SLATE WORKING NOTES 13

Implementing Singular Value and Symmetric/Hermitian Eigenvalue Solvers

Mark Gates
Mohammed Al Farhan
Ali Charara
Jakub Kurzak
Dalal Sukkari
Asim YarKhan
Jack Dongarra

Innovative Computing Laboratory

April 2, 2020

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Revision	**Notes**
09-2019	first publication
04-2020	added generalized Hermitian definite eigenvalues (Section 2.3) and eigenvectors (Section 2.5)

```
@techreport{gates2019implementing,
  author={Gates, Mark and Al Farhan, Mohammed and Charara, Ali and
          Kurzak, Jakub and Sukkari, Dalal and YarKhan, Asim and
          Dongarra, Jack},
  title={{SLATE} Working Note 13:
         Implementing Singular Value and Symmetric/Hermitian Eigenvalue Solvers},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2019},
  month={September},
  number={ICL-UT-19-07},
  note={revision 04-2020}
}
```

Contents

List of Figures

CHAPTER 1

Introduction

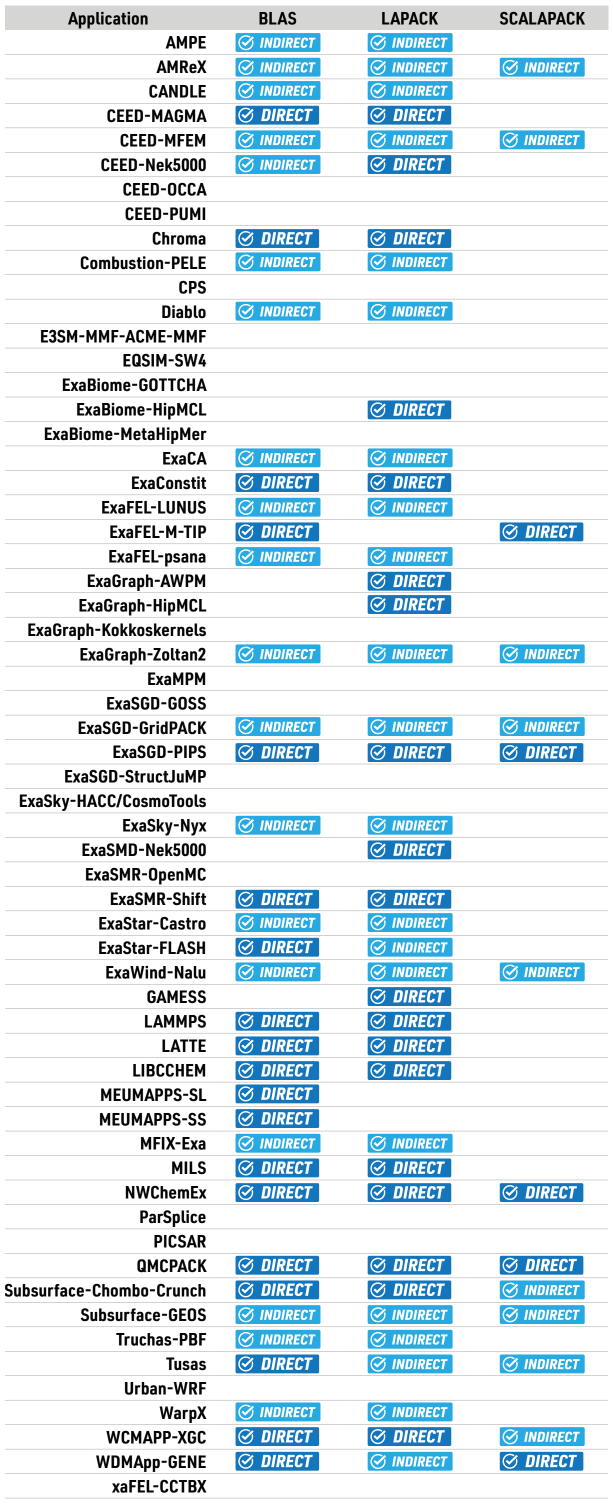
1.1 Significance of SLATE

Software for Linear Algebra Targeting Exascale (SLATE) [1] is being developed as part of the Exascale Computing Project (ECP) [2], which is a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration (NNSA). SLATE will deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems, including GPU-accelerated systems as well as more traditional multi core–only systems.

SLATE will provide coverage of existing LAPACK and ScaLAPACK functionality, including parallel implementations of Basic Linear Algebra Subroutines (BLAS), linear systems solvers, least squares solvers, and singular value and eigenvalue solvers. In this respect, SLATE will serve as a replacement for LAPACK and ScaLAPACK, which, after two decades of operation, cannot be adequately retrofitted for modern, GPU-accelerated architectures.

Figure 1.1 shows how heavily ECP applications depend on dense linear algebra software. A direct dependency means that the application's source code contains calls to the library's routines. An indirect dependency means that the applications needs to be linked with the library due to another component depending on it. Out of 60 ECP applications, 38 depend on BLAS – either directly on indirectly – 40 depend on LAPACK, and 14 depend on ScaLAPACK. In other words, the use of dense linear algebra software is ubiquitous among the ECP applications.

[1] http://icl.utk.edu/slate/
[2] https://www.exascaleproject.org



Application	BLAS	LAPACK	SCALAPACK
AMPE	✓ INDIRECT	✓ INDIRECT	
AMReX	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
CANDLE	✓ INDIRECT	✓ INDIRECT	
CEED-MAGMA	✓ DIRECT	✓ DIRECT	
CEED-MFEM	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
CEED-Nek5000	✓ INDIRECT	✓ DIRECT	
CEED-OCCA			
CEED-PUMI			
Chroma	✓ DIRECT	✓ DIRECT	
Combustion-PELE	✓ INDIRECT	✓ INDIRECT	
CPS			
Diablo	✓ INDIRECT	✓ INDIRECT	
E3SM-MMF-ACME-MMF			
EQSIM-SW4			
ExaBiome-GOTTCHA			
ExaBiome-HipMCL		✓ DIRECT	
ExaBiome-MetaHipMer			
ExaCA	✓ INDIRECT	✓ INDIRECT	
ExaConstit	✓ DIRECT	✓ DIRECT	
ExaFEL-LUNUS	✓ INDIRECT	✓ INDIRECT	
ExaFEL-M-TIP	✓ DIRECT		✓ DIRECT
ExaFEL-psana	✓ INDIRECT	✓ INDIRECT	
ExaGraph-AWPM		✓ DIRECT	
ExaGraph-HipMCL		✓ DIRECT	
ExaGraph-Kokkoskernels			
ExaGraph-Zoltan2	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
ExaMPM			
ExaSGD-GOSS			
ExaSGD-GridPACK	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
ExaSGD-PIPS	✓ DIRECT	✓ DIRECT	✓ DIRECT
ExaSGD-StructJuMP			
ExaSky-HACC/CosmoTools			
ExaSky-Nyx	✓ INDIRECT	✓ INDIRECT	
ExaSMD-Nek5000		✓ DIRECT	
ExaSMR-OpenMC			
ExaSMR-Shift	✓ DIRECT	✓ DIRECT	
ExaStar-Castro	✓ INDIRECT	✓ INDIRECT	
ExaStar-FLASH	✓ DIRECT	✓ INDIRECT	
ExaWind-Nalu	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
GAMESS		✓ DIRECT	
LAMMPS	✓ DIRECT	✓ DIRECT	
LATTE	✓ DIRECT	✓ DIRECT	
LIBCCHEM	✓ DIRECT	✓ DIRECT	
MEUMAPPS-SL	✓ DIRECT		
MEUMAPPS-SS	✓ DIRECT		
MFIX-Exa	✓ INDIRECT	✓ INDIRECT	
MILS	✓ DIRECT	✓ DIRECT	
NWChemEx	✓ DIRECT	✓ DIRECT	✓ DIRECT
ParSplice			
PICSAR			
QMCPACK	✓ DIRECT	✓ DIRECT	✓ DIRECT
Subsurface-Chombo-Crunch	✓ DIRECT	✓ DIRECT	✓ INDIRECT
Subsurface-GEOS	✓ INDIRECT	✓ INDIRECT	✓ INDIRECT
Truchas-PBF	✓ INDIRECT	✓ INDIRECT	
Tusas	✓ DIRECT	✓ INDIRECT	✓ INDIRECT
Urban-WRF			
WarpX	✓ INDIRECT	✓ INDIRECT	
WCMAPP-XGC	✓ DIRECT	✓ DIRECT	✓ INDIRECT
WDMApp-GENE	✓ DIRECT	✓ INDIRECT	✓ DIRECT
xaFEL-CCTBX			

