CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3152

Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems

Ichitaro Yamazaki [1,*,†], Tingxing Dong [1], Raffaele Solcà [2], Stanimire Tomov [1], Jack Dongarra [1] and Thomas Schulthess [2]

[1] *Electrical Engineering and Computer Science, University of Tennessee, Knoxville, Tennessee, U.S.A.*
[2] *Institute for Theoretical Physics and Swiss National Supercomputer Center, Eidgenössische Technische Hochshule (ETH), Zürich, Switzerland*

SUMMARY

For software to fully exploit the computing power of emerging heterogeneous computers, not only must the required computational kernels be optimized for the specific hardware architectures but also an effective scheduling scheme is needed to utilize the available heterogeneous computational units and to hide the communication between them. As a case study, we develop a static scheduling scheme for the tridiagonalization of a symmetric dense matrix on multicore CPUs with multiple graphics processing units (GPUs) on a single compute node. We then parallelize and optimize the Basic Linear Algebra Subroutines (BLAS)-2 symmetric matrix-vector multiplication, and the BLAS-3 low rank symmetric matrix updates on the GPUs. We demonstrate the good scalability of these multi-GPU BLAS kernels and the effectiveness of our scheduling scheme on twelve Intel Xeon processors and three NVIDIA GPUs. We then integrate our hybrid CPU-GPU kernel into computational kernels at higher-levels of software stacks, that is, a shared-memory dense eigensolver and a distributed-memory sparse eigensolver. Our experimental results show that our kernels greatly improve the performance of these higher-level kernels, not only reducing the solution time but also enabling the solution of larger-scale problems. Because such symmetric eigenvalue problems arise in many scientific and engineering simulations, our kernels could potentially lead to new scientific discoveries. Furthermore, these dense linear algebra algorithms present algorithmic characteristics that can be found in other algorithms. Hence, they are not only important computational kernels on their own but also useful testbeds to study the performance of the emerging computers and the effects of the various optimization techniques. Copyright © 2013 John Wiley & Sons, Ltd.

Received 8 November 2012; Revised 3 September 2013; Accepted 4 September 2013

KEY WORDS: dense linear algebra; GPU accelerators; symmetric tridiagonal reduction; symmetric matrix-vector multiplication; parallel eigensolver

1. INTRODUCTION

Emerging high-performance computers are based on heterogeneous multicore architectures, where each compute node consists of multicore or manycore CPUs, and accelerators or coprocessors like NVIDIA GPUs, Advanced Micro Devices, Inc. (AMD) Fusion accelerated processing unit, or Intel Xeon Phi coprocessors. For example, the Keeneland Initial Delivery System [1] at the Georgia Institute of Technology has 120 compute nodes, each of which consists of two six-core Intel Xeon processors and three NVIDIA Fermi GPUs. For software to fully utilize the computing power of such heterogeneous architectures, computational kernels running at the compute node level of the software stack must be redesigned to utilize the computing power of the heterogeneous node architecture.

*Correspondence to: Ichitaro Yamazaki, Electrical Engineering and Computer Science, University of Tennessee, Knoxville, Tennessee, U.S.A.
†E-mail: ic.yamazaki@gmail.com

Linear Algebra Package (LAPACK) [2] is a set of dense linear algebra routines on shared-memory CPUs. These routines are used extensively as important building blocks in many scientific and engineering simulations on shared-memory and distributed-memory computers. Matrix Algebra on GPU and Multicore Architectures (MAGMA) [3] extends LAPACK to heterogeneous architectures, and its performance is critical for these simulations to utilize the computing power of such architectures. To exploit the heterogeneous architecture, MAGMA is based on a hybrid programming paradigm and a static scheduling scheme; namely, an algorithm is first split into small computational tasks, which are then statically scheduled either on the CPUs or GPUs to match the algorithmic requirements of different computational tasks to the architectural strengths of different computational units. To obtain high performance of these individual computational tasks, MAGMA uses threaded Basic Linear Algebra Subroutines (BLAS) [4] on the CPUs and MAGMA BLAS [5] on the GPUs, which are optimized for a specific CPU and GPU architecture, respectively. By carefully scheduling these tasks on the CPUs and GPUs, MAGMA obtains significant speedups over vendor-optimized LAPACKs [6–8]. Moreover, because many dense linear algebra algorithms present algorithmic characteristics that are found in other algorithms, MAGMA is not only an important practical package on its own but it also provides a useful testbed to study the performance of emerging computers and the effects of various optimization techniques. [‡]

As a case study, in this paper, we examine the tridiagonal reduction of a symmetric dense matrix, which is implemented in the MAGMA routine **xSYTRD** and utilizes a single GPU (where **x** can be either **S**, **D**, **C**, or **Z** denoting either single, double, single-complex, or double-complex precision used for the reduction, respectively).[§] This symmetric tridiagonal reduction is often the first step of solving symmetric dense eigenvalue problems, and it dominates the solution time. Because such eigenvalue problems arise in many scientific and engineering simulations, **xSYTRD** is an important kernel to be optimized on the emerging computers. [¶] For example, to simulate $\mathcal{O}(10^2)$ atoms for electronic structures calculation [Chapter 12.1 , 9–11], Exciting [12] or Elk [13] simulation code routinely solves complex generalized dense symmetric eigenvalue problems of dimension $\mathcal{O}(10^3)$ on a compute node, but several projects are underway to enable a simulation of $\mathcal{O}(10^3)$ atoms. In addition, a dense eigensolver is often needed in a subspace projection method to solve large-scale sparse eigenvalue problems; there, a projection subspace is first computed using multiple compute nodes, and then a smaller projected system is solved on each node. For instance, TRLan [14] implements a thick restart Lanczos method [15] on a distributed-memory system and uses **xSYTRD** to redundantly solve the projected system on each node at every restart. Similarly, a state-of-the-art electronic structure calculation simulation [16, 17] uses a conjugate gradient method to generate a projection subspace on multiple nodes and solves a projected system on each compute node. Hence, in these methods, it is critical to have an efficient solver on a compute node for solving the projected dense systems. Finally, there are methods like a matrix sign function [18], which may allow us to split a matrix into smaller independent submatrices. Then, **xSYTRD** may be used to solve each of the independent subproblems on one node.

In this paper, we extend **xSYTRD** to utilize the multiple GPUs on a compute node. This is an important extension in two ways: (i) it allows us to exploit the aggregated computing power of multiple GPUs to shorten the reduction time; and (ii) it utilizes the aggregated GPU memory to solve larger-scale problems. The rest of the paper is organized as follows: in Sections 2 and 3, we first describe **xSYTRD** of LAPACK and its hybrid extension to use multiple GPUs, respectively. In the latter section, several techniques to enhance the performance of the hybrid **xSYTRD** are also described. Then, in Section 4, we present our multi-GPU extension of a BLAS-2 dense symmetric matrix-vector multiplication routine, **xSYMV**, that often dominates the reduction time of **xSYTRD**. Finally, in Section 5, we present the performance of **xSYMV** and **xSYTRD** on the Keeneland system. In this section, we also present three test cases for using our multi-GPU kernels: (i) a standard

[‡]For example, the LINPACK benchmark that solves a dense linear system of equations is still used to rank HPC computers for the TOP500 list, http://www.top500.org.

[§]For complex double or single precision, the tridiagonalization routine that works on a Hermitian matrix is named **CHETRD** or **ZHETRD**, respectively.

[¶]A list of applications requiring the solution of large dense symmetric eigenvalue problems can be found at http://elpa.rzg.mpg.de/goal.

DOI: 10.1002/cpe

dense symmetric eigensolver on a single node with multiple GPUs and its performance comparison with a distributed-memory dense eigensolver [19]; (ii) a generalized dense symmetric eigensolver, a critical component in state-of-the-art electronic structure calculation simulations [12, 13, 20, 21], which currently use MAGMA and can immediately exploit our multi-GPU extensions; and (iii) an integration of our multi-GPU kernels into a distributed-memory sparse eigensolver TRLan. We conclude with final remarks in Section 6.

