SLATE

Software for Linear Algebra Targeting Exascale

Mark Gates

Jakub Kurzak, Asim YarKhan, Ali Charara, Jamie Finney,
Dalal Sukkari, Mohammed Al Farhan, Ichitaro Yamazaki,
Panruo Wu, Jack Dongarra

Examples & Slides

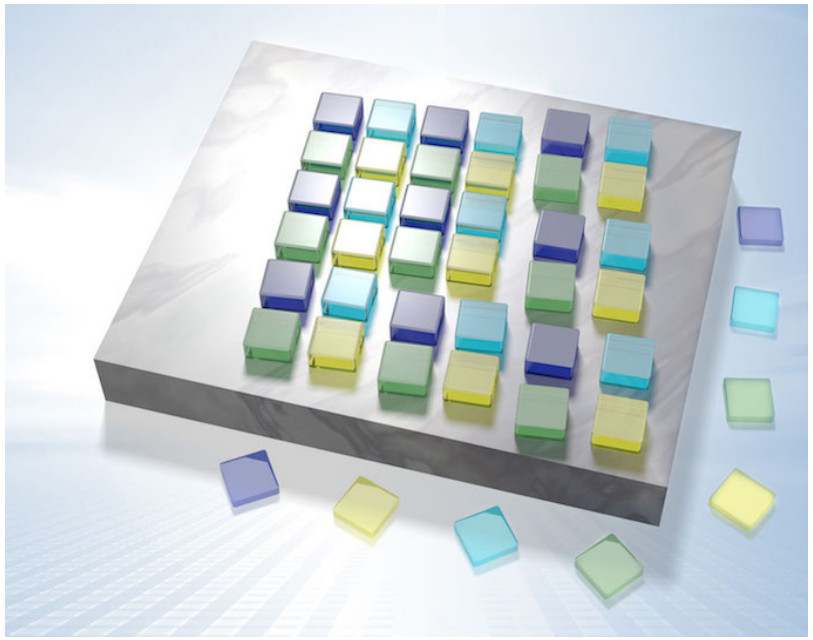
https://bitbucket.org/icl/slate-tutorial

https://confluence.exascaleproject.org/display/2020ECPAM/Sessions

ICL INNOVATIVE COMPUTING LABORATORY

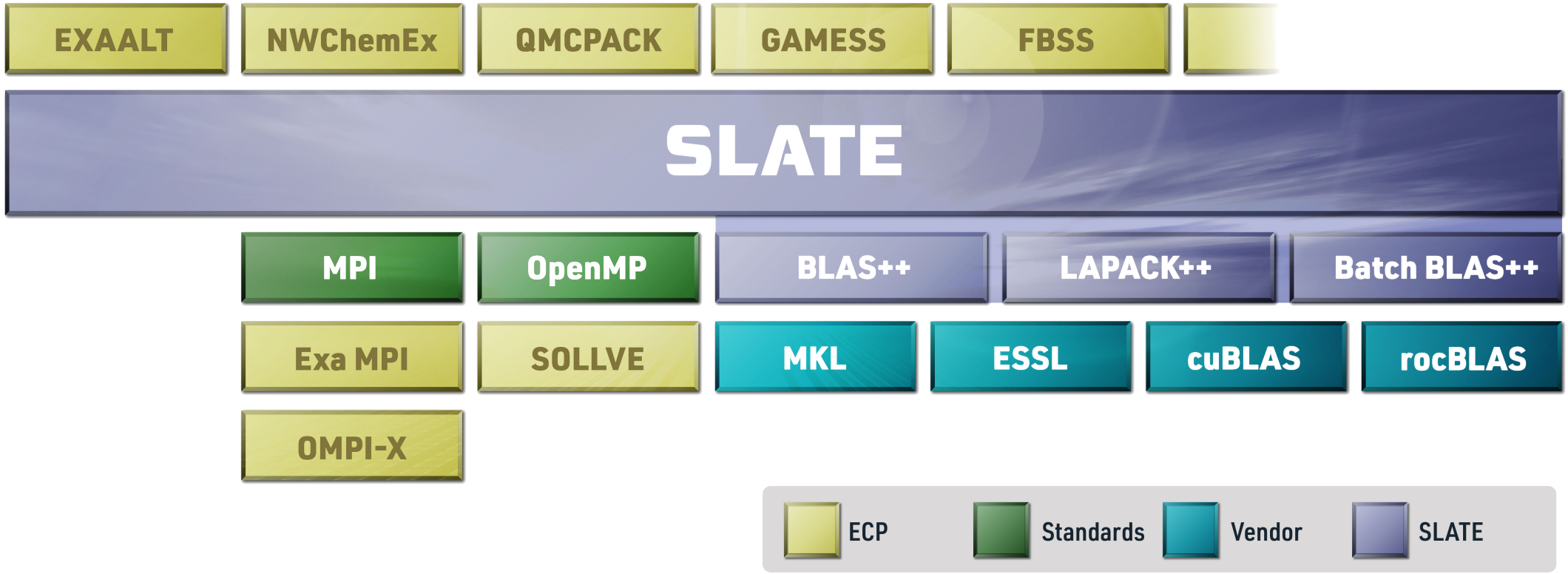
THE UNIVERSITY OF TENNESSEE KNOXVILLE

SLATE design



- **Modern C++ replacement for ScaLAPACK**
  - Code templated for precision
  - Backwards compatibility layer
- **Flexible**
  - Non-uniform block sizes
  - Arbitrary distributions; default 2D block-cyclic
- **Standards based**
  - MPI for distributed communication
  - OpenMP 4.5 tasks for shared memory parallelism
  - GPU support — currently CUDA; future HIP and Intel
- **Developed from scratch as ECP project**

SLATE in ECP ecosystem



https://bitbucket.org/icl/slate-tutorial

Outline

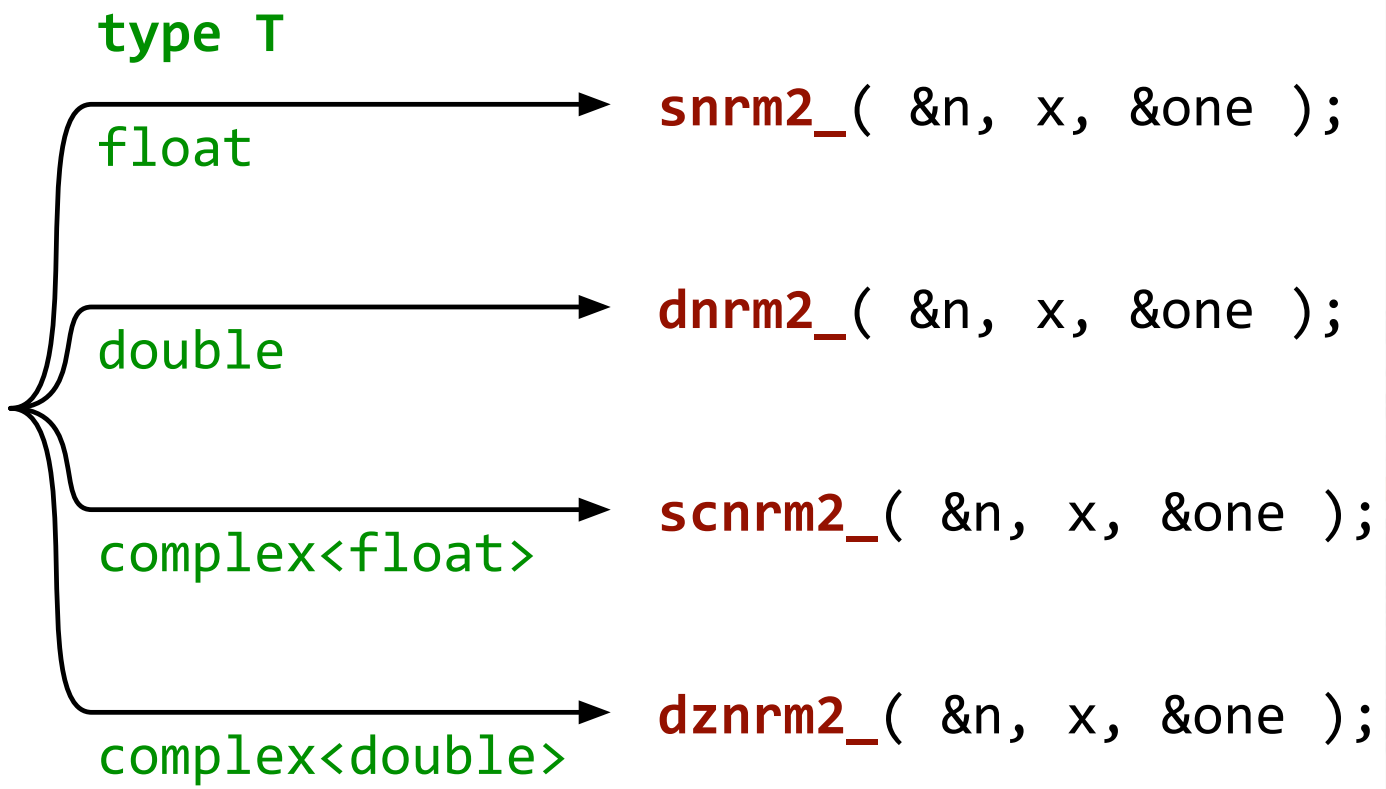
- **BLAS++ and LAPACK++**
- Creating & accessing matrices
- SLATE routines
  - Norms and BLAS
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - Least squares: $AX \approx B$ using QR or LQ
  - SVD and Hermitian eigenvalues
- Future
  - Non-symmetric eigenvalues

BLAS++ and LAPACK++

blas00_gemm.cc
lapack00_potrf.cc

- Overloaded C++ wrappers around Fortran BLAS and LAPACK
- Make real & complex versions identical

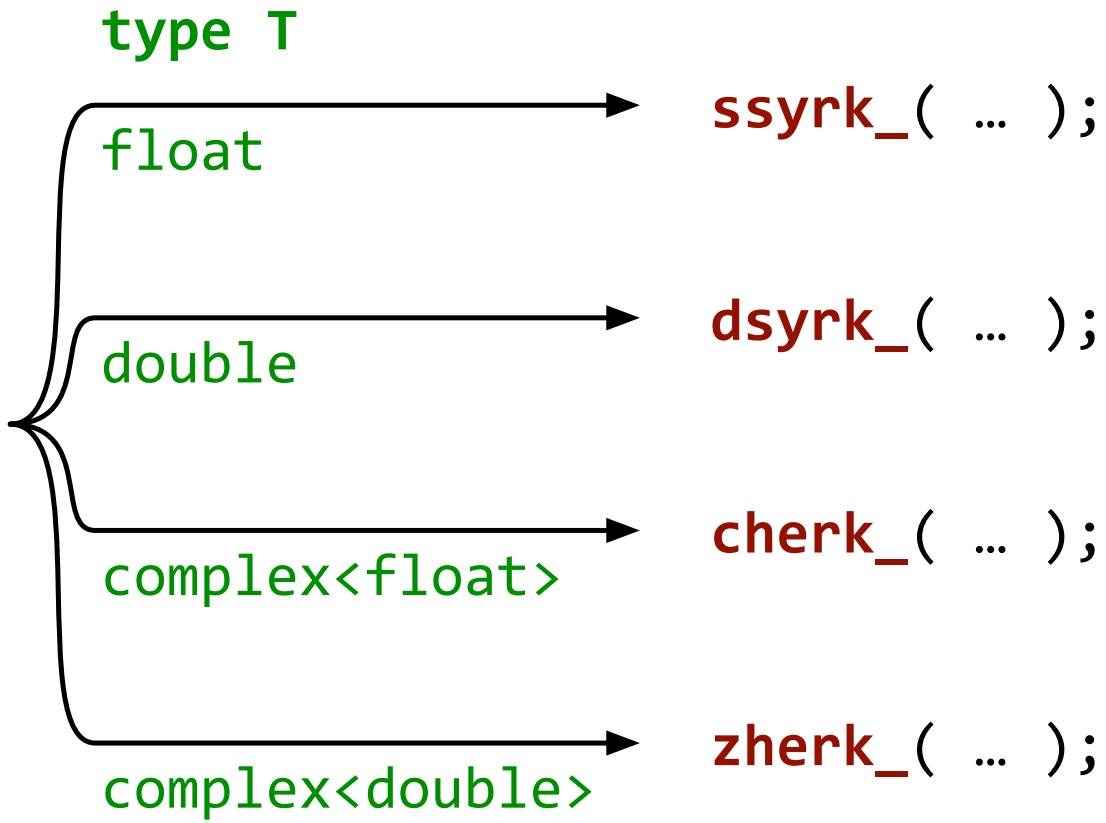
```
template <typename T>
void myfunc( int n, T* x )
{
    …
    blas::real_type<T> norm
     = blas::nrm2( n, x, 1 );
    …
}
```