Figure 1.1: Dependencies of ECP applications on dense linear algebra software.

1.2 Design of SLATE

SLATE is built on top of standards, such as MPI and OpenMP and de facto standard industry solutions such as NVIDIA CUDA and AMD HIP. SLATE also relies on high performance implementations of numerical kernels from vendor libraries, such as Intel MKL, IBM ESSL, NVIDIA cuBLAS, and AMD rocBLAS. SLATE interacts with these libraries through a layer of C++ APIs. Figure 1.2 shows SLATE's position in the ECP software stack.

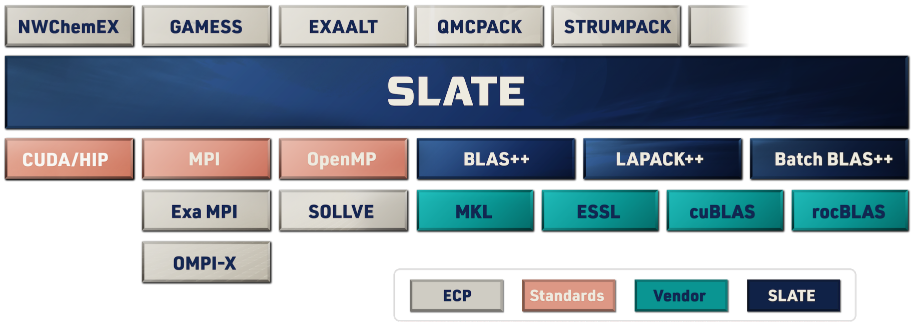


Figure 1.2: SLATE in the ECP software stack.

The following paragraphs outline the foundations of SLATE's design.

Object-Oriented Design: The design of SLATE revolves around the Tile class and the Matrix class hierarchy. The Tile class is intended as a simple class for maintaining the properties of individual tiles and implementing core serial tile operations, such as tile BLAS, while the Matrix class hierarchy maintains the state of distributed matrices throughout the execution of parallel matrix algorithms in a distributed-memory environment. Currently, the classes are structured as follows:

- **BaseMatrix** is an abstract base class for all matrices.
  - **Matrix** represents a general $m \times n$ matrix.
  - **BaseTrapezoidMatrix** is an abstract base class for all upper-trapezoid or lower-trapezoid, $m \times n$ matrices. For upper matrices, tiles $A(i, j)$ are stored for $i \leq j$. For lower matrices, tiles $A(i, j)$ are stored for $i \geq j$.
    - **TrapezoidMatrix** represents an upper-trapezoid or a lower-trapezoid, $m \times n$ matrix. The opposite triangle is implicitly zero.
      - **TriangularMatrix** represents an upper-triangular or a lower-triangular, $n \times n$ matrix.
    - **SymmetricMatrix** represents a symmetric, $n \times n$ matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry ($A_{j,i} = A_{i,j}$).
    - **HermitianMatrix** represents a Hermitian, $n \times n$ matrix, with only the upper or lower triangle stored. The opposite triangle is implicitly known by symmetry ($A_{j,i} = \bar{A}_{i,j}$).

Tiled Matrix Layout: The new matrix storage introduced in SLATE is one of its most impactful features. In this respect, SLATE represents a radical departure from other distributed linear algebra software such as ScaLAPACK or Elemental, where the local matrix occupies a contiguous memory region on each process. In contrast, tiles are first class objects in SLATE that can be individually allocated and passed to low-level tile routines. In SLATE, the matrix consists of a collection of individual tiles with no correlation between their positions in the matrix and their memory locations. At the same time, SLATE also supports tiles pointing to data in a traditional ScaLAPACK matrix layout, thereby easing an application's transition from ScaLAPACK to SLATE.

Handling of side, uplo, trans: The classical BLAS takes parameters such as `side`, `uplo`, `trans` (named "`op`" in SLATE), and `diag` to specify operation variants. Traditionally, this has meant that implementations have numerous cases. The reference BLAS has nine cases in `zgemm` and eight cases in `ztrmm` (times several sub-cases). ScaLAPACK and PLASMA likewise have eight cases in `ztrmm`. In contrast, by storing both `uplo` and `op` within the matrix object itself, and supporting inexpensive shallow copy transposition, SLATE can implement just one or two cases and map all the other cases to that implementation by appropriate transpositions. For instance, SLATE only implements one case for `gemm` (`NoTrans, NoTrans`) and handles all other cases by swapping indices of tiles and setting `trans` appropriately for the underlying tile operations.

Templating of Precisions: SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. Operations are defined so that they can be applied consistently across all precisions. SLATE's BLAS++ component provides overloaded, precision-independent wrappers for all underlying, node-level BLAS, and SLATE's PBLAS are built on top of these. Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE code should be able to accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS.

Templating of Execution Targets: Parallelism is expressed in SLATE's computational routines. Each computational routine solves a sub-problem, such as computing an LU factorization (`getrf`) or solving a linear system given an LU factorization (`getrs`). In SLATE, these routines are templated for different targets (CPU or GPU), with the code typically independent of the target. The user can choose among various target implementations:

Target::HostTask means multithreaded execution by a set of OpenMP tasks.

Target::HostNest means multithreaded execution by a nested "`parallel for`" loop.

Target::HostBatch means multithreaded execution by calling a batched BLAS routine.

Target::Devices means (multi-)GPU execution using calls to batched BLAS.

MPI Communication: Communication in SLATE relies on explicit dataflow information. When a tile is needed for computation, it is broadcast to all the processes where it is required. Rather than explicitly listing MPI ranks, the broadcast is expressed in terms of the destination (sub)matrix to be updated. This way, SLATE's messaging layer is oblivious to the mapping of tiles to processes. Also, multiple broadcasts are aggregated to allow for pipelining of MPI messages with transfers between the host and the devices. Since the set of processes involved in a broadcast is determined dynamically, the use of MPI collectives is not ideal, as it would require setting up a new subcommunicator for each broadcast. Instead, SLATE uses point-to-point MPI communication following a hypercube pattern to broadcast the data.

