Toward a Scalable Multi-GPU Eigensolver via Compute-Intensive Kernels and Efficient Communication

Azzam Haidar
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
haidar@eecs.utk.edu

Mark Gates
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
mgates3@eecs.utk.edu

Stanimire Tomov
Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
tomov@eecs.utk.edu

Jack Dongarra[*] [†]
Computer Science and Mathematics Division
Oak Ridge National Laboratory
dongarra@eecs.utk.edu

ABSTRACT

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 a successful methodology on how 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 tridiagonal reduction algorithm for the symmetric eigenvalue problem. This is a fundamental linear algebra problem with many engineering and physics applications. We use a combination of a task-based approach to parallelism and a new algorithmic design to achieve high performance. 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. This may increase the number of flops, but the increase is offset by the more efficient execution and reduced data transfers. Our performance results are the best available, providing an enormous performance boost compared to current state-of-the-art solutions. In particular, our software scales up to 1070 Gflop/s using 16 Intel E5-2670 cores and eight M2090 GPUs, compared to 45 Gflop/s achieved by the optimized Intel Math Kernel Library (MKL) using only the 16 CPU cores.

[*]University of Tennessee, USA

[†]School of Mathematics & School of Computer Science, University of Manchester, United Kingdom

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Singular value decomposition, Eigenvalues and eigenvectors*

Keywords

Eigenvalue, GPU Communication, GPU Computation, Performance, Reduction to tridiagonal, Singular Value Decomposition, heterogeneous programming model, task parallelism

1. INTRODUCTION

An exponentially increasing gap over the years of processor speed vs. memory and interconnect speeds has established the notion that "flops are free" compared to the cost of fetching data. Indeed, if we consider the speed of current GPUs vs. bandwidth of CPU-to-GPU communications, the cost of sending one double-precision floating-point (DP) number through PCIe 3.0, disregarding latency, can be the same as 1,000 DP flops on the latest NVIDIA Kepler K20 GPU. In other words, a computation in this setting is bound by the CPU-to-GPU bandwidth, unless data reuse exceeds 1,000. To address this issue, algorithms must be redesigned in order to increase the computational intensity of their major compute kernels, and to reduce synchronization and data transfers between processors. We describe how to achieve this in the development of a scalable high-performance multi-GPU tridiagonal reduction for the symmetric eigenvalue problem.

Eigenvalue problems are fundamental for many engineering and physics applications. For example, image processing, compression, facial recognition, vibrational analysis of mechanical structures, and computing energy levels of electrons in nanostructure materials can all be expressed as eigenvalue problems. There are many ways to formulate mathematically and solve these problems numerically [11], but in all cases, designing an efficient computation is challenging because of the nature of the algorithms. In particular, the basic transformations (orthogonal similarity transformations) applied to the matrix are two-sided, i.e., transformations are applied on both the left and right side of the matrix. This creates data dependencies that prevent the use of standard techniques to increase the computational intensity of the computation [26], such as blocking and look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations [27]. New algorithms are needed in order to overcome the bottlenecks associated with the standard techniques, as described in Section 4.

Besides the algorithmic design, the development of high-performance scalable software is linked to the programming model and languages used in the implementation. Our development is done in C/C++ and, when available, we rely on existing routines in numerical libraries like CUBLAS and MAGMA BLAS (for GPUs), MKL, ACML, and PLASMA (for multicore CPUs), or MAGMA (for hybrid CPU-GPU systems) to provide the basic building blocks for our algorithms. Our programming model is based on task parallelism where the computation is split into tasks and data dependencies among the tasks, and the execution is scheduled over all available heterogeneous hardware components without violating the dependencies. This model has been successfully used in the development of dense linear algebra libraries [3], and is self-evident in the presentation.

2. RELATED WORK

LAPACK [4] and ScaLAPACK [10] are the standard linear algebra libraries for dense and banded matrices that include eigensolver routines. LAPACK is for shared-memory systems, while ScaLAPACK is its extension to distributed-memory systems. Neither of them supports GPUs. Hardware vendors in general provide well tuned LAPACK and ScaLAPACK versions based on optimized BLAS — the building block for these libraries.

Recent work specifically on eigenvalue problems has concentrated on accelerating separate components of the solvers, and in particular, the reduction to condensed form, which is the most time consuming phase. This includes reductions to Hessenberg, tridiagonal, and bidiagonal forms, each of which is the result of a similarity transformation for general eigenvalues, symmetric eigenvalues, or the singular value decomposition, respectively. The standard one-stage reduction algorithms have been challenged by the method of splitting the reduction phase into multiple stages.

One of the first uses of a two-stage reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [16]. Then, a multi-stage method was used to reduce a matrix to tridiagonal, bidiagonal and Hessenberg forms [22]. 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 Reduction (SBR) was designed [8], 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 bandwidth is reached. The off-diagonal elements are then annihilated, which produces fill-in blocks or bulges that need to be chased down toward the bottom right corner of the matrix, hence this stage is termed “bulge chasing.” If eigenvectors are required in addition to the eigenvalues, the bulge chasing transformations can be efficiently accumulated through Level 3 BLAS operations and used to generate the eigenvectors. Communication bounds for such two-sided reductions have been established under the Communication Avoiding framework [6]. A multi-stage approach has also been applied to the Hessenberg reduction [21] as well as the QZ algorithm [20] for the generalized non-symmetric eigenvalue problem.

Tile algorithms have recently seen a rekindled interest when applied to the two-stage tridiagonal [17, 24] and bidiagonal reductions [23]. The first stage is implemented using high performance kernels combined with a data translation layer to execute on top of the tile data layout format. The second stage is implemented based on cache-aware kernels and a task coalescing technique [17]. The performance achieved is far greater than other available implementations. 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. [26] presented novel hybrid reduction algorithms for the two-sided factorizations, which take advantage of the high bandwidth of the GPU by offloading the expensive Level 2 BLAS operations of the panel factorization to the GPU. Bientinesi et al. [7] instead 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 from band to tridiagonal) still remains on the host though. Vomel et al. [28] extended the tridiagonalization approach in [26] to the symmetric standard eigenvalue problem, and Dong et al. [12] to multi-GPUs.