Strategy: write complex precision version

- Hermitian (he) ⇒ symmetric (sy) in real

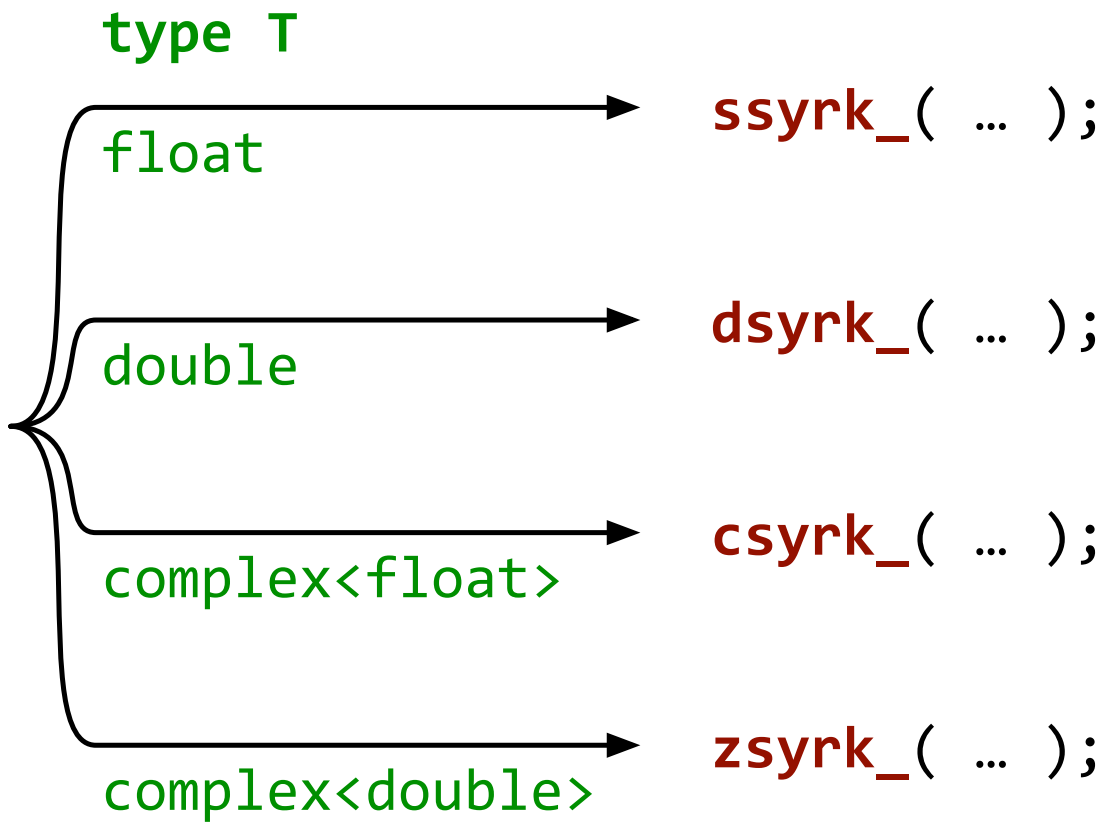
```
template <typename T>
void myfunc( T* A, … )
{
    …
    blas::herk( …, A, … );
    …
}
```



Strategy: write complex precision version

- Complex symmetric matrices

```
template <typename T>
void myfunc( T* A, … )
{
    …
    blas::syrk( …, A, … );
    …
}
```



BLAS++ and LAPACK++

- **C++ enums**
  - BLAS++ supports both column and row major, with no to minimal overhead, as in CBLAS
  - LAPACK++ supports only column major
  - Op represents transpose operation, $\mathrm{op}(\mathrm{A}) = \mathrm{A}, \mathrm{A}^{\mathrm{T}}, \mathrm{A}^{\mathrm{H}}$

```
// triangular solve matrix, A^H X = alpha B
blas::trsm( blas::Layout::ColMajor,
            blas::Side::Left,
            blas::Uplo::Lower,
            blas::Op::ConjTrans,
            blas::Diag::Unit,
            m, n, alpha, A, lda, B, ldb );
```

BLAS++ and LAPACK++

- **LAPACK++ allocates optimal workspace sizes**
- **Throws hard errors (e.g., lda < m)**
- **Returns numerical errors (e.g., singular matrix, LAPACK info > 0)**
- **int64_t for all integer arguments**
  - If vendor library is ILP64 (64-bit int), int64_t passed through
  - If vendor library is LP64 (32-bit int), int64_t converted; throws overflows

BLAS++ and LAPACK++ coverage

- Level 1 BLAS on CPU
- Level 2 BLAS on CPU
- Level 3 BLAS on CPU and GPU (cuBLAS)
- Level 3 Batched BLAS on CPU and GPU (cuBLAS)
- Most LAPACK functions of interest

BLAS++ types

blas01_util.cc

- `blas::real_type<T1, ...>` **strips complex**

```
real_type< double >          == double
real_type< complex<float> > == float
```

- `blas::complex_type<T1, ...>` **adds complex**

```
complex_type< double >          == complex<double>
complex_type< complex<float> > == complex<float>
```

- `blas::scalar_type<T1, ...>` **simplest type to represent types T1, ...**

```
scalar_type< float, double >          == double
scalar_type< double, complex<float> > == complex<double>
std::common_type_t< double, complex<float> > == complex<float>
```

BLAS++ types

blas01_util.cc

- `blas::real_type<T1, ...>` **strips complex**

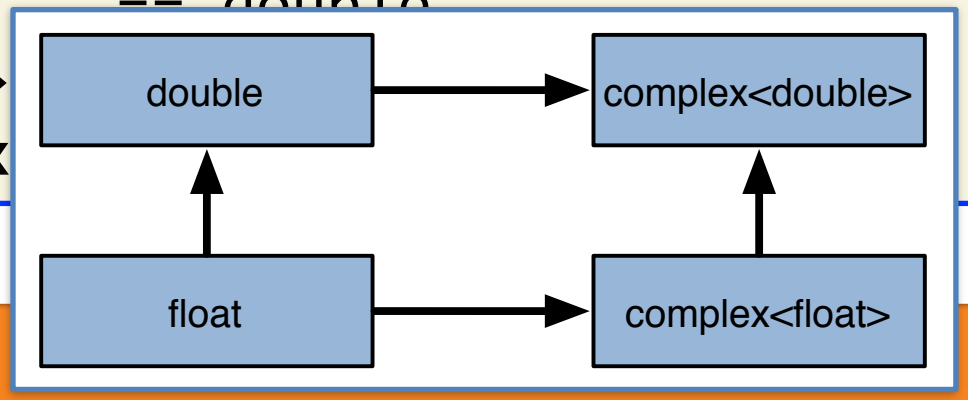
```
real_type< double >          == double
real_type< complex<float> > == float
```

- `blas::complex_type<T1, ...>` **adds complex**

```
complex_type< double >          == complex<double>
complex_type< complex<float> > == complex<float>
```

- `blas::scalar_type<T1, ...>` **simplest type to represent types T1, ...**

```
scalar_type< float, double >
scalar_type< double, complex<float>
std::common_type_t< double, complex
```



BLAS++ utilities

blas01_util.cc

- `std` **provides** `real, imag, conj` **for complex only**
- `blas` **provides** `real, imag, conj` **for any type**
- `blas::is_complex<T>::value` **is true if type T is complex**

```
template <typename T>
void test_util( T x ) {
    using blas::real;
    using blas::imag;
    if (blas::is_complex<T>::value)
        printf( "x = %7.4f + %7.4fi\n", real(x), imag(x) );
    else
        printf( "x = %7.4f\n”, real(x) );
```

BLAS++ utilities

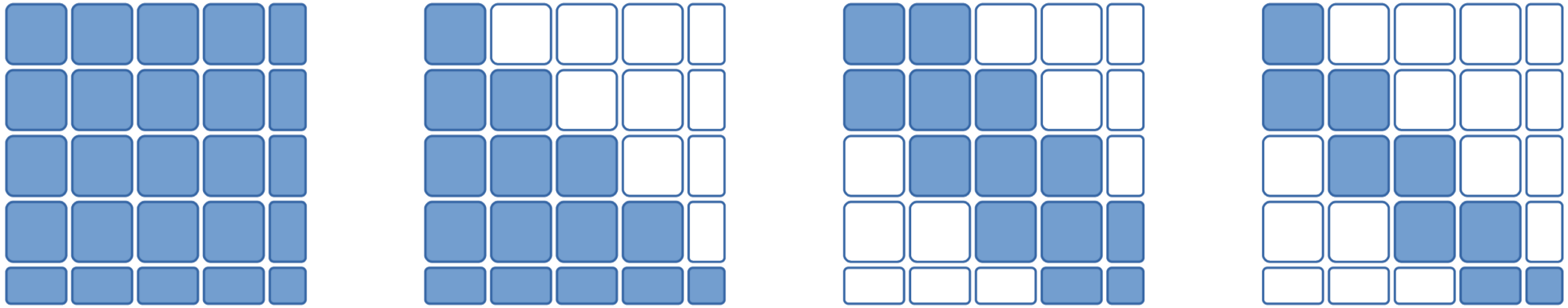
- `blas::min( a, b, c, ... )`
- `blas::max( a, b, c, ... )`
  - Arbitrary number of items
  - Arbitrary types — returns scalar_type<T1, T2, ...>
  - `std::max( n, 0 )` may not compile, e.g., max(int64_t, int)
  - `blas::max( n, 0 )` works when sensible

Outline

- BLAS++ and LAPACK++
- **Creating & accessing matrices**
- SLATE routines
  - Norms and BLAS
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - Least squares: $AX \approx B$ using QR or LQ
  - SVD and Hermitian eigenvalues
- Future
  - Non-symmetric eigenvalues

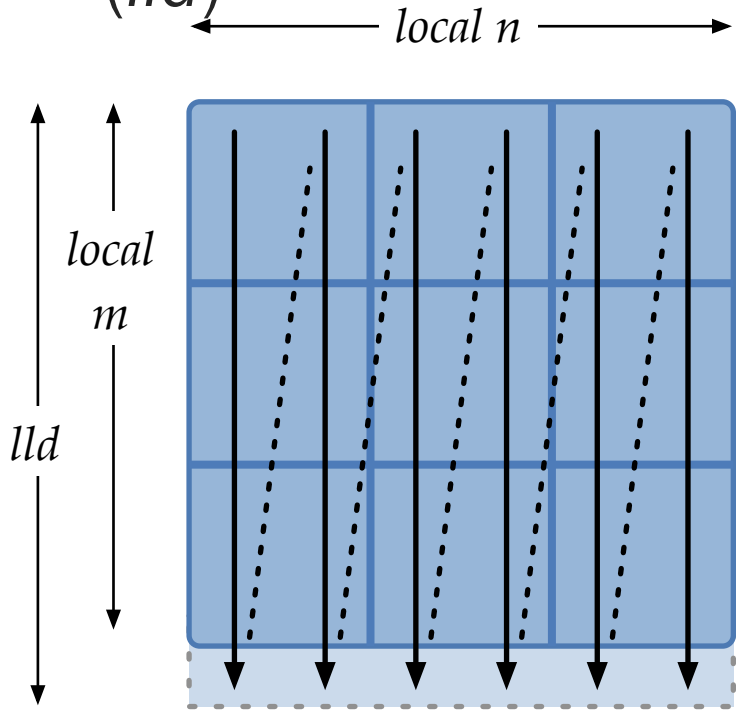
slate::Matrix

- **Map from tile indices { i, j, device } to Tile**
  - Global addressing of tiles
  - Distributed & GPU support is easy
  - No wasted memory for symmetric and triangular matrices
  - Accommodates band & block-sparse matrices
  - Backwards compatible with (Sca)LAPACK