The following notations are used throughout the rest of the paper: the (i, j)-th element of a matrix A is denoted by $a_{i,j}$, while the j-th column and the i-th row of A is $\mathbf{a}_{:,j}$ and $\mathbf{a}_{i,:}$, respectively. Furthermore, the submatrix consisting of the i_1-th through the i_2-th rows and the j_1-th through the j_2-th columns of A is denoted as $A_{i_1:i_2,j_1:j_2}$. Our discussion in this paper assumes the lower-triangular part of the symmetric matrix A is stored, but it can easily be extended to the case where the upper-triangular part of A is stored. Unless otherwise stated, all the experiments in this paper were conducted on the Keeneland system. All the dense test matrices are random matrices with the uniform distribution between zero and one.

2. TRIDIAGONAL-REDUCTION ON MULTICORES

In the tridiagonalization of a symmetric matrix, an n-by-n symmetric matrix A is reduced to a tridiagonal matrix T by an orthogonal similarity transformation; that is, $Q^T A Q = T$, where Q is an n-by-n orthogonal matrix. The LAPACK routine **xSYTD2** computes this orthogonal matrix Q as a product of $n-1$ elementary Householder reflectors; i.e., $Q = H_1 H_2 ... H_{n-1}$, where $H_j = I - \tau_j \mathbf{v}_j \mathbf{v}_j^T$, τ_j is a scalar, and $\mathbf{v}_j$ is an n-length vector. Another LAPACK routine **xLARFG** computes this j-th Householder reflector $\mathbf{v}_j$ that zeroes out the elements of the j-th column $\mathbf{a}_{:,j}$ below the subdiagonal; that is, $(I^{(j+1)} - \tau_j \mathbf{v}_j^{(j+1)} \mathbf{v}_j^{(j+1)^T}) \mathbf{a}_j^{(j+1)} = \|\mathbf{a}_j^{(j+1)}\|_2 \mathbf{e}_1^{(n-j)}$, where $\mathbf{v}_j^{(j+1)}$ is the vector consisting of the $(j+1)$-th through the n-th elements of $\mathbf{v}_j$, and $\mathbf{e}_1^{(n-j)}$ is the first column of an $(n-j)$-by-$(n-j)$ identity matrix $I^{(j+1)}$. Finally, the LAPACK routine **xSY2RK** updates the trailing submatrix using the j-th Householder reflector; that is,

$$\begin{aligned} A^{(j+1)} &:= H_j^{(j+1)} A^{(j+1)} H_j^{(j+1)} \\ &= A^{(j+1)} - \mathbf{v}_j^{(j+1)} \mathbf{w}_j^{(j+1)^T} - \mathbf{w}_j^{(j+1)} \mathbf{v}_j^{(j+1)^T}, \end{aligned}$$

where $A^{(j+1)}$ is the trailing submatrix at the j-th step (i.e., $A^{(j+1)} = A_{j+1:n,j+1:n}$), and $\mathbf{w}_j^{(j+1)} = \tau_j (I - \frac{\tau_j}{2} \mathbf{v}_j^{(j+1)} \mathbf{v}_j^{(j)^T}) A^{(j+1)} \mathbf{v}_j^{(j+1)}$. Unfortunately, **xSYTD2** often obtains only a fraction of the peak performance on modern CPUs because most of the computation is performed using BLAS-1 or BLAS-2, which is bandwidth-limited.

To improve the data locality of **xSYTD2**, we can delay the application of b Householder transformations and use BLAS-3 to update the trailing submatrix with the accumulated transformations at once; that is, $A^{(J+1)} := A^{(J+1)} - V_J^{(J+1)} W_J^{(J+1)^T} - W_J^{(J+1)} V_J^{(J+1)^T}$, where $V_J^{(J+1)}$ is the $(J+1)$-th through the N-th blocks of the J-th block column V_J of the matrix $V = [v_1, v_2, \ldots, v_n]$; that is, $V_J^{(J+1)} = V_{Jb+1:n,(J-1)b+1:Jb}$, and $N = \frac{n}{b}$.[‖] The resulting blocked algorithm is implemented in **xSYTRD** of LAPACK, and its pseudocode is shown in Figure 1.

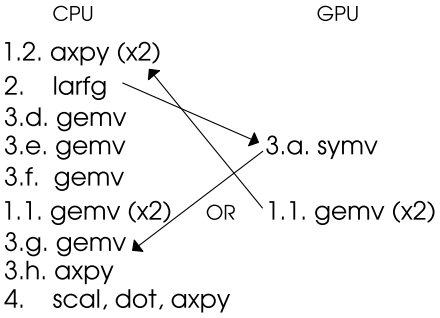
While **xSYTRD** requires about $\frac{4}{3}n^3$ floating-point operations (flops) to tridiagonalize an n-by-n matrix A, about 50% of these flops are performed using the BLAS-2 **xSYMV** to compute the symmetric matrix-vector multiplication (Step 3.a of Figure 1(c)), while most of the remaining flops are performed using the BLAS-3 **xSYR2K** to update the trailing submatrix (Step 2 of Figure 1(a)). Therefore, the tridiagonalization time of LAPACK is often dominated and limited by that of the BLAS-2 **xSYMV**, and **xSYTRD** still obtains a fraction of the peak performance, especially on a large number of CPUs.

[‖]Our discussion here assumes that n is a multiple of b, but it can be easily extended for other cases.

DOI: 10.1002/cpe

```
for J = 1, 2, ..., N do
1. (Panel factorization with xLATRD in Figure 1(c))
   [V_J^(J), W_J^(J)] =xLATRD(A_J^(J))
2. (trailing submatrix update with xSYR2K)
   A^(J+1) -= V_J^(J+1) W_J^(J+1)^T + W_J^(J+1) V_J^(J+1)^T
end for
```

(a) Psedocode of xSYTRD.



(b) Algorithmic flow of xLATRD

```
for j = 1, 2, ..., m do
1.1.(apply previous 1-st through
     (j − 2)-th reflectors to j-th column)
   a. a_j^(j) -= V_{j:n,1:j-2} w_{j,1:j-2}^T
   b. a_j^(j) -= W_{j:n,1:j-2} v_{j,1:j-1}^T
1.2.(apply previous (j − 1)-th reflector to j-th column)
   a. a_j^(j) -= v_{j-1}^(j) w_{j,j-1}
   b. a_j^(j) -= w_{j-1}^(j) v_{j,j-1}
2. (compute j-th Householder reflector)
   a. [v_j^(j), τ_j] = xLARFG(a_j^(j+1))
   b. // start sending v_j^(j+1) to GPU
3. (perform matrix-vector multiplication)
   a. w_j^(j+1) = A^(j+1) v_j^(j+1)
   b. //start sending w_j^(j+1) to CPU
   c. if j > 1 then
   d.    f = W_{j+1:n,1:j-1}^T v_j^(j+1)
   e.    f = V_{j+1:n,1:j-1} f
   f.    g = V_{j+1:n,1:j-1}^T v_j^(j+1)
   g.    w_j^(j+1) -= W_{j+1:n,1:j-1} g
   h.    w_j^(j+1) -= f
   i. end if
4. (compute auxiliary vector w_1^(j+1))
   a. w_j^(j+1) = τ_j w_j^(j+1)
   b. α = (τ_j/2) w_j^(j+1)^T v_j^(j+1)
   c. w_j^(j+1) += α v_1^(j+1)
   d. //start sending w_j^(j+1) to GPU
end for
```

(c) Pseudocode of xLATRD.

Figure 1. Tridiagonalization algorithm.

3. TRIDIAGONAL-REDUCTION USING MULTIPLE GPUS

On a heterogeneous compute node architecture, the computational units of a different type are adapted for particular types of tasks. For instance, the GPUs are designed to maximize the throughput of multiple tasks, and they are specifically adapted to handle tasks that exhibit high data or thread-level parallelism. On the other hand, the CPUs are designed to minimize the latency of a single task using deep memory-hierarchy and instruction-level parallelism. To exploit the architectural strengths of these different computational units, MAGMA splits the algorithm into smaller computational tasks and statically schedules their executions on the CPUs or GPU. As discussed in Section 2, the reduction time of **xSYTRD** is often limited by the BLAS-2 **xSYMV** and the BLAS-3 **xSY2RK**. Hence, MAGMA schedules these two computational kernels on the GPU. To effectively use both CPUs and GPU, **xSYMV** on the GPU (Step 3.a of Figure 1(c)) is overlapped with **xGEMV** to compute the auxiliary vectors **f** and **g** on the CPUs (Steps 3.d, 3.e, and 3.f). Furthermore, the application of the 1-st through the $(j-2)$-th reflectors to the j-th vector (Step 1.1) is scheduled either on the CPUs or GPU, and it is overlapped with either **xSYMV** on the GPU (Step 3.a) or the computation of $\mathbf{w}_j$ on the CPUs (Steps 3.g and 3.h), respectively. The decision on whether to schedule Step 1.1 on the CPUs or GPU can be made at run time by querying the GPU whether **xSYMV** has completed after Step 3.f. Specifically, if **xSYMV** has not completed, then Step 1.1 is scheduled on the CPUs. Otherwise, it is scheduled on the GPU. Figure 1(b) illustrates this algorithmic flow. As soon as the auxiliary vector $\mathbf{w}_j$ is computed, it is asynchronously sent to the GPU (Steps 4.c and 4.d of Figure 1(c)), which is then used to update the trailing submatrix (Step 2 of Figure 1(a)). All other computation and communication on the GPU is performed on a single GPU stream to avoid explicit synchronizations.

