SLATE Developers’ Guide

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

Innovative Computing Laboratory

August 18, 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.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-2019</td>
<td>first publication</td>
</tr>
<tr>
<td>02-2020</td>
<td>executing multiple internal routines on devices subsection</td>
</tr>
<tr>
<td>04-2020</td>
<td>add work routines</td>
</tr>
<tr>
<td>08-2020</td>
<td>copy editing</td>
</tr>
</tbody>
</table>

@techreport{charara2019slate,
  author={Charara, Ali and Gates, Mark and Kurzak, Jakub and YarKhan, Asim and
           Al Farhan, Mohammed and Sukkari, Dalal and Dongarra, Jack},
  title={SLATE Developers’ Guide, {SWAN} No. 11},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2019},
  month={December},
  number={ICL-UT-19-02},
  note={revision 08-2020},
  url={https://www.icl.utk.edu/publications/swan-011},
}
Contents

1 Introduction 1

2 API Layers 2
   2.1 Drivers ................................................................. 3
   2.2 Computational Routines .............................................. 3
   2.3 Internal routines for major, parallel tasks ...................... 8
   2.4 Tile operations for small, sequential tasks ................... 19
   2.5 BLAS++, Batched BLAS++, and LAPACK++ ....................... 19
   2.6 Work routines for actual OpenMP work ......................... 20

3 Matrix Storage 26
   3.0.1 Tile management .................................................. 28

4 Matrix Hierarchy 30

5 Handling of Side, Uplo, Trans, etc. 32

6 Handling of Precisions 34

7 Parallelism Model 36

8 Message Passing Communication 40

9 MOSI Coherency Protocol 41
   9.1 Coherency control .................................................. 41
      9.1.1 Tile States .................................................... 42
      9.1.2 MOSI API ...................................................... 43
      9.1.3 Data transfer .................................................. 44
      9.1.4 State diagrams .............................................. 44
9.2 Developer hints .............................................................. 48

10 Column Major and Row Major Layout 49
  10.1 Layout representation and API ..................................... 49
  10.2 Layout conversion .................................................... 50
     10.2.1 Layout conversion of extended tiles ...................... 53
  10.3 Layout-aware MOSI .................................................. 53

11 Compatibility APIs ...................................................... 56
  11.1 LAPACK Compatibility API ...................................... 56
  11.2 ScaLAPACK Compatibility API .................................. 57

Bibliography ................................................................. 58
CHAPTER 1

Introduction

SLATE (Software for Linear Algebra Targeting Exascale)\(^1\) 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). The objective of SLATE is to provide fundamental dense linear algebra capabilities to the U.S. Department of Energy and to the high-performance computing (HPC) community at large.

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

This Developers’ Guide is intended to describe the internal workings of SLATE, to be of use for SLATE developers and contributors. A companion SLATE Users’ Guide \([1]\) is available for application-end users, which focuses on the public SLATE API. These guides will be periodically revised as SLATE develops, with the revision noted in the front matter notes and BibTeX.

---

\(^1\)http://icl.utk.edu/slate/
\(^2\)https://www.exascaleproject.org
SLATE’s API is composed of several layers, as depicted in Figure 2.1. The drivers and computational routines are the primary public API; the internal task and tile routines implement major (parallel) and minor (sequential) tasks, respectively. The LAPACK++ and BLAS++ packages, including Batched BLAS++, are independent packages developed for SLATE that interface to the vendor-optimized LAPACK and BLAS routines.

Figure 2.1: Software layers in SLATE.
SLATE’s routine names are derived from traditional BLAS and LAPACK names, minus the traditional initial letter denoting the precision (s, d, c, z). We have also developed simplified names using overloaded functions, using the Matrix types to identify the operation to be performed. For instance, `multiply(A, B, C)` can map to a general, symmetric, or Hermitian matrix-matrix multiply (`gemm`, `symm`, or `hemm`) depending on whether the type of A is a general Matrix, SymmetricMatrix, or HermitianMatrix, respectively.