ELPA [5], a recent distributed-memory eigensolver library, was developed for electronic structure codes. ELPA is based on 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 part of it. Haidar et al. [18] developed a two-stage approach for multicore and single GPU. The main thrust of the work presented here is the extension of this two-stage approach to multi-GPUs.

3. MAIN CONTRIBUTIONS

We developed the following new algorithms, software, and techniques for heterogeneous multi-GPU computing:

- **Efficient and scalable multi-GPU BLAS**, including the HER2K and HEMM Level 3 BLAS kernels;
- **Multi-GPU two-stage tridiagonalization**, utilizing techniques to map computational tasks to the strengths of heterogeneous hardware components, and to overlap computation on GPUs with computation on CPUs;
- **Hierarchical multi-GPU communication model** to optimize communication for multi-GPUs, which can be applied in general, beyond the scope of the algorithms developed;
- **Examining trade-offs between communication and extra computation** to reduce overall execution time, which we believe will become increasingly important for current and up-coming hardware.

Further, the algorithms presented are included in MAGMA [1], an open source library of next generation LAPACK-compliant linear algebra routines for hybrid GPU-CPU architectures.

4. BACKGROUND

To solve a Hermitian (symmetric) eigenproblem of the form $Ax = \lambda x$, finding its eigenvalues Λ and eigenvectors Z such that $A = Z\Lambda Z^*$, where * denotes conjugate-transpose, the standard algorithm follows three steps [2,15,25]. First, reduce the matrix to tridiagonal form using an orthogonal transformation Q such that $A = QTQ^*$, where T is a tridiagonal matrix (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"). 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. Thus, in this paper, we will focus on improving the reduction to tridiagonal step, and present a multi-GPU implementation of it.

4.1 Tridiagonal reduction

Several approaches exist to compute the tridiagonalization of a dense matrix.

4.1.1 Standard approach

Standard software for reducing a symmetric dense matrix to tridiagonal form is available in LAPACK [4] and in MAGMA [1] through the ZHETRD (DSYTRD) routine. This approach suffers from a lack of efficiency, and so it is important to first identify the bottlenecks of this standard one-stage approach. It is characterized by iterating two computational phases: the panel factorization and the update of the trailing submatrix. The panel factorization computes the transformations (reflectors) to introduce zeros below the subdiagonal within a block column using memory-bound Level 2 BLAS operations. The trailing submatrix update applies the accumulated block reflector using compute-bound Level 3 BLAS operations. 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 computational phases and memory transfers between the CPU and GPU. The panel factorization step is actually the critical phase because computing each reflector column of the panel requires a substantial symmetric matrix-vector product involving the trailing submatrix. Although this approach scales on multi-GPUs, the performance is memory-bound [12].

4.1.2 Two-stage approach

Because of the expense of the reduction step, renewed research has focused on improving this step, resulting in a novel technique based on a two-stage reduction [17]. We developed a two-stage algorithm for one GPU [18] which achieved good performance compared to the standard approach [12, 26, 28], motivating us to focus in this paper on developing and optimizing a multi-GPU two-stage approach.

The two-stage reduction is designed to increase the utilization of compute-intensive operations. Many algorithms have been investigated using this two-stage approach [8, 9]. The idea is to split the original one-stage approach into a compute-intensive phase (first stage) and a memory-bound phase (second or "bulge chasing" stage). The first stage reduces the original symmetric dense matrix to a symmetric band form, while the second stage reduces from band to tridiagonal form.

First stage.

The first stage applies a sequence of block Householder transformations to reduce a symmetric dense matrix to a symmetric band matrix. This stage uses compute-intensive matrix-multiply kernels, eliminating the memory-bound matrix-vector product in the one-stage panel factorization, and has been shown to have a good data access pattern and large portion of Level 3 BLAS operations [8, 13, 14, 18]. It also enables the efficient use of GPUs by minimizing communication and allowing overlap of computation and communication. Given a dense $n \times n$ symmetric matrix A, the matrix is divided into $nt = n/b$ block-columns of size b. The algorithm proceeds panel by panel, performing a QR decomposition for each panel to generate the Householder reflectors V (i.e., the orthogonal transformations) required to zero out elements below the bandwidth b. Then the generated block Householder reflectors are applied from the left and the right to the trailing symmetric matrix, according to

$$A = A - WV^* - VW^*, \tag{1}$$

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

$$\begin{aligned} W &= X - \tfrac{1}{2}VT^*V^*X, \text{ where} \\ X &= AVT. \end{aligned} \tag{2}$$

Second stage.

The second 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 use the novel bulge chasing algorithm (reduction from band to tridiagonal) developed in [17]. This algorithm is designed to extensively use cache friendly kernels combined with fine grained, memory aware tasks in an out-of-order scheduling technique which considerably enhances data locality. This reduction has been designed for multicore architectures, and results have shown its efficiency. This step has been well optimized such that it takes between 5% to 10% of the global time of the reduction from dense to tridiagonal. We refer the reader to [17,18] for a detailed description of the bulge chasing algorithm.

We decide to develop a hybrid CPU-GPU implementation of only the first stage of this algorithm, and leave the second stage executed entirely on the CPU. The main motivation is that the first stage is the most expensive computational phase of the reduction. Results show that 90% of the time is spent in the first stage reduction. Another motivation for this direction is that accelerators perform poorly when dealing with memory-bound fine-grained computational tasks (such as bulge chasing), limiting the potential benefit of a GPU implementation of the second stage.