To utilize the multiple GPUs, the matrix A is distributed among the GPUs in a 1D block-column cyclic layout, using the same block size b on the CPUs and GPUs. On the other hand, the two n-by-b block-columns V_J and W_J are duplicated on the GPUs, which are then used to update the trailing submatrix by our multi-GPU extensions of **xSYR2K**. Currently, our multi-GPU **xSYR2K** is implemented using a sequence of MAGMA BLAS calls. Specifically, a GPU updates each of the local diagonal blocks and the off-diagonal blocks of each block column through single calls to **xSYR2K**

DOI: 10.1002/cpe

and **xGEMM**, respectively. Multiple GPU streams are used cyclically on these block columns to potentially execute small kernels in parallel on each GPU. This allows us to exploit the high data parallelism of the BLAS-3 kernel on the GPUs. On the other hand, **xSYMV** is bandwidth-limited. Even though the GPU has a greater memory bandwidth than the CPU does, it requires a careful implementation to obtain the high performance of **xSYMV** on the GPU, which will be discussed in the next section.

4. SYMMETRIC-MATRIX VECTOR MULTIPLICATION USING MULTIPLE GPUS

The reduction time of MAGMA **xSYTRD** is often dominated by the BLAS-2 **xSYMV** on a GPU. This is mainly because **xSYMV** is not only a bandwidth-limited operation but it also exhibits irregular data access patterns that are difficult to optimize on a GPU. In this section, we extend **xSYMV** to utilize the multiple GPUs. We acknowledge that many parallel **xSYMV** and **xSYTRD** algorithms have been proposed and implemented on CPUs (e.g., PBLAS [22] and PLAPACK [23]). However, our contributions are their efficient implementations and their performance studies on GPUs. Our experimental results will demonstrate that significant speedups can be obtained by carefully tuning the codes on GPUs and effectively utilizing both CPUs and GPUs. Furthermore, the performance of **xSYMV** on multiple GPUs can be used to infer the performance of other computational kernels with irregular data accesses. In fact, many researchers have recognized **xSYMV** as an important kernel and have optimized it on a single GPU [24–27]. In this section, we first outline our multi-GPU **xSYMV** algorithm (Section 4.1). We then discuss an optimization parameter and techniques (Sections 4.2 and 4.3). Finally, we present a couple of algorithm extensions of **xSYMV** to obtain high performance of **xSYTRD** (Sections 4.4 and 4.5).

4.1. Algorithm

To compute the matrix-vector multiplication $\mathbf{y} := \alpha A\mathbf{x} + \beta\mathbf{y}$ on multiple GPUs, the symmetric matrix A is distributed in a 1D block-column cyclic layout, while the vector $\mathbf{x}$ is duplicated on each GPU. Then, each GPU computes the partial result of **xSYMV** using its local matrix. Figure 2(a) shows the pseudocode of our multi-GPU **xSYMV**, where $G(k)$ is the set of the block-column indices belonging to the k-th GPU. Our I-th thread block on the k-th GPU accesses the I-th block row

1. (multiplication with diagonal blocks)
`for` each $I \in G(k)$ `do`
  a. accumulate into I-th block:
    $\mathbf{y}_I^{<k,I>} = B\mathbf{x}_I$, where
    $B = A_{I,I} + A_{I,I}^H - \mathrm{diag}(A_{I,I})$.
`end for`
2. (multiplication with off-diagonal blocks)
`for` $I = 1, 2, \ldots, N$ `do`
  `for` each $J \in G(k)$ and $J < I$ `do`
  a. accumulate into I-th block:
    $\mathbf{y}_I^{<k,I>} += A_{I,J}\mathbf{x}_J$.
  b. accumulate into J-th block:
    $\mathbf{y}_J^{<k,I>} += A_{I,J}^H\mathbf{x}_I$.
  `end for`
`end for`

(a) Code executed by the k-th GPU.

1. (multiplication with diagonal block $A_{I,I}$)
`if` $I \in G(k)$ `then`
  `for` $i = 0, 1$ with $i_1 = 32i + 1$, $i_2 = 32(i + 1)$ `do`
  a. expand $(A_{I,I})_{i_1:i_2,i_1,i_2}$ into B
  b. accumulate into I-th block:
    $(\mathbf{y}_I^{<K,I>})_{i_1:i_2} += B(\mathbf{x})_{i_1:i_2}$
  `end for`
  e. load $(A_{I,I})_{33:64,1:32}$ into B
  f. accumulate into I-th block:
    $(\mathbf{y}_I^{<K,I>})_{33:64} += B(\mathbf{x}_I)_{1:32}$
    $(\mathbf{y}_I^{<K,I>})_{1:32} += B^H(\mathbf{x}_I)_{33:64}$
`end if`
2. (multiplication with off-diagonal blocks)
`for` each $J \in G(k)$ and $J < I$ `do`
  `for` $i = 0, 1, 2, 3$ with $i_1 = 16i + 1$, $i_2 = 16(i + 1)$
  `do`
  a. load $(A_{I,J})_{:,i_1:i_2}$ into B
  b. accumulate into I-th and J-th blocks:
    $\mathbf{y}_I^{<k,I>} += B(\mathbf{x}_J)_{i_1:i_2}$, and
    $(\mathbf{y}_J^{<k,I>})_{i_1:i_2} += B^H\mathbf{x}_I$.
  `end for`
`end for`

(b) Code executed by I-th thread-block on k-th GPU.

Figure 2. Pseudocode of multi-GPU **xSYMV** ($b = 64$).

DOI: 10.1002/cpe

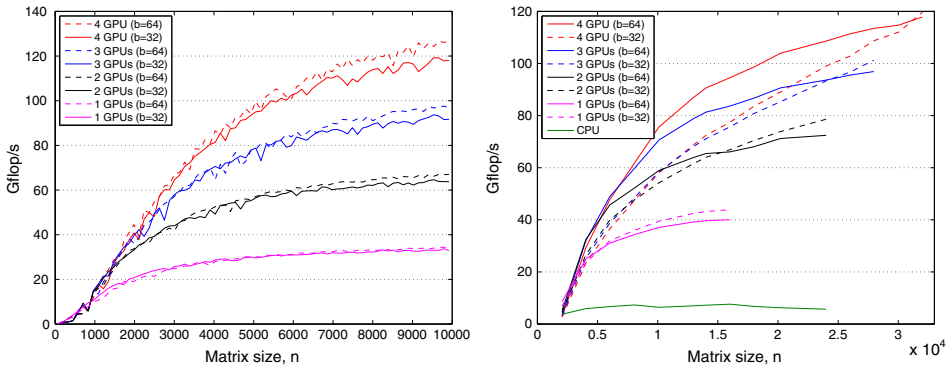
of the local matrix and computes its contributions to $\mathbf{y}^{<k>}$. To avoid the synchronizations among the thread blocks, the I-th thread block writes its contribution to its own workspace $\mathbf{y}^{<k,I>}$. Once all the contributions $\mathbf{y}^{<k,I>}$ are computed, another kernel is launched to sum these partial results on the GPU; that is, $\mathbf{y}^{<k>} = \sum_{I=1}^{N} \mathbf{y}^{<k,I>}$. Finally, the partial sums $\mathbf{y}^{<k>}$ are sent to the CPUs, and the CPUs compute the final results $\mathbf{y}$ by accumulating the contributions from all the GPUs; that is, $\mathbf{y} = \sum_k \mathbf{y}^{<k>}$. We note that even though we use the 1D block-column distribution of A among GPUs, the block rows of the local matrix are distributed among the thread-blocks on each GPU. Hence, A is distributed in a 2D block layout among the thread-blocks.