Node-Level Coherency: For offload to GPU accelerators, SLATE implements a memory consistency model, inspired by the MOSI cache coherency protocol [1, 2], on a tile-by-tile basis. For read-only access, tiles are mirrored in the memories of, possibly multiple, GPU devices and deleted when no longer needed. For write access, tiles are migrated to the GPU memory and returned to the CPU memory afterwards if needed. A tile's instance can be in one of three states: *Modified*, *Shared*, or *Invalid*. Additional flag *OnHold* can be set along any state, as follows:

Modified (M) indicates that the tile's data is modified. Other instances should be *Invalid*. The instance cannot be purged.

Shared (S) indicates that the tile's data is up-to-date. Other instances may be *Shared* or *Invalid*. The instance may be purged unless it is on hold.

Invalid (I) indicates that the tile's data is obsolete. Other instances may be *Modified*, *Shared*, or *Invalid*. The instance may be purged unless it is on hold.

OnHold (O) is a flag orthogonal to the other three states that indicates a hold is set on the tile instance, and the instance cannot be purged until the hold is released.

Dynamic Scheduling: Dataflow scheduling (`omp task depend`) is used to execute a task graph with nodes corresponding to large blocks of the matrix. Dependencies are tracked using dummy vectors, where each element represents a block of the matrix, rather than the matrix data itself. For multi-core execution, each large block is dispatched to multiple cores—using either nested tasking (`omp task`) or batched BLAS. For GPU execution, calls to batched BLAS are used specifically to deliver fast processing of matrix blocks that are represented as large collections of tiles.

One of the main benefits of SLATE's architecture is dramatic reduction in the size of the source code, compared to ScaLAPACK (Figure 1.3). As of August 2019, with more than two thirds of ScaLAPACK's functionality covered, SLATE's source code is 8× to 9× smaller than ScaLAPACK's.

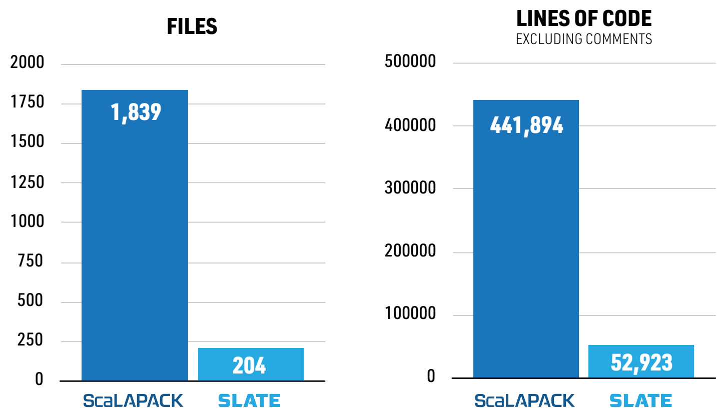


Figure 1.3: Code size comparison - ScaLAPACK vs SLATE (numbers from August 2019).

CHAPTER 2

Implementation

2.1 Singular Value Decomposition

In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix A of the form $U\Sigma V^H$, where U is an $m \times m$ real or complex unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and V is $n \times n$ real or complex unitary matrix. The diagonal entries σ_i of Σ are known as the singular values of A. The columns of U and the columns of V are known as the left-singular vectors and the right-singular vectors of A, respectively. Typically the values σ_i are ordered such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$. Typically, only the first $\min(m, n)$ columns of U and rows of V are computed, yielding the "reduced" or "economy-size" SVD, since the remaining columns of U and rows of V are multiplied by the zero part of Σ and do not contribute to A.

The SVD is the generalization of the eigendecomposition of a positive semidefinite normal matrix to any $m \times n$ matrix via an extension of the polar decomposition. The SVD is related to the eigendecomposition in the following way. The singular values are the square roots of the eigenvalues of A^TA, the columns of V are the corresponding eigenvectors, and the columns of U are the eigenvectors of AA^T.

The discovery of the SVD is attributed to four famous mathematicians, who seem to have come across it independently: Eugenio Beltrami (in 1873), Camille Jordan (in 1874), James Joseph Sylvester (in 1889), Léon César Autonneand (in 1915). The first proof of the singular value decomposition for rectangular and complex matrices seems to be by Carl Eckart and Gale J. Young in 1936 [3].

First practical methods for computing the SVD are attributed to Kogbetliantz and Hestenes [4] and resemble closely the Jacobi eigenvalue algorithm, which uses Jacobi (Givens) plane rotations.

These were replaced by the method of Golub and Kahan [5], which uses Householder reflections to reduce to bidiagonal, then plane rotations to continue the reduction to diagonal. The most popular algorithm used today is the variant of the Golub/Kahan algorithm published by Golub and Reinsch [6].

2.2 Hermitian Eigenvalue Problem

In linear algebra, an eigendecomposition or spectral decomposition is the factorization of a matrix into a canonical form, where the matrix is represented in terms of its eigenvalues and eigenvectors. An eigenvector or characteristic vector of a linear transformation is a nonzero vector that changes by a scalar factor when that linear transformation is applied to it. That is, a (non-zero) vector x of dimension n is an eigenvector of a square $n \times n$ matrix A if it satisfies the linear equation $Ax = \lambda x$. In other words, the eigenvectors are the vectors that the linear transformation A merely elongates or shrinks, and the amount that they elongate/shrink by is the eigenvalue.

A square $n \times n$ matrix A with n linearly independent eigenvectors q_i (where $i = 1, ..., n$) can be factored as $A = X \Lambda X^{-1}$ where X is the square $n \times n$ matrix whose ith column is the eigenvector x_i of A, and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{ii} = \lambda_i$. Only diagonalizable matrices can be factorized in this way.

Any Hermitian matrix can be diagonalized by a unitary matrix, and the resulting diagonal matrix has only real entries. This implies that all eigenvalues of a Hermitian matrix A with dimension n are real, and that A has n linearly independent eigenvectors. Moreover, a Hermitian matrix has orthogonal eigenvectors for distinct eigenvalues. Given that conjugate transpose of a unitary matrix is also its inverse, the Hermitian eigenvalue problem boils down to $A = X \Lambda X^H$. This means $A = X \Lambda X^T$ in the case of real symmetric matrices.

Historically, eigenvalues arose in the study of quadratic forms and differential equations. The initial discoveries are attributed to Euler, Lagrange, and Cauchy. The list of mathematicians who contributed to the field includes such famous names as Fourier, Sturm, Hermite, Brioschi, Clebsch, Weierstrass, Liouville, Schwarz, and Poincaré. Generally, Hilbert is credited with using the German word *eigen*, which means "own", to denote eigenvalues and eigenvectors, though he may have been following a related usage by Helmholtz.

The first numerical algorithm for computing eigenvalues and eigenvectors appeared in 1929, when Von Mises published the power method. One of the most popular methods today, the QR algorithm, was proposed independently by Francis [7] and Kublanovskaya [8] in 1961.

2.3 Generalized Hermitian Definite Eigenvalue Problem

The generalized Hermitian definite eigenvalue problem has various types:

- Type 1: $Az = \lambda Bz$,
- Type 2: $ABz = \lambda z$,
- Type 3: $BAz = \lambda z$,

where A is Hermitian and B is Hermitian positive-definite.