5. MULTI-GPU ALGORITHM

Our hybrid CPU-GPU algorithm is similar to the two-stage tridiagonal reduction developed for multicore CPUs [17]. However, a hybrid CPU-GPU implementation needs to take advantage of the GPUs by offloading expensive operations to the GPUs, while overlapping the CPU and the GPU computation and minimizing communication. In other words, our goal is to keep the GPUs busy, making the CPUs wait for the GPUs, and not the reverse. The relatively slow CPU computation and the communications with the GPUs are hidden by overlapping them with the GPU computation. To characterize our work, we start by describing the single GPU implementation of the reduction to banded form.

5.1 Hybrid CPU–single GPU band reduction

The hybrid CPU-GPU algorithm is illustrated in Figure 1. We first run the QR decomposition (GEQRT kernel) of a panel on the CPUs. Once the panel factorization of step i is finished, then we compute W on the GPU, as defined by equation (2). This kernel is called xPNCXW. Its computation involves a matrix-matrix multiplication (GEMM) to compute VT, then a Hermitian matrix-matrix multiplication to compute $X = AVT$ (HEMM), which is the dominant cost of computing W, consisting of 95% of the time spent in computing W, and finally another inexpensive GEMM. Once W is computed, the trailing matrix update (applying transformations on the left and right) defined by equation (1) can be performed using a rank-$2k$ update (HER2K). Note that for real symmetric matrices, the operations corresponding to HEMM and HER2K are SYMM and SYR2K.

However, to allow overlap of CPU and GPU computation, the trailing submatrix update is split into two pieces. First, the next panel for step $i+1$ (green panel of Figure 1) is updated using two GEMMs on the GPU. The corresponding kernel is called xPNRFB. Next, the remainder of the trailing submatrix (orange triangle of Figure 1) is updated using a HER2K. While the HER2K is executing, the CPUs receive the panel for step $i+1$, perform the next panel factorization (GEQRT), and send the resulting V_{i+1} back to the GPU. In this way, the factorization of panels $i = 2, \ldots, nt$ and the associated communication are hidden by overlapping with GPU computation, as demonstrated in Figure 2.

This is similar to the look-ahead technique typically used in the one-sided dense matrix factorizations. Figure 2 shows a snapshot of the execution trace of the reduction to band

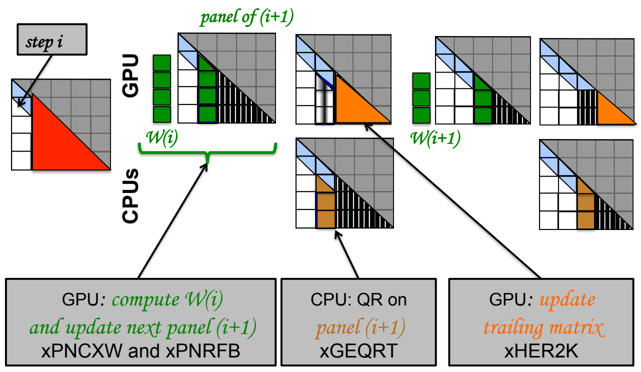

step i
panel of (i+1)
GPU
CPUs
W(i)
W(i+1)
GPU: compute W(i) and update next panel (i+1) xPNCXW and xPNRFB
CPU: QR on panel (i+1) xGEQRT
GPU: update trailing matrix xHER2K

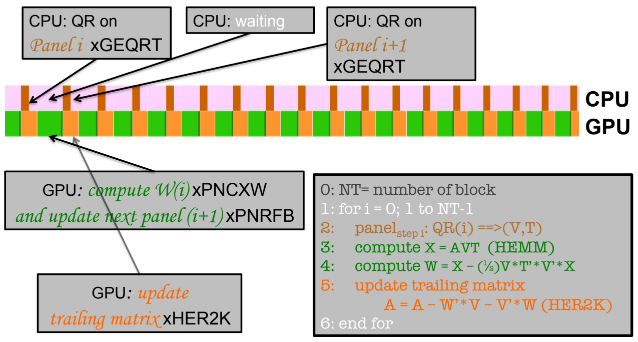
Figure 1: Description of the reduction to banded form, stage 1.

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, while the lower performance CPUs wait for the GPU as necessary. We compare hybrid GPU-CPU implementations of the one-stage and two-stage algorithms in Figure 3, where the two-stage algorithm is about six times faster than the standard one-stage approach. Also, both hybrid implementations are significantly faster than the CPU-only implementation available in the Intel MKL.

5.2 Hybrid multi CPU-GPU band reduction

In this section, we describe the development of our multi-GPU algorithm, including the constraints to consider, how to deal with communication, and how to achieve good scalability over both CPUs and GPUs. Our target algorithm, the first stage reduction to band form, involves two main multi-GPU kernels: the computation of W and the trailing submatrix update.

Recall that the computation of W involves two small GEMMs (5% of the time to compute W), and one expensive HEMM (95% of the time). Rather than computing the two small GEMMs on one GPU and broadcasting the results to the other GPUs, it is faster to duplicate these two small computations on all the GPUs. A scalable implementation of the multi-GPU HEMM, in which a data distribution is chosen in order to minimize communications, is described below.


CPU: QR on Panel i xGEQRT
CPU: waiting
CPU: QR on Panel i+1 xGEQRT
CPU
GPU
GPU: compute W(i) xPNCXW and update next panel (i+1) xPNRFB
GPU: update trailing matrix xHER2K

```
0: NT= number of block
1: for i = 0; 1 to NT-1
2:     panel_{step i}: QR(i) ==>(V,T)
3:     compute X = AVT  (HEMM)
4:     compute W = X - (½)V*T'*V'*X
5:     update trailing matrix
           A = A - W'*V - V'*W (HER2K)
6: end for
```

Figure 2: Execution trace of reduction to banded form.