If the upper-triangular part of the matrix A is stored, then A is again distributed among GPUs in a 1D block-column cyclic layout, and each thread block on a GPU processes a block row of the local matrix. Our experiments have shown that the performance of **xSYMV** is comparable when either the lower or upper triangular part of A is stored. Our discussion in the rest of this paper assumes the lower-triangular part of A is stored.

4.2. Optimization parameter, block size

Because **xSYMV** is a bandwidth-limited operation, its main optimization parameter is the block size to utilize the shared memory on the GPU. Our single-GPU **xSYMV** [26] uses the block size of 64 dividing the matrix A into 64-by-64 blocks. Then, each thread block consisting of 64-by-4 threads further subdivides each off-diagonal block $A_{I,J}$ into 64-by-16 sub-blocks. Each sub-block is then loaded into the shared memory and multiplied with the corresponding part of $\mathbf{x}_J$, and its transpose is multiplied with $\mathbf{x}_I$. This recursive-blocking is also used on a diagonal block, where the 64-by-64 block is subdivided into 32-by-32 sub-blocks, and the GPU threads are mapped into a 32-by-8 grid. The lower-triangular part of a diagonal sub-block of $A_{I,I}$ is expanded into the full square matrix in the shared memory before being multiplied with the corresponding part of $\mathbf{x}_I$. Once the off-diagonal sub-block of the diagonal block is loaded into shared memory, we multiply the sub-block and its transpose with the corresponding parts of $\mathbf{x}_I$ before moving on to another sub-block. Hence, the matrix A is loaded into the shared memory only once. Figure 2(b) shows the pseudocode of this algorithm executed by a GPU thread-block, where B is in the shared memory and $(A_{I,J})_{:,i_1:i_2}$ is the sub-block consisting of the i_1-th through the i_2-th columns of $A_{I,J}$.

The optimal performance of **xSYMV** may be obtained using a different block size for a different matrix dimension, in a different precision, or on a different GPU architecture. Furthermore, **xSYMV** may have to use the same block size as that of a higher-level routine because the matrix is statically distributed among the GPUs, and the optimal performance of the higher-level routine may be obtained using a block size different from the optimal block size of **xSYMV**. For example, Figure 3 shows the performance of **DSYMV** and **DSYTRD** on two six-core 2.1GHz AMD Opteron 6172 processors and up to four NVIDIA C2050 Tesla GPUs. We used MKL 2011.1.069 and CUDA 4.0. We clearly see that especially for a small-scale matrix on multiple GPUs, **xSYTRD**



(a) DSYMV (b) DSYTRD

Figure 3. Performance of `DSYMV` and `DSYTRD` on AMD Opteron and NDIVIA Tesla C2050 GPUs.

DOI: 10.1002/cpe

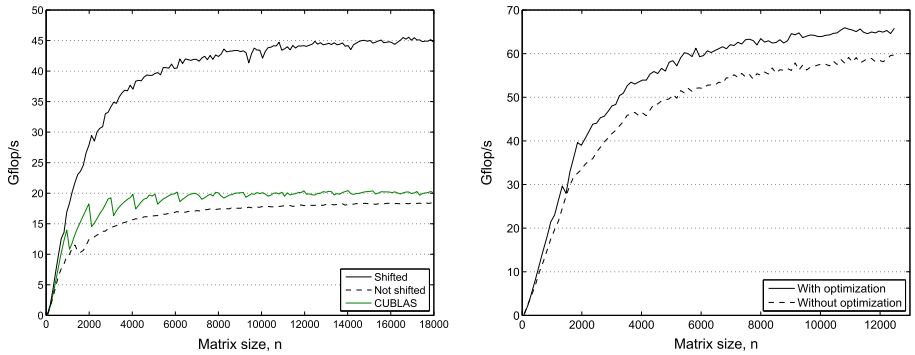
performed better using the block size of 32 rather than using the block size of 64, which was optimal for **xSYMV**. This is mainly because when the CPU has a low computing power, the BLAS-2 **xGEMV** required by **xLATRD** becomes expensive for a larger block size on the CPUs. As a result, using the block size of 64, the time spent on the CPUs became a bottleneck, and the reduction time did not scale as well as that using the block size of 32. In this paper, we study the performance of **xSYMV** using the block sizes of 16, 32, and 64.

On a Fermi GPU, each streaming multiprocessor (SM) has 48KB of the shared memory, which is large enough to store the 32-by-32 numerical values in all of the four precisions. Hence, when the block size is less than or equal to 32, our **xSYMV** has an option of not using the recursive-blocking, which allows us to simplify the data access of each thread block and remove some synchronizations. Using the block size of 32 without the recursive-blocking, our **xSYMV** uses the same amount of the shared memory as that using the block size of 64 with the recursive-blocking. Furthermore, without recursive-blocking, we use the 32-by-8 thread block on both diagonal and off-diagonal blocks.

4.3. Optimization techniques

Our multi-GPU **xSYMV** uses all the optimization techniques in the original single-GPU **xSYMV** [26]. For instance, reducing the number of shared memory bank conflicts is vital to obtain high performance. Because the shared memory on a Fermi GPU has 32 memory banks, when the 32-by-8 thread block is used, every 8-th thread with the same thread index t_x accesses the same memory bank at each clock cycle, where each GPU thread is identified by a 2D thread index of (t_x, t_y). Hence, when accessing the shared memory, these threads with the same t_x must cooperate with each other. To avoid these bank conflicts, we shift the memory location that each thread accesses so that the threads with the same t_x do not access the same memory bank at the same time. Since data in the shared memory is heavily used, this technique improved the performance of **xSYMV** by a factor of more than two (Figure 4(a)).

In addition, we have reduced the number of branches and synchronizations in our implementations. For instance, while processing off-diagonal blocks, once a thread block computes a multiplication with a sub-block in shared memory, a warp of 32 threads with $t_y = 1$ accumulate the results, while other warps prefetch the next sub-block into shared memory. In the original code, the same shared memory were used to store the sub-block and then the partial sum, and hence, this prefetching of the next sub-block was not possible. Another optimization is specific to the multi-GPU kernels, where some thread blocks do not own diagonal blocks. To compute the multiplication with the diagonal blocks, each thread block performs the multiplication with the last block in its block row. However, only the thread blocks owning diagonal blocks accumulate the results, while other thread blocks simply discard them. We found that this optimization avoids an expensive conditional statement and can lead to up to 20% increase in the Gflop/s for single precision on a single



(a) Bank conflict optimization ($b = 32$). (b) Conditional statement optimization ($b = 64$).

Figure 4. Effects of `DSYMV` optimizations on a Tesla M2090 (Left) and C2050 (Right) GPU.

DOI: 10.1002/cpe

GPU (Figure 4(b)). However, this introduces additional computation, which could limit the parallel scaling of our **xSYMV**.

4.4. Multiplying with a submatrix

At the j-th step of **xSYTRD**, **xSYMV** is called on the $(n-j)$-by-$(n-j)$ trailing submatrix. On a single GPU, **xSYMV** can take a pointer to the starting point of the submatrix and use the pointer redirection [26] to improve the data access during the multiplication with the submatrix of a general size. On the other hand, to specify the submatrix, our multi-GPU **xSYMV** uses an global offset into the submatrix in addition to an array of pointers, each of which points to the beginning of the local matrix stored on a GPU; that is, if the offset is set to be $j+1$, then it computes $A^{(j+1)}\mathbf{x}^{(j+1)}$. Furthermore, to align the memory access during the multiplication with the submatrix $A^{(j+1)}$, each GPU first identifies the leading diagonal block $A_{J,J}$ of the submatrix (i.e., $J = \left\lceil \frac{j+1}{b} \right\rceil$). Then, the matrix-vector multiplication $A^{(J)}\mathbf{x}^{(J)}$ is computed. With our block size of 32 or 64, we always meet the alignment requirements when a set of 32 memory banks are read. To eliminate the contribution from the 1-st through the j-th elements of $\mathbf{x}$, when the first block $\mathbf{x}_J$ is copied into shared memory, the corresponding elements of $\mathbf{x}_J$ are set to be zero; that is, $\mathbf{x}_{(J-1)b+1:j} = 0$. This avoids conditional statements, which are not suited for GPU computing. When the block size is 32, the multiplication with these padded zeros does not add any overhead because all of the 32 threads in the same warp executes one common instruction [28]. There is an overhead of loading extra data when the block size is 64. However, because this overhead is needed only for the first block $\mathbf{x}_J$, it is negligible, especially when the matrix size is large.

