Leading Edge Hybrid Multi-GPU Algorithms for Generalized Eigenproblems in Electronic Structure Calculations*

Azzam Haidar[1], Raffaele Solcà[4], Mark Gates[1], Stanimire Tomov[1], Thomas Schulthess[4,5], and Jack Dongarra[1,2,3]

[1] University of Tennessee Knoxville
[2] Oak Ridge National Laboratory
[3] University of Manchester
[4] Institut for Theoretical Physics, ETH Zurich
[5] Swiss National Supercomputer Center

Abstract. Today's high computational demands from engineering fields and complex hardware development make it necessary to develop and optimize new algorithms toward achieving high performance and good scalability on the next generation of computers. The enormous gap between the high-performance capabilities of GPUs and the slow interconnect between them has made the development of numerical software that is scalable across multiple GPUs extremely challenging. We describe and analyze a successful methodology to address the challenges—starting from our algorithm design, kernel optimization and tuning, to our programming model—in the development of a scalable high-performance generalized eigenvalue solver in the context of electronic structure calculations in materials science applications. We developed a set of leading edge dense linear algebra algorithms, as part of a generalized eigensolver, featuring fine grained memory aware kernels, a task based approach and hybrid execution/scheduling. The goal of the new design is to increase the computational intensity of the major compute kernels and to reduce synchronization and data transfers between GPUs. We report the performance impact on the generalized eigensolver when different fractions of eigenvectors are needed. The algorithm described provides an enormous performance boost compared to current GPU-based solutions, and performance comparable to state-of-the-art distributed solutions, using a single node with multiple GPUs.

1 Introduction

In the context of electronic structure problems in material science and chemistry, the solution of the generalized Hermitian-definite eigenvalue problem is the most expensive task, dominating the entire computation [4, 20, 26]. In parallel electronic structure codes, many independent eigenvalue problems must be

* The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and MathWorks for supporting this research effort.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 67–80, 2013.

solved, allowing each problem to be solved independently on a different node. In this work we are thus interested in dense eigensolvers, and in particular, for generalized Hermitian-definite problems of the form

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

where A is a dense Hermitian matrix and B is Hermitian positive definite. Solving (1) requires the development of a number of routines. First, the matrix B is decomposed using a Cholesky factorization into $B = LL^H$, where ^{H} denotes conjugate-transpose. The resulting L factors are used to transform (1) to a standard Hermitian eigenproblem $\tilde{A}z = \lambda z$, where $\tilde{A} = L^{-1}AL^{-H}$. After solving the standard Hermitian eigenproblem, the eigenvectors X of the generalized problem (1) are then computed by backsolving with the Cholesky factor, $X = L^{-H}Z$. To solve the standard Hermitian (symmetric) eigenproblem of the form $\tilde{A}z = \lambda z$, finding its eigenvalues Λ and eigenvectors Z such that $\tilde{A} = Z\Lambda Z^H$, the standard strategy follows three steps [1,12,24]. First, reduce the matrix to a tridiagonal matrix T using an orthogonal transformation Q such that $\tilde{A} = QTQ^H$ (called the "reduction phase"). Note that when a two-sided orthogonal transformation is applied to generate T, 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"). All of these steps are computationally expensive, so we will develop an efficient multi-GPU implementation of each step.

2 Related Work

Solving the generalized eigenvalue problem is an active research field. Recently many researchers have been interested in this area and have developed various strategies, with a number of software implementations. The robust and conventional software LAPACK [3] and ScaLAPACK [7] are for shared-memory and distributed-memory systems, respectively. Recent work on symmetric eigenvalue problems has concentrated on accelerating separate components of the solvers, and in particular, the reduction to tridiagonal form, which is the most time consuming phase, and also the eigensolver. A new type of algorithm that challenges the standard one-stage reduction algorithms has been introduced. The idea behind this new technique is to split the reduction phase into two or more stages, recasting expensive memory-bound operations that occur during the panel factorization into compute-bound operations. One of the first uses of a two-stage reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [13]. Then, a multi-stage method was used to reduce a matrix to tridiagonal, bidiagonal and Hessenberg forms [21]. Consequently, a framework called Successive Band Reduction (SBR) was developed [5,6]. A multi-stage approach has also been applied to the Hessenberg reduction [18,19]. Tile algorithms have also recently seen a rekindled interest when applied to the two-stage tridiagonal [15,23] and bidiagonal reductions [22]. Their first stage is