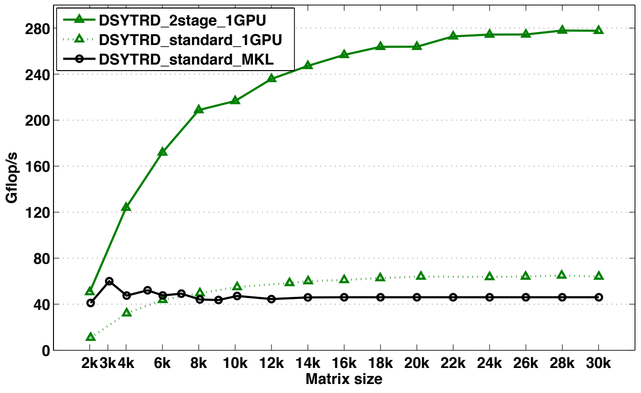


Figure 3: Comparison between the standard and the two-stage approaches for the reduction to tridiagonal. The reference Gflops formula is $4n^3/3$ for all.

As previously described for the single GPU implementation, the trailing submatrix update is split into two parts. First we update the next panel, which is a block column belonging to a single GPU, so it can be updated with two GEMMs on a single GPU. Then a multi-GPU HER2K updates the rest of the trailing matrix, which is distributed across multiple GPUs.

5.2.1 *Data distribution*

In the LAPACK storage for Hermitian and symmetric matrices, only one of the lower or upper triangles of A has valid entries; the other triangle is not referenced. We shall assume that the lower triangle of A is referenced. Due to the low communication bandwidth between GPUs relative to their high floating point performance, a 1-D block-column cyclic distribution of the matrix was chosen, as shown in Figure 4. However, the block size for distribution does not need to be the same as the bandwidth. It may be more efficient to have a larger block size for distribution than the bandwidth. This only slightly complicates the code, as the first and last block can be partial blocks.

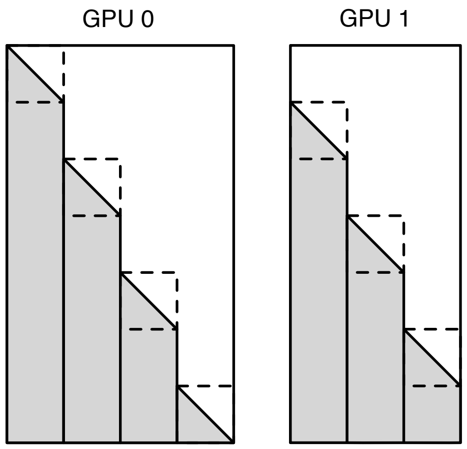


Figure 4: Block-cyclic data distribution. Shaded areas contain valid data. Dashed lines indicate diagonal blocks.

5.2.2 *Multi-GPU HER2K (SYR2K)*

The Hermitian rank-$2k$ update implements the operation $A = A - VW^* - WV^*$, where A is an $n \times n$ Hermitian matrix, and V and W are $n \times k$ matrices. After distribution, the portion of A on each GPU no longer appears as a symmetric matrix, but instead has a stepped appearance, as in Figure 4. Because of this stepped storage, the multi-GPU HER2K cannot be assembled purely from regular HER2K calls on each GPU. Instead, each block column must be processed individually. The diagonal blocks require special attention. In the BLAS standard, elements above the diagonal are not accessed; the user is free to store unrelated data there and the BLAS library will not touch it. To achieve this, one can use a HER2K to update each diagonal block, and a GEMM to update the remainder of each block column below the diagonal block. However, these small HER2K operations have little parallelism and so are inefficient on a GPU. This can be improved to some degree by using multiple streams to execute several HER2K updates simultaneously. However, because we have copied the data to the GPU, we can consider the space above the diagonal to be a scratch workspace. Thus, we update the entire block column, including the diagonal block, writing extra data into the upper triangle of the diagonal block, which is subsequently ignored. We do extra computation for the diagonal block, but gain efficiency overall by launching fewer BLAS kernels on the GPU and using the more efficient GEMM kernels, instead of small HER2K kernels.

```
1: for $i = 1$ to $nt$ do
2:     if this GPU owns block-column $i$ then
3:         $A_{i:nt,i} \mathrel{-}= V_{i:nt} W_i^*$
4:     end if
5: end for
6: for $i = 1$ to $nt$ do
7:     if this GPU owns block-column $i$ then
8:         $A_{i:nt,i} \mathrel{-}= W_{i:nt} V_i^*$
9:     end if
10: end for
```

Algorithm 1: Multi-GPU Hermitian rank-$2k$ update.

We replicate the V and W matrices on all the GPUs. Each block column of A, denoted $A_{i:nt,i}$, is updated with two GEMMs, as shown in Algorithm 1. These can be done in multiple streams, to allow multiple simultaneous GEMMs when a single GEMM does not fill up the GPU. The code loops over all block columns and executes the required GEMMs on the GPU that owns each block column. Other than broadcasting V and W to all GPUs, there is no additional communication required during the HER2K operation, so it scales well to multiple GPUs, as demonstrated in Figure 5.

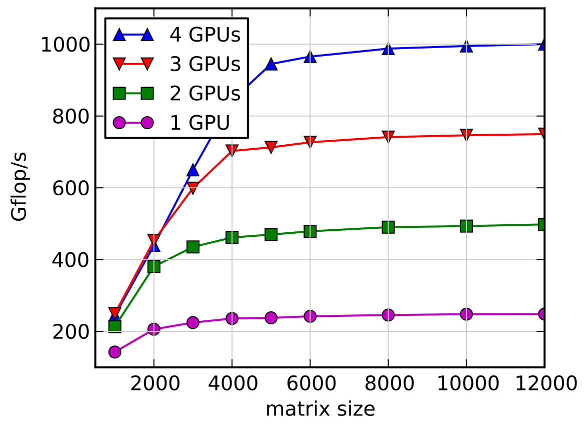


Figure 5: Multi-GPU DSYR2K scales perfectly with number of GPUs.

