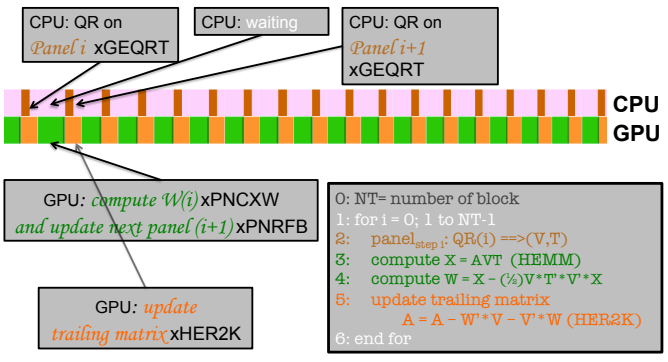
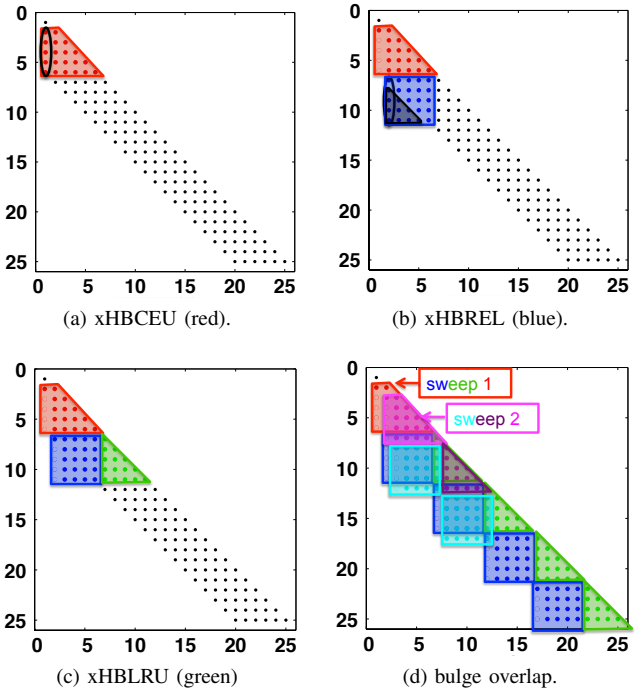


Fig. 2. Execution trace of reduction to band form.

down to the bottom right side of the matrix using successive orthogonal transformations at each sweep. This stage involves memory-bound operations and requires the band matrix to be accessed from multiple disjoint locations. In other words, there is an accumulation of substantial latency overhead each time different portions of the matrix are loaded into cache memory, which is not compensated for by the low execution rate of the actual computations (the so-called surface-to-volume effect). To overcome these critical limitations, we developed a bulge chasing algorithm, very similar to the bulge chasing techniques developed in [19], but we differ from it in using a column-wise elimination instead of an element-wise elimination which allows us to have better locality for computing or applying the orthogonal matrix resulting from this phase. Our bulge chasing



(a) xHBCEU (red). (b) xHBREL (blue). (c) xHBLRU (green) (d) bulge overlap.

Fig. 3. Kernel execution of the TRD algorithm during the second stage.

algorithm is based on three new kernels that are designed to considerably enhance the data locality of the computation. The first one is the xHBCEU kernel. This kernel triggers the beginning of each sweep by annihilating, using the xLARFG function, the extra non-zero entries within a single column, as shown in Figure 3a. It then applies the computed elementary Householder reflector from the left and the right to the corresponding symmetric data block (red triangle) loaded into the cache memory. The second kernel, xHBREL, continues the application from the right derived from the previous kernel, either xHBCEU or xHBLRU. This subsequently generates triangular bulges as shown in Figure 3b (the black triangular), which must be annihilated by an appropriate technique in order to eventually avoid the excessive growth of the fill-in structure. A classical implementation will eliminate the whole triangular bulge. However, as an appropriate study of the bulge chasing

procedure, let us remark that, the elimination of the column $i+1$ (the sweep $i+1$), at the next step, creates a triangular bulge which will overlap this one by one column shift to the right and one row to the bottom, as shown in Figure 3d where the reader can see that the lower triangular portion of the cyan block (the bulge created by sweep $i+1$) overlaps with the lower triangular portion of the blue block (corresponding to the bulges created by the previous sweep i). As a result, we can reduce the computational cost and instead of eliminating the whole triangular bulge created for sweep i, we only eliminate the non-overlapped region of it: its first column. The remaining columns can be delayed to the upcoming annihilation sweeps. In this way, we can avoid the growth of the bulges and reduce the extra cost accrued when the whole bulge is eliminated. Moreover, we designed a cache friendly kernel that takes advantage of the fact that the created bulge (the black block) remains in the cache and therefore it directly eliminates its first column and applies the corresponding left update to the remaining column of the blue block. Talking about the third kernel, xHBLRU, it loads the next block (the green block of Figure 3c) and it applies the necessary left updates derived from the previous kernel (xHBREL). Since, the green block is remaining in cache, hence the kernel proceeds with the application from the right to the symmetric portion.