2.1 Drivers

As in LAPACK and ScaLAPACK, driver routines solve an entire problem, such as a linear system $Ax = b$ (routines `gesv`, `posv`), a least squares problem $Ax \approx b$ (`gevs`), or a singular-value decomposition $A = U \Sigma V^H$ (`gesvd`). Drivers in turn call computational routines to solve subproblems. Drivers are typically independent of the target (CPU or device), delegating those details to lower level routines. Algorithm 2.1 gives an example of the Cholesky driver, `posv`, which relies on computational routines `potrf` and `potrs` to factor the matrix $A$ and solve the system $Ax = b$.

Note that since it is independent of the target, we do not need to template it based on the target, as the computational routines will be. Nor do we need to unpack the `opts` argument; simply pass it along to the computational routines.

Algorithm 2.1 Cholesky solve driver, `slate::posv`

```cpp
1 // Distributed parallel Cholesky solve, AX = B
2 // A: Matrix to factor; overwritten by L
3 // B: On input, matrix B; overwritten by X
4 // opts: User options such as Target and Lookahead
5 // scalar_t: Datatype: float, double, std::complex, etc.
6 template <typename scalar_t>
7 void posv( HermitianMatrix<scalar_t>& A,
8            Matrix<scalar_t>& B,
9            Options const& opts )
10 {
11   potrf( A, opts ); // factor A = LL^H
12   potrs( A, B, opts ); // solve AX = B using factorization
13 }
```

2.2 Computational Routines

Again, as in LAPACK and ScaLAPACK, computational routines solve a sub-problem, such as computing an LU factorization (`getrf`), or solving a linear system given an LU factorization (`getrs`). In SLATE, these are templated on target (CPU or device), with the code typically independent of the device. However, if needed, code can be optimized for a specific target by providing an overloaded version. Communication between processes and dependencies between tasks are managed at this level. SLATE’s Parallel Basic Linear Algebra Subprograms (PBLAS) exists at this level.

Algorithm 2.2 gives an example of the Cholesky factorization computational routine (`potrf`),
used by the Cholesky driver. SLATE’s potrf routine is approximately the same length as the
LAPACK dpotrf code, and roughly half the length of the ScaLAPACK and MAGMA code (all
excluding comments). Yet SLATE’s code handles all precisions, multiple targets, distributed-
memory and shared-memory parallelism, a lookahead to overlap communication and com-
putation, and GPU acceleration. Of course, there is significant code in lower levels, but this
demonstrates that writing driver and computational routines can be simplified by delegating
code complexity to lower-level abstractions.

Comments on the code

Normally, matrices are passed by reference (Matrix<scalar_t>& A), as this avoids invoking
(shallow) copy constructors. For Cholesky, however, the matrix may get transposed, so it must
be passed by value; see Chapter 5.

Dependencies are tracked via a dummy vector—not based on the actual data—unlike in pure
dataflow implementations like PLASMA. For Cholesky, entries in the dummy vector represent
each column. The dummy vector is allocated using std::vector for exception safety, but
OpenMP needs a raw pointer to its data.

The variable A_nt is defined instead of using A.nt() directly because some compilers complain
about using A.nt() in OpenMP pragmas.

Template dispatch

The routine in Algorithm 2.2 is the internal implementation, templated on the target. It takes
a dummy TargetType argument, which is the C++ idiom for specialization of a function. In
this case it is templated on the target, but an overload can be given for a specific target type, as
shown in Algorithm 2.4.