5.2.3 *Multi-GPU HEMM (SYMM)*

In this section, we will discuss the different implementations for the HEMM kernel, which is one of the two primary kernels of the reduction to band algorithm. We recall that the role of HEMM is to compute $X = AVT$, where A is an $n \times n$ Hermitian matrix and $B = VT$ is a block-column matrix of size $n \times nb$. We mention that only the lower or the upper triangle of the matrix A is referenced, the opposite

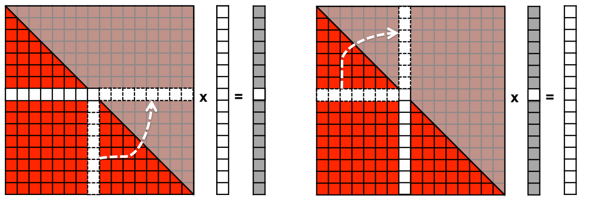

(a) Row-oriented schema (b) Column-oriented schema

Figure 6: Representation of a one GPU HEMM for both Row or Column oriented schema.

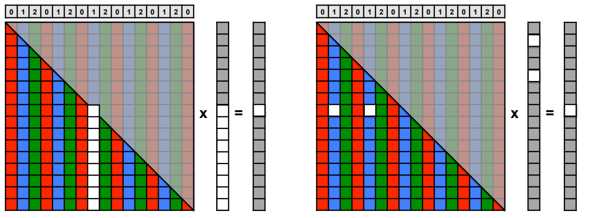
triangle being obtained by symmetry. Two loop orderings can be considered when implementing this kernel.

Row-oriented.

A row-oriented HEMM multiplies block-row $A_{i,:}$ by B to compute each block X_i, for $i = 1$ to nt, as illustrated in Figure 6a. Elements of A right of the diagonal are obtained by transposing block-column i below the diagonal (assuming lower-triangular storage). The multi-GPU implementation of this design is given in Algorithm 2 and diagrammed in Figure 7. This implementation suffers from a lack of parallelism due to the small size of its output (small square $nb \times nb$ block) which is known as an *inner product* GEMM and cannot perform efficiently on GPUs (compared to *outer product* GEMM). Because the data is distributed over the GPUs, each GPU computes a partial $X^{(g)}$. A sum-allreduce across the GPUs is then required to accumulate $X = \sum_{g=1}^{\text{ngpu}} X^{(g)}$ and distribute the final result back to all the GPUs.

```
1: $X = 0$
2: for $i = 1$ to $nt$ do
3:     $X_i$ += $A_{i,\text{myblk}} B_{\text{myblk}}$,
4:     if this GPU owns block-column $j$ then
5:         $X_i$ += $A^*_{i:nt,i} B_{i:nt}$,
6:     end if
7: end for
8: sum–allreduce $X$
```

Algorithm 2: Row-oriented multi-GPU HEMM. myblk denotes blocks owned by current GPU.



(a) X_i += $A^*_{i:nt,i} B_{i:nt}$ (b) X_i += $A_{i,\text{myblk}} B_{\text{myblk}}$

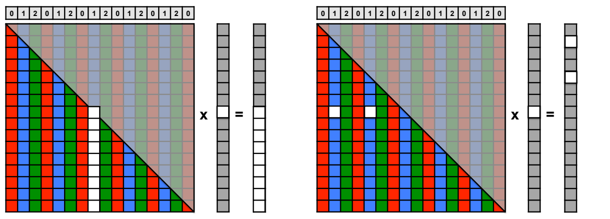
Figure 7: Block-row i of row-oriented multi-GPU HEMM writes small result with little parallelism. Each color represents different GPU.

Column-oriented.

A column-oriented HEMM multiplies block-column $A_{:,j}$ by block B_j to compute a partial sum $X^{(j)}$, for $j = 1$ to nt, as illustrated in Figure 6b. Elements of A above the diagonal are obtained by transposing block-row j left of the diagonal. The multi-GPU implementation of this design is given in Algorithm 3 and diagrammed in Figure 8. It has a large amount of parallelism because each block-column GEMM results in a large $(n - j * nb) \times nb$ block. Again, each GPU computes a partial $X^{(g)}$, which requires a sum-allreduce across the GPUs to accumulate X and distribute the final result back to all the GPUs. This HEMM algorithm is used in our multi-GPU implementation (version 1 and 2), as detailed in Section 5.2.4.

```
1: $X = 0$
2: for $j = 1$ to $nt$ do
3:     $X_{\text{myblk}}$ += $A^*_{j,\text{myblk}} B_j$,
4:     if this GPU owns block-column $j$ then
5:         $X_{j:nt}$ += $A_{j:nt,j} B_j$
6:     end if
7: end for
8: sum–allreduce $X$
```

Algorithm 3: Column-oriented multi-GPU HEMM.



(a) $X_{j:nt}$ += $A_{j:nt,j} B_j$ (b) X_{myblk} += $A^*_{j,\text{myblk}} B_j$

Figure 8: Block-column j of column-oriented multi-GPU HEMM writes large result with significant parallelism.

5.2.4 *Communication schema*

Reducing communication or overlapping communication with computation is critical in order to maximize the time GPUs spend in compute-intensive kernels, especially for multi-GPU implementations. In the context of our algorithm, the assembly of $X^{(g)}$ of the HEMM described above becomes a synchronization point because the HER2K depends on the result of the HEMM, preventing overlap of communication and computation. Our goal is therefore to minimize the communication time. We investigate various implementations of the communication model that could be used for the sum-allreduce and discuss the trade-offs in each.

Model 1.

In our first implementation, all the GPUs send their $X^{(g)}$ to the CPU, which does the sum-allreduce and then broadcasts the final result X back to all the GPUs. Figure 9 shows an execution trace of the HEMM and its communication with this approach on three GPUs.