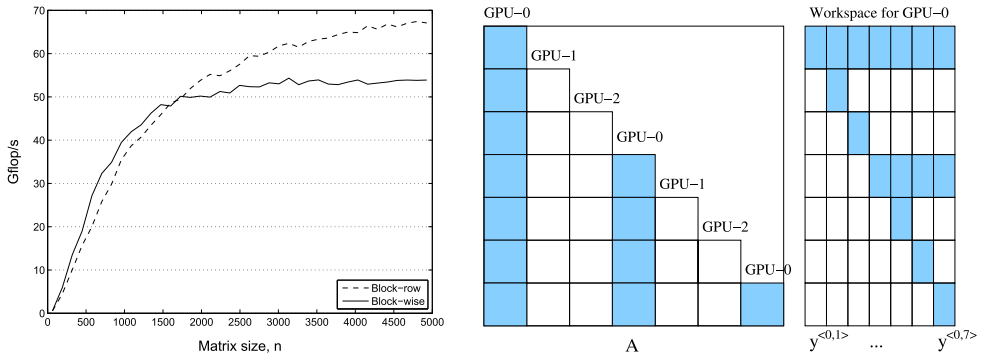
4.5. Exploiting fine-grained parallelism in a small-scale matrix

Because **xSYTRD** calls **xSYMV** on the trailing submatrices of dimensions $n-1$ through 1, it is critical that **xSYMV** is optimized over the range of the submatrix dimensions. We observed that our block-row algorithm in Section 4.1 (each thread block accesses a block row) performs well for a large-scale submatrix, but for a small-scale submatrix, its performance is lower than what is expected. This is because each thread block computes the contributions from one block row, and the total number of thread blocks is only N (i.e., $N = \frac{n}{b}$). For example, when the submatrix dimension is 1024 and the block size is 32, there are only 32 thread blocks, each of which requires about 16KB of the shared memory. This leads to only two or three thread blocks per SM on a Tesla C2050, which has 14 SMs and 48KB of the shared memory per SM. Hence, the occupancy of this block-row algorithm could be too low to obtain high performance.

The low occupancy of the algorithm could also lead to load imbalance. Because of the symmetric matrix storage, when a single GPU is used, the workload of the I-th thread block is $\triangle + (I-1)\triangledown$, where $\triangle$ and $\triangledown$ are the workload required to process a diagonal and an off-diagonal blocks, respectively. Hence, the load imbalance between the two adjacent thread blocks is $\triangledown$. If the number of block rows is much greater than the number of SMs, then multiple thread blocks are assigned to an SM, and the workloads are likely balanced among SMs [26]. For instance, if the thread blocks are cyclically assigned to SMs in an increasing order of their thread block IDs, I, then the maximum load imbalance between SMs is given by $\triangle + 13\lceil \frac{N}{14} \rceil \triangledown$ on a Fermi GPU, which has 14 SMs. For a large-scale matrix, this imbalance is relatively small in comparison to the total workload, which is $N(\triangle + \frac{(N-1)}{2}\triangledown)$. However, for a small N, this block-row algorithm may not provide enough tasks to balance the workloads among SMs, and the workload imbalance may be significant relative to the total workload.

To enhance the performance of **xSYMV** on a small-scale matrix, we developed another algorithm, where each thread block computes a contribution from a single block. In comparison to the block-row algorithm, this block-wise algorithm increases the number of thread blocks to be $\frac{N^2+N}{2}$ and exploits a finer-grained parallelism. Furthermore, this block-wise algorithm typically has enough thread blocks to improve the load balance among SMs, even for a small-scale matrix. For instance, if the thread blocks are cyclically assigned to SMs in an increasing order of their thread block IDs,

DOI: 10.1002/cpe



(a) ZHEMV in block-row and block-wise algorithms. (b) GPU-0 allocates workspace to store blocks colored in blue for $\mathbf{y}_I^{<k,I>}$.

Figure 5. Performance of ZHEMV on a Tesla M2090 (Left) and Illustration of per-GPU workspace (Right).

then the maximum difference in the workload per SM is reduced to be $\triangle + \triangledown$. As a result, this block-wise algorithm improved the performance of the block-row algorithm by up to 30% for a small-scale matrix (Figure 5(a)). On the other hand, for a large-scale matrix, the performance of the block-wise algorithm was lower than that of the block-row algorithm. This is because the overhead of scheduling a thread block can be relatively large for the small task generated by our block-wise algorithm. Furthermore, when the block-wise algorithm is used, the threads working on the blocks in the same block row write into the same workspace. Hence, there is an extra shared memory reduction among these threads working on the same block row. In our experiments, we used the block-wise algorithm for the matrix dimension less than 1400, while the block-row algorithm is used for a larger matrix.

4.6. Memory overhead

Figure 5(b) illustrates the workspace requirement of our multi-GPU **xSYMV** kernel. Each GPU requires the workspace to store about $n + \frac{n(N-1)}{2n_g}$ numerical values, where n_g is the number of GPUs. On a single GPU with $b = 32$ or 64, this is about 1.56% or 0.78% overhead in comparison to that required to store the matrix A. When **xSYMV** is used in **xSYTRD**, the workspace needs to be allocated only once, and the memory allocation overhead is amortized over $n - 1$ calls to **xSYMV**.

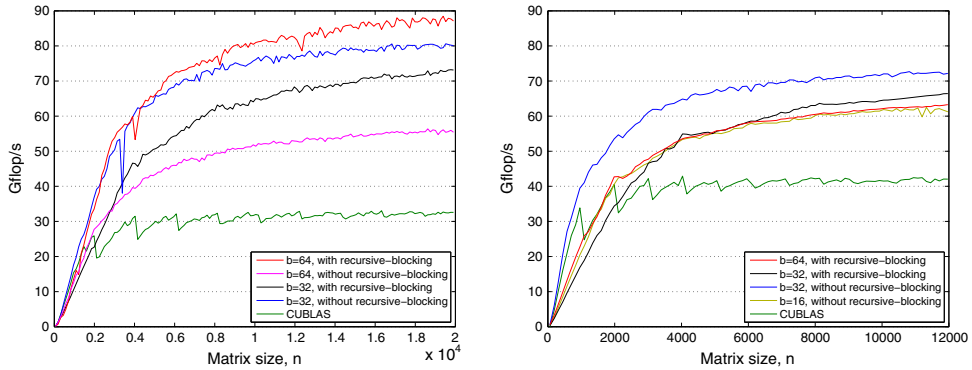
5. PERFORMANCE EVALUATION

We now study the performance of our multi-GPU **xSYMV** and **xSYTRD** on a single compute node of the Keeneland system, which consists of twelve 2.8GHz Intel Xeon processors and three NVIDIA Tesla M2090 GPUs. The total size of the page-caches available on the CPU is 18GB, while each GPU is equipped with 6GB of memory, but 12% of the GPU memory is used by Error Correction Code (ECC). The dimension of the largest matrix in double precision, which can be stored in the CPU memory, is about 45,000. For all the experiments, we used MKL 2011.3.174 and CUDA 4.1, and all 12 cores of the CPUs.