implemented using high performance kernels and asynchronous execution while the second stage is implemented based on cache-aware kernels and a task coalescing technique [15]. Recently, a distributed-memory eigensolver library called ELPA [4] was developed for electronic structure codes. ELPA is similar to ScaLAPACK and does not support GPUs. It includes one-stage and two-stage tridiagonalizations, the corresponding eigenvector transformation, and a modified divide and conquer routine that can compute the entire eigenspace or a portion of it. These approaches, in contrast to our own, are not for hybrid GPU-CPU systems.

With the emergence of high-bandwidth, high-performance GPUs, memory-bound and compute-bound operations can be accelerated by an order of magnitude or more. Tomov et al. [29, 30] presented a hybrid CPU-GPU implementation for the one stage reduction algorithms, which take advantage of the high-bandwidth of the GPU by offloading the expensive Level 2 BLAS operations to the GPU. Dong et al. [9] extended this to multi-GPUs. Haidar et al. [16] developed a two-stage approach for multicore and a single GPU. The main thrust of the work presented here is the extension of this two-stage approach to multi-GPUs.

3 Main Contributions

Besides the software development efforts that we investigate to accomplish an efficient implementation, we highlight three main contributions related to the algorithm's design:

- **Fine grained memory aware and computationally intense tasks.** Our approach to efficient hardware use and parallelism relies on splitting the computation into tasks that either increase computational intensity or reduce data movement. Two main issues should be taken into consideration here. First, the task splitting and determination of granularity is essential for obtaining high performance. Second, the data distribution among the CPUs and GPUs should also be taken into consideration to minimize communication and achieve good performance.
- **Hybrid multi CPU-GPU execution.** Along with the computation splitting, a hybrid multi CPU-GPU implementation combined with task scheduling is an indispensable ingredient for obtaining high performance algorithms. We map computational tasks to the strengths of heterogeneous hardware components and overlap computation on GPUs with computation on CPUs.
- **A hierarchical multi-GPU communication model,** which optimizes communication for multi-GPUs and can be applied in general, beyond the scope of the algorithms developed.

4 Hybrid Multi GPU-CPU Algorithm

In this section we describe our multi CPU-GPU algorithm, presenting a detailed study to explain how we achieve good performance while dealing with a heterogeneous system. To make our description fruitfully interesting and clear we will

also describe implementation issues and give performance results of each kernel's multi-GPU implementation. Before developing our kernels, we should pay attention to the communication schema that our algorithm will use, as communication is an important component that affects any multi-GPU implementation.

4.1 Hierarchical Communication Model

As GPUs are well known for their high-performance capabilities and the relatively slow interconnect between them, reducing communication or overlapping communication with computation is critical in order to maximize the time GPUs spend in compute-intensive kernels, and minimize the time spent only in communication. To address the increase in communication when a large number of GPUs are used together, we developed a hierarchical communication model. Each PCIe switch connecting GPUs is viewed as a node in a distributed system, with one GPU in each node assigned to be the master. Within each node, GPUs communicate locally in a "free GPU-GPU mode" between the master GPU and other GPUs; between two nodes, the master GPUs communicate together directly. This hierarchical communication model is easily adaptable to a distributed environment, where communication between master GPUs of different nodes should be done via the CPU using MPI.

4.2 Transformation from Generalized to Standard Eigenvalue

As described above, the transformation from a generalized to a standard eigenvalue problem consists, first, in performing the multi-GPU Cholesky factorization of B. We refer the reader to [31] for a detailed description of our multi-GPU Cholesky implementation. Then, the resulting factor L is used to compute $\tilde{A} = L^{-1}AL^{-H}$. This operation is equivalent to the xHEGST function of the LAPACK library. We split this operation into three phases: (1) partially compute a panel A_i (blue portion of Figure 1a), then (2) use it to update the trailing matrix $A_{i+1:n}$ (red and green portion) by a xHER2K, and finally (3) continue the computation of the panel A_i (blue portion). Our multi-GPU algorithm distributes the matrix A in a 1D column block cyclic distribution, thus each GPU owns many blocks of A.