To understand the communication behavior requires understanding the underlying hardware topology. Systems with multiple GPUs often use PCIe switches or I/O hubs, with several GPUs connected to each switch. This is the case, for instance, with an NVIDIA S2070, which has four GPUs connected, in pairs, to two PCIe switches. GPUs connected to a PCIe switch share bandwidth to the host CPU. As a result, sending data from multiple GPUs requires the same total time, whether sent in parallel or sequentially.

In the context of our HEMM, we choose to send the data sequentially, so that once the CPU receives data from the first two GPUs, it can start performing the sum-allreduce while receiving the next data. However, the communication time increases linearly with the number of GPUs, creating a severe bottleneck that limits the scalability of this approach. While adding GPUs decreases the HEMM compute time, this is not sufficient to offset the increased communication time. This serialization of communication is easily observed in Figure 9, and from this understanding of the communication behavior, we expect poor performance as the number of GPUs increases. We refer to our implementation of the column-oriented HEMM using this communication model as version 1.

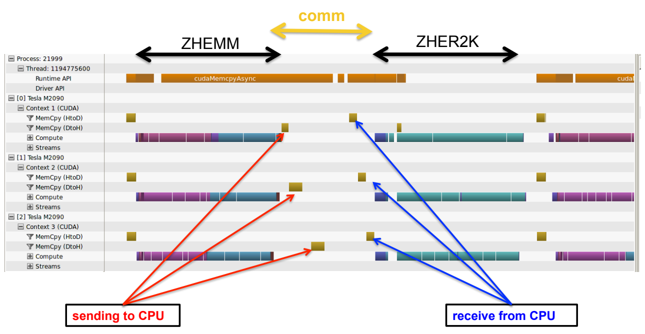


Figure 9: Trace of the execution of ZHEMM using communication model 1 when the CPU is responsible for the communication.

Model 2: **Hierarchical Communication Model.**

To address this increase in communication time, we developed a second, hierarchical, communication schema. Each PCIe switch 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" with each other or with the master; between two nodes, the master GPUs communicate directly together. This hierarchical communication model is easily adaptable to a distributed environment where the communication between masters of different nodes should be done via the CPU using MPI. This communication model is depicted in Figure 10. Thus, in the context of our HEMM, the sum-allreduce is first done independently by the master GPU within each node, where all the GPUs send their $X^{(g)}$ to the master GPU, which does the addition. After receiving contributions from GPUs within each node, the master GPUs exchange data and each, redundantly, makes the final addition. Then, each master independently broadcasts the final result X to all GPUs within its node.

We refer to our implementation of the column-oriented HEMM using this communication model as version 2. Similarly to version 1, we depict in Figure 11 the execution trace of the HEMM with this communication schema. Here, GPU 0 belongs to node 0, while GPU 1 and GPU 2 belong to node 1. Hence, within each node the communication is

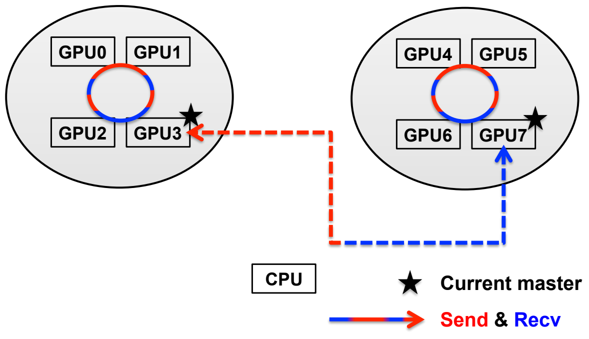


Figure 10: Mapping communication to hardware.

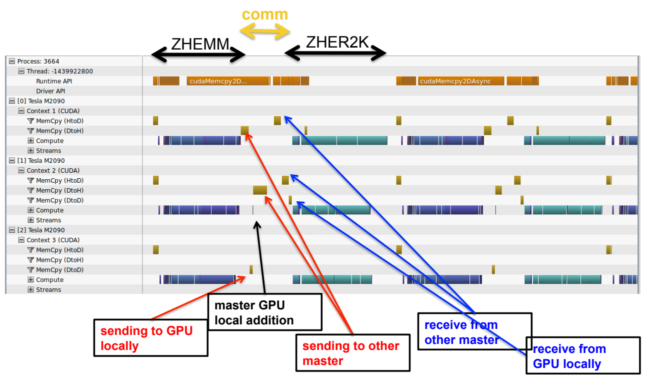


Figure 11: Trace of the execution of ZHEMM using the Hierarchical Communication Model.

done independently and the exchange of data between nodes is reduced to master communication, significantly reducing the communication time.

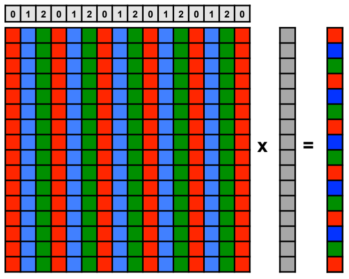
In Table 1, we report the time in seconds needed by the reduction from dense to band form using these two communication models when varying the number of GPUs. As expected, the second model of communication has good performance and good scalability compared to the first one. We note here that the size of the matrix is too small to achieve linear scaling up to five GPUs.

	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs
version 1	60.1	43.2	36.7	34.0	33.2
version 2	60.0	29.1	21.8	21.2	20.4
version 3	67.3	30.2	21.7	19.4	16.3

Table 1: Comparison of three versions of reduction to band algorithm, showing time in seconds while varying number of GPUs for matrix of size $n = 15000$ in double complex precision.