To solve it, we first reduce it to the standard eigenvalue form, $\hat{A}x = \lambda x$. The reductions for types (2) and (3) are identical; they differ in the back-transformation. First, form the Cholesky factorization of B as either $B = LL^H$ with lower triangular L, or $B = U^H U$ with upper triangular U. Then form $\hat{A}$, which overwrites A, as:

- Type 1: compute $\hat{A} = L^{-1}AL^{-H}$ or $\hat{A} = U^{-H}AU^{-1}$, as shown in Algorithm 1;
- Type 2 or 3: compute $\hat{A} = L^H AL$ or $\hat{A} = UAU^H$, as shown in Algorithm 2.

Only the lower or upper triangles of A, $\hat{A}$, and B are stored and computed on, the opposite triangle being known from symmetry. The `hegst` routine (Hermitian generalized to standard) takes A and the Cholesky factor L or U of B as input; the lower or upper triangle of $\hat{A}$ overwrites the lower or upper triangle of A on output.

After solving the standard eigenvalue problem, $\hat{A}x = \lambda x$, an eigenvector x is back-transformed to be an eigenvector z of the generalized eigenvalue problem as follows:

- Type 1 or 2: $z = L^{-H}x$ or $z = U^{-1}x$ using `trsm`;
- Type 3: $z = Lx$ or $z = U^H x$ using `trmm`.

2.4 Three Stage Algorithms

We solve both the SVD and the Hermitian eigenvalue problem by a three stage algorithm, shown in Figure 2.1:

(1) First stage reduction from full to triangular band (SVD) or Hermitian band (eigenvalue) form, which uses Level 3 BLAS.

(2) Second stage reduction band to real bidiagonal (SVD) or real symmetric tridiagonal (eigenvalue) form. This uses a bulge chasing algorithm.

(3) Third stage reduction to diagonal form, revealing the singular values or eigenvalues. Currently we use QR iteration, but could also use divide and conquer, MRRR, bisection, or other solver.

This is in contrast to the traditional algorithm used in LAPACK and ScaLAPACK that goes directly from full to bidiagonal or symmetric tridiagonal, which uses Level 2 BLAS and is memory-bandwidth limited. If $m \gg n$ (or $m \ll n$), the SVD has an optional initial reduction from tall (or wide) to square, using a QR (or LQ) factorization.

For the SVD, the first stage proceeds by computing a QR factorization of a block column to annihilate entries below the diagonal, and updating the trailing matrix, as shown in Figure 2.2. It then computes an LQ factorization of a block row to annihilate entries right of the upper bandwidth, and updates the trailing matrix. It repeats factoring block columns and block rows, until the entire matrix is brought to band form. The width of the block columns and rows is the resulting matrix bandwidth, n_b.

Algorithm 1 Reduction to standard form (type 1) pseudocode.

```
function hegst(type, $A$, $B$)
    for $k = 1, \ldots, nt$  // $nt$ = number of block rows in $A$.
        // $A(k,k) = B(k,k)^{-1} * A(k,k) * B(k,k)^{-H}$.
        hegst(type, $A(k,k)$, $B(k,k)$)
        // $A(k+1:nt,k) = A(k+1:nt,k) * B(k,k)^{H}$.
        for $m = k+1, \ldots, nt$
            trsm($B(k,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k) = [B(k+1:nt,k) * A(k,k)] + A(k+1:nt,k)$.
        for $m = k+1, \ldots, nt$
            hemm($A(k,k)$, $B(m,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k+1:nt) = [A(k+1:nt,k) * B(k+1:nt,k)] + A(k+1:nt,k+1:nt)$.
        for $m = k+1, \ldots, nt$
            for $n = k+1, \ldots, nt$
                her2k($A(m,k)$, $B(m,k)$, $A(m,n)$)
            end
        end
        // $A(k+1:nt,k) = [B(k+1:nt,k) * A(k,k)] + A(k+1:nt,k)$.
        for $m = k+1, \ldots, nt$
            hemm($A(k,k)$, $B(m,k)$, $A(m,k)$)
        end
        // $A(k+1:nt,k) = B(k+1:nt,k+1:nt) * A(k+1:nt,k)$.
        for $m = k+1, \ldots, nt$
            for $n = k+1, \ldots, nt$
                trsm($B(m,n)$, $A(m,k)$)
            end
        end
    end
    return $A$
end function
```

Algorithm 2 Reduction to standard form (type 2 or 3) pseudocode.

```
function hegst(type, $A$, $B$)
    for $k = 1, \ldots, nt$  // $nt$ = number of block rows in $A$.
        // $A(k, 1:k) = [A(k, 1:k) * B(1:k, 1:k)]$.
        for $m = 1, \ldots, k$
            for $n = 1, \ldots, k$
                trmm($B(m, n)$, $A(k, m)$)
            end
        end
        // $A(k, 1:k) = [A(k, k) * B(k, 1:k)] + A(k, 1:k)$.
        for $m = 1, \ldots, k$
            hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
        end
        // $A(1:k, 1:k) = [A(k, 1:k)^H * B(k, (1:k)^H] + A(1:k, 1:k)$.
        for $m = 1, \ldots, k$
            for $n = 1, \ldots, k$
                her2k($A(k, m)$, $B(k, m)$, $A(m, n)$)
            end
        end
        // $A(k, 1:k) = [A(k, k) * B(k, 1:k)] + A(k, 1:k)$.
        for $m = 1, \ldots, k$
            hemm($A(k, k)$, $B(k, m)$, $A(k, m)$)
        end
        // $A(k, 1:k) = [B(k, k)^H * A(k, 1:k)]$.
        for $m = 1, \ldots, k$
            trmm($B(k, k)$, $A(k, m)$)
        end
        // $A(k, k) = B(k, k)^H * A(k, k) * B(k, k)$.
        hegst(type, $A(k, k)$, $B(k, k)$)
    end
    return $A$
end function
```