Accordingly, the annihilation of each sweep can be described as, one call to the first kernel followed by a repetitive calls to a cycle of the second and the third kernels.

C. Modified Divide and Conquer

The standard Divide and Conquer algorithms compute all the eigenvalues and eigenvectors of a real tridiagonal matrix. A detailed description of the algorithm can be found in [33]. To illustrate how our GPU implementation is designed, let us describe it for two subproblems. Let the matrix T of size n be split into two subproblems, T_1 of size n_1 and T_2 of size $n_2 = n - n_1$, as described in (6). Let the eigensolution of those two sons be given by $T_1 = \tilde{E}_1 \tilde{\Lambda}_1 \tilde{E_1^T}$ and $T_2 = \tilde{E}_2 \tilde{\Lambda}_2 \tilde{E_2^T}$, where $(\tilde{\Lambda}_i, \tilde{E}_i)$, $i = 1, 2$ are the eigenvalues and eigenvectors pair of T_i. Assuming that $(\tilde{E}_0, \tilde{\Lambda}_0)$ are the eigenpairs solution of the system inside the bracket of (7) (that we call it the rank-one modified system M), then $\Lambda = \tilde{\Lambda}_0$ and $E = \tilde{E}_i \tilde{E}_0$ are the eigenpairs of T.

$$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T \tag{6}$$

$$= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left\{ \begin{pmatrix} \tilde{\Lambda}_1 & 0 \\ 0 & \tilde{\Lambda}_2 \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T \right\} \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T \tag{7}$$

$$= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left(\tilde{E}_0 \tilde{\Lambda}_0 {\tilde{E}_0}^T \right) \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T = E \Lambda E^T \tag{8}$$

To find the eigensolution of each rank-one modified system M requires solving its secular equation. This is a memory bound process computed by the xLAED4 routine that requires only $\mathcal{O}(n^2)$ operations, and can be easily parallelized over the different CPU cores. So in our implementation we keep this computation on the CPUs side, while the GPUs perform the multiplication of the intermediate eigenvector matrices $\tilde{E}_i$, which can requires upto n^3 flop.

D. The application of the orthogonal matrices Q_1 and Q_2

In this section, we discuss the application of the Householder reflectors generated from the two stages of the reduction to tridiagonal. The first stage reduces the original Hermitian matrix $\tilde{A}$ to a band matrix by applying a two-sided transformation to $\tilde{A}$ such that $\tilde{A} = Q_1 S Q_1^H$. Similarly, the second stage (bulge chasing) reduces the band matrix S to tridiagonal by applying the transformation from both the left and the right side to S such that $S = Q_2 T Q_2^H$. Thus, when the eigenvectors matrix Z of $\tilde{A}$ are requested, the eigenvectors matrix E resulting from the eigensolver needs to be updated from the left by the Householder reflectors generated during the reduction phase, according to

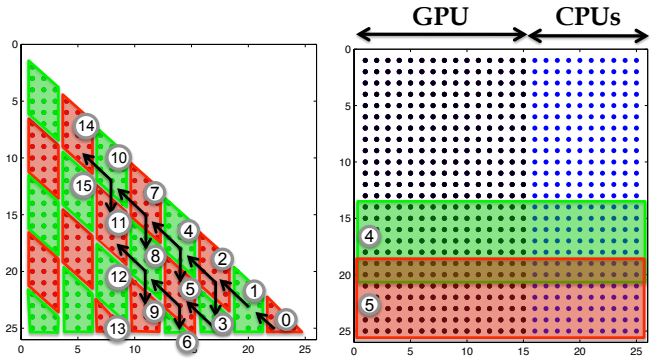
$$Z = Q_1 Q_2 E = (I - V_1 t_1 V_1^H)(I - V_2 t_2 V_2^H) E, \tag{9}$$

where (V_1, t_1) and (V_2, t_2) represent the Householder reflectors generated during the reduction stages one and two, respectively.

In our implementation, to obtain the complete eigenvectors of the matrix $\tilde{A}$, the matrix Z is updated by the V_2 reflectors, and the resulting matrix is updated by the V_1 reflectors. The application of the V_2 reflectors is not as simple as the application of V_1, and requires special attention. We represent the V_2 in Figure 4a. Note that these reflectors represent the annihilation of the band matrix, and thus each is of length nb, where nb is the bandwidth size. A naive implementation would take each reflector and apply it to the matrix E. Such an implementation is memory bound, relying on BLAS 2 operations and thus gives poor performance. However, if we want to group them to take advantage of the efficiency of BLAS 3 operations, we must pay attention to the overlap between them and that their application must follow the specific dependency order of the bulge chasing procedure in which they were created. Let's note that for sweep i (e.g., the column at position S(i,i):S(i,i+nb)), its annihilation creates a set of k Householder reflectors v_i^k, each of length nb represented in column i of the matrix V_2 depicted in Figure 4a. We can group the reflectors v_i^k from sweep i with those from sweep i+1, i+2,..., $i+l$ to apply them together using a blocked technique according to the diamond shape region as defined in Figure 4a. While each of those diamonds is considered as one block, their application needs to follow the chasing dependency order. For example, applying the green diamond 4 and the red diamond 5 of the V_2's in Figure 4a modifies the green block row 4 and the red block row 5, respectively, of the eigenvector matrix E drawn in Figure 4b, where we can easily observe the overlapped region. According to the chasing order, block 4 needs to be applied before block 5. We have drawn a sample of those dependencies by the arrows in Figure 4a. We designed our parallelism based on the matrix E, where we split E by block column over both