To optimize the code, phase 3 is delayed to the end of the computation, since its final result is not needed by any of the subsequent steps $i+1, i+2, \ldots$, while the GPU that owns it also owns other blocks involved in the update phase 2, so computing phase 3 at step i may delay the computation. To further reduce the synchronization between steps, during the update phase 2, the GPU that owns the next panel, A_{i+1}, will prioritize it and update it first, perform its partial computation (phase 1), broadcast it to other GPUs, and then continue the update of phase 2. In this way, while GPUs are updating (phase 2) at step i, they will receive the next panel A_{i+1} required to perform the next update of step $i+1$. Finally, once all updates are done, the GPUs will compute the remaining phase 3 computation of their blocks independently, without requiring any further communication.

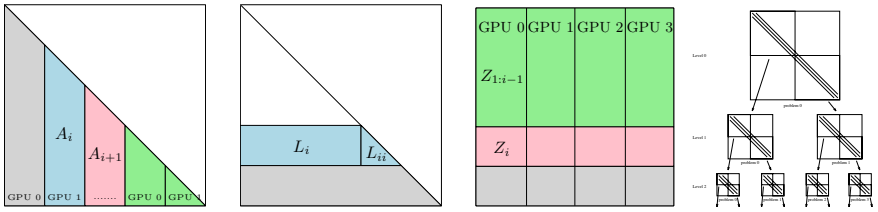


Fig. 1. (a) The multi-GPU xHEGST algorithm. (b) and (c) The i^{th} step of the blocked xTRSM. (d) the divide and conquer partitioning tree.

When the eigenvectors X are requested, the back transformation operation $X = L^{-H}Z$ needs to be performed. This operation can be performed in a parallel fashion where L^{-H} is applied independently to each vector of Z. Thus, if we split the matrix Z between the GPUs using a 1-D column block cyclic distribution, as depicted in Figure 1c, while the stored factor L is in block rows as shown in Figure 1b, then this operation can be performed independently by each GPU as follows. Each block row of the matrix L is broadcast over the GPUs (e.g., the blue block of Figure 1b). Once received, each GPU perform two operations. First, it computes $Z_i = L_{ii}^{-H} Z_i$ (red portion of Figure 1c), and then it updates $Z_{1:i-1} = Z_{1:i-1} - L_i^H Z_i$ (green portion). During these two operations the CPU will broadcast the next L_{i+1} to all the GPUs, so as to overlap the copy and the computation. The idea is to minimize the communication and overlap it as much as possible. We represent in Figure 2a the speedup obtained over the 1-GPU implementation by this multi-GPU implementation of this kernel when we vary the matrix size from 2000 to 40000, and also as we increase the number of GPUs from 2 to 8 GPUs. We see that this kernel asymptotically reaches near-perfect scaling.

4.3 Hybrid Multi CPU-GPU Tridiagonal Reduction

Due to its computational complexity and data access patterns, the tridiagonal reduction phase is the most challenging to develop, both algorithmically and implementation-wise. There are two algorithmic approaches — the standard one-stage approach from LAPACK [2], where block Householder transformations are used to directly reduce the dense matrix to tridiagonal form, and a newer two-stage (or more) approach, 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 [15]. The one-stage approach is well known to be memory bound as it relies on symmetric matrix-vector multiplications (50% of the flops).

The two-stage approach overcomes the memory-bound limitations of the one-stage. The reduction to band is done very efficiently using Level 3 BLAS. In particular, the dense matrix is first spread among the GPUs in a 1-D column block-cyclic distribution. The panel that must be factored at each iteration is

sent and factored on the CPU. The result is broadcast back to the GPUs and used for their local updates. A look ahead techniques is used — that is, the next panel to be factored is updated first and sent to the CPU for factorization while the GPUs complete the rest of their updates. This allows us to overlap CPU and GPU work. Moreover, all communication required during the update process is performed in an efficient manner using our hierarchical model.