*5.1. Performance of **xSYMV***

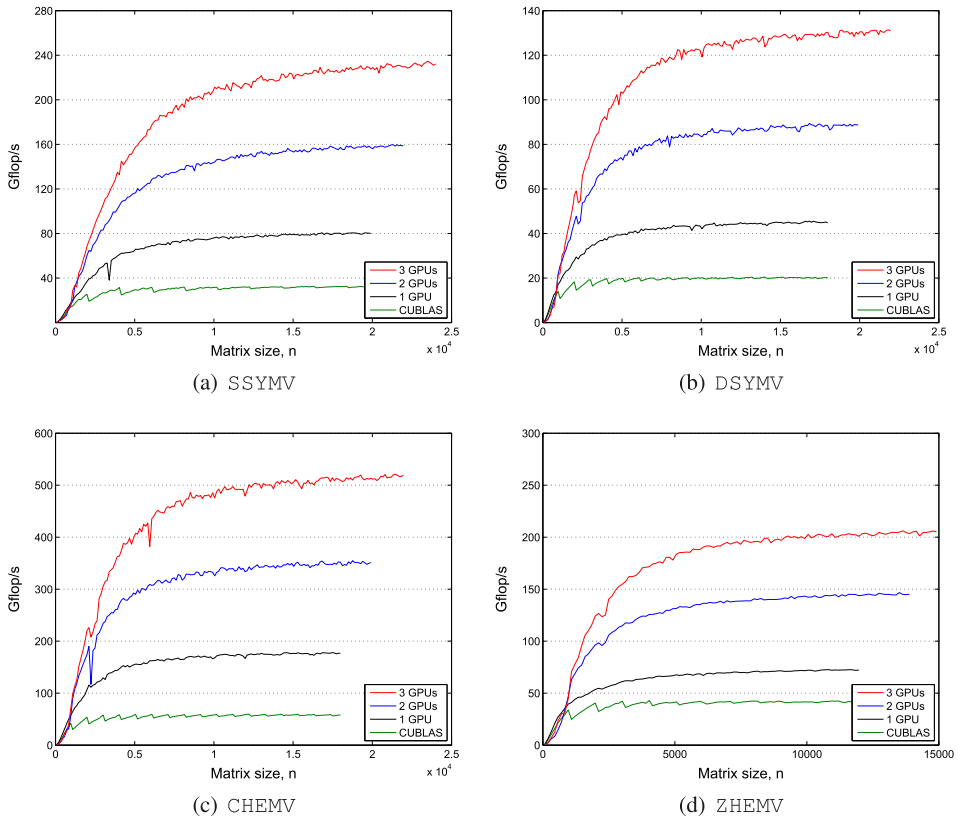
Figure 6 shows the Gflop/s obtained using our **SSYMV** and **ZHEMV** on one GPU. We see that in most cases, the best performance was obtained using the block size of 64 and 32 for **SSYMV** and **ZHEMV**, respectively. The Tesla M2090 GPU has a memory bandwidth of 177GB/s, but according to the bandwidth utility of NVIDIA SDK, with ECC on, the practical bandwidth is about 120GB/s. Hence, the respective theoretical peak performances of **SSYMV**, **DSYMV**, **CHEMV**, and **ZHEMV** on one GPU are about 120, 60, 240, and 120Gflop/s. In comparison to **SSYMV**, the performance

DOI: 10.1002/cpe



(a) SSYMV (b) ZHEMV

Figure 6. Performance of xSYMV on a single Tesla M2090 GPU.



(a) SSYMV (b) DSYMV

(c) CHEMV (d) ZHEMV

Figure 7. Performance of xSYMV on multiple Tesla M2090 GPUs ($b = 32$ without recursive-blocking).

of **ZHEMV** was lower than expected. An auto-tuning technique like [29] may help improve the performance of **ZHEMV**.

Next, in Figure 7, we show the Gflop/s obtained using our **xSYMV** on multiple GPUs. The block size was fixed at $b = 32$, which provided good performances in all the four precisions. We see that the Gflop/s of **SSYMV** was about twice as much as **DSYMV** because the performance of **xSYMV** is limited by the memory bandwidth. These figures also demonstrate the excellent scaling of our algorithm on multiple GPUs, especially when the matrix size is large enough.

DOI: 10.1002/cpe

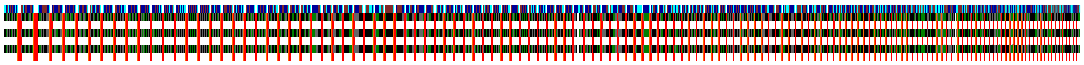
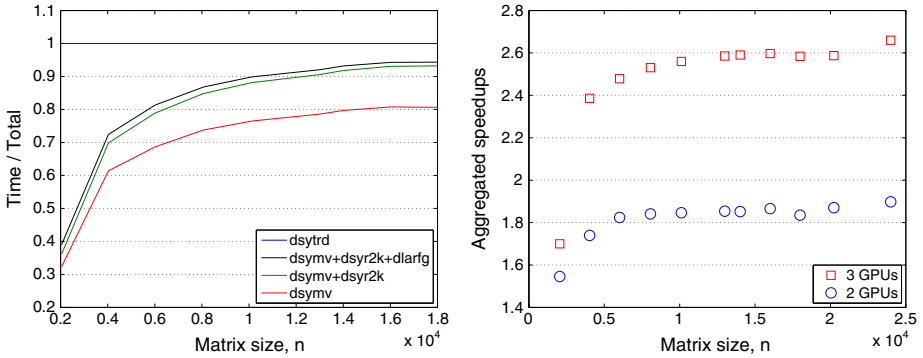


Figure 8. Trace of **DSYTRD** ($n = 10,000$). The top trace is on the Intel Xeon CPUs, and the following three pairs of the traces are for the three Telsa M2090 GPUs, each of which uses two streams. The blue, cyan, and brown traces on the CPU represent **DGEMV**, **DAXPY**, and **DLARFG**, respectively, while the green, red, and orange on the GPUs represent **DSYMV**, **DGEMV**, and **DSYR2K**, respectively.



(a) Time break-down on one GPU. (b) Speedups of aggregated time of `DSYMV`.

Figure 9. Performance profile of `DSYTRD` on Intel Xeon and Tesla M2090 ($b = 64$ with recursive-blocking).

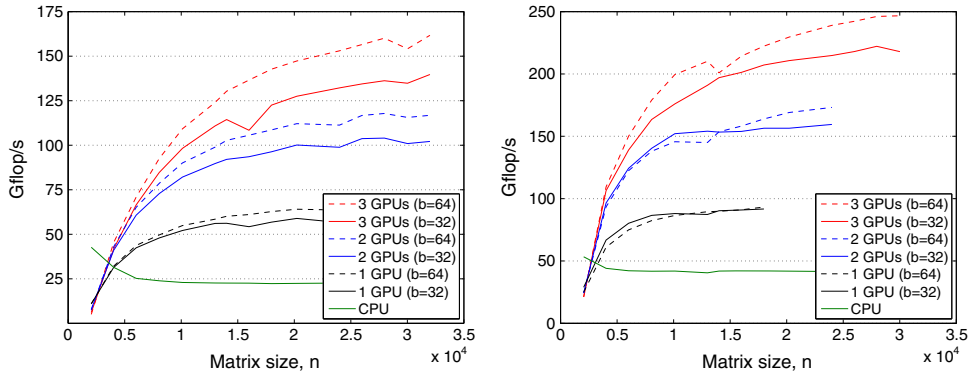
5.2. *Performance of **xSYTRD***

We now examine the performance of **xSYTRD**. Figure 8 shows the trace of **DSYTRD** with $n = 10,000$. We clearly see that the total reduction time is dominated by **DLATRD**. Figure 9(a) shows the breakdown of the reduction time on one GPU, and shows that even with $b = 64$, up to 80% of the reduction time was spent on **DSYMV**. Figure 9(b) shows the speedups of the aggregated time spent by **DSYMV** for the submatrix sizes of $n - 1$ through 1 within **DSYTRD**. In many cases, **xSYMV** does not scale on a smaller-scale submatrix as well as it does on a larger-scale submatrix, and hence, on a larger number of GPUs, a larger portion of the reduction time is spent in **xSYMV** on the small-scale submatrices. As a result, the scalability of **xSYTRD** could depend on the scalability of **xSYMV** on these small to medium sized submatrices.

Our **xSYTRD** distributes the matrix A from the CPU to the GPUs only at the beginning, and each panel of A is copied back to the CPU only before the panel factorization. Then, at each step, a vector is asynchronously copied between the CPU and the GPUs, overlapping some of the PCI-data transfer with the BLAS calls on the CPU. As a result, **xSYTRD** is typically not bounded by the PCI-data transfer for a large enough matrix. Figure 9(a) shows that more than 90% of the reduction time was spent in the BLAS calls, which do not include this data-transfer time, and the data-transfer time was less than 10% on one GPU. This ratio was about the same on the three GPUs when the matrix size is large enough. On the other hand, our **xSYMV** reads the local matrix from the GPU memory and could be limited by the GPU memory bandwidth. As shown in Sections 4 and 5.1, our **xSYMV** is designed to minimize this data transfer.