the CPUs and the GPU as shown in Figure 4b, where we can apply each diamond independently to each portion of E.

The application of V_1 to the resulting matrix from above, $G = (I - V_2 T_2 V_2^T)E$, can be done easily. First, there is no overlap between the different V_1's. Second, their application is computing intensive and involves efficient BLAS 3 kernels and it is done by using the GPU function "`magma_xunmtr`", which is the GPU implementation of the standard LAPACK function (xUNMTR).



(a) blocking for V_2 (b) eigenvectors matrix

Fig. 4. Blocking technique to apply the Householder reflectors V_2 with a hybrid implementation on GPU and CPU.

E. Parallel implementation and task scheduling

Restructuring linear algebra algorithms as a sequence of tasks that operate on blocks of data can remove certain fork-join synchronizations, and moreover, can facilitate a design that overlaps the CPUs and GPU computations. A schematic trace representation of the execution of our algorithm is depicted in Figure 5. The first stage, the reduction from dense to band, is done on both CPUs and GPU, where the CPUs compute the panel factorization while the GPU computes the update. The implementation results show that the CPUs' computation is mostly overlapped by the GPU operation. In contrast to the kernels of the first stage, the kernels of the second stage are memory-bound and rely on Level 2 BLAS operations involving small matrices. For that, the second stage is fully scheduled on the CPUs using a static runtime environment that we developed. The second stage is the most challenging when it comes to tracking data dependencies. The annihilation of the subsequent sweeps generates tasks, which partially overlap data from tasks from previous sweeps (see Figure 3d), and thus generating complex, inherent for the algorithm, data dependencies that are challenging to schedule. We have used the data dependency layer (DTL) and the function dependencies proposed by [13], [19] to handle those data dependencies and to provide crucial information to the runtime to achieve the correct scheduling. The first stage is, to an extent, oblivious to data locality, while for the second stage (the bulge chasing), and for the application of the Householder reflectors V_2 resulting from it, the data locality is of the utmost importance. The performance of the latter on the CPUs is guided by the memory bus speed, the scheduling sequence, and the cache memory sizes, whereas on the GPU, performance is dependent on the block size of the operations performed.

On CPUs: Although we optimized the CPU kernels for cache

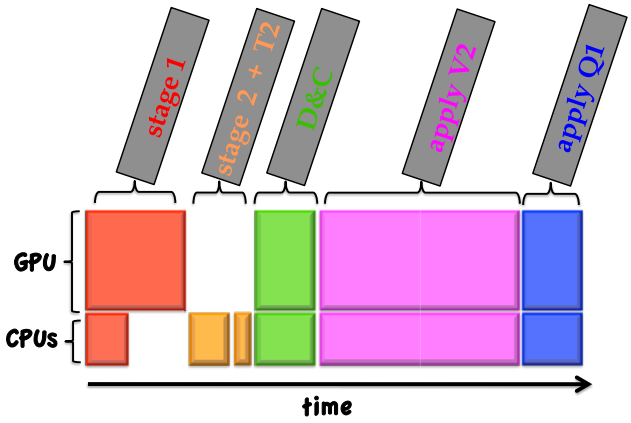


Fig. 5. Schematic trace execution of our hybrid eigensolver

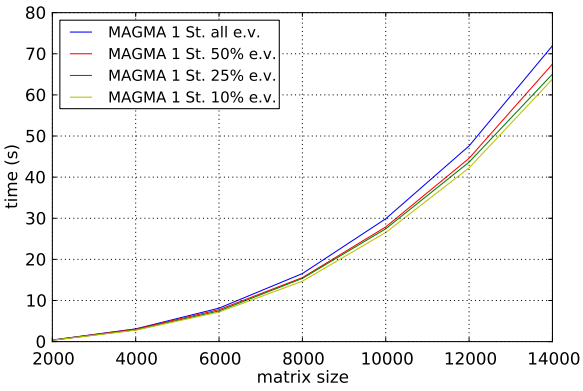
reuse, a single kernel call cannot, on its own, take advantage of running in-cache. Once the data is fetched into the high level caches and the registers, it needs to stay there and be reused between computational tasks as much as possible. Our idea to improve the locality, and hence the cache reuse, is to store the matrix in 1-D block cyclic fashion, and to aggregate computational tasks together in an appropriate manner. The data storage allows each CPU to work on its first block, then to move to its second block and so on, increasing the cache reuse. By aggregating tasks together into groups, we ensure that the data will be reused among the various tasks belonging to a certain group. As a consequence, operations that are supposedly memory-bound, increase their ratio of computation to off-chip communication and become compute-bound, which renders them amenable to efficient execution on multicore architectures.