Figure 2b shows the performance and scalability of the multi-GPU reduction to band. We see that this implementation of the two-stage approach provides good scalability. For two and three GPUs the scalability is perfect, while for four GPUs it approaches perfect scaling for a large matrix. For larger number of GPUs, it would require larger matrices to be able to reach the asymptotic perfect behavior. For example, if we split a matrix of size 20K over 8 GPUs, then each GPU will hold a small portion of size 20K×2.5K, which is not enough to perform intensive operations. The second stage that reduces the band matrix to tridiagonal is done on the multicore host using a multi-threaded bulge chasing implementation. Further detail can be found in Haidar et al. [14]. We observe that this multi-GPU implementation of the reduction to tridiagonal provides a jump in the performance compared to its one stage multi-GPU counterpart, being approximately four times faster than the one-stage approach. We will not detail more on this as the purpose of this paper is the overall generalized eigensolver problem.

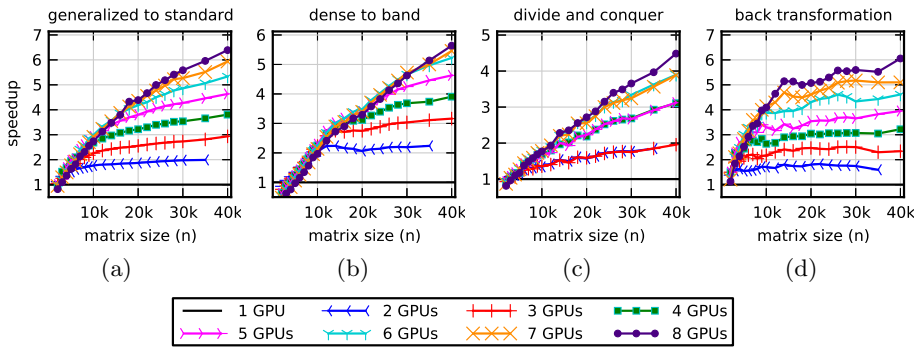


Fig. 2. Speedup relative to one GPU for major components of generalized eigensolver

4.4 Flexible Multi-GPU Divide and Conquer Algorithm

Introduced by Cuppen [8], the divide and conquer (D&C) algorithm computes the eigenvalues of the tridiagonal matrix T. Many serial and parallel Cuppen-based eigensolver implementations for shared and distributed memory have been proposed in the past [10, 11, 17, 25, 27, 28]. The overall D&C approach consists in splitting the problem into two subproblems (son nodes) representing a rank-one modification. Each of these subproblems is an independent problem without any data dependencies with the other subproblems. This process is repeated recursively, constructing a binary tree where the bottom nodes will have two

independent sons of small size considered as two *simple* eigenvalue problems. Then, on each parent node, merge the two subproblems (left and right son) which are defined by a rank-one modification of a diagonal matrix, and proceed to the next level in a bottom-up fashion.

To illustrate how our multi-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 (2). 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 (2), then $\Lambda = \tilde{\Lambda}_0$ and $E = \tilde{E}_i \tilde{E}_0$ are the eigenpairs of T.

$$\begin{aligned} T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T &= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left\{ \begin{pmatrix} \tilde{\Lambda}_1 & 0 \\ 0 & \tilde{\Lambda}_2 \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T \right\} \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T \\ &= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left(\tilde{E}_0 \tilde{\Lambda}_0 \tilde{E}_0{}^T \right) \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T = E \Lambda E^T \end{aligned} \tag{2}$$

To find the eigensolution of each rank-one modified system M requires solving its secular equation. This is a memory bound process that requires only $\mathcal{O}(n^2)$ operations, so in our implementation we keep this computation on the CPU side, while the GPUs perform the multiplication of the intermediate eigenvector matrices $\tilde{E}_i$.