5.2.5 *Alternative implementation*

An alternative third implementation that minimizes the communication can be of interest. Assume that the Hermitian matrix is available in both its lower and upper part. Thus a row-oriented large HEMM can be applied as a single GEMM operation on each GPU. Because the matrix is assumed to be full, then the result of this HEMM is large, of size $n \times n/ngpu$, and thus the resulting parallelism is enough to get good performance. The resulting X_{myblk} is the final result of X for the blocks belonging to each GPU, and thus there is no need for a sum-allreduce. Each GPU needs to broadcast only its computed blocks of X to the other GPUs. The amount of communication is divided by the number of GPUs, resulting in significantly less communication than the two approaches described above. Each GPU must broadcast $n/(nb \times \text{ngpu})$ blocks instead of n/nb blocks. We use the same hierarchical communication model described above in Figure 10 to organize this communications in an optimized fashion. The assembly inside each node is done by local GPU-GPU communication independently from other nodes, while assembly between nodes is done via the master GPU of each node. In Figure 12, we depict a representation of this implementation of HEMM. Note that the number of operations remains the same; the only requirement is that the symmetric part of the matrix needs to be available.



$$X_{\text{myblk}} = A^*_{:,\text{myblk}} B$$

Figure 12: A fast multi-GPU implementation of an HEMM minimizing communication and increasing parallelism.

The main difficulty here is to have the symmetric part of the matrix. To do this we have two choices. First is to symmetrize the matrix just before starting the HEMM kernel. However, this requires significant communication and we would lose any gains we achieved, so we disregard this choice. Second is to symmetrize the matrix *once*, and modify the HER2K so that it maintains the symmetry by updating both the lower and upper parts of the matrix. This incurs no extra communication but does twice the number of operations. We trade off extra computation in the HER2K for a large reduction in communication cost in the HEMM, particularly for a large number of GPUs. This is further justified because the HER2K is overlapped with the panel factorization on the CPUs; with enough GPUs, the HER2K becomes faster than the panel, so the GPUs must wait for the panel instead of the HER2K. Thus, adding extra work to the HER2K will not impact the overall execution time so long as it is faster than the panel.

In Figure 13, we depict the execution trace of this approach, showing that the communication time becomes marginal. We also report in Table 1 the time in seconds for the reduction to band form, and compare it to the previous two versions defined above. We see that for more than three GPUs, the extra work needed by this approach is hidden, and it is able to reach better performance than the two previous versions. For one GPU, it is normal that version 3 is slower, as no communication is needed while the HER2K has twice as many operations, but it is a good example to show

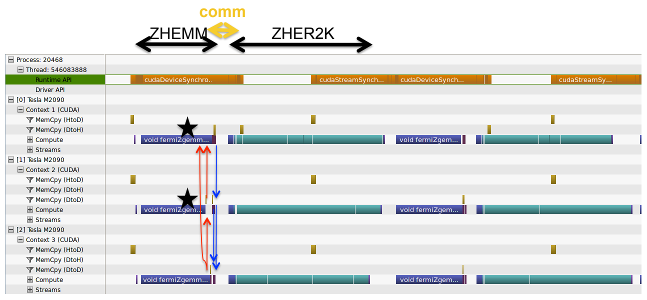


Figure 13: Trace of the execution of ZHEMM using the Hierarchical Communication Model.

that the extra cost is not prohibitive. It is 12% of the time on one GPU, and so becomes less than 6% on two GPUs and so on. Thus we consider version 3 also as interesting as version 2.

6. EXPERIMENTAL RESULTS

This section presents the performance comparisons of our hybrid multi CPU-GPU two-stage TRD against the multi-GPU standard TRD approach, and against the state-of-the-art multicore numerical linear algebra libraries.

6.1 Experimental Environment

Our experiments were performed on the largest shared-memory system we could access at the time of writing this paper. It is representative of a vast class of servers and workstations commonly used for computationally intensive workloads. We benchmark all implementations on an Intel multicore system with dual-socket, 8 core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket has a 24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1. The system is equipped with 52 Gbytes of memory. The theoretical peak for this architecture in double precision is 20.8 Gflop/s per core. The system is also equipped with eight NVIDIA M2090 cards with 6 Gbytes per card running at 1.3 GHz, connected to the host via PCIe I/O hubs at 8 Gbytes/s bandwidth.

There are a number of software packages that implement the tridiagonal reduction. For comparison, we used the MKL (Math Kernel Library) [19] which is a commercial software from Intel that is a highly optimized programming library. It includes a comprehensive set of mathematical routines implemented to run well on multicore processors. In particular, MKL includes the LAPACK-equivalent routine performing the tridiagonal reduction DSYTRD. The MAGMA [1] library is used for the multi-GPU standard approach. We recall that the algorithmic complexity of the standard full tridiagonal reduction is $\frac{4}{3}N^3$ and this is used as a reference to draw the performance of all the graphs we show.

6.2 Effect of block size or bandwidth

Both stages in the two-stage reduction to tridiagonal form algorithm depend strongly on the bandwidth size. The reduction to band form can be achieved with efficient Level 3 BLAS operations for large bandwidth size, e.g., $kb > 40$, while the bulge chasing is a memory-bound algorithm, meaning that it performs better for small kb rather than for large kb. The bulge chasing stage has been implemented [17] with cache-friendly kernels and smart scheduling to allow good performance for large $kb > 40$, but it is recommended to keep kb as small as possible. Thus, the choice of bandwidth is a compromise between these two competing factors. Overall, a bandwidth of 64 was found to be a good compromise.

6.3 Performance results

We compare the performance of our hybrid multi-GPU two-stage TRD algorithm against the one-stage approach from optimized state-of-the-art numerical linear algebra libraries. In particular, Figure 14 shows the comparison with (1) the DSYTRD routine from the Intel MKL, and (2) the hybrid multi-GPU DSYTRD from MAGMA. Our implementation asymptotically achieves more than a $7\times$ speedup on one GPU and more than a $20\times$ speed up on six GPUs against MKL. It also achieved a significant performance compared to its multi-GPU counterpart using the standard one-stage approach. It is around 4 times faster than the multi-GPU standard approach. The fact that the first stage is a compute intensive stage, and the fact that the implementation design maps both the algorithm and the communication to the hardware, allows this algorithm to achieve good scalability. We believe that this attractive behavior makes our algorithm a good candidate for the next generation of distributed multi-GPU machines.