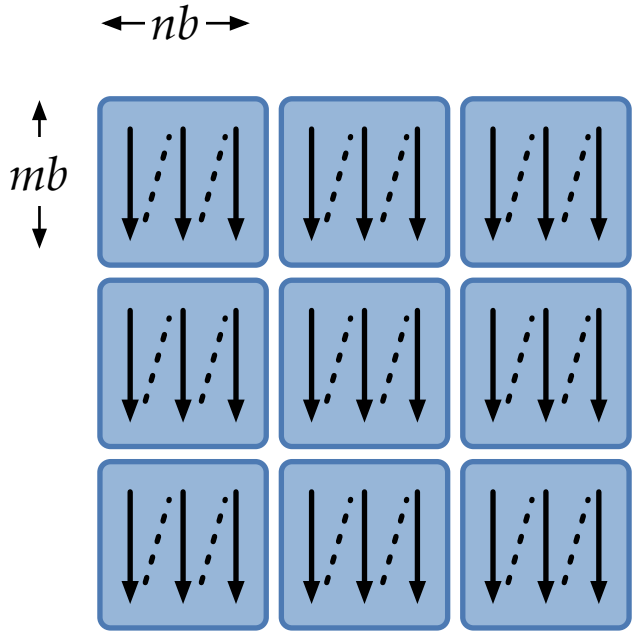
slate::Tile Format

- **ScaLAPACK format**
  - 2D block cyclic with stride (*lld*)



OR

- **Individually allocated tiles**



Distributed Tile Layout

Global matrix

$q = 2$

$p = 2$

0	2	0	2
1	3	1	3
0	2	0	2
1	3	1	3

Rank 0

0		0	
0		0	

Rank 2

	2		2
	2		2

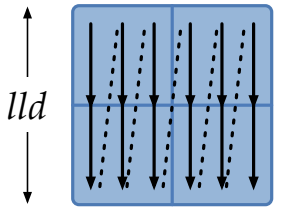
Rank 1

1		1	
1		1	

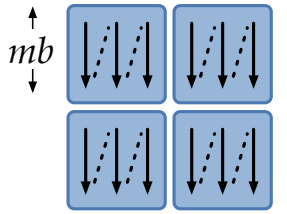
Rank 3

	3		3
	3		3

Stored as ScaLAPACK tiles



or individual tiles



Distributed Tile Layout

Global matrix

$q = 2$

$p = 2$

0	2	0	2
1	3	1	3
0	2	0	2
1	3	1	3

Rank 0

0		0	
0		0	

Rank 2

0	2		2
0	2		2

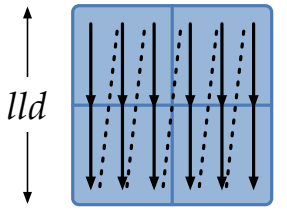
Rank 1

0			
1		1	
0			
1		1	

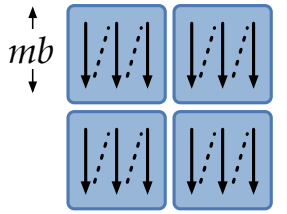
Rank 3

0			
	3		3
0			
	3		3

Stored as ScaLAPACK tiles



or individual tiles



```
// slate00_matrix.cc
// create 2000 x 1000 matrix on 2 x 2 MPI process grid
#include <slate.hh>
int main( int argc, char** argv ) {
  int provided = 0;
  int err = MPI_Init_thread( &argc, &argv,
                             MPI_THREAD_MULTIPLE, &provided );
  int64_t m=2000, n=1000, nb=256, p=2, q=2;
  slate::Matrix<double> A( m, n, nb, p, q, MPI_COMM_WORLD );
  A.insertLocalTiles();
  for (int64_t j = 0; j < A.nt(); ++j)
    for (int64_t i = 0; i < A.mt(); ++i)
      if (A.tileIsLocal( i, j )) {
        slate::Tile<double> T = A( i, j );
        random_matrix( T.mb(), T.nb(), T.data(), T.stride() );
      }
  err = MPI_Finalize();
}
```

```
# Compile, with BLAS++, LAPACK++, and SLATE
# installed in /opt/slate
mpicxx -fopenmp -Wall \
       -I/opt/slate/include \
       -c -o slate00_matrix.o slate00_matrix.cc

# Link. Assumes -lslate pulls in its dependencies
# (BLAS, LAPACK, CUDA, ...)
mpicxx -fopenmp -Wall \
       -L/opt/slate/lib \
       -Wl,-rpath,/opt/slate/lib -lslate \
       -o slate00_matrix slate00_matrix.o

# Run
mpirun -np 4 ./slate00_matrix
```

Creating Matrix from ScaLAPACK format data

- **Insert all local tiles pointing to ScaLAPACK data**

```
auto A = slate::Matrix<double>::fromScaLAPACK(
    m, n, data, lld, nb, p, q, mpi_comm );
```

  - m × n: Global matrix dimensions
  - data: Local array
  - lld: Local leading dimension (stride) for data
  - nb: Block size
  - p × q: MPI process grid
  - mpi_comm: MPI communicator
- **User retains ownership of data!**

Create Matrix with SLATE allocating data

- Create empty Matrix (no tiles)

```
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );
```

- Insert SLATE allocated tiles

```
A.insertLocalTiles();
```

equivalent to

```
for (int64_t j = 0; j < A.nt(); ++j)
    for (int64_t i = 0; i < A.mt(); ++i)
        if (A.tileIsLocal( i, j ))
            A.tileInsert( i, j );
```

Create Matrix with SLATE allocating data

- Create empty Matrix (no tiles)

```
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );
```

- Insert SLATE allocated tiles — on GPU devices

```
A.insertLocalTiles( Target::Devices );
```

equivalent to

```
for (int64_t j = 0; j < A.nt(); ++j)
    for (int64_t i = 0; i < A.mt(); ++i)
        if (A.tileIsLocal( i, j ))
            A.tileInsert( i, j, A.tileDevice(i, j) );
```

Create Matrix with user allocated data

- Create empty Matrix (no tiles)

```
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );
```

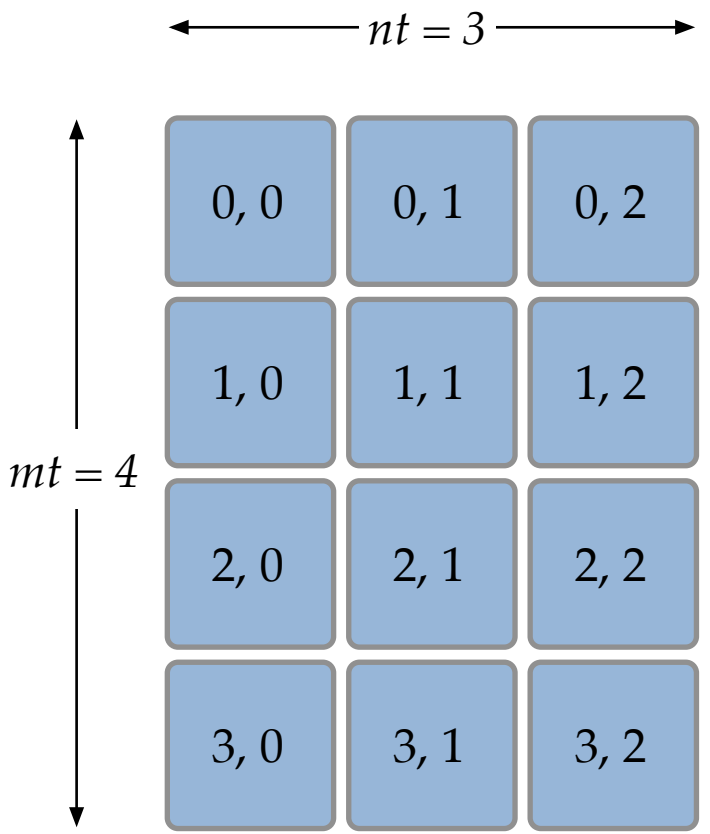
- User allocated tiles, from pointers in *data(i, j)*

```
for (int64_t j = 0; j < A.nt(); ++j)
    for (int64_t i = 0; i < A.mt(); ++i)
        if (A.tileIsLocal( i, j ))
            A.tileInsert( i, j, data(i, j), lld );
```

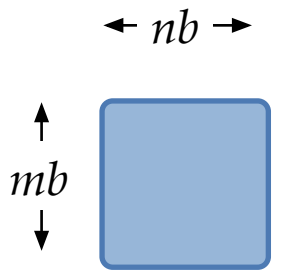
- User retains ownership of data!

Accessing data

- **For Matrix A**
  - A.mt(), A.nt() ⇒ # block rows & block cols
  - T = A(i, j) ⇒ (i, j) tile on CPU host
  - T = A(i, j, dev) ⇒ (i, j) tile on GPU device



- **For Tile T**
  - T.mb(), T.nb() ⇒ # rows & cols in tile
  - T(ii, jj) ⇒ (ii, jj) entry of tile T; includes conj
  - T.at(ii, jj) ⇒ reference to (ii, jj) entry of tile T; no conj



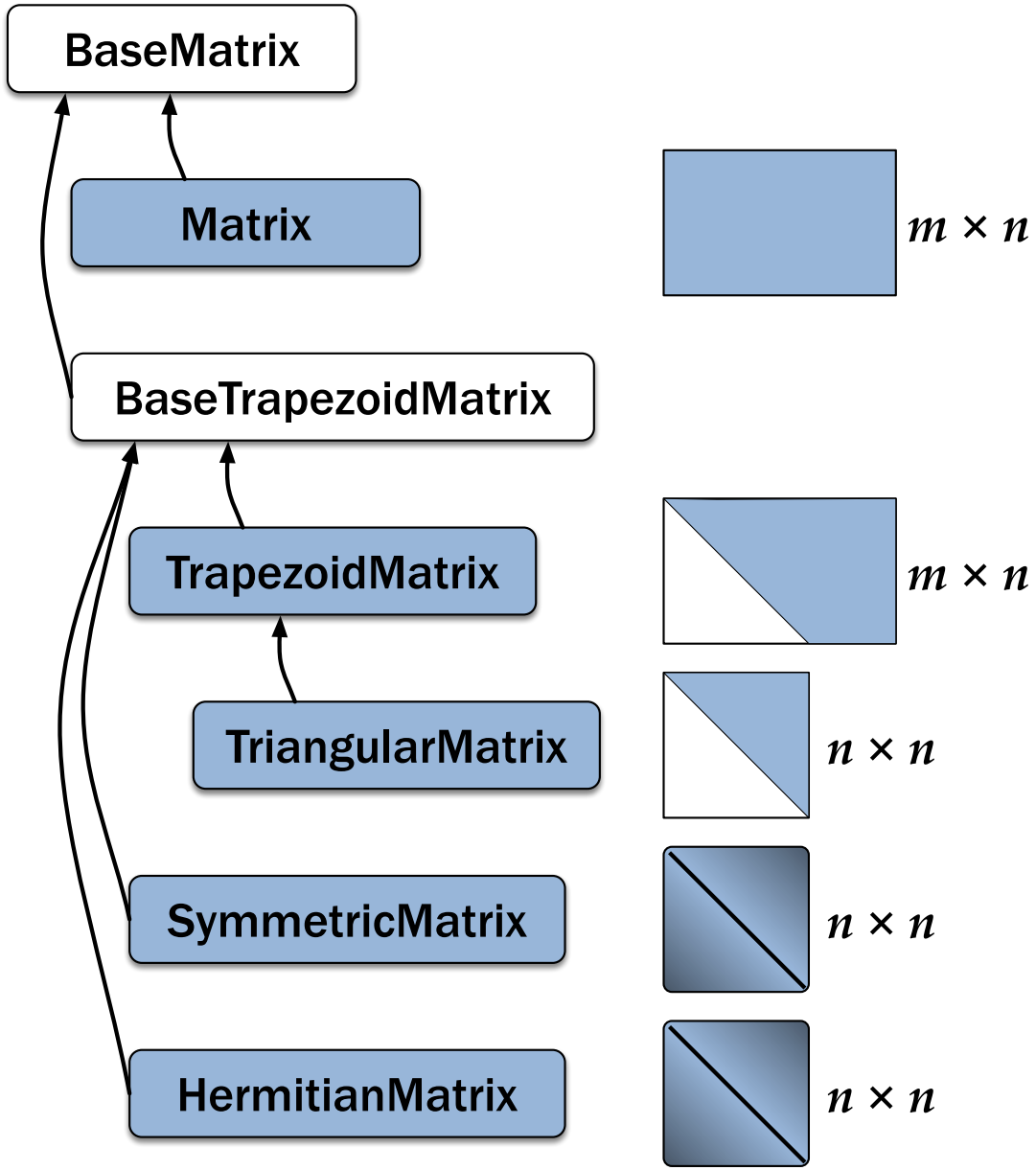
Accessing data

```
// loop over tiles
for (int64_t j = 0; j < A.nt(); ++j) {
    for (int64_t i = 0; i < A.mt(); ++i) {
        if (A.tileIsLocal( i, j )) {
            // loop over entries in tile
            // (assuming it exists on CPU)
            auto T = A( i, j );
            for (int64_t jj = 0; jj < T.nb(); ++jj)
                for (int64_t ii = 0; ii < T.mb(); ++ii)
                    T.at( ii, jj ) = ...;
        }
    }
}
```