The independent parallelism generated by the D&C approach allows us to distribute each division over half of the GPUs recursively. For example, half of the GPUs will compute the upper part of the eigenvectors (involving E_1), and the rest the lower part (involving E_2). This imposes a constraint on the number of GPUs to be multiple of 2. The implementation could be generalized to deal with any number of GPUs, but based on the way the eigenvectors are generated we don't expect this would give more improvement. We plot in Figure 2c the speedup of this kernel for various matrix sizes when increasing the number of GPUs. When the computing intensive kernels are performed on the GPUs, it reduces the time to compute such expensive operations, and thus the time required to solve the memory bound secular equation becomes dominant, so we can expect that the performance of this kernel to have limited scalability. Nonetheless, the obtained scalability remains very attractive. We also modified the algorithm such that when only a portion of the eigenvectors are required, the multiplication is done with only this portion, reducing the total amount of computation.

4.5 Back Transformation

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

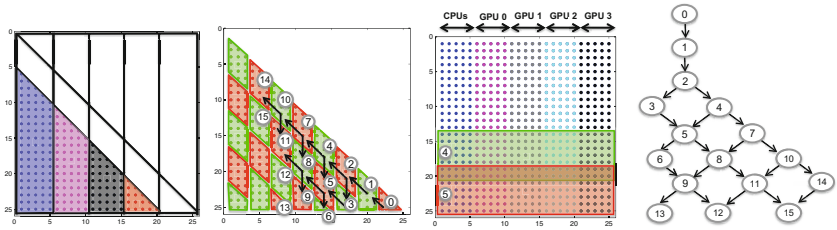


Fig. 3. (a) Tiling of V_1, (b) Blocking technique to apply V_2, (c) Distribution of the eigenvectors matrix that create independent fashion of applying Q_2 which increase locality per core, (d) Portion of the DAG showing the dependency of the V's of V_2

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

$$Z = Q_1 Q_2 E = (I - V_1 t_1 V_1^H)(I - V_2 t_2 V_2^H) E, \tag{3}$$

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

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 3b. 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 us give an example that explain those issues. 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 3b. Similarly, the ones related to sweep $i+1$ are those presented in column $i+1$. They are shifted one element down compared to those of sweep i. After analyzing the dependencies of the bulge chasing procedure as explained by the example above, we notice that 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 3b. While each of those diamonds is considered as one block, their application needs to follow the dependency order. For example, applying the green block 4 and the red block 5 of the V_2's in Figure 3b modifies the green block row 4 and the red block row 5, respectively, of the eigenvector matrix E drawn in Figure 3c, 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 3b. For clarity, we also represented them by the DAG in Figure 3d. A little effort studying the pattern of dependencies of this DAG leads us to the conclusion that designing an algorithm based on such schema provides a very limited number of parallel and pipelined tasks. Despite all of those constraints, a nice feature to create efficiency is that, if we design our parallelism based on the matrix E, where we split E by block column over the number of cores/GPUs as shown in Figure 3c, then we can apply each diamond block independently to each portion of E. Moreover, this way does not require any data communication between GPUs. The overlap between each application of V's as described above increase the cache reuse. For the CPUs, we also define the size of the block of E in a way to fit more than one region of it in the L2 cache level to increase locality. We implemented a new kernel to deal with these diamonds to increase the cache reuse and overlap the communication to the GPUs. These blocks are broadcast to the GPUs in a look ahead fashion, meaning that when GPUs are using the block V_2^m to update their portion of E, the CPU broadcasts the next V_2^{m+1} in a overlapped technique.

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, they can be blocked as shown in Figure 3a. Thus their application is computing intensive and involves efficient BLAS 3 kernels. Using the same parallelism design, the V_1 can be applied independently to each block column of Figure 3c, which are now the block of G and thus the distribution remains the same. This operation does not require any communication between GPUs or between the previous kernel (apply V_2) and the current kernel (apply V_1). Similarly, each block of V_1's is broadcast over the GPUs in a look ahead fashion. This implementation is independent and very suitable for parallel and heterogeneous implementation, especially when communication is expensive. Speedup results for matrices raising from 2000 to 40000 when varying the number of GPUs are presented in Figure 2d, showing a very good speedup is obtained.