On GPU: If the block size chosen during the computation of the first stage is too large, the application of Q_2 resulting from the second stage may encounter difficulties on the overlapped computation because having a large amount of operations involving triangular blocks would reduce the efficiency of their GPU execution. The challenge is the following: on the one hand, the block size needs to be large enough to extract high performance from the GPU, and on the other hand, it must be small enough to minimize the use of low performance kernels on the GPU (TRMM). This trade-off between the block size and the kernel performance has been tackled by using two nested levels of blocking where the initial block size, which is the same as the band reduction size, is chosen as the minimum size that gives good performance for the first stage, while a new, smaller blocking is defined for the computation of Q_2 to avoid the use of a low performance GPU kernel.

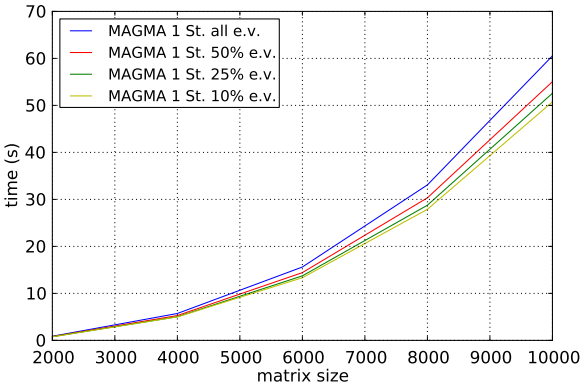
VI. Environment setup

Heterogeneous CPU-GPU supercomputing systems can be built from homogeneous CPU systems by replacing some CPU sockets with GPUs. For example, the Cray XK6 system is adapted from its XE6 counterpart by replacing one CPU socket with a GPU socket on each node. To make a fair comparison we compare our hybrid routines tested on a system with eight-core Intel Xeon E5-2670 2.6 GHz CPUs and an Nvidia K20c Kepler GPU, (8 CPU threads and one GPU), against non-GPU routines, tested on a system with an sixteen-core (two sockets) Intel Xeon E5-2670 2.6 GHz CPUs (16 threads for shared-memory routines, or 16 MPI processes for distributed-memory solvers).

A. Standard eigensolver results



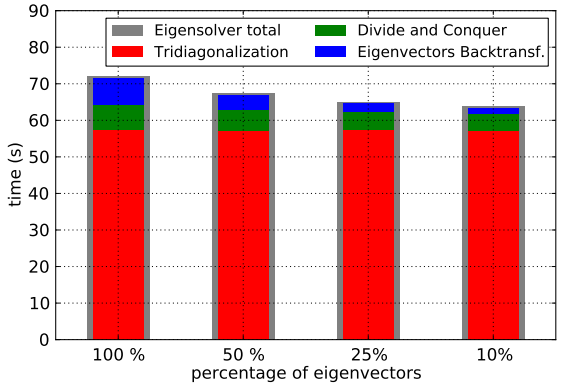
(a) double precision



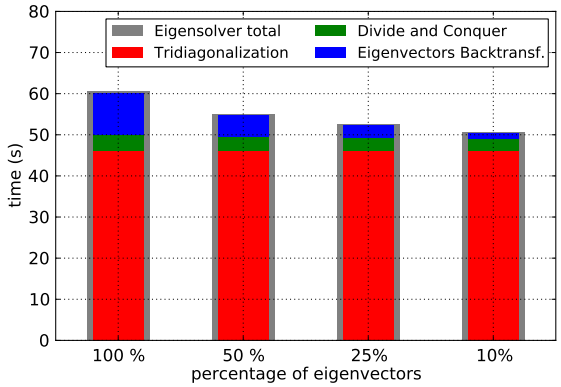
(b) double complex precision

Fig. 6. Time needed to solve a symmetric double precision eigenproblem (top) and a Hermitian double complex precision eigenproblem (bottom) with the one-stage tridiagonalization approach on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.