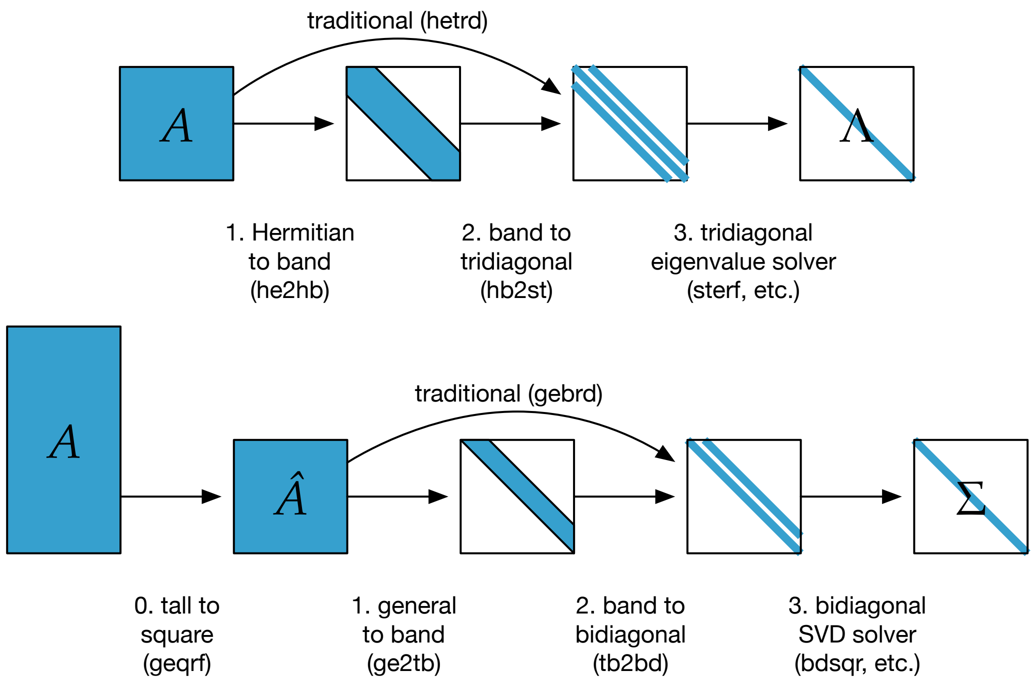

traditional (hetrd)
A
Λ
1. Hermitian to band (he2hb)
2. band to tridiagonal (hb2st)
3. tridiagonal eigenvalue solver (sterf, etc.)
A
traditional (gebrd)
Â
Σ
0. tall to square (geqrf)
1. general to band (ge2tb)
2. band to bidiagonal (tb2bd)
3. bidiagonal SVD solver (bdsqr, etc.)

Figure 2.1: Three stage Hermitian eigenvalue and SVD algorithms.
Three stage Hermitian eigenvalue (top) and SVD (bottom) algorithms.

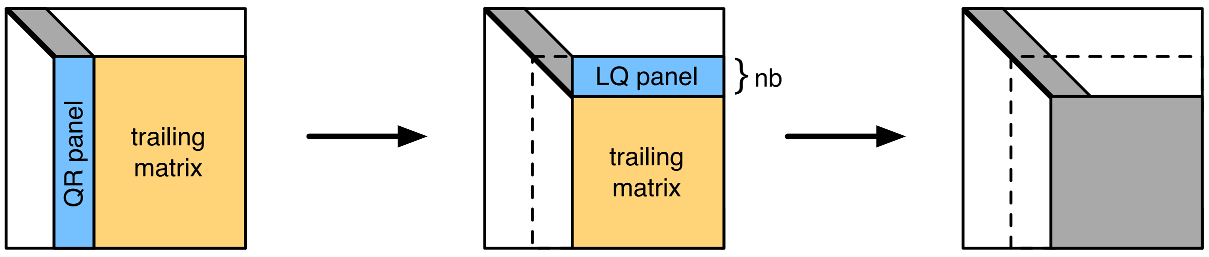


Figure 2.2: One panel of the first stage reduction to band form.

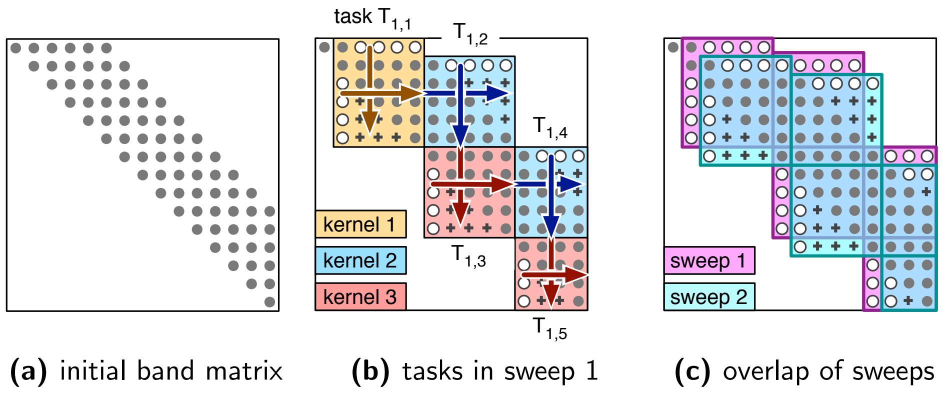


Figure 2.3: Bulge-chasing algorithm. "o" indicates eliminated elements; "+" indicates fill. Arrows show application of Householder reflector on left ($\rightarrow$), which update a block row, and on right ($\downarrow$), which update a block column.

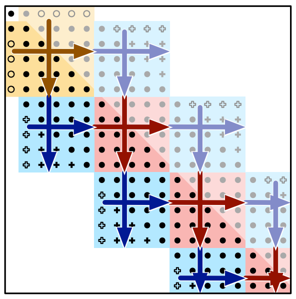


Figure 2.4: Hermitian bulge-chasing algorithm. Only the lower triangle is accessed; the upper triangle is known implicitly by symmetry.

The second stage reduces the band form to the final bidiagonal form using a bulge chasing technique. It involves $6n_bn^2$ operations, so it takes a small percentage of the total operations, which decreases with n. The operations are memory bound, but are fused together as Level 2.5 BLAS [9] for cache efficiency. We designed the algorithm to use fine-grained, memory-aware tasks in an out-of-order, data-flow task-scheduling technique that enhances data locality [10, 11].

The second stage proceeds in a series of sweeps, each sweep bringing one row to bidiagonal and chasing the created fill-in elements down to the bottom right of the matrix using successive orthogonal transformations. It uses three kernels. Kernel 1 (yellow task $T_{1,1}$ in Figure 2.3b) applies a Householder reflector from the right (indicated by the down arrow) to eliminate a row right of the superdiagonal, which also creates a bulge of fill-in beneath the diagonal. It then applies a Householder reflector from the left (indicated by the right arrow) to eliminate the first column of the bulge below the diagonal, and applies the update to the first block column only. The remainder of the bulge is not eliminated, but is instead left for subsequent sweeps to eliminate, as they would reintroduce the same nonzeros.

Kernel 2 (blue task $T_{1,2}$) continues to apply the left Householder reflector from kernel 1 (or kernel 3) to the next block column, creating a bulge above the upper bandwidth. It then applies a right Householder reflector to eliminate the first row of the bulge right of the upper bandwidth, updating only the first block row.

Kernel 3 (red task $T_{1,3}$) continues to apply the right Householder reflector from kernel 2, creating a bulge below the main diagonal. As in kernel 1, it then applies a left Householder reflector to eliminate the first column of the bulge below the diagonal and updates just the current block column. After kernel 3, kernel 2 is called again (blue task $T_{1,4}$) to continue application of the left Householder reflector in the next block column. A sweep consists of calling kernel 1 to bring a row to bidiagonal, followed by repeated calls to kernels 2 and 3 to eliminate the first column or row of the resulting bulges, until the bulges are chased off the bottom-right of the matrix.

For parallelism, once a sweep has finished the first kernel 3, a new sweep can start in parallel. This new sweep is shifted over one column and down one row, as shown in Figure 2.3c. Before task i in sweep s, denoted as $T_{s,i}$, can start, it depends on task $T_{s-1,\,i+3}$ in the previous sweep being finished, to ensure that kernels do not update the same entries simultaneously. To maximize cache reuse, tasks are assigned to cores based on their data location. Ideally, the band matrix fits into the cores' combined caches, and each sweep cycles through the cores as it progresses down the band.

For the Hermitian eigenvalue problem, the second stage shown in Figure 2.4 is very similar to the SVD second stage. Where the SVD has different reflectors from the right and left, here the same reflector is applied from the left and the right. Symmetry is taken into account, so only entries in the lower triangle are computed, while entries in the upper triangle are known by symmetry.