- Caveat: accessing tile entries this way is inefficient in inner loops

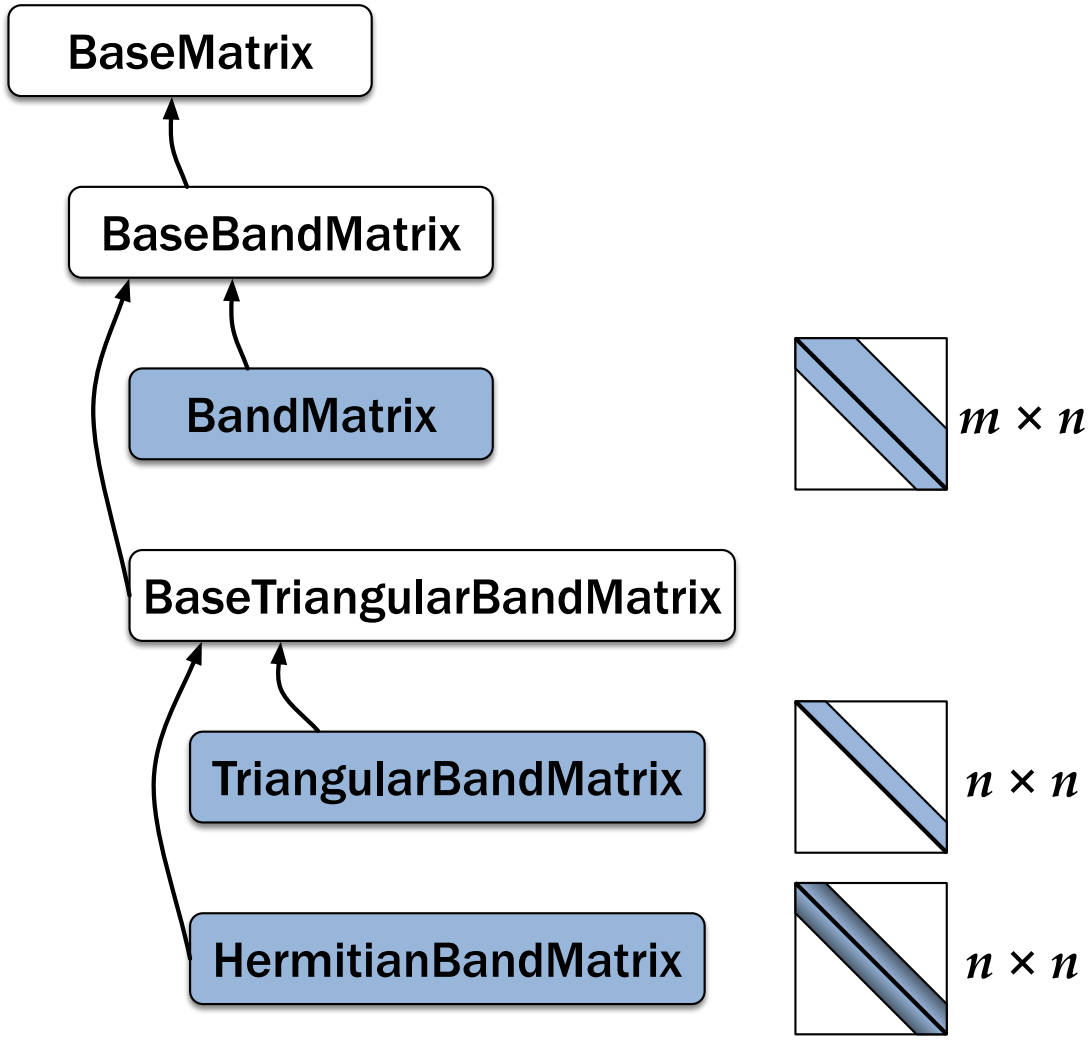
Matrix types

- **Class hierarchy**
  - Enforces correct argument types
- **BaseTrapezoid**
  - Adds upper & lower
- **Trapezoid**
  - Adds unit & non-unit diagonals
  - Opposite triangle is zero
- **Symmetric and Hermitian**
  - Opposite triangle is symmetric



Band Matrix types

- **BaseBand**
  - Adds bandwidths
- **BaseTriangularBand**
  - Adds upper & lower
- **Triangular**
  - Adds unit & non-unit diagonals
  - Opposite triangle is zero
- **Hermitian (and Symmetric)**
  - Opposite triangle is symmetric
- **Currently need to explicitly zero out tiles**
- **Relatively new, so expect more development — give feedback**



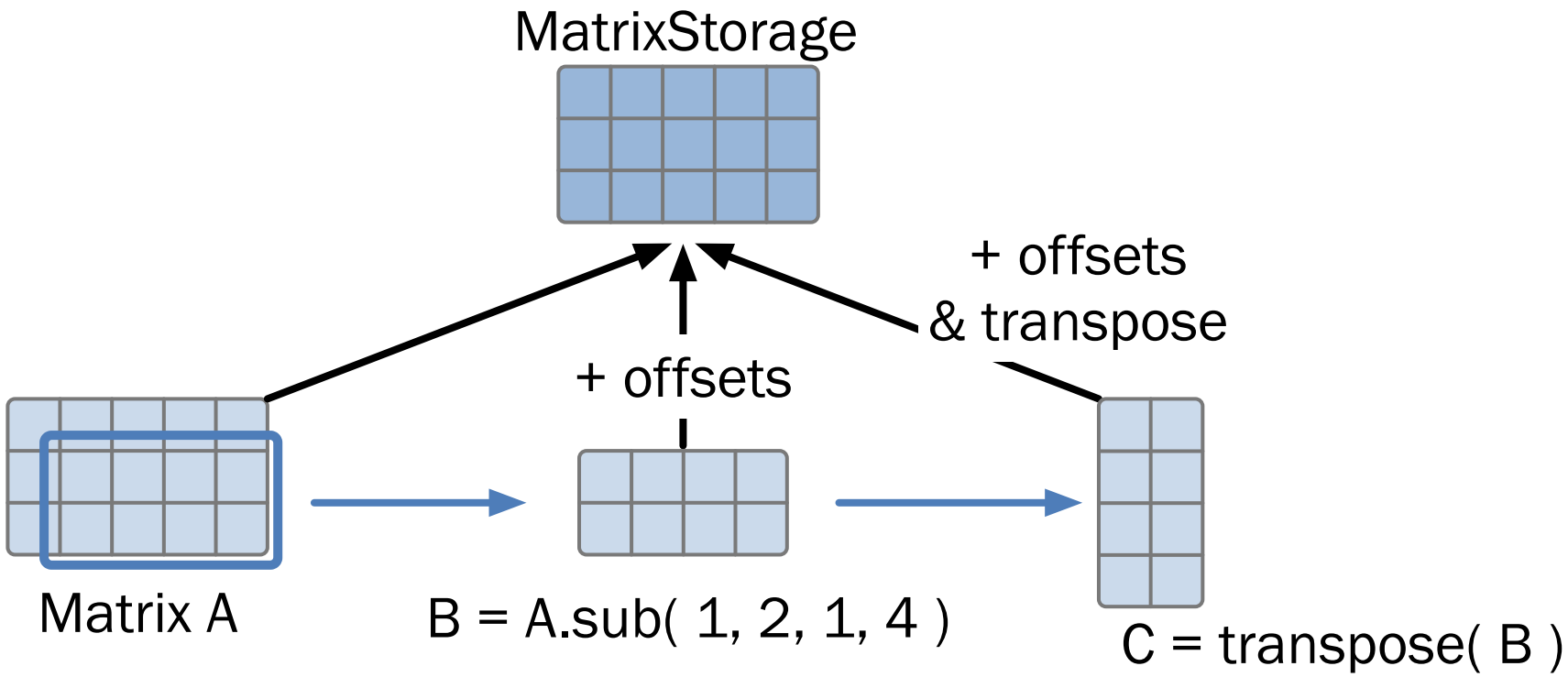
Conversions between matrix types

- Conversion constructors between types
- Matrices share same underlying data (shallow copy)
- May need to slice matrix to convert to triangular or symmetric (see slate01_conversion.cc)

```
// slate01_conversion.cc
slate::Matrix<double>
    A( n, n, nb, p, q, mpi_comm );

slate::TriangularMatrix<double>
    L( slate::Uplo::Lower,
       slate::Diag::Unit, A );

slate::TriangularMatrix<double>
    U( slate::Uplo::Upper,
       slate::Diag::NonUnit, A );

slate::SymmetricMatrix<double>
    S( slate::Uplo::Upper, A );
```

Shallow copy semantics

- **Matrix is a view onto a map of tiles, using a shared pointer**
- **Matrices and tiles use *shallow copies*, not deep copies**
  - A, B, C all view a subset of the same data



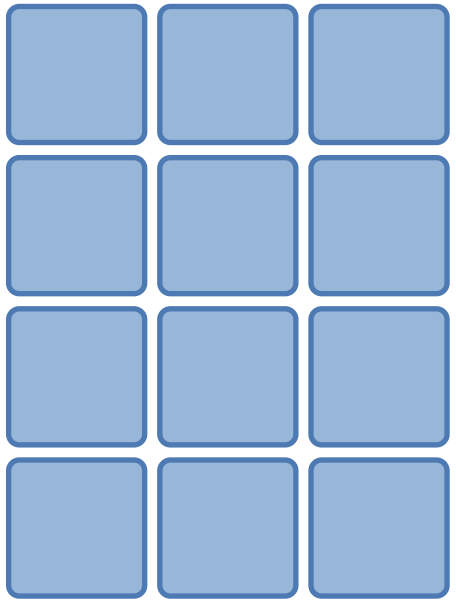
Matrix transpose

- **Lightweight operation sets flag in shallow copy of matrix or tile**
- **Transpose**
  - `AT = transpose( A );`
  - AT is shallow copy of A, with flag AT.op() == Op::Trans
  - Tile AT(i, j) == transpose(A(j, i))
- **Conjugate transpose**
  - `AH = conj_transpose( A );`
  - AH is shallow copy of A, with flag AH.op() == Op::ConjTrans
  - Tile AH(i, j) == conj_transpose(A(j, i))

Sub-matrix

slate02_submatrix.cc

- **Sub-matrices based on tile indices**
  - A.sub(i1, i2, j1, j2) is A(i1 : i2, j1 : j2) inclusive