Figure 6 shows the standard eigensolver results using the one-stage approach with different matrix sizes and different percentages of eigenvectors computed. The overhead observed for increasing the fraction of the eigenvectors computed is small. The reason is that, as Figure 7 shows, most of the time is spent in the tridiagonalization. Since the tridiagonalization does not depend on the fraction of eigenvectors wanted, it limits the speedup that can be achieved for computing a part



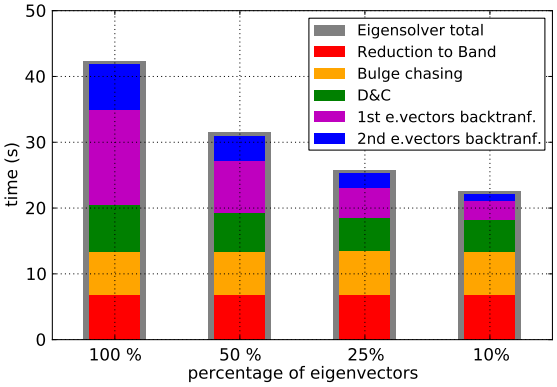
(a) double precision



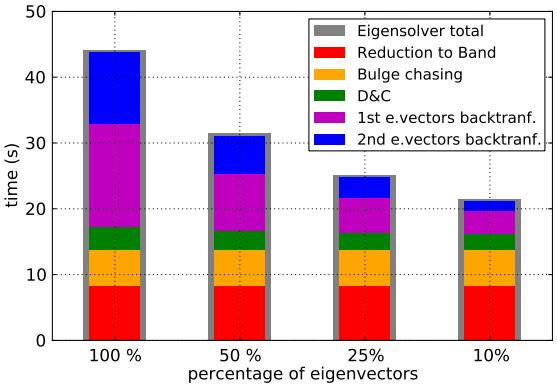
(b) double complex precision

Fig. 7. Time needed by the different subroutines to solve a symmetric double precision eigenproblem, matrix size $14,000$ (top) and a Hermitian double complex precision eigenproblem, matrix size $10,000$ (bottom) with the one-stage tridiagonalization approach on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.

of the eigenspace. Figure 8 presents a breakdown for the time spent on the different routines for computing the standard eigensolver using the two-stage approach. We note that here the time for the tridiagonalization is depicted by two routines, namely, the reduction to band and the bulge chasing. Similarly, the time for the back transformation of the eigenvectors is also reported in two routines. Next, we compare the one- and the two-stage approaches. Numerical experiments show that the two-stage tridiagonalization process is around 3-4 times faster than the one-stage. The back transformation, though, is faster in the one-stage approach. This is due to the fact that in the two-stage approach we have two back transformations to be applied (see equation 9), and the first back transformation (i.e., the application of V_2) cannot be implemented as efficiently as the second because of the irregular, diamond-shaped blocks of V_2 (Figure 4a). Overall however, the eigensolver using the two-stage approach is always faster than the one using the one-stage approach, even if the computation of all the eigenvectors is required. When reducing the number of eigenvectors requested, the speedup observed by the two-stage approach becomes larger. This is quantified in Figure 9 where we show the time ratio for the symmetric eigensolver

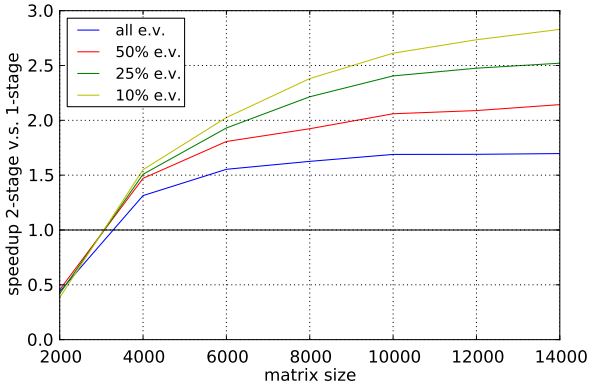


(a) double precision

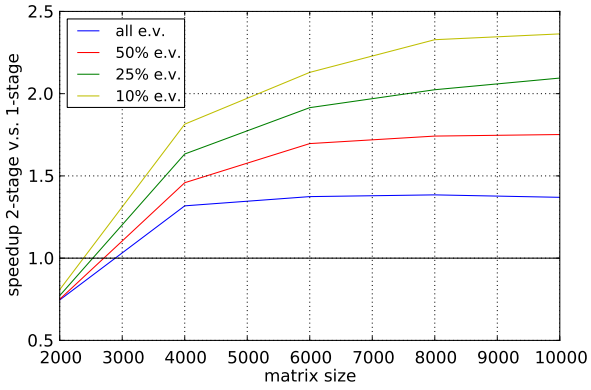


(b) double complex precision

Fig. 8. Time needed by the different subroutines for a symmetric double precision eigenproblem, matrix size $14,000$ (top) and a Hermitian double complex precision eigenproblem, matrix size $10,000$ (bottom) with the two-stage tridiagonalization approach on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.