2.5 Eigenvector Computation

The three stage Hermitian approach to solve the eigenvalue problem of a dense matrix is to first reduce it to Hermitian band matrix form, $A = Q_1 B Q_1^H$ using Householder reflectors, then reduce the banded matrix further into a real symmetric tridiagonal matrix $B = Q_2 T Q_2^H$, finally, compute the eigenpairs of the tridiagonal matrix using an iterative method such as QR iteration, or the recursive approach of divide-and-conquer, such that $T = Q_3 \Lambda Q_3^H$. The subsequent eigenvectors are then accumulated during the back transformation phase, i.e., $X = Q_1 Q_2 Q_3$ to calculate the eigenvectors X of the original matrix A.

2.5.1 Eigenvectors of tridiagonal matrix

Once the tridiagonal reduction is achieved, the implicit QR eigensolver `steqr2` calculates the eigenvalues and optionally its associated eigenvectors of the condensed matrix structure. In SLATE (and ScaLAPACK), the `steqr2` is a modified version of the LAPACK routine `steqr` which allows each process to perform updates on the distributed matrix Q_2, and achieve parallelization during this step.

Algorithm 3 shows the call to the tridiagonal eigensolver `steqr2`. First, a matrix to store the eigenvectors $Q_{3,1D}$ of the tridaigonal matrix T is created using a 1D block row cyclic with a $n_p \times 1$ process grid, where n_p is the number of MPI processes. Then each process updates up to $(n/n_b)/n_p$ rows of the matrix $Q_{3,1D}$, where n is the matrix size and n_b is the block size used to distribute the rows of $Q_{3,1D}$. Finally, the matrix of the eigenvectors is redistributed to a 2D

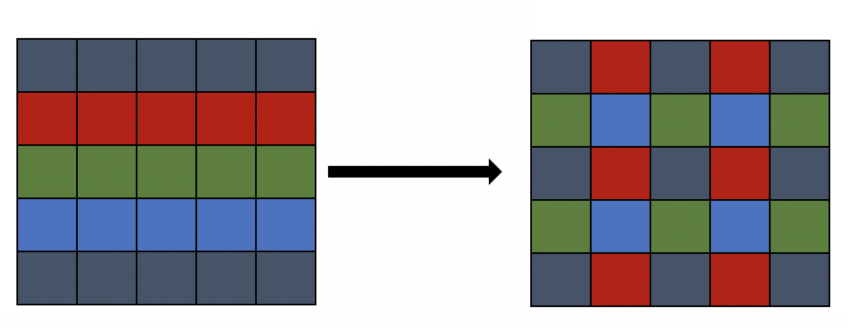


Figure 2.5: Redistribute 1D block row cyclic distributed matrix using 4×1 grid into a 2D block cyclic distribution using 2×2 grid.

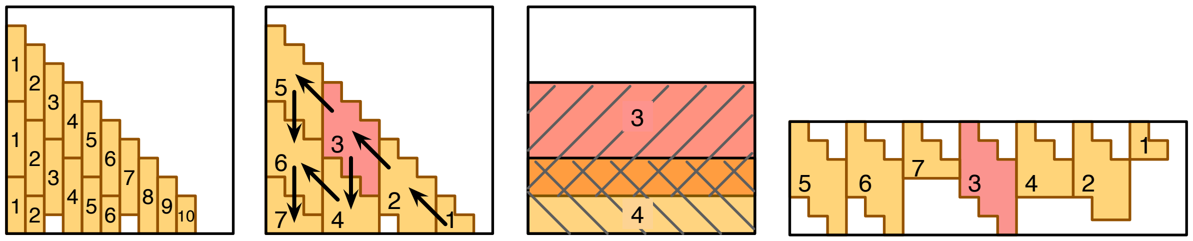
block cyclic distribution as illustrated in Figure 2.5.

Algorithm 3 Tridiagonal Eigensolver using steqr2 pseudocode.

```
function steqr2($T$, $Q_3$)
    // The “1D block row cyclic” grid configuration
    $1D = n_p \times 1$
    // Compute the number of rows owned by each processor
    $nrc = (n/n_b)/n_p$
    // Build SLATE matrix $Q_{3,1D}$ using the 1-dim grid
    $Q_{3,1D} = \text{Matrix}(nrc, n_b, n_p, 1)$
    // Call steqr2 to compute the eigenpairs of the tridiagonal matrix
    $(Q_{3,1D} \Lambda Q_{3,1D}^H) = \text{steqr2}(T)$
    // The “2D block cyclic” grid configuration
    $2D = p \times q$
    // Redistribute the 1-dim eigenvector matrix into 2-dim matrix
    $Q_3 = \text{redistribute}(Q_{3,1D})$
end function
```

2.5.2 Second stage back-transformation

The second stage back-transformation multiplies the vectors Q_3 by Q_2 from the second stage reduction from band to tridiagonal form (“bulge chasing”), to form Q_2Q_3. SLATE uses a distributed version of the scheme developed by [12]. The Householder vectors generated during the bulge chasing (Figure 2.4) are stored in a matrix V, shown in Figure 2.6. Conceptually, the vectors from each sweep i are stored in column i of the lower triangular matrix V. The vectors are blocked together into parallelograms, as shown in Figure 2.6b, to form block Householder reflectors, $H_r = I - V_r T_r V_r^H$ where V_r is the rth block of V, using the compact WY format [13]. Thus $Q_2 = H_k \cdots H_2 H_1$. Application of these H_r overlap, illustrated in Figure 2.6c, creating the dependencies between them shown in Figure 2.6b. These dependencies allow up to $\lceil \frac{mt}{2} \rceil$ updates to occur in parallel. Figure 2.7 shows these blocks and the corresponding tasks for a 10×10 block matrix. For instance, all four dark blue tasks update different rows of Q_3 and so can run in parallel. Using the OpenMP task scheduler makes taking advantage of this parallelism very easy.



(a) Householder vectors, numbered by sweep.

(b) Dependencies between blocks of vectors.

(c) Application of block reflectors 3 and 4 overlaps.

(d) Blocks stored in packed order.

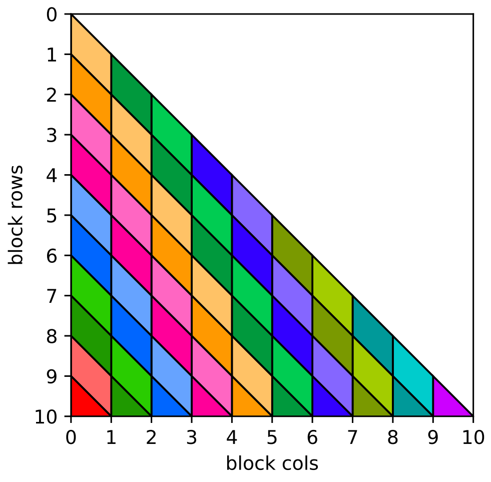
Figure 2.6: Second stage back transformation, with V block size $j_b = 3$ vectors. Block reflector 3 is highlighted to show overlap.

The routine `unmtr_hb2st`, outlined in Algorithm 4, applies Q to a matrix C; for eigenvectors, $C = Q_3$. Application of each H_r becomes a single task, with dependencies on the two rows it updates, row$[i]$ and row$[i+1]$. The parallelism in Figure 2.7b occurs automatically based on these dependencies. Within a row of C, updating each tile is independent, so we can use nested parallelism in the `parallel for` loop.