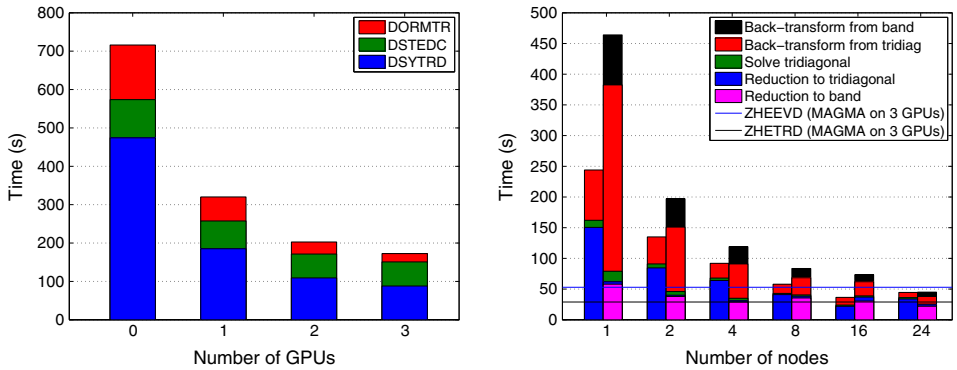
Finally, Figure 10 shows the Gflop/s of **DSYTRD** and **ZHETRD**, where we used recursive-blocking with $b = 64$ but did not use it with $b = 32$. On Keeneland, we see that **xSYTRD** obtained higher performance using the block size of 64 than using the block size of 32, while on the AMD machine in Figure 3, a higher performance was obtained using the block size of 32. A reason for this could be that the CPUs on Keeneland have a much greater computing power. As a result, on Keeneland, the reduction time was dominated more by the GPU kernels like **xSYMV** and **xSYR2K**, and the time spent by the CPU kernel like **xGEMV** was completely hidden behind the time spent on the GPUs. In the end, the performance of **xSYTRD** was higher using the block size of 64 on Keeneland because the improvement obtained by **xSYMV** using the block size of 32 was lost in

DOI: 10.1002/cpe



(a) DSYTRD (b) ZHETRD

Figure 10. Performance of xSYTRD on Intel Xeon and Tesla M2090.



(a) DSYEVD, $n = 20,000$. (b) ELPA and ZHEEVD, $n = 10,000$.

Figure 11. Performance of shared-memory xSYEVD and distributed-memory ELPA.

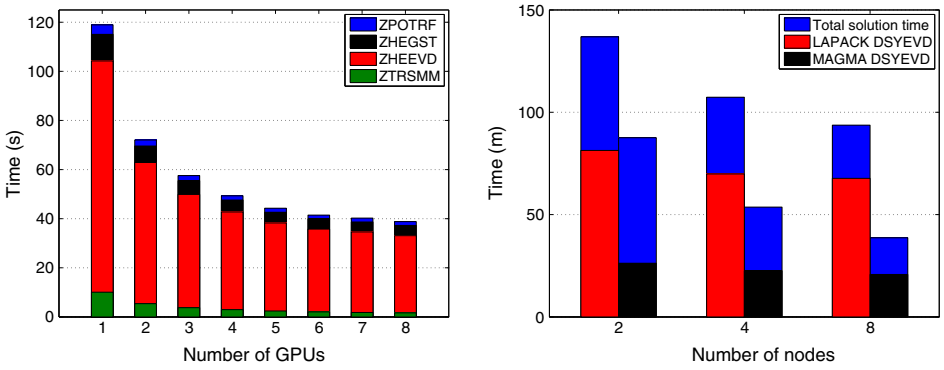
the slowdown in **xSYR2K**. For the rest of the experiments, we used the block size of 64 with recursive-blocking.

*5.3. Case study 1: performance of **xSYEVD***

We study the performance of our multi-GPU **xSYTRD** in our new multi-GPU dense symmetric eigensolver **xSYEVD**. In our experiments, we computed all the eigenvalues and eigenvectors of each dense symmetric ransom matrix. Once the matrix is reduced to a tridiagonal form, the eigenvalues and eigenvectors of the tridiagonal matrix are computed by a multi-GPU extension of the divide-conquer algorithm **xSTEDC** [30, 31]. Then, the eigenvectors of the original matrix is computed using a multi-GPU extension of **xORMTR**, where the eigenvectors are distributed in a block-column layout, and each GPU independently applies the Householder transformation to the local set of eigenvectors. Figure 11(a) shows the strong scalability of **DSYEVD**, where the matrix size is fixed at $n = 20,000$. If only the eigenvalues of A are requested, then we only need the first two steps using **xSYTRD** and **xSTEDC**.

Figure 11(b) compares the performance of our multi-GPU **ZHEEVD** with a distributed-memory dense symmetric eigensolver called Eigenvalue SoLvers for Petaflop-Applications (ELPA) [19] using OpenMPI 1.4.3 and the default block size of 40. ELPA implements two approaches to solve the eigenvalue problem: (i) one-stage approach: just like our **xSYEVD**, the matrix A is directly reduced to a tridiagonal form; and (ii) two-stage approach: the matrix A is first reduced to a banded form using BLAS-3, then the banded matrix is reduced to a tridiagonal form using the so-called bulge chasing technique. In the figure, for each node count, the first bar shows the solution time of the one-stage approach, while the second bar shows that of the two-stage approach. The solid lines

DOI: 10.1002/cpe



(a) ZHEGVD, $n = 10,000$. (b) TRLan, $n = 100,000$.

Figure 12. Performance of shared-memory ZHEGVD (Left) and distributed-memory TRLan (Right).

represent the solution time of MAGMA using three GPUs on a single node. The figure shows that because of the use of BLAS-3, the two-stage approach was much faster to reduce A into a tridiagonal form in most cases. However, the two-stage approach requires additional flops to compute the eigenvectors, and if the eigenvectors are requested, its total solution time was greater than that of the one-stage approach. We see that the performance of our **ZHETRD** was competitive with that of ELPA on up to 24 compute nodes. On the other hand, the performance of our **ZSTEDC** was lower than that of ELPA because ELPA utilizes all the cores on the multiple compute nodes, while MAGMA can exploit only a single node. In the end, our **ZHEEVD** obtained the performance similar to that of ELPA on eight nodes. We also observed that for a smaller-scale matrix, ELPA may not scale to a large number of nodes, and in this case, the performance of our **ZHEEVD** on three GPUs was comparable to that of ELPA using an optimal number of nodes.

*5.4. Case study 2: performance of **xSYGVD***

We extended the dense symmetric generalized eigensolver **xSYGEVD** of MAGMA to utilize multiple GPUs. First, our multi-GPU one-sided factorization kernel **xPOTRF** [8] is used to compute the Cholesky factorization of the symmetric positive definite matrix. Then, to transform the generalized problem into and from a standard problem, multi-GPU **xSYGST** and **xTRSMM**, respectively, apply the corresponding transformations to an independent set of vectors on each GPU. Finally, the standard eigenvalue problem is solved using our multi-GPU **xSYEVD** of Section 5.3. Figure 12(a) shows the time spent in each step of **ZHEGVD** on twelve 2.8GHz Intel Xeon processors and up to eight Tesla M2090 GPUs at ETH, Zürich. This multi-GPU **ZHEGVD** is being integrated into electronic structure calculation simulations [13].

*5.5. Case study 3: performance of **TRLan***

TRLan implements a thick restart Lanczos method [15] to compute eigenpairs of a large-scale sparse Hermitian matrix on a distributed-memory system. At each iteration, TRLan generates a new basis vector of a projection subspace using multiple MPI processes. Then, to restart the iteration, each MPI process redundantly solves the projected eigenvalue problem using **xSYEVD** of LAPACK. To enhance its performance, TRLan integrates a communication-avoiding technique to generate the projection subspace [32], and an auto-tuning technique to adaptively select the vectors to keep at each restart and the next subspace dimension [33].

To integrate our multi-GPU **DSYEVD**, we modified TRLan so that even when we have multiple MPI processes per node, the projected problem is solved by one MPI process on a node. Because MAGMA has the same interface as LAPACK, it was easy to integrate our kernel into TRLan. Figure 12(b) shows the effects of using the GPUs in TRLan, where $5,000$ eigenpairs of a synthetic diagonal matrix $\mathrm{diag}(1, 2, \ldots, 100000)$ were computed using the maximum projection subspace of

DOI: 10.1002/cpe

10, 000. [**] To solve this eigenvalue problem, TRLan required about 44, 000 iterations and 15 restarts. We clearly see that the serial bottleneck of TRLan was reduced using our multi-GPU **DSYEVD**.