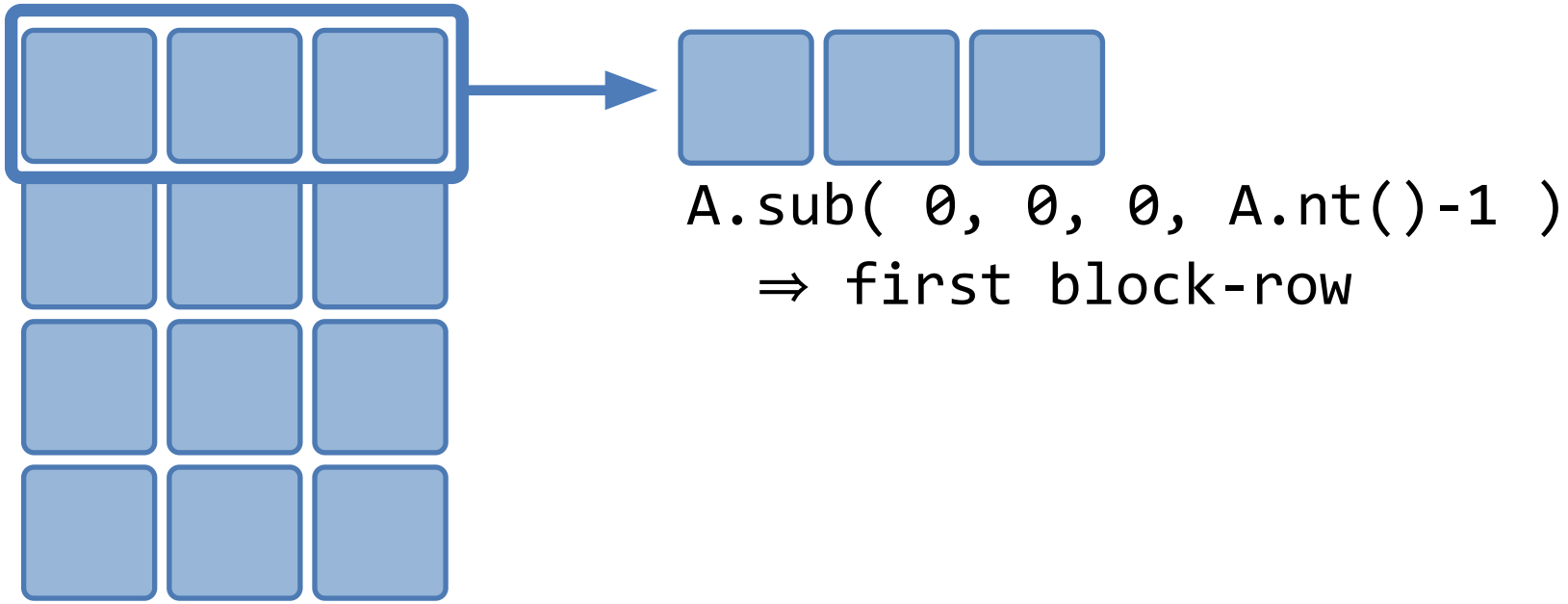


- **Shallow copy semantics!**

Sub-matrix

slate02_submatrix.cc

- **Sub-matrices based on tile indices**
  - A.sub(i1, i2, j1, j2) is A(i1 : i2, j1 : j2) inclusive



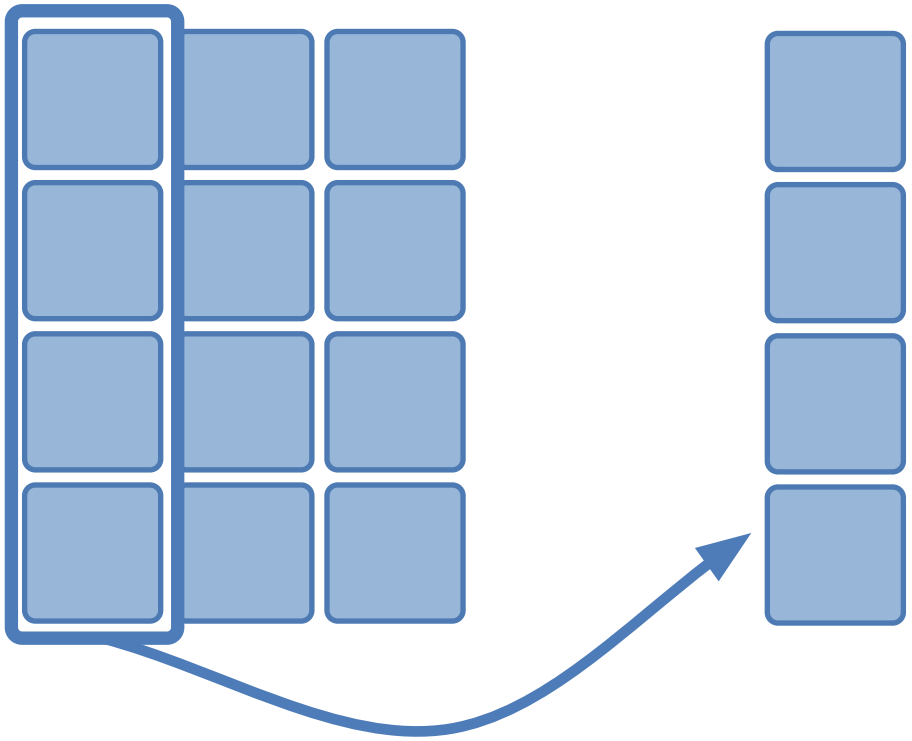
```
A.sub( 0, 0, 0, A.nt()-1 )
   ⇒ first block-row
```

- **Shallow copy semantics!**

Sub-matrix

slate02_submatrix.cc

- **Sub-matrices based on tile indices**
  - A.sub(i1, i2, j1, j2) is A(i1 : i2, j1 : j2) inclusive



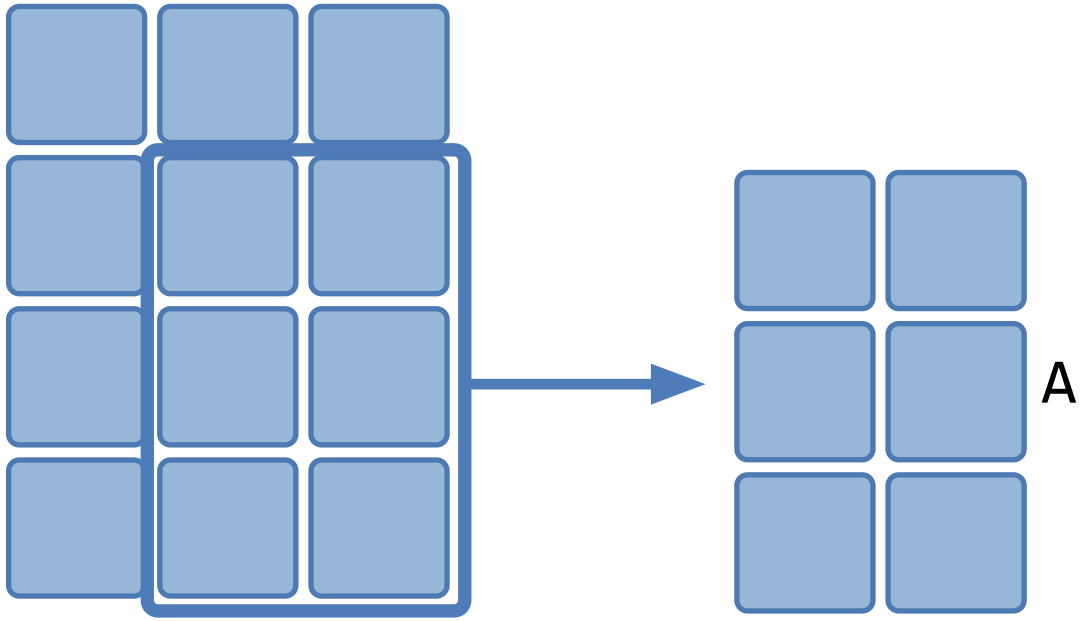
```
A.sub( 0, A.mt()-1, 0, 0 )
  ⇒ first block-column
```

- **Shallow copy semantics!**

Sub-matrix

slate02_submatrix.cc

- **Sub-matrices based on tile indices**
  - A.sub(i1, i2, j1, j2) is A(i1 : i2, j1 : j2) inclusive



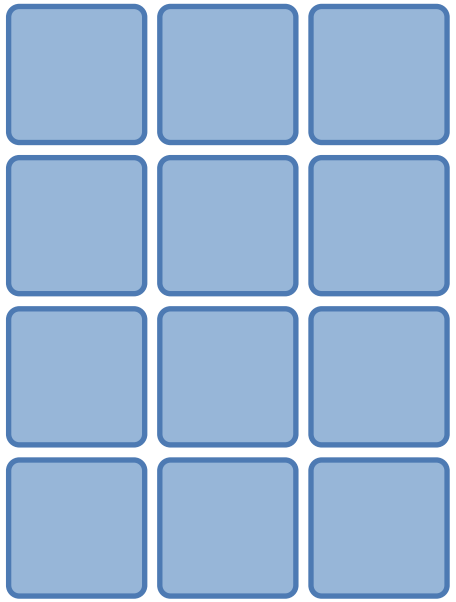
```
A.sub( 1, A.mt()-1, 1, A.nt()-1 )
   ⇒ trailing matrix
```

- **Shallow copy semantics!**

Sliced Matrix

slate02_submatrix.cc

- **Slicing uses row & column indices, instead of tile indices.**
  - A.slice(row1, row2, col1, col2) is A(row1 : row2, col1 : col2), inclusive

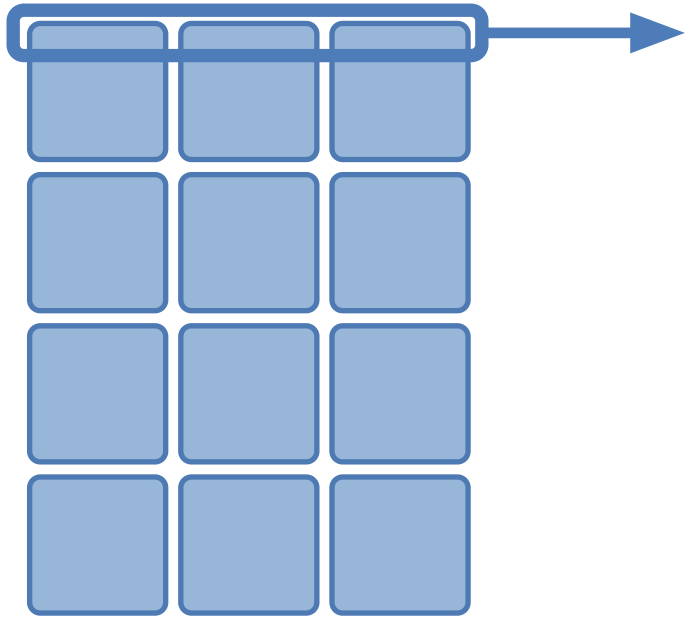


- **Less efficient than A.sub. Less algorithm support, esp. on GPUs.**

Sliced Matrix

slate02_submatrix.cc

- **Slicing uses row & column indices, instead of tile indices.**
  - A.slice(row1, row2, col1, col2) is A(row1 : row2, col1 : col2), inclusive



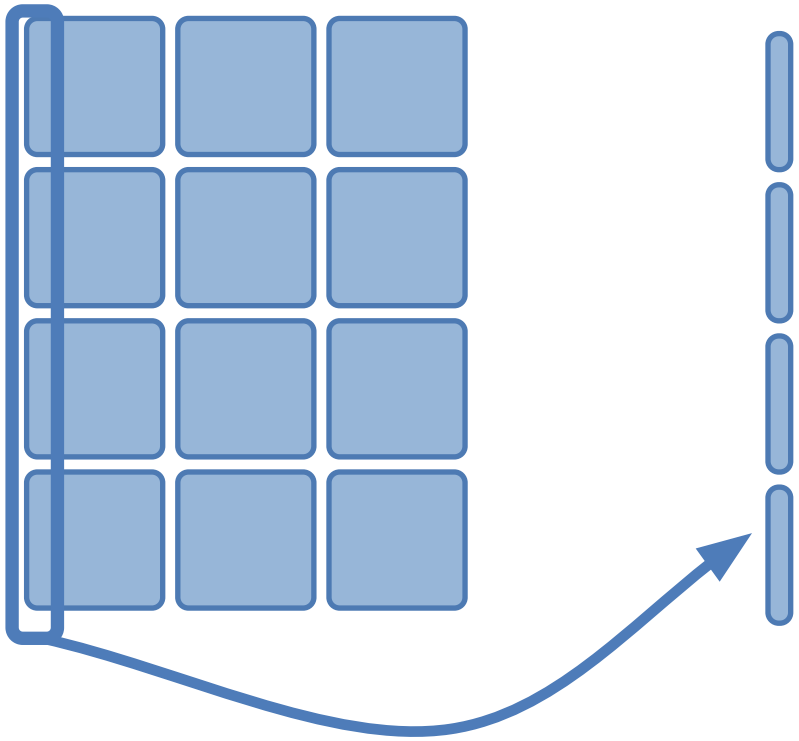
```
A.slice( 0, 0, 0, A.n()-1 )
   ⇒ first row
```