(a) double precision



(b) double complex precision

Fig. 9. Ratio between time needed by the one-stage and two-stage eigensolver on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.

using the one-stage vs. the two-stage approach. Note that the two-stage approach overcomes the one-stage approach in both double and double complex precision when all or a fraction of the eigenspace is requested. Figure 10 presents a global comparison between our MAGMA one- and two-stage eigensolvers, the eigensolvers available in MKL version 10.3, and ELPA when either all or 10% (excluding MKL) of the eigenvectors are requested. The comparison shows that we exceed the performance of both the shared-memory (MKL) and the distributed-memory (ELPA) libraries.

The electronic structure problem at hand requires a fraction of the eigenvectors. In this case, the two-stage approach becomes of great interest.

B. Generalized eigensolver results

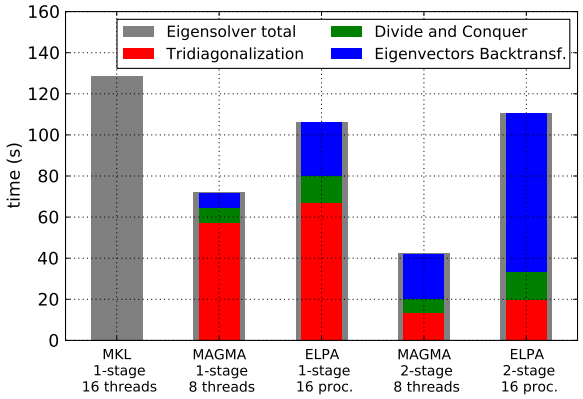
Figure 11 summarizes our results of the one-stage approach for the generalized Hermitian eigensolver with different percentages of eigenvectors computed. Similar to the standard eigensolver results from Section VI-A, the overhead for increasing the fraction of the eigenvectors computed is small.

Figure 12 shows the comparison between the one- and the two-stage approaches. The speedup observed grows with reducing the number of eigenvectors requested. As a result, we can claim that the two-stage approach always achieves higher performance than the one-stage approach.

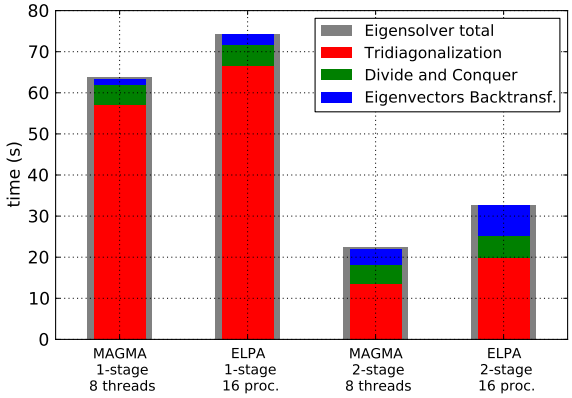
Figure 13 presents the comparison between the double complex generalized eigensolvers of our (MAGMA) one- and two-stage approaches, the eigensolvers available in MKL version 10.3, and ELPA when either all or 10% (excluding MKL) of the eigenvectors are requested. The comparison shows that we overcome in time to solution the shared-memory MKL library and the distributed-memory ELPA library.

VII. Conclusions and Future Directions

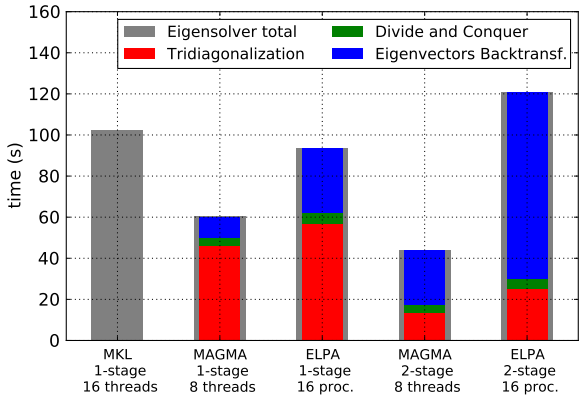
We developed a compute-bound approach to solve the standard eigenvalue problem on hybrid CPU-GPU systems and compared it with other approaches. Both the one- and the two-stage approaches, when used on a CPU socket+GPU, show a significant speedup with respect to the best currently available shared-memory and distributed-memory libraries using two CPU sockets. We then extended the one- and two-stages standard eigenproblem solvers to solve the generalized eigenproblem on hybrid systems. Both the one-stage and two-stage approaches showed a relevant speedup in comparison with the shared-memory and distributed-memory libraries. Also, our generalized eigensolver achieved a performance speedup against the other libraries presented in the paper. Finally, understanding the challenges in developing algorithms for hybrid