5 Experimental Results

The eigensolver presented in this paper was tested on an experimental machine offering a dual-socket six-core Intel Xeon 5675 running at 3.07 GHz, with 48 GB of main system memory and 8 NVIDIA Fermi M2090 GPUs. We tested the distributed memory libraries on a tightly coupled computing cluster system, offering nodes based on the dual-socket six-core Intel Xeon 5650 processor architecture, running at 2.6 GHz, with 24 GB of main system memory per node.

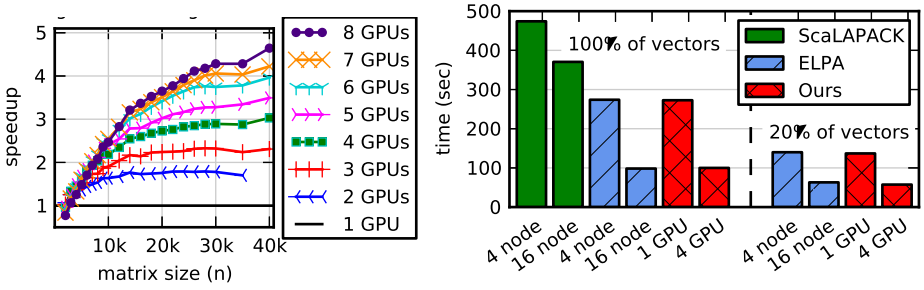
5.1 Accuracy Analysis

We mention that the only difference, numerically, between our algorithm and the LAPACK algorithm is that we use a two-stage algorithm for the reduction to tridiagonal. The reduction to tridiagonal relies on the Householder elimination,

which has been proved to be backward stable and accurate, and hence we expect the same accuracy as LAPACK. Our implementation of the divide and conquer is based on the main Cuppen algorithm and is exactly the same as the one implemented in the LAPACK library. All of our experiments obtained the same order of accuracy as that computed by the LAPACK solver, using the metrics described in the LAPACK Users' Guide [2].

5.2 Performance Scalability

We performed an extensive study with a large number of experimental tests to give the reader as much information as possible. We computed the eigenpairs of the generalized eigenvalue problem, varying the size of matrices from 2000 to 40000 and varying from 1 to 8 GPUs. Figure 4a shows the speedup obtained by our hybrid multi-GPU symmetric generalized eigenvalue solver as compared to its one GPU implementation. As expected, the speedup performance obtained has a similar trend to the ones presented above in Figure 4 for each kernel of the algorithm. Strong scalability is observed as, for a fixed matrix size, when we increase the number of GPUs the time should decrease linearly. The results show a very good scalability for our implementation on such an heterogeneous hybrid system with a huge computing intensive component (8 GPUs) connected to it. We can see that although some kernels of this algorithm are strictly multicore CPU implementations (the bulge chasing and the secular equation solver), our hybrid multi-GPU implementation provides a very attractive scalability. On four GPUs, our approach is able to run three times faster than on one GPU, which is considered to be very good scalability. On a larger number of GPUs, it requires large matrices to be able to see the asymptotic scaling behavior.



(a) Generalized eigensolver scalability (b) Performance comparison

Fig. 4. (a) Speedup relative to one GPU for generalized eigensolver. (b) Time comparison of various implementations to solve $n = 20000$ generalized eigenvalue problem, computing either 100% of eigenvectors (on left) or only 20% of eigenvectors (on right).