- **Less efficient than A.sub. Less algorithm support, esp. on GPUs.**

Sliced Matrix

slate02_submatrix.cc

- **Slicing uses row & column indices, instead of tile indices.**
  - A.slice(row1, row2, col1, col2) is A(row1 : row2, col1 : col2), inclusive



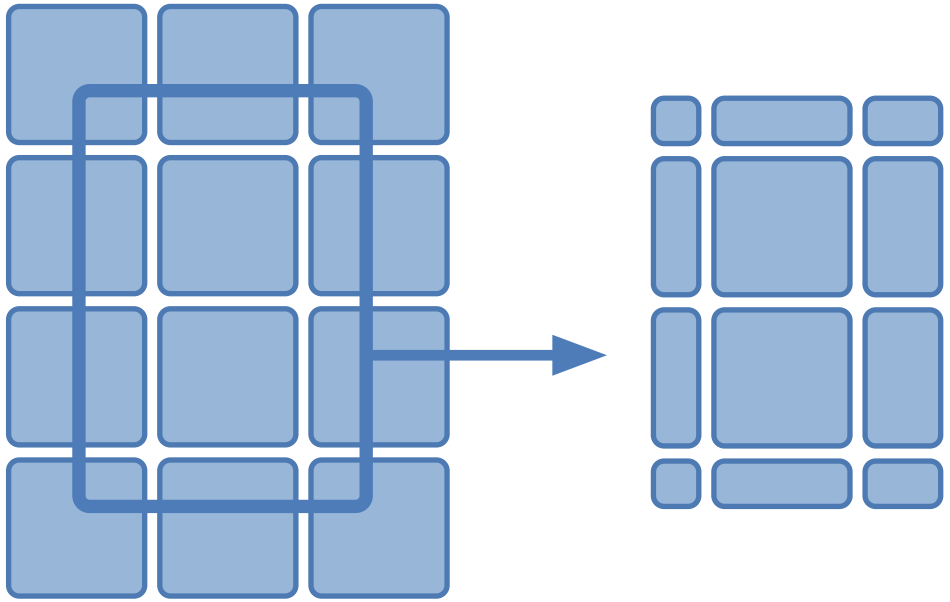
```
A.slice( 0, A.m()-1, 0, 0 )
   ⇒ first column
```

- **Less efficient than A.sub. Less algorithm support, esp. on GPUs.**

Sliced Matrix

slate02_submatrix.cc

- **Slicing uses row & column indices, instead of tile indices.**
  - A.slice(row1, row2, col1, col2) is A(row1 : row2, col1 : col2), inclusive



```
A.slice( i1, i2, j1, j2 )
  ⇒ arbitrary region
```

- **Less efficient than A.sub. Less algorithm support, esp. on GPUs.**

Outline

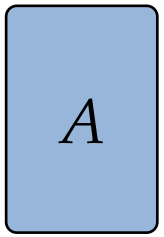
- BLAS++ and LAPACK++
- Creating & accessing matrices
- SLATE routines
  - **Norms and BLAS**
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - Least squares: $AX \approx B$ using QR or LQ
  - SVD and Hermitian eigenvalues
- Future
  - Non-symmetric eigenvalues

Matrix norm

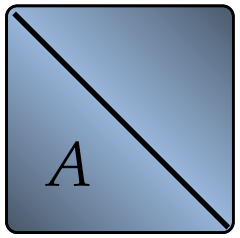
slate03_norm.cc

- `enum class Norm { Max, One, Inf, Fro };`

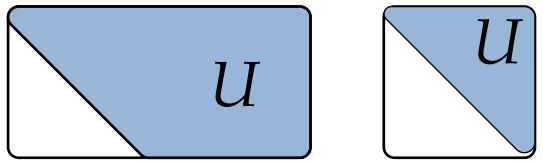
- `slate::Matrix<double> A(…);`
- `double Anorm = norm( Norm::One, A );`



- `slate::SymmetricMatrix<double> S(…);`
- `double Snorm = norm( Norm::One, S );`



- `slate::TrapezoidMatrix<double> T(…);`
- `double Tnorm = norm( Norm::One, T );`

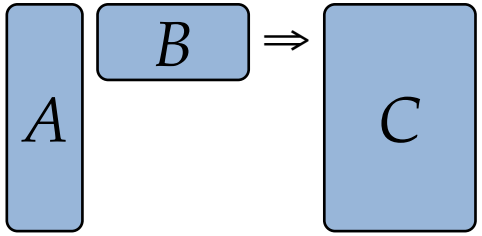


BLAS: matrix-matrix multiply

slate04_blas.cc

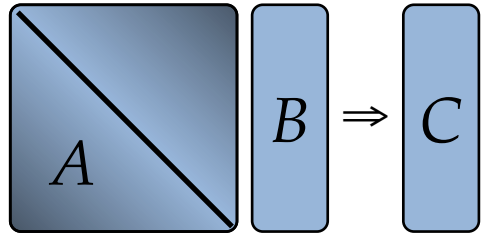
- $C = AB + C$ **where** A, B, C **are general**

```
slate::gemm( alpha, A, B, beta, C );
```



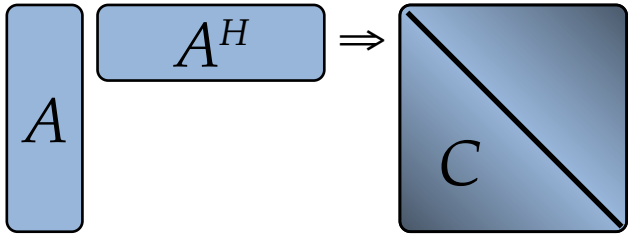
- $C = AB + C$ **where** A **is symmetric**

```
slate::symm( Side::Left,
             alpha, A, B, beta, C );
```



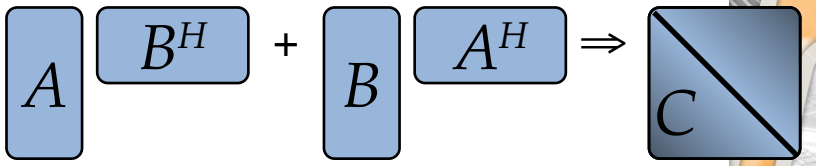
- $C = AA^T + C$ **where** C **is symmetric**

```
slate::syrk( alpha, A, beta, C );
```



- $C = AB^T + BA^T + C$ **where** C **is symmetric**

```
slate::syr2k( alpha, A, B, beta, C );
```



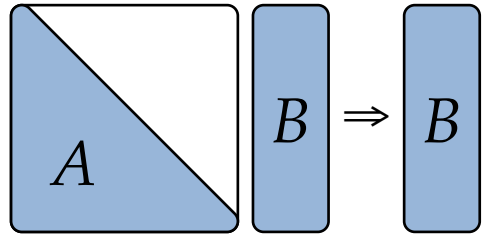
BLAS: matrix-matrix multiply

slate04_blas.cc

- $B = AB$ **where** A **is triangular.**

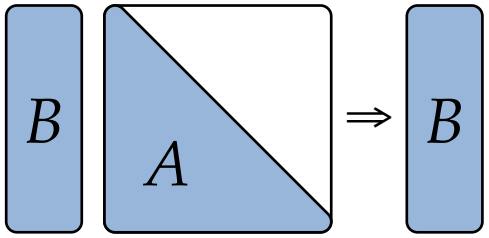
```
slate::trmm( Side::Left, alpha, A, B );
```

  - Note diag is property of matrix A.



- **Solve** $AB = B$ **where** A **is triangular**

```
slate::trsm( Side::Right, alpha, A, B );
```



- **Flags in matrix itself**
  - transpose operation op = NoTrans, Trans, ConjTrans
  - uplo = Lower or Upper
  - diag = Unit or NonUnit

Options

- **SLATE routines take optional map of options as last argument:**

```
slate::gemm( alpha, A, B, beta, C );
slate::gemm( alpha, A, B, beta, C,
  // Run on GPU devices
  { slate::Option::Target, slate::Target::Devices,
    slate::Option::LookAhead, 1
  } );
```

  - Current default Target is HostTask.
    Will probably change to auto-detect if GPUs are available.
  - Generally, default LookAhead = 1 is sufficient
  - Some routines have inner blocking (ib) as tuning parameter

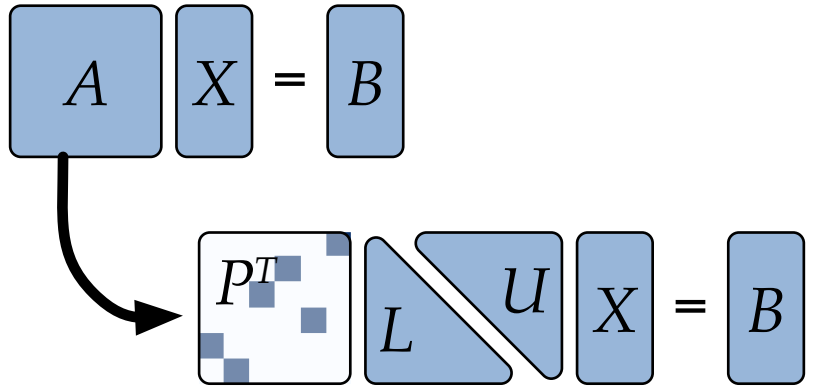
Outline

- BLAS++ and LAPACK++
- Creating & accessing matrices
- SLATE routines
  - Norms and BLAS
  - **Linear systems: $AX = B$ using LU, Cholesky, or LDL^T**
  - Least squares: $AX \approx B$ using QR or LQ
  - SVD and Hermitian eigenvalues
- Future
  - Non-symmetric eigenvalues

Solving linear system

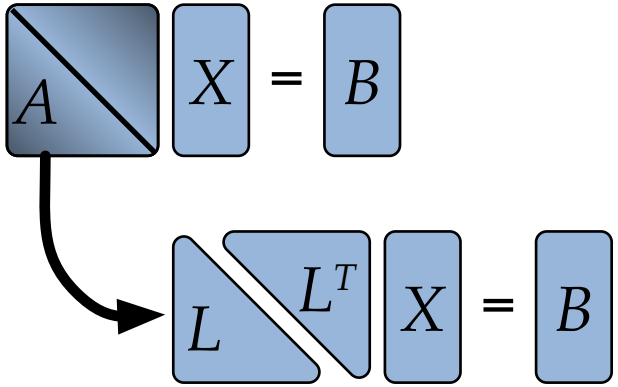
- General square matrix (LU)

```
// slate05a_linear_system_lu.cc
slate::Matrix<double> A( n, n,    … );
slate::Matrix<double> B( n, nrhs, … );
slate::Pivots pivots;
slate::gesv( A, pivots, B );
```



- Symmetric positive definite (SPD, Cholesky)

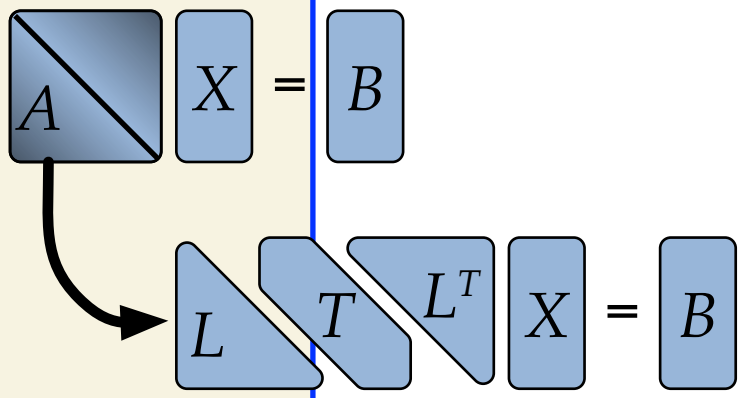
```
// slate05b_linear_system_cholesky.cc
slate::SymmetricMatrix<double>
    A( slate::Uplo::Lower, n, … );
slate::Matrix<double> B( n, nrhs, … );
slate::posv( A, B );
```