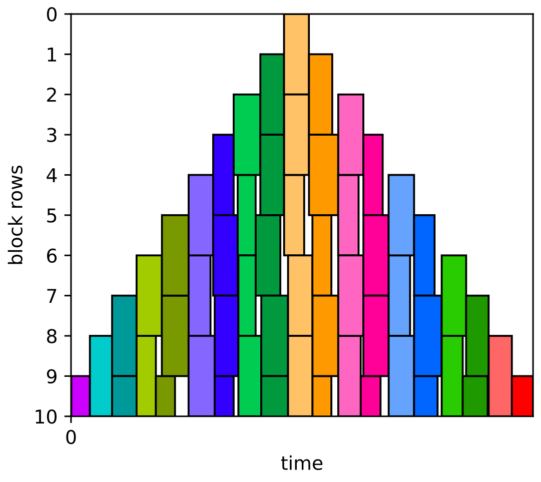
In SLATE, each parallelogram block V_r is $2n_b \times n_b$. To ease computation, instead of storing blocks in a lower triangular matrix (Figure 2.6b), each block is stored as one $2n_b \times n_b$ tile, with explicit zeros in the upper and lower triangular areas, as shown in Figure 2.6d. This allows us, for instance, to use LAPACK's `larft` function to compute T_r from V_r, and to use `gemm` instead of `trmm`. Normally, V_r has unit diagonal. SLATE stores the Householder τ values on the diagonal of V_r. During computation, the diagonal is set to 1's, and the τ values are restored afterwards.

2.5.3 First stage back-transformation

The first stage back-transformation multiplies the vectors $(Q_2 Q_3)$ by Q_1 from the first stage reduction to band, to form $X = Q_1(Q_2 Q_3)$. The routine `unmtr_he2hb` applies Q_1 or Q_1^H on the left or right of a matrix C, which is then overwritten by $Q_1 C$, $Q_1^H C$, $C Q_1$, or $C Q_1^H$. For eigenvectors, we need only the left, no-transpose case with $C = Q_2 Q_3$, to form the eigenvectors $X = Q_1(Q_2 Q_3)$. It is essentially identical to applying Q from a QR factorization, but shifted by one block-row since we reduced to band form instead of triangular form, as in QR. Thus, as in LAPACK, we can leverage the existing `unmqr` routine that applies Q from a QR factorization.



(a) Blocks of vectors, colored by independent blocks.



(b) Simulated run showing task parallelism.

Figure 2.7: Dependencies allow up to $\lceil \frac{mt}{2} \rceil$ parallel tasks.

Algorithm 4 `unmtr_hb2st` back-transformation pseudocode. Indices are block rows/cols.

```
function unmtr_hb2st($V, C$)
    // $C$ is $mt \times nt$ block rows/cols, blocksize $n_b \times n_b$
    // $V$ is $mt(mt+1)/2$ blocks, blocksize $2n_b \times n_b$
    for $j = mt - 1$ to 0
        for $i = j$ to $mt - 1$
            task depend in, out on row[$i$] and row[$i+1$]
                $r = i - j + j \cdot mt - j(j-1)/2$
                Broadcast $V_r$
                Compute $T$ from $V_r$ (larft)
                $D = V_r T$ (gemm or trmm)
                parallel for $k = 0$ to $nt - 1$
                    if $C_{i:i+1,k}$ are local then
                        // Compute $QC = (I - VTV^H)C$
                        $W = V_r^H C_{i:i+1,k}$
                        $C_{i:i+1,k} = C_{i:i+1,k} - DW$
                    end
                end
            end task
        end
    end
end function
```

CHAPTER 3

Performance

3.1 Environment

3.1.1 Hardware

Performance numbers were collected using the Summit system [1,2] at the Oak Ridge Leadership Computing Facility (OLCF). Summit is equipped with IBM POWER9 processors and NVIDIA V100 (Volta) GPUs. Each of Summit's nodes contains two POWER9 CPUs (with 22 cores each) and six V100 GPUs. Each node has 512 GB of DDR4 memory, and each GPU has 16 GB of HBM2 memory. NVLink 2.0 provides all-to-all 50 GB/s connections for one CPU and three GPUs (i.e., one CPU is connected to three GPUs with 50 GB/s bandwidth each, and each GPU is connected to the other two with 50 GB/s bandwidth each). The two CPUs are connected with a 64 GB/s X Bus. Each node has a Mellanox enhanced-data rate (EDR) InfiniBand network interface controller (NIC) that supports 25 GB/s of bi-directional traffic. Figure 3.1 shows the hardware architecture of a Summit node.

3.1.2 Software

The software environment used for the SVD experiments included:

- GNU Compiler Collection (GCC) 6.4.0,
- NVIDIA CUDA 10.1.105,
- IBM Engineering Scientific Subroutine Library (ESSL) 6.1.0,

[1] https://www.olcf.ornl.gov/summit/
[2] https://en.wikichip.org/wiki/supercomputers/olcf-4

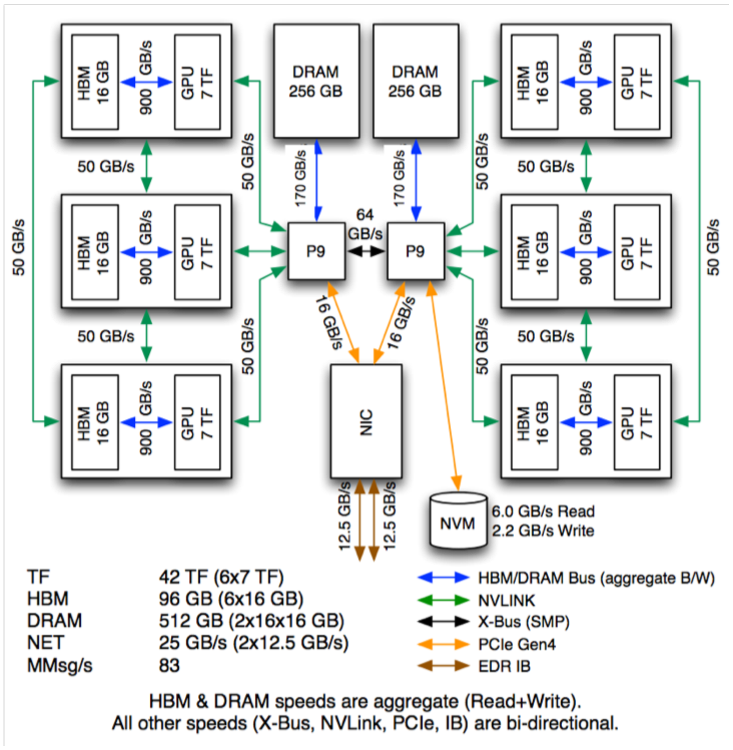


Figure 3.1: Summit node architecture.

- IBM Spectrum MPI 10.3.0.0,
- Netlib LAPACK 3.8.0, and
- Netlib ScaLAPACK 2.0.2.