The user can specify the target as HostTask, HostNest, HostBatch, or Devices via the opts
parameter. The public routine that the user actually calls unpacks the opts dictionary and
dispatches to the target-specific version, as shown in Algorithm 2.5. Note in this routine the
matrix A is passed by reference, unlike the internal implementation where it is passed by
value (Algorithm 2.2). (As of 12/2019, this is split into an additional wrapper, but will likely be
simplified as shown here in the near future.)
Algorithm 2.2 Cholesky factorization computational routine, `slate::potrf`, with variable lookahead. Continued in Algorithm 2.3.

```cpp
namespace internal {
namespace specialization {

// Distributed parallel Cholesky factorization, A = L L^H
// target: Computation method: HostTask, Devices, etc.
// A: Matrix to factor; overwritten by L
// lookahead: Lookahead depth

template <Target target, typename scalar_t>
void potrf( slate::internal::TargetType<target>,
            HermitianMatrix<scalar_t> A, int64_t lookahead ) {
    using real_t = blas::real_type<scalar_t>;
    scalar_t one = 1.0;
    real_t r_one = 1.0;
    const int64_t A_nt = A.nt();
    const int priority_one = 1;

    // if upper, change to lower (see Chapter 5)
    if (A.uplo() == Uplo::Upper)
        A = conj_transpose(A);

    // dummy vector to track dependencies
    std::vector<uint8_t> column_vector(A_nt);
    uint8_t * column = column_vector.data();

    #pragma omp parallel
    #pragma omp master
    {
        omp_set_nested(1);
        for (int64_t k = 0; k < A_nt; ++k) {
            // panel, high priority
            #pragma omp task depend(inout:column[k]) priority(priority_one)
            {
                // factor A(k, k)
                internal::potrf<Target::HostTask>(A.sub(k, k), priority_one);

                // send A(k, k) down col A(k+1:nt-1, k)
                if (k+1 < A_nt-1)
                    A.tileBcast(k, k, A.sub(k+1, A_nt-1, k, k));

                // A(k+1:nt-1, k) * A(k, k)^{-H}
                if (k+1 < A_nt-1) {
                    auto Akk = A.sub(k, k);
                    auto Tkk = TriangularMatrix<scalar_t>({Diag::NonUnit, Akk});
                    internal::trsm<Target::HostTask>(
                        Side::Right,
                        one, conj_transpose(Tkk),
                        A.sub(k+1, A_nt-1, k, k), priority_one);
                }
            }
        }
        typename Matrix<scalar_t>::BcastList bcast_list_A;
        for (int64_t i = k+1; i < A_nt; ++i) {
            // send A(i, k) across row A(i, k+1:i)
            #pragma omp task depend(inout:column[i])
            {
                // and down col A(i:nt-1, i)
                bcast_list_A.push_back(
                    {i, k, {A.sub(i, i, k+1, i),
                      A.sub(i, A_nt-1, i, i)}});
            }
        }
    }
}
```

Algorithm 2.3 Cholesky factorization. Continued from Algorithm 2.2.

```cpp
// update lookahead column(s), high priority
for (int64_t j = k+1; j < k+1+lookahead && j < A_nt; ++j) {
    #pragma omp task depend(in:column[k]) \ 
    depend(inout:column[j]) priority(priority_one)
    {
        // A(j, j) -= A(j, k) * A(j, k)'H
        internal::herk<Target::HostTask>(
            -r_one, A.sub(j, j, k, k),
            r_one, A.sub(j, j), priority_one);
        
        // A(j+1:nt-1, j) -= A(j+1:nt-1, k) * A(j, k)'H
        if (j+1 <= A_nt-1) {
            auto Ajk = A.sub(j, k, k);
            internal::gemm<Target::HostTask>(
                -one, A.sub(j+1, A_nt-1, k, k),
                conj_transpose(Ajk),
                one, A.sub(j+1, A_nt-1, j, j), priority_one);
        }
    }
    
    // update trailing submatrix, normal priority
    if (k+1+lookahead < A_nt) {
        #pragma omp task depend(in:column[k]) \ 
        depend(inout:column[k+1+lookahead]) \ 
        depend(inout:column[A_nt-1])
        {
            // A(kl+1:nt-1, kl+1:nt-1) -=
            // A(kl+1:nt-1, k) * A(kl+1:nt-1, k)'H
            // where kl = k + lookahead
            internal::herk<Target>(
                -r_one, A.sub(k+1+lookahead, A_nt-1, k, k),
                r_one, A.sub(k+1+lookahead, A_nt-1));
        }
    }
}
#pragma omp taskwait
A.tileUpdateAllOrigin();
} // omp parallel master
A.releaseWorkspace();
} // namespace specialization
} // namespace internal
```