Solving linear system

- Hermitian indefinite (Aasen's)

```
// slate05c_linear_system_hesv.cc
slate::HermitianMatrix<double>
    A( slate::Uplo::Lower, n, ... );
slate::Matrix<double> B( n, nrhs, ... );

// workspaces
slate::Matrix<double> H( n, n, ... );
slate::BandMatrix<double> T( n, n, nb, nb, ... );
slate::Pivots pivots, pivots2;

slate::hesv( A, pivots, T, pivots2, H, B );
```

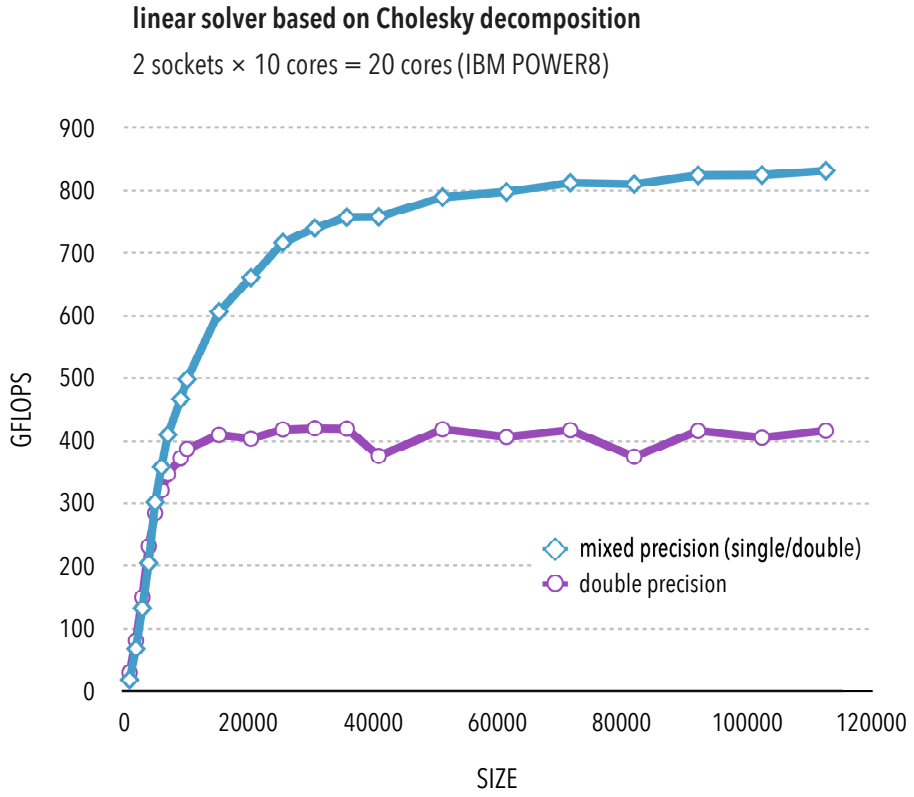


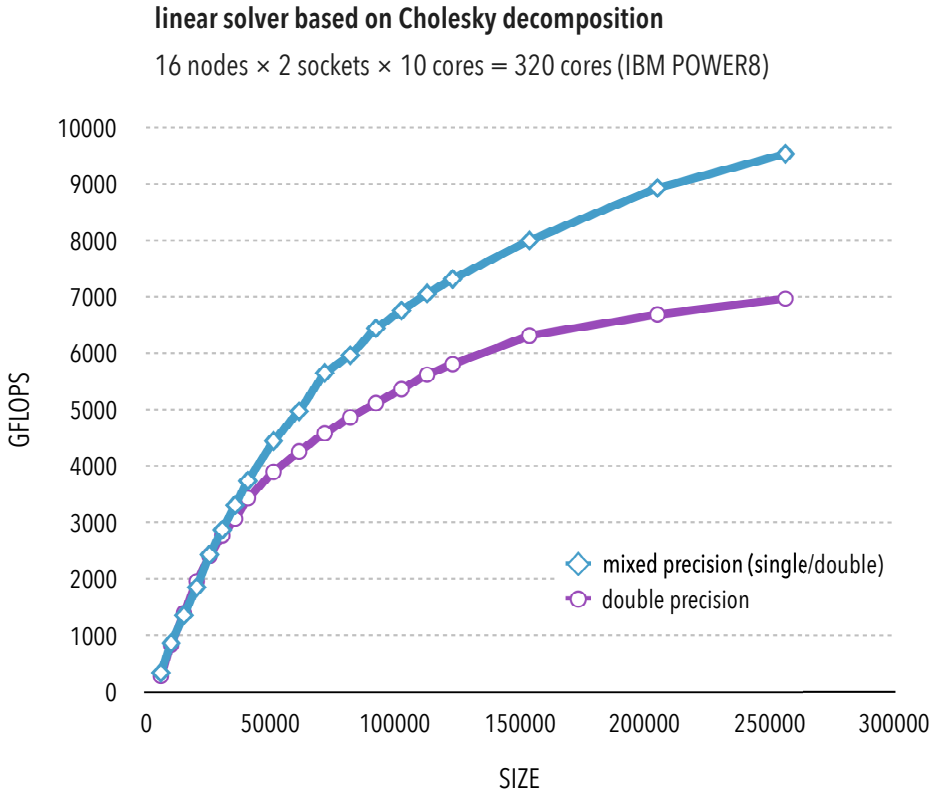
- Currently CPU-only implementation

Mixed precision

slate05a_linear_system_lu.cc
slate05b_linear_system_cholesky.cc

- Single precision is twice as fast as double precision
- Factor matrix in single, doing $O(n^3)$ work
- Iterative refine result with $O(n^2)$ work to double precision accuracy





https://bitbucket.org/icl/slate-tutorial

Matrix inverse

- Factor matrix, then compute inverse from factors

```
// slate05a_linear_system_lu.cc
slate::Matrix<double> A( n, n, … );
slate::Pivots pivots;
slate::getrf( A, pivots );  // factor
slate::getri( A, pivots );  // inverse
```

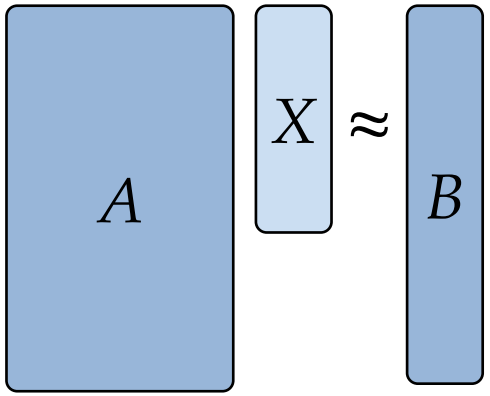
- Caveat: generally, solve $AX = B$ using gesv, posv, etc. rather than computing inverse and multiplying $X = A^{-1} B$. Solves are both faster and more accurate.

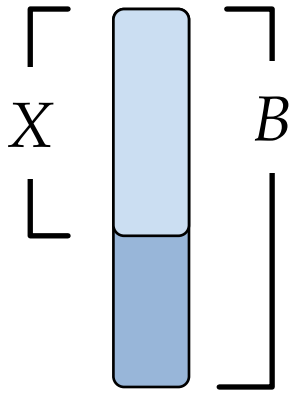
Outline

- BLAS++ and LAPACK++
- Creating & accessing matrices
- SLATE routines
  - Norms and BLAS
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - **Least squares: $AX \approx B$ using QR or LQ**
  - SVD and Hermitian eigenvalues
- Future
  - Non-symmetric eigenvalues

Least squares (over-determined)

- B and X stored in same array

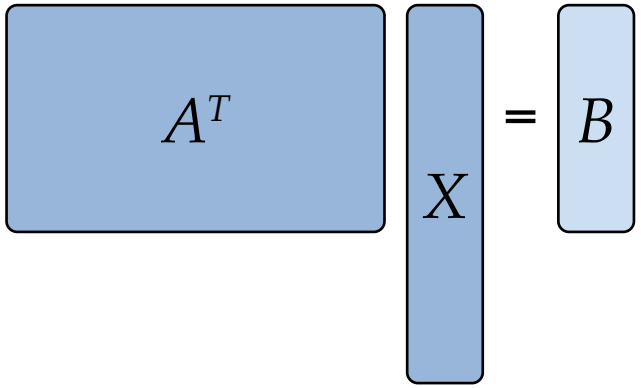


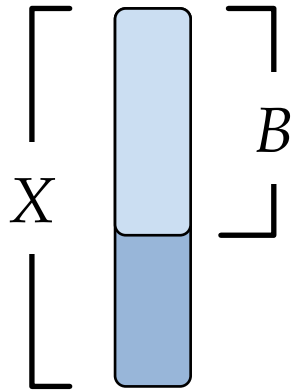



```
// slate06_least_squares.cc
int64_t m=2000, n=1000, nrhs=100,
        nb=100, p=2, q=2;
int64_t max_mn = std::max( m, n );
slate::Matrix<double>
    A( m, n, nb, p, q, mpi_comm );
slate::Matrix<double>
    BX( max_mn, nrhs, nb, p, q, mpi_comm );
// todo: fill in A, B
auto B = BX;
auto X = BX.slice( 0, n-1, 0, nrhs-1 );
slate::TriangularFactors<double> T;

// solve AX = B, solution in X
slate::gels( A, T, BX );
```

Under-determined (min. norm solution)

- Using A^T



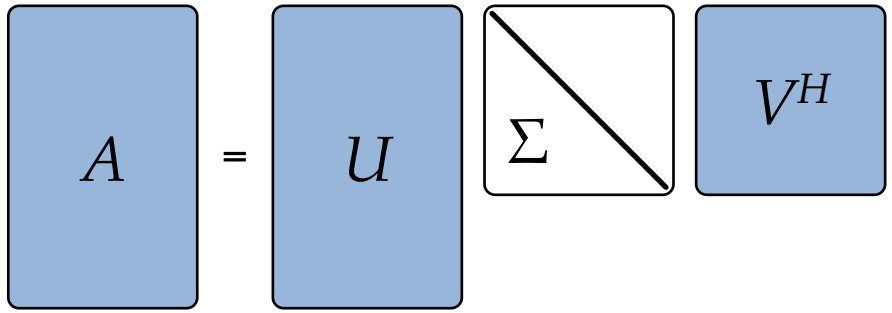



```
// slate06_least_squares.cc
int64_t m=2000, n=1000, nrhs=100,
        nb=100, p=2, q=2;
int64_t max_mn = std::max( m, n );
slate::Matrix<double>
    A( m, n, nb, p, q, mpi_comm );
slate::Matrix<double>
    BX( max_mn, nrhs, nb, p, q, mpi_comm );
// todo: fill in A, B
auto B = BX.slice( 0, n-1, 0, nrhs-1 );
auto X = BX;
slate::TriangularFactors<double> T;

// solve A^T X = B, solution in X
auto AT = transpose(A);
slate::gels( AT, T, BX );
```