5.3 Performance Results

We compare the performance of our hybrid multi-GPU eigensolver against the optimized state-of-the-art numerical linear algebra library ScaLAPACK, and the ELPA library, using both the one stage and the two stage reduction. We ran tests for ELPA and ScaLAPACK for various problem sizes, while varying the number of processors from 24 to 240. The results reported here are the best results achieved. In our tests we found that both one stage and two stage ELPA implementations show good scalability compared to ScaLAPACK; the required time decreases three times when increasing the number of processors from 48 to 192. We also found that the two-stage approach of the ELPA solver was faster than its one-stage approach for any number of nodes and any percentage of the eigenvectors requested, and so we omit results of the one stage of ELPA from our graphs. Figure 4b shows the time needed for all the solvers to find the solution of the generalized eigenvalue problem for a matrix of size 20000. For ScaLAPACK and ELPA, we give results on four nodes, each dual-socket six-core (48 processors), and also on 16 nodes (192 processors), which have reasonable time compared to our multi-GPU solver. These distributed and GPU-based systems have comparable peak matrix-matrix multiply performance: 48 processors has 460 Gflop/s peak, compared to 500 Gflop/s using 12 cores plus one GPU, while 192 processors has 1.9 Tflop/s peak, compared to 1.6 Tflop/s using 12 cores plus four GPUs. Comparing to our approach, we can see that, to solve the generalized eigenvalue problem computing all (100%) of its eigenpairs, the time needed by ScaLAPACK on 192 processors and by the ELPA implementation on 48 processors is very close to the time needed by our hybrid multi-GPU solver running with only one GPU. Similarly, the time needed by the ELPA solver running on 192 processors is similar to the time needed by our multi-GPU solver running on only four Fermi GPUs. This behavior has also been observed for other matrices size, in particular, for a matrix of size 30000, the ELPA solver running on 192 cores requires 327 seconds to find all of its eigenpairs, while our hybrid multi-GPU solver running with four Fermi GPUs needs 314 seconds. These results show that with only a small numbers of devices, an efficient multi-GPU implementation can achieve as much speed as one of the best solvers on 192 processors.

As many applications need only a portion of the eigenvectors, we also present in Figure 4b the comparison with the ELPA solver when only 20% of the eigenvectors are needed. The trend shown here is again similar to the performance shown when all the eigenpairs are computed. For example, for matrices of size 20000 and 30000, the ELPA solver running on 192 processors requires 63 seconds and 218 seconds respectively, while our solver running on four GPUs requires 57 seconds and 174 seconds respectively. We note here that when a fraction of the eigenvectors are computed, both our approach and the two stage ELPA are significantly faster than the one stage approach implemented in either ELPA or ScaLAPACK.

5.4 Real Electronic Structure Application

We did preliminary experiments in the context of a real electronic structure application. We performed a ground state computation for the Eu_6C_{60} compound with the density functional method. The density functional calculations was done with a new prototype of a linearized augmented plane wave (LAPW) library []. During each iteration a generalized eigenvalue problem of size 25383 in double complex precision, requiring the 1335 eigenvectors with the lowest eigenvalues, has to be solved. We note that this latter consists of 67% of the total iteration time when solved on a distributed system and could be decreased down to around 20% when solved with our multi-GPU implementations meaning that the time per iteration could be speeded up 3 to 4 times.

The experiment has been done on a cluster of Intel Xeon E5-2670 (Sandy Bridge) 2.6 GHz processors. The time required to perform each iteration using the ScaLapack library is 787 and 263 seconds using 64 and 256 MPI processes, respectively. Of that, 525 and 175 seconds, respectively, are spent solving the generalized eigenvalue problem (67% of the time per iteration). Using only one node (consisting of 16 processors of the same type) with 4 Nvidia K20 GPUs, we were able to reduce the time of the generalized eigensolver to 145 seconds, which will provide a huge boost. The total number of iterations needed depends on the compound and on the mixer used, and usually it varies between 20 and 100 iterations.

6 Conclusions and Future Directions

We demonstrated that it is possible to develop efficient and scalable algorithms for heterogeneous systems with an enormous gap between their computing power and interconnection bandwidth. Our hybrid multi-CPU-GPU implementation demonstrated very promising results in terms of performance as well as in terms of scalability on heterogeneous architecture systems. It has been extensively tested using different matrix types and many parallel configuration against other well-known generalized symmetric eigenvalue solvers. The performance obtained is very encouraging. These results show the impact of our work on applications, especially the field of electronic structure computations where a large number of dense generalized eigenvalue problem need to be solved in the solution of Schrödinger equation, thus the choice of a suitable method is of great importance. We believe that these techniques will only increase in relevance for upcoming architectures. We plan to further study the implementation of multi-GPU algorithms in a distributed computing environment.