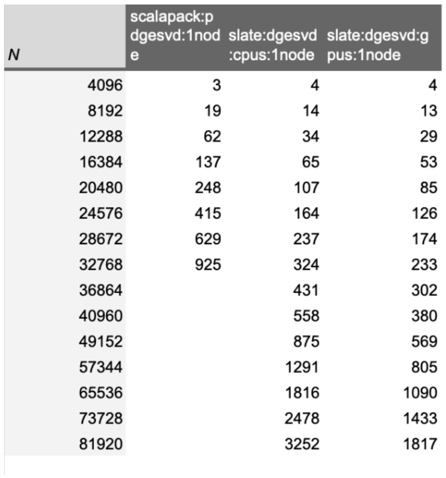
For the generalized Hermitian eigenvalues, these were updated to:

- GNU Compiler Collection (GCC) 8.1.1,
- NVIDIA CUDA 10.1.243,
- IBM Engineering Scientific Subroutine Library (ESSL) 6.1.0,
- IBM Spectrum MPI 10.3.1.2,
- Netlib LAPACK 3.8.0, and
- Netlib ScaLAPACK 2.0.2.

3.2 Results

Here, we present the results of our preliminary performance experiment with the singular value solve. Figure 3.2 shows the execution time of ScaLAPACK compared to SLATE with and without GPU acceleration. Two MPI ranks are mapped to one node of Summit, i.e., one rank is mapped to one CPU socket (22 cores) and three GPU devices. Only singular values are computed in all cases (no vectors).

For a matrix of size 32,768 $\times$ 32,768, ScaLAPACK took 925 seconds, while SLATE took 324 seconds using CPUs only and 233 seconds with GPU acceleration. That is, SLATE was almost



N	scalapack:pdgesvd:1node	slate:dgesvd:cpus:1node	slate:dgesvd:gpus:1node
4096	3	4	4
8192	19	14	13
12288	62	34	29
16384	137	65	53
20480	248	107	85
24576	415	164	126
28672	629	237	174
32768	925	324	233
36864		431	302
40960		558	380
49152		875	569
57344		1291	805
65536		1816	1090
73728		2478	1433
81920		3252	1817

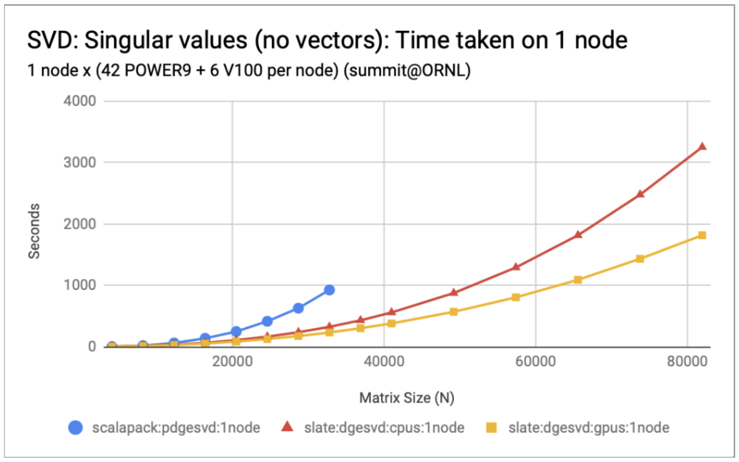


Figure 3.2: SVD performance comparison.

3 times faster without acceleration and almost 4 times faster with acceleration. Since the performance gap increases with the problem size, we expect SLATE to be an order of magnitude faster for matrices in the $O(100K)$ range without acceleration, and further benefit $3\times$ to $4\times$ from acceleration.

For the generalized Hermitian definite eigenvalue problem, Figure 3.3 shows the performance for conversion from the generalized form to standard form (`hegst`). On the CPU host, SLATE closely matches ScaLAPACK's performance, while when using GPUs, SLATE gets a modest acceleration. We will continue to investigate ways to optimize the performance.

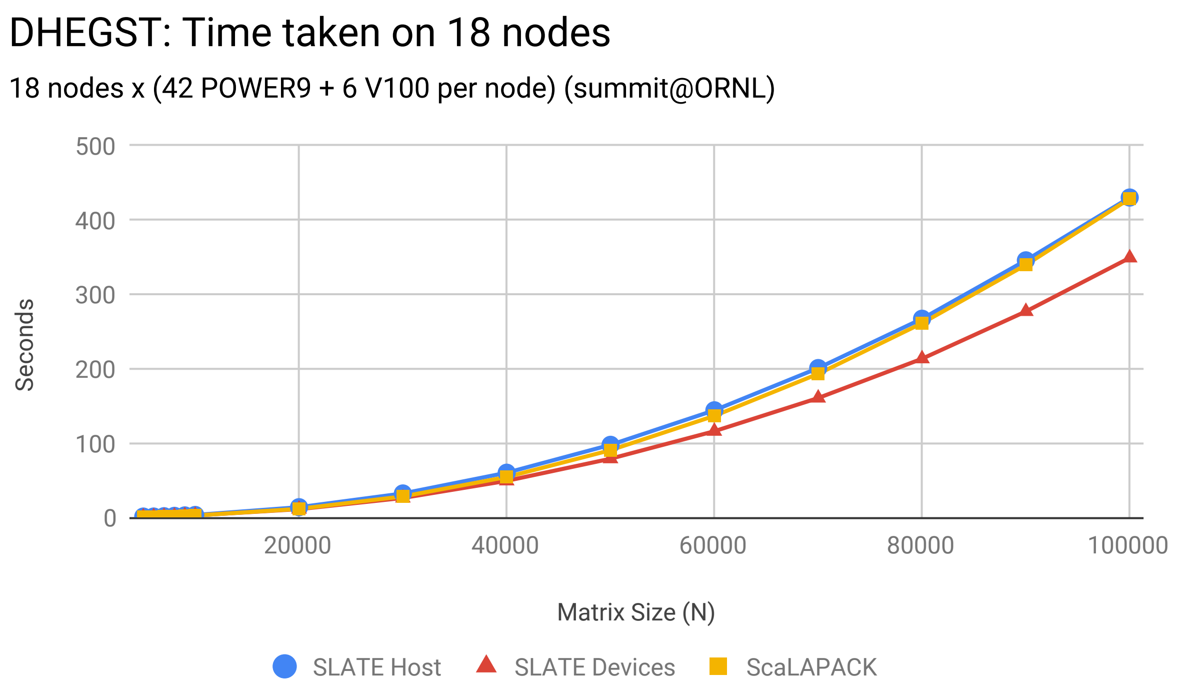


Figure 3.3: Generalized to standard eigenvalue performance comparison.

Bibliography

[1] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News*, 14(2): 414–423, 1986.

[2] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[3] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[4] Magnus R Hestenes. Inversion of matrices by biorthogonalization and related results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51–90, 1958.

[5] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[6] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear Algebra*, pages 134–151. Springer, 1971.

[7] John GF Francis. The qr transformation a unitary analogue to the lr transformation—part 1. *The Computer Journal*, 4(3):265–271, 1961.

[8] Vera N Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *USSR Computational Mathematics and Mathematical Physics*, 1(3):637–657, 1962.

[9] Gary W Howell, James W Demmel, Charles T Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):14, 2008. doi: 10.1145/1356052.1356055.

[10] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International*

Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), page 90. ACM, 2013. doi: 10.1145/2503210.2503292.

[11] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 8:1–8:11. ACM, 2011. doi: 10.1145/2063384.2063394.

[12] Azzam Haidar, Stanimire Tomov, Jack Dongarra, Raffaele Solca, and Thomas 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*, 28(2):196–209, 2014. doi: 10.1177/1094342013502097.

[13] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10 (1):53–57, 1989. doi: 10.1137/0910005.