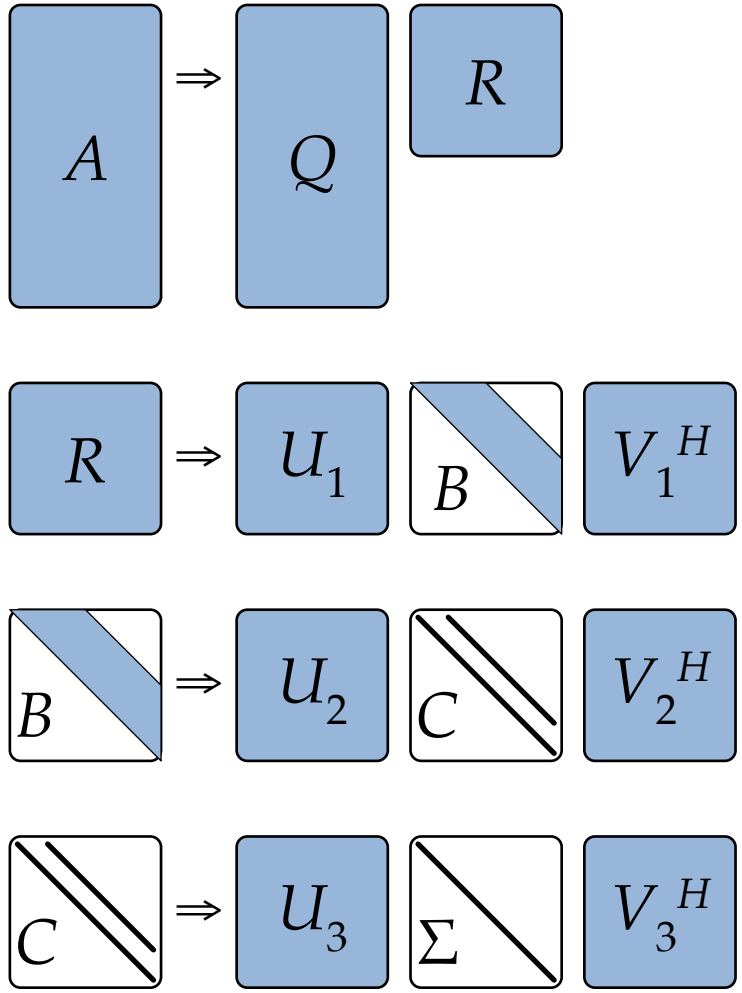
Outline

- BLAS++ and LAPACK++
- Creating & accessing matrices
- SLATE routines
  - Norms and BLAS
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - Least squares: $AX \approx B$ using QR or LQ
  - **SVD and Hermitian eigenvalues**
- Future
  - Non-symmetric eigenvalues

SVD

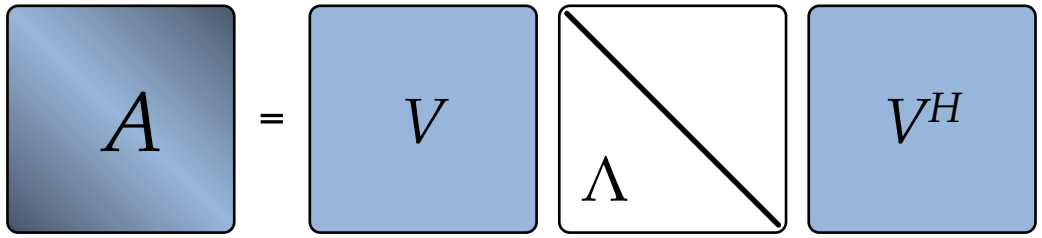


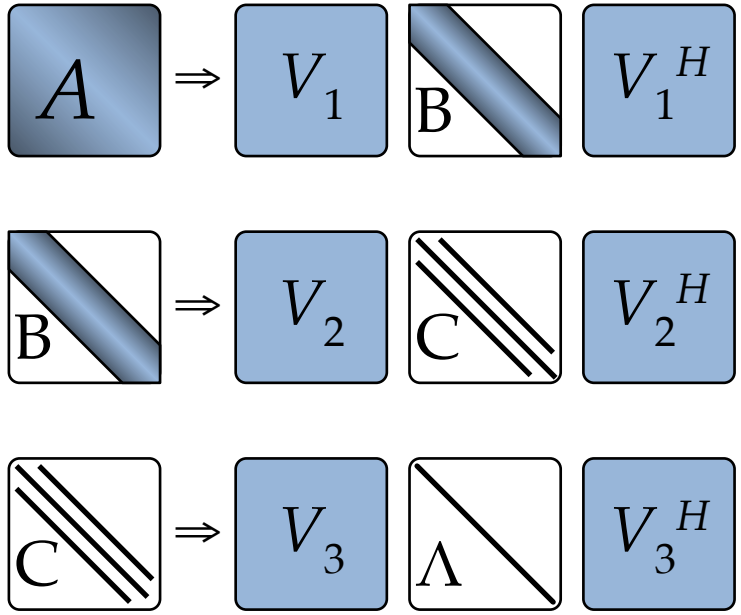
- Uses 2-stage reduction to bidiagonal
- Currently only singular values
- Singular vectors coming soon! (1Q 2020)




```
// slate07_svd.cc
slate::Matrix<double> A( m, n, … );
std::vector<double> Sigma( min(m, n) );
slate::gesvd( A, Sigma );

// singular vectors forthcoming
slate::svd( A, U, Sigma, VH );
```

Hermitian (symmetric) eigenvalues



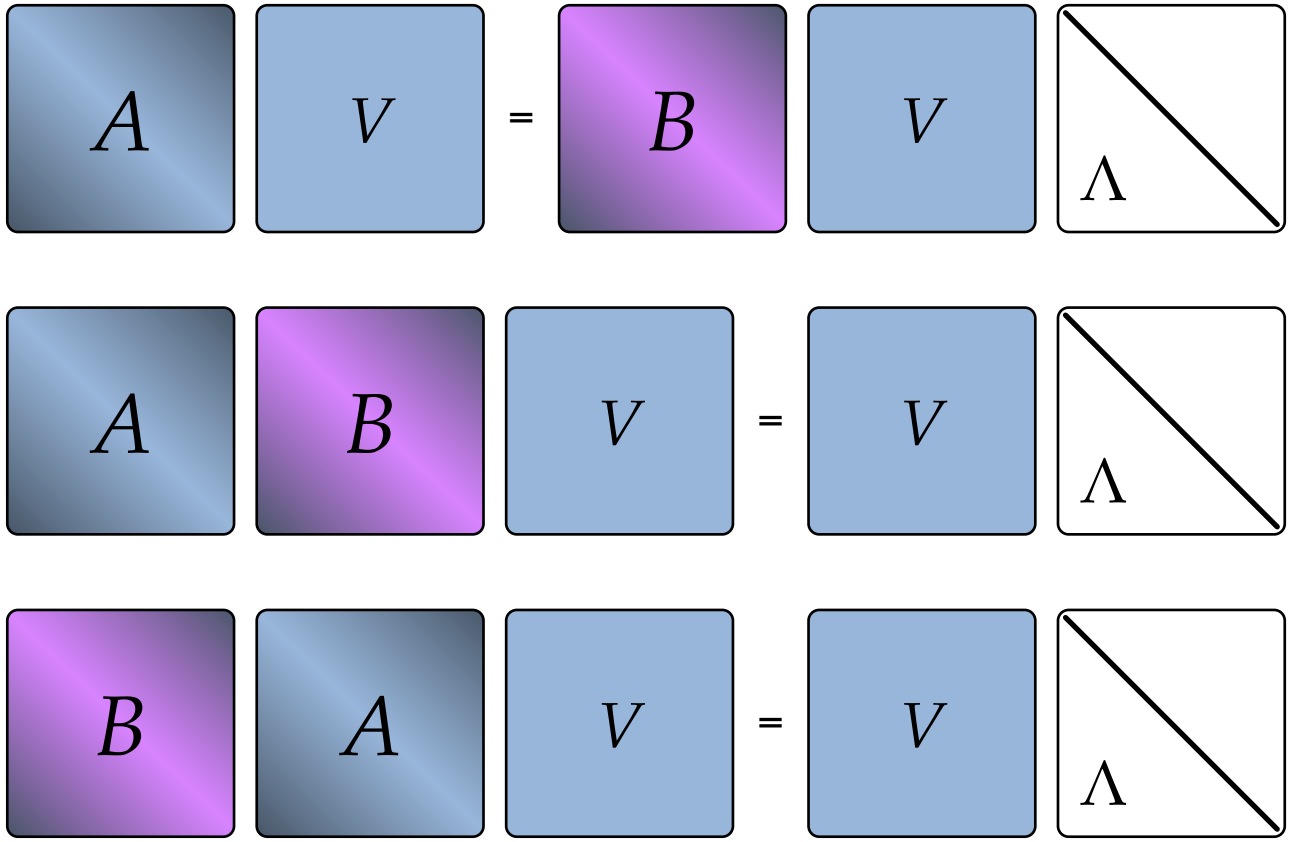


- Uses 2-stage reduction to tridiagonal
- Currently only eigenvalues
- Eigenvectors coming soon! (1Q 2020)

```
// slate08_hermitian_eig.cc
slate::HermitianMatrix<double> A( Uplo::Lower, n, … );
std::vector<double> Lambda( n );
slate::heev( A, Lambda );

// eigenvectors forthcoming
slate::heev( A, Lambda, X );
```

Generalized Hermitian (symmetric) eigenvalues

- Coming soon! (1Q 2020)
- A is Hermitian, B is Hermitian positive definite



Outline

- BLAS++ and LAPACK++
- Creating & accessing matrices
- SLATE routines
  - Norms and BLAS
  - Linear systems: $AX = B$ using LU, Cholesky, or LDL^T
  - Least squares: $AX \approx B$ using QR or LQ
  - SVD and Hermitian eigenvalues
- **Future**
  - Non-symmetric eigenvalues

Future: Simplified naming scheme

- **multiply(alpha, A, B, beta, C)**

  ⇒ gemm (all general), symm/hemm (A or B symmetric/Hermitian)

- **multiply_rankk(alpha, A, beta, C)**

  ⇒ syrk/herk (C is symmetric/Hermitian)

- **multiply(alpha, A, B)**

  ⇒ trmm (A or B is triangular)

- **solve(alpha, A, B)**

  ⇒ trsm (A or B is triangular)

Future: Simplified naming scheme

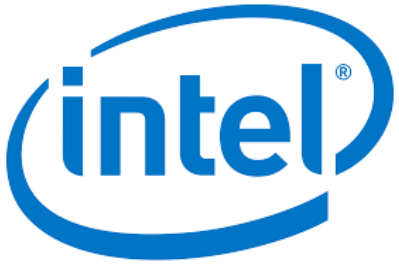
- **Less cryptic names, such as:**
  - cholesky(A), cholesky_solve(A, B)
  - lu(A), lu_solve(A, B)
- **solve(A, B)**
  - Triangular A ⇒ triangular solve (trsm)
  - Symmetric A ⇒ Cholesky (posv); fall back LTL^T or LU?
  - General & square A ⇒ LU (gesv)
  - Rectangular A ⇒ Least squares (gels)
- **Options for preferred algorithm, etc.**

Future: Simplified naming scheme

- **svd(A, U, Sigma, VH)**
  - Options for preferred algorithm (QR iter., D&C, QDWH, Jacobi, etc.)
  - Reduced (economy size) or Full SVD
- **eig(A, Lambda, X)**
  - Hermitian
- **eig(A, B, Lambda, X)**
  - Generalized Hermitian
- **eig(A, Lambda, U, V)**
  - Non-symmetric

Future: Functionality

- SVD singular vectors: 1Q 2020
- Hermitian eigenvectors: 1Q 2020
- Generalized Hermitian eigenvalues: 1Q 2020
- Non-symmetric eigenvalues: 3Q 2020
- More mixed precision solvers
  - Multiprecision Focus Effort in the ECP Math Libraries, Thu @ 10:30
- AMD (HIP) and Intel GPU support
- Spack and xSDK



Future: Iterators

- Iterators that are aware of parallel distribution. Current code:

```
for (int64_t j = 0; j < A.nt(); ++j)
    for (int64_t i = 0; i < A.mt(); ++i)
        if (A.tileIsLocal( i, j )) {
            auto T = A( i, j );
            ...
        }
```

- would become:

```
for (auto iter = A.begin(); iter != A.end(); ++iter) {
    auto i = iter.i();
    auto j = iter.j();
    auto T = *iter;
    ...
}
```

Availability

- **Papers & Guides**
  - http://icl.utk.edu/slate/
- **Bitbucket**
  - https://bitbucket.org/icl/slate/
  - https://bitbucket.org/icl/blaspp/
  - https://bitbucket.org/icl/lapackpp/
  - Mercurial repo (changing to Git)
  - Issue tracking
  - Pull requests for user contributions
- **SLATE user email list**
  - slate-user@icl.utk.edu

Speed Dating for ECP:
1-on-1 Conversations between AD Teams and Math Library Developers
Wed @ 10:30