References

1. Aasen, J.O.: On the reduction of a symmetric matrix to tridiagonal form. BIT 11, 233–242 (1971)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992), `http://www.netlib.org/lapack/lug/`

3. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
4. Auckenthaler, T., Blum, V., Bungartz, H.J., Huckle, T., Johanni, R., Krämer, L., Lang, B., Lederer, H., Willems, P.R.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. 37(12), 783–794 (2011)
5. Bientinesi, P., Igual, F.D., Kressner, D., Quintana-Ortí, E.S.: Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 387–395. Springer, Heidelberg (2010)
6. Bischof, C.H., Lang, B., Sun, X.: Algorithm 807: The SBR Toolbox—software for successive band reduction. ACM Transactions on Mathematical Software 26(4), 602–616 (2000)
7. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1997)
8. Cuppen, J.J.M.: A divide and conquer method for the symmetric eigenproblem. Numer. Math. 36, 177–195 (1981)
9. Dong, T., Dongarra, J., Schulthess, T., Solca, R., Tomov, S., Yamazaki, I.: Matrix-vector multiplication and tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. Parallel Comput. (July 2012) (submitted)
10. Dongarra, J.J., Sorensen, D.C.: A fully parallel algorithm for the symmetric eigenvalue problem. SIAM J. Sci. Statist. Comput. 8, s139–s154 (1987)
11. Gates, K., Arbenz, P.: Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem (1994)
12. Golub, G.H., Loan, C.F.V.: Matrix Computations, 2nd edn. The Johns Hopkins University Press, Baltimore (1989)
13. Grimes, R.G., Simon, H.D.: Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. ACM Transactions on Mathematical Software 14, 241–256 (1988)
14. Haidar, A., Gates, M., Tomov, S., Dongarra, J.: Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication. In: ICS 2013: 27th International Conference on Supercomputing, Eugene, Oregon, USA, June 10-14 (submitted, 2013)
15. Haidar, A., Ltaief, H., Dongarra, J.: Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In: SC 2011: International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA, USA, November 12-18 (2011)
16. Haidar, A., Tomov, S., Dongarra, J., Solca, R., Schulthess, T.: A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. International Journal of High Performance Computing Applications (September 2012) (accepted)
17. Ipsen, L.C.F., Jessup, E.R.: Solving the symmetric tridiagonal eigenvalues problem on the hypercube. SIAM J. Sci. Stat. Comput. 11, 203–229 (1990)
18. Kågström, B., Kressner, D., Quintana-Orti, E., Quintana-Orti, G.: Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. BIT Numerical Mathematics 48, 563–584 (2008)

19. Karlsson, L., Kågström, B.: 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
20. Kent, P.: Computational challenges of large-scale, long-time, first-principles molecular dynamics. Journal of Physics: Conference Series 125(1), 012058 (2008)
21. Lang, B.: Efficient eigenvalue and singular value computations on shared memory machines. Parallel Computing 25(7), 845–860 (1999)
22. Ltaief, H., Luszczek, P., Dongarra, J.: High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. In: ACM TOMS (2011) (accepted)
23. Luszczek, P., Ltaief, H., Dongarra, J.: 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)
24. Parlett, B.N.: The Symmetric Eigenvalue Problem. Prentice-Hall, Englewood Cliffs (1980)
25. Rutter, J., Rutter, J.D.: A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem (1994)
26. Singh, D.J.: Planewaves, Pseudopotentials, and the LAPW Method. Kluwer, Boston (1994)
27. Sorensen, D.C., Tang, P.T.P.: On the orthogonality of eigenvectors computed by divide-and-conquer techniques. SIAM J. Numer. Anal. 28(6), 1752–1775 (1991)
28. Tisseur, F., Dongarra, J.: Parallelizing the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem on distributed memory architectures. SIAM J. SCI. Comput. 20, 2223–2236 (1998)
29. Tomov, S., Nath, R., Dongarra, J.: Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Comput 36(12), 645–654 (2010)
30. Vomel, C., Tomov, S., Dongarra, J.: Divide and conquer on hybrid GPU-accelerated multicore systems. SIAM Journal on Scientific Computing 34(2), C70–C82 (2012)
31. Yamazaki, I., Tomov, S., Dongarra, J.: One-sided dense matrix factorizations on a multicore with multiple gpu accelerators. In: Proc. of ICCS 2012, Procedia CS, vol. 9, pp. 37–46 (2012)