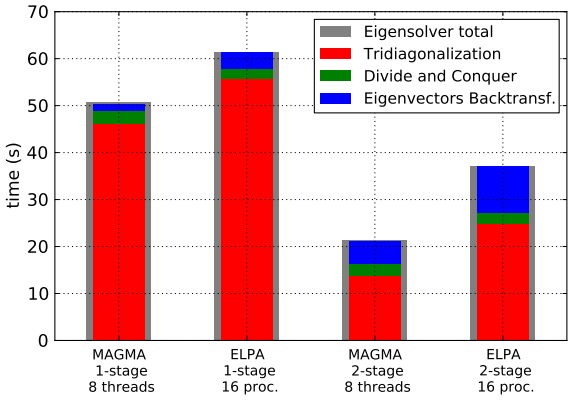
(a) double precision, all eigenvectors



(b) double precision, 10% eigenvectors



(c) double complex precision, all eigenvectors



(d) double complex precision, 10% eigenvectors

Fig. 10. Comparison between different divide and conquer routines for a symmetric double precision eigenproblem, matrix size $14,000$ (top) and a Hermitian double complex precision eigenproblem, matrix size $10,000$ (bottom) with both a one and two-stage tridiagonalization approach. MAGMA runs are on a system eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU, MKL and ELPA runs are on a system with sixteen-core (two sockets) Intel Xeon E5-2670 CPUs.

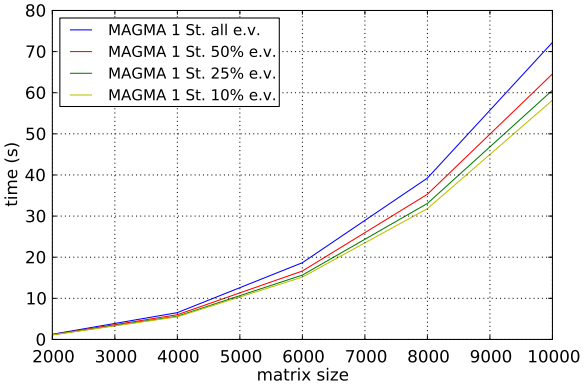


Fig. 11. Time needed to solve a Hermitian double complex precision generalized eigenproblem on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.

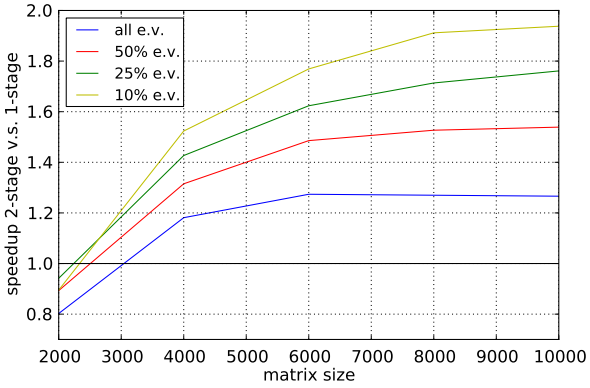
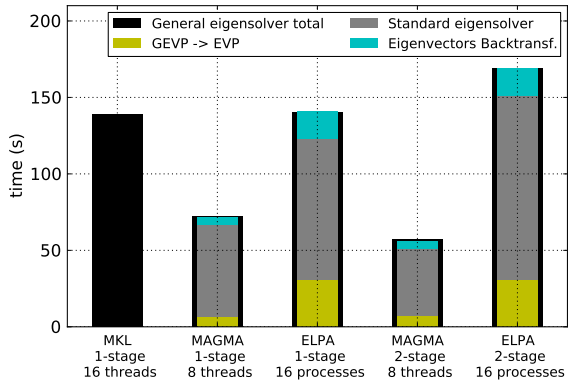


Fig. 12. Ratio between time needed by the one-stage and two-stage double complex generalized eigensolver on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU.

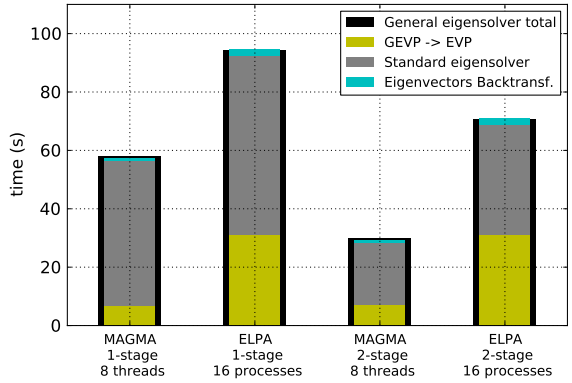
shared memory CPUs-GPU systems is critical before tackling the distributed CPUs-GPUs environment, since the routines developed for a hybrid CPU-GPU node emulate to some extent the distributed context, and can be used as building blocks in the development. A future distributed implementation will eventually be developed.

ACKNOWLEDGMENTS

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and MathWorks



(a) double complex precision, all eigenvectors



(b) double complex precision, 10% eigenvectors

Fig. 13. Comparison between different divide and conquer routines for a Hermitian double complex precision general eigenproblem, matrix size 10,000 with both a one and two-stage tridiagonalization approach. GEVP $\rightarrow$ EVP (General eigenvalue problem to eigenvalue problem) denotes both the Cholesky decomposition and the reduction to standard form. MAGMA runs are on a system with eight-core Intel Xeon E5-2670 and an Nvidia K20c GPU, MKL and ELPA runs are on a system with sixteen-core (two sockets) Intel Xeon E5-2670 CPUs.

for supporting this research effort.