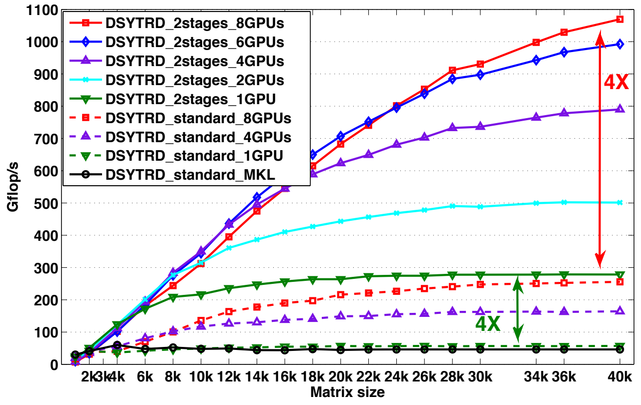


Figure 14: Performance comparison of our two-stage multi-GPU implementation to standard one-stage MKL and MAGMA implementations, in double precision.

6.4 Scalability

Figure 15 shows the performance scalability of the reduction from dense to band. The curves show performance in terms of Gflop/s. We note that this also reflects the elapsed time, e.g., a performance that is two times higher, corresponds to an elapsed time that is two times less. The multi-GPU implementation shows very good scalability. We mention here that Figure 15 shows the performance for only the reduction to band form. We show this graph here only to highlight the scalability of the multi-GPU implementation of the reduction to band. The performance shown in Figure 14 (the full reduction to tridiagonal form) drops slightly compared to Figure 15, because it consists of the time to reduce the matrix to band added to the time to reduce the band to tridiagonal, which, as pointed out earlier, is a multicore only implementation, and so, although it is an efficient CPU code, its elapsed time does not change when the number of GPUs increases. However, the overall performance of our implementation scales very well. We also mention that when integrated into the full eigensystem solver, the second stage will be less than 5% of the time, and so its effect on performance will further decrease.

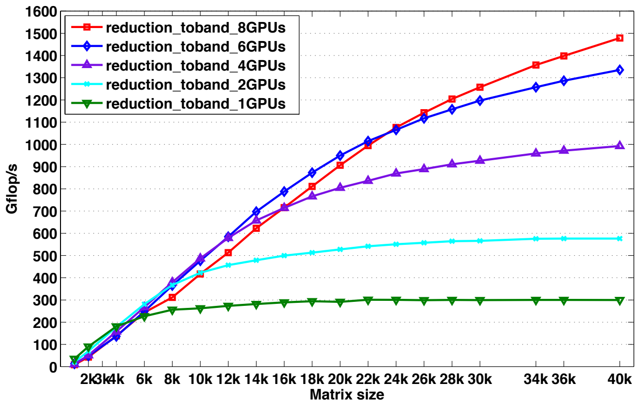


Figure 15: Scalability of our multi-GPU implementation of the reduction to band (first stage) when the number of GPUs increases from one to eight, in double precision.

7. CONCLUSIONS

We demonstrated that it is possible to transform a classical, memory-bound algorithm into a compute-bound algorithm, even on systems with an enormous gap between their computing power and interconnection bandwidth. Moreover, we developed and presented the techniques that make this possible for a highly demanded algorithm that naturally features communication and complex data dependencies. A key technique is to map the algorithm to best take advantage of heterogeneous hardware. In the context of the tridiagonal reduction, we chose a two-stage algorithm due to its richness in compute-intensive kernels, which are scheduled on the GPU, while the memory-bound bulge-chasing stage is scheduled on the CPU using cache-friendly kernels. We also developed a hierarchical communication model for multi-GPUs, based on the underlying communication topology of the host PCIe system.

We believe that these techniques will increase in relevance for upcoming architectures. Indeed, our experience over the last few years, and the results here, show that although GPUs were used to accelerate some standard memory-bound algorithms more than $10\times$, the improvements stagnated from one generation of GPUs to the next, following the marginal improvements in GPU bandwidth. Thus techniques such as trading-off between communication and extra computation will increase in relevance as computation continues to become exponentially cheaper than communication.

We plan to further study the implementation of multi-GPU algorithms in a distributed computing environment. We believe that the techniques presented will become more popular and will be integrated into dynamic runtime system technologies. The ultimate goal is that this integration will help to tremendously decrease development time while retaining high-performance.

Acknowledgments

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

8. REFERENCES

[1] MAGMA 1.3. `http://icl.cs.utk.edu/magma/`, 2012.

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

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

[4] 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*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[5] 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.*, 37(12):783–794, 2011.

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

[7] P. Bientinesi, F. D. Igual, D. Kressner, and E. S. Quintana-Ortí. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. PPAM'09, pages 387–395, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Transactions on Mathematical Software*, 26(4):602–616, 2000.

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

[10] 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*. Society for Industrial and Applied Mathematics, PA, 1997.

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

[12] T. Dong, J. Dongarra, T. Schulthess, R. Solca, S. Tomov, and I. Yamazaki. 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).

[13] 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*, 27(1-2):215 – 227, 1989.

[14] W. Gansterer, D. Kvasnicka, and C. Ueberhuber. Multi-sweep algorithms for the symmetric eigenproblem. In *Vector and Parallel Processing - VECPAR'98*, volume 1573 of *Lecture Notes in Computer Science*, pages 20–28. Springer, 1999.

[15] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.

[16] R. G. Grimes and H. D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14:241–256, September 1988.

[17] 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*, pages 8:1–8:11, New York, NY, USA, 2011. ACM.

[18] A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess. 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).

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

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

[21] 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.

[22] B. Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25(7):845–860, 1999.

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

[24] 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.

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

[26] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.*, 36(12):645–654, 2010.

[27] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[28] C. Vomel, S. Tomov, and J. Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal on Scientific Computing*, 34(2):C70–C82, 2012.