Algorithm 2.4 Overload specialization for Target::Devices.

```cpp
template<typename scalar_t>
void potrf( slate::internal::TargetType<Target::Devices>,
    HermitianMatrix<scalar_t> A, int64_t lookahead )
{
    // ... code specific to GPU Devices implementation ...
}
```
Algorithm 2.5 Dispatch to target implementations.

```c++
template <typename scalar_t>
void potrf( HermitianMatrix<scalar_t>& A,
            Options const& opts )
{
    // todo: replace with opt()
    Target target;
    try {
        target = Target(opts.at(Option::Target).i_);
    }
    catch (std::out_of_range&) {
        target = Target::HostTask;
    }
    int64_t lookahead;
    try {
        lookahead = opts.at(Option::Lookahead).i_;
        assert(lookahead >= 0);
    }
    catch (std::out_of_range&) {
        lookahead = 1;
    }
    switch (target) {
        case Target::Host:
        case Target::HostTask:
            internal::specialization::potrf(
                internal::TargetType<Target::HostTask>() ,
                A, lookahead);
            break;
        case Target::HostNest:
            internal::specialization::potrf(
                internal::TargetType<Target::HostNest>() ,
                A, lookahead);
            break;
        case Target::HostBatch:
            internal::specialization::potrf(
                internal::TargetType<Target::HostBatch>() ,
                A, lookahead);
            break;
        case Target::Devices:
            internal::specialization::potrf(
                internal::TargetType<Target::Devices>() ,
                A, lookahead);
            break;
        default:
    }
}
```
2.3 Internal routines for major, parallel tasks

SLATE adds a third layer of internal routines that generally perform one step or major task of a computational routine. These are typically executed in parallel across multiple CPU cores, or as a batch routine on the GPU. (See Chapter 7 for how algorithms are implemented as tasks.) For instance, in the outer $k$ loop, slatet::gemm calls a sequence of slatet::internal::gemm, each of which performs one block outer product. Most internal routines consist of a set of independent tile operations that can be issued as a batched gemm or an OpenMP parallel-for loop, with no task dependencies to track. Internal routines provide device-specific implementations such as OpenMP nested tasks, parallel for loops, or batched BLAS operations. In many linear algebra algorithms, these internal routines implement the trailing matrix update.

Algorithm 2.6 gives an example of the internal gemm routine, CPU HostTask implementation, used in the PBLAS gemm routine and for the update in the Cholesky factorization routine. This code reveals several features of SLATE. Currently, routines loop over all tiles in the matrix $C$, and select just the local tiles to operate on. By filtering for local tiles via the tileIsLocal call, SLATE is agnostic to the actual distribution. To reduce overheads, we are developing 2D iterators that are aware of the distribution, enabling iteration over just the local tiles without needing to check if tiles are local, while the code can still be agnostic to the distribution.

In the potrf call, the internal::gemm call is an OpenMP task. Within internal::gemm, each tile gemm call is a nested OpenMP task, with no dependencies. Before each tile gemm, tileGetForReading and tileGetForWriting ensures that the tiles are in CPU memory, initiating a transfer from accelerator memory if necessary. Remote tiles are given a life counter to track the number of tiles they update. After each tile gemm, the $A$ and $B$ tiles have their lives decremented by tileTick; once all local tiles in row $i$ of $C$ are updated, the life of tile $A(i, 0)$ reaches zero and the tile is deleted if it is a workspace tile (i.e., not an origin tile). Similarly, when all local tiles in column $j$ of $C$ are updated, the life of tile $B(0, j)$ reaches zero and the tile is deleted, if it is workspace.