REFERENCES

[1] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*, Z. Bai, Ed. PA: Society for Industrial and Applied Mathematics, 2000.

[2] P. Kent, "Computational challenges of large-scale, long-time, first-principles molecular dynamics," *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012058, 2008. [Online]. Available: http://stacks.iop.org/1742-6596/125/i=1/a=012058

[3] T. Auckenthaler, V. Blum, H. J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Comput.*, vol. 37, no. 12, pp. 783–794, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.05.002

[4] D. J. Singh, *Planewaves, Pseudopotentials, and the LAPW Method*. Boston: Kluwer, 1994.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. PA: Society for Industrial and Applied Mathematics, 1997.

[7] Intel, "Math Kernel Library," Available at http://software.intel.com/en-us/articles/intel-mkl/.

[8] AMD, "AMD Core Math Library (ACML)," available at http://developer.amd.com/tools/.

[9] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[10] R. G. Grimes and H. D. Simon, "Solution of large, dense symmetric generalized eigenvalue problems using secondary storage," *ACM Transactions on Mathematical Software*, vol. 14, pp. 241–256, September 1988. [Online]. Available: http://doi.acm.org/10.1145/44128.44130

[11] B. Lang, "Efficient eigenvalue and singular value computations on shared memory machines," *Parallel Computing*, vol. 25, no. 7, pp. 845–860, 1999.

[12] C. H. Bischof, B. Lang, and X. Sun, "Algorithm 807: The SBR Toolbox—software for successive band reduction," *ACM Transactions on Mathematical Software*, vol. 26, no. 4, pp. 602–616, 2000.

[13] P. Luszczek, H. Ltaief, and J. Dongarra, "Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures," in *IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, USA, May 16-20 2011.

[14] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, Aug. 1990, dOI 10.1145/79173.79181.

[15] G. Ballard, J. Demmel, and I. Dumitriu, "Communication-optimal parallel and sequential eigenvalue and singular value algorithms," EECS University of California, Berkeley, CA, USA, Technical Report EECS-2011-14, February 2011, LAPACK Working Note 237.

[16] L. Karlsson and B. Kågström, "Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures," *Parallel Computing*, 2011, dOI:10.1016/j.parco.2011.05.001.

[17] B. Kågström, D. Kressner, E. Quintana-Orti, and G. Quintana-Orti, "Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited," *BIT Numerical Mathematics*, vol. 48, pp. 563–584, 2008.

[18] H. Ltaief, P. Luszczek, and J. Dongarra, "High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures," *ACM TOMS*, 2011, Accepted.

[19] A. Haidar, H. Ltaief, and J. Dongarra, "Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels," in *Proceedings of SC '11*. New York, NY, USA: ACM, 2011, pp. 8:1–8:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063394

[20] S. Tomov, R. Nath, and J. Dongarra, "Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing," *Parallel Comput.*, vol. 36, no. 12, pp. 645–654, 2010.

[21] P. Bientinesi, F. D. Igual, D. Kressner, and E. S. Quintana-Ortí, "Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures," ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 387–395. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882839

[22] C. Vomel, S. Tomov, and J. Dongarra, "Divide and conquer on hybrid GPU-accelerated multicore systems," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C70–C82, 2012.

[23] "MAGMA 1.3," http://icl.cs.utk.edu/magma/, 2012.

[24] "PLASMA," http://icl.cs.utk.edu/plasma/.

[25] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 2nd ed. Baltimore, MD, USA: The Johns Hopkins University Press, 1989.

[26] J. O. Aasen, "On the reduction of a symmetric matrix to tridiagonal form," *BIT*, vol. 11, pp. 233–242, 1971.

[27] B. N. Parlett, *The Symmetric Eigenvalue Problem*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1980.

[28] 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/.

[29] C. H. Bischof and C. V. Loan, "The WY representation for products of Householder matrices," *SIAM J. Sci. Statist. Comput.*, vol. 8, pp. s2–s13, 1987.

[30] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling, "Block reduction of matrices to condensed forms for eigenvalue computations," *Journal of Computational and Applied Mathematics*, vol. 27, no. 1-2, pp. 215 –

227, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYH-45H2DG8-D/2/c56b22700b2d79feb9f4048732043c49

[31] W. Gansterer, D. Kvasnicka, and C. Ueberhuber, "Multi-sweep algorithms for the symmetric eigenproblem," in *Vector and Parallel Processing - VECPAR'98*, ser. Lecture Notes in Computer Science. Springer, 1999, vol. 1573, pp. 20–28. [Online]. Available: http://dx.doi.org/10.1007/10703040_3

[32] H. Ltaief, J. Kurzak, and J. Dongarra, "Parallel band two-sided matrix bidiagonalization for multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, April 2010.

[33] J. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.