6. CONCLUSION

In this paper, we parallelized a matrix-vector multiplication (**xSYMV**) and tridiagonal reduction (**xSYTRD**) of a symmetric dense matrix on a single compute node with multiple GPUs. The experimental results on the Keeneland system demonstrated the excellent scalability of our algorithms, especially on a large-scale matrix. We then integrated our multi-GPU kernels into dense symmetric standard and generalized eigensolvers on a compute node and into a sparse symmetric eigensolver on a distributed-memory system. Our experimental results demonstrated the significant impacts of our kernels on the performance of these higher-level kernels. We are currently studying the performance of our kernels in real simulations. Because our multi-GPU extension can exploit both aggregated computing power and memory storage of multiple GPUs for solving large-scale problems, it may lead us to new scientific discoveries. [††] We also plan to extend our performance comparison of state-of-the-art dense eigensolvers for the specific applications of our interests. For instance, in our experiments, we have computed all the eigenvalues and eigenvectors. If only a small number of the eigenvalues are needed, then other algorithms like an algorithm based on multiple relatively robust representations [34] may be preferred for computing the eigenvalues and eigenvectors of the tridiagonal matrix (even though their GPU-extensions have not been explored, yet). Similarly, if only a few eigenvalues and eigenvectors were needed, or if only the eigenvalues were needed, then the two-stage approach often has an advantage over the one-stage approach because the overhead of computing the eigenvectors is removed or reduced. Finally, in our experiments, we used random matrices. However, unlike the performance of the tridiagonalization, the performance of the eigensolvers depends on the spectral properties of the matrix.

ACKNOWLEDGEMENTS

This research was supported in part by NSF SDCI - National Science Foundation Award #OCI-1032815, "Collaborative Research: SDCI HPC Improvement: Improvement and Support of Community Based Dense Linear Algebra Software for Extreme Scale Computational Science," NSF Keeneland - Georgia Institute of Technology subcontract #RA241-G1 on NSF Prime Grant #OCI-0910735, and DOE MAGMA - Department of Energy Office of Science grant #DE-SC0004983, "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems." This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Foundation under contract OCI-0910735.

REFERENCES

1. Vetter J, Glassbrook R, Dongarra J, Schwan K, Loftis B, McNally S, Meredith J, Rogers J, Roth P, Spafford K, Yalamanchili S. Keeneland: bringing heterogeneous GPU computing to the computational science community. *IEEE Computing in Science and Engineering* 2011; **13**:90–5. available also at http://dx.doi.org/10.1109/MCSE.2011.83.
2. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK Users' Guide*, 3rd Edition. Society for Industrial and Applied Mathematics: Philadelphia, PA, 1999.
3. Tomov S, Nath R, Du P, Dongarra J. MAGMA users' guide, 2009. available at http://icl.eecs.utk.edu/magma.
4. Lawson C, Hanson R, Kincaid D, Krogh F. Basic linear algebra subprograms for FORTRAN usage. *Acm Transactions on Mathematical Software* 1979; **5**:308–323.
5. Nath R, Tomov S, Dongarra J. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications* 2010; **24**:511–515.
6. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 2010; **36**:232–240.
7. Tomov S, Nath R, Dongarra J. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing* 2010; **36**:645–654.

[**] This test matrix was used to study the performance of TRLan in [33].
[††] We have investigated the potential of using distributed GPUs in our recent paper [35].

DOI: 10.1002/cpe

8. Yamazaki I, Tomov S, Dongarra J. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. *International Conference on Computational Science (ICCS)*, Omaha, Nebraska, 2012; 37–46.
9. Auckenthaler T, Blum V, Bungartz H, Huckle T, Jahanni R, Krämer L, Lang B, Lederer H, Willems P. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing* 2011; **37**:783–794.
10. Kent P. Computational challenges of large-scale long-time first-principles molecular dynamics. *Journal of Physics: Conference Series* 2008; **125**:012–058.
11. Martin R. *Electronic Structure: Basic Theory and Practical Methods*. Cambridge University Press, 2004.
12. Exciting. software (Available at http://exciting-code.org/) [Accessed date: September 12, 2013].
13. Dewhurst J, Sharma S, Nordström L, Cricchio F, Bultmark F, Gross E. The ELK code manual version 1.4.18. software (Available at http://elk.sourceforge.net/) [Accessed date: September 12, 2013].
14. Yamazaki I, Wu K, Simon H. nu-TRLan User Guide. *Technical Report LBNL-1288E*, Lawrence Berkeley National Lab, 2008. software available at https://codeforge.lbl.gov/projects/trlan/.
15. Wu K, Simon H. Thick-restart Lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications* 2000; **22**:602–616.
16. Tomov S, Langou J, Dongarra J, Canning A, Wang LW. Conjugate-gradient eigenvalue solvers in computing electronic properties of nanostructure architectures. *International Journal of Computational Science and Engineering* 2006; **2**:205–212.
17. Wang LW, Zunger A. Solving Schrodingers equation around a desired energy: application to silicon quantum dots. *Journal of Chemical Physics* 1994; **100**:2394–2397.
18. Higham N. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008.
19. Auckenthaler T, Bungartz HJ, Huckle T, Kramer L, Lang B, Willems P. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*; **2**:272–278. software available at http://elpa.rzg.mpg.de/.
20. QuantumEspresso. software (Available at http://www.quantum-espresso.org/) [Accessed date: September 12, 2013].
21. Gonze X, Amadon B, Anglade P-M, Beuken J-M, Bottin F, Boulanger P, Bruneval F, Caliste D, Caracas R, Côté M, Deutsch T, Genovese L, Ghosez P, Giantomassi M, Goedecker S, Hamann DR, Hermet P, Jollet F, Jomard G, Leroux S. ABINIT: First-principles approach of materials and nanosystem properties, 2009. software (Available at www.http://abinit.org/) [Accessed date: September 12, 2013].
22. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley R. A proposal for a set of parallel basic linear algebra subprograms. *The Proceedings of the Second International Workshop on Applied Parallel Computing (PARA)*, Lyngby, Denmark, 1995; 107–114.
23. van de Geijn R. *Using Plapack: Parallel Linear Algebra Package*. MIT press: Cambridge, MA, 1997.
24. Abdelfattah A, Keyes D, Dongarra J, Ltaief H. Optimizing memory-bound numerical kernels on GPU hardware accelerators. *The Proceedings of the 10th International Meeting on High-Performance Computing for Computational Science (VECPAR)*, Kobe, Japan, 2012; 72–79.
25. Imamura T, Yamada S, Machida M. A high performance SYMV kernel on a Fermi-core GPU. *The Proceedings of the 10th International Meeting on High-Performance Computing for Computational Science (VECPAR)*, Kobe, Japan, 2012; 59–71.
26. Nath R, Tomov S, Dong T, Dongarra J. Optimizing symmetric dense matrix-vector multiplication on GPUs. *Acm/IEEE Conference on Supercomputing (SC11)*, 2011.
27. Yamada S, Imamura T, Machida M. Dynamic variation of eigenvalue problems in density-matrix renormalization-group code, 2012. presented at SIAM conference on Parallel Processing for Scientific Computing.
28. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide.
29. Kurzak J, Tomov S, Dongarra J. Autotuning GEMMs for Fermi. *LAPACK working note 245: Technical Report UT-CS-11-671*, Electrical Engineering and Computer Science Department, University of Tennessee, 2011.
30. Cuppen J. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerical Mathematics* 1981; **36**:177–195.
31. Vömel C, Tomov S, Dongarra J. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal of Scientific Computing* 2012; **34**:C70–C82.
32. Yamazaki I, Wu K. A communication-avoiding thick-restart Lanczos method on a distributed-memory system. *Workshop on Algorithms and Programming Tools for Next-Generation High-Performance Scientific and Software (HPCC)*, 2011.
33. Yamazaki I, Bai Z, Simon H, Wang LW, Wu K. Adaptive projection subspace dimension for the thick-restart Lanczos method. *ACM Transactions on Mathematical Software* 2010; **37**:1–1.
34. Dhillon I, Parlett B. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and its Applications* 2004; **387**:1–28.
35. Yamazaki I, Dong T, Tomov S, Dongarra J. Tridiagonalization of a symmetric dense matrix on a GPU cluster. *The Proceedings of the Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, Boston, Massachusetts, 2013; 1070–1079.

DOI: 10.1002/cpe