Panel operations, such as the LU and QR parallel panels, also exist as internal routines. However, unlike trailing matrix updates, panels create a set of interdependent tasks.
2.3. INTERNAL ROUTINES FOR MAJOR, PARALLEL TASKS

Algorithm 2.6 Host task implementation of internal matrix multiply routine, `slate::internal::gemm`, corresponding to a single block outer product.

```c++
// C = alpha AB + beta C; A is one block col, B is one block row
template <typename scalar_t>
void gemm(internal::TargetType<Target::Devices>,
    scalar_t alpha, Matrix<scalar_t>& A,
    Matrix<scalar_t>& B,
    scalar_t beta, Matrix<scalar_t>& C,
    Layout layout, int priority)
{
    // todo: update to recent code that uses MOSI set interfaces.
    LayoutConvert convert(layout);
    for (int64_t i = 0; i < C.mt(); ++i) {
        for (int64_t j = 0; j < C.nt(); ++j) {
            if (C.tileIsLocal(i, j)) {
                #pragma omp task shared(A, B, C, err)
                priority(priority)
                {
                    A.tileGetForReading(i, 0, convert);
                    B.tileGetForReading(0, j, convert);
                    C.tileGetForWriting(i, j, convert);
                    gemm(alpha, A(i, 0),
                          B(0, j),
                          beta, C(i, j));
                    A.tileTick(i, 0);
                    B.tileTick(0, j);
                }
            }
        }
    }
    #pragma omp taskwait
}
```
2.3. INTERNAL ROUTINES FOR MAJOR, PARALLEL TASKS

Batched GPU tasks

Compared to the CPU implementation in Algorithm 2.6, the batched GPU implementation is significantly more complicated. Each device is handled by a separate task in parallel. First, it loops over all the relevant tiles to copy them to the GPU device if they aren’t already resident (Algorithm 2.7). Second, it loops over the tiles again to construct the batch arrays, and copies the batch arrays to the GPU (Algorithm 2.8). Third, it executes the batched gemm call, and finally cleans up any workspace tiles in matrices A and B (Algorithm 2.9).

This uses the newer Modified, OnHold, Shared, Invalid (MOSI) set API, which builds a set of tiles to transfer, then transfers them with a single call. Copying of the sets is launched as nested tasks for increased parallelism. See Chapter 9 for more details.

All the batch arrays for the A, B, and C matrices are stored contiguously, one after another, to make a single cudaMemcpy transfer to the GPU.

There are several limitations with this current approach. It assumes 4 areas of the matrix, with uniform tile sizes within each area. Batch 00 is the main batch excluding border tiles, batch 01 is for border tiles in the right column, batch 10 is for border tiles in the bottom row, and batch 11 is for the border tile in the bottom-right corner. Not only the tile size but also the strides (lda, ldb, ldc) must be uniform within each batch; effectively, this means all the tiles must be contiguous, not strided. Supporting sliced matrices with uniform interior tiles would require 9 areas, to accommodate border tiles along the left column and top row. To generalize this code, ideally it would leverage Batched BLAS++ to use a group or variable-sized batched BLAS interface to allow arbitrary tile sizes.

Recent work [2] has investigated splitting this gemm into two pieces: a prep step to copy data to the GPU and prepare the batch arrays, and an exec step to execute the batched gemm. There is also recent work [2] to allow multiple simultaneous gemm operations without conflicts on the batch arrays.
Algorithm 2.7 Batched GPU device implementation of internal matrix multiply routine, `slate::internal::gemm`, corresponding to a single block outer product. Handling transposed $C$ and row-major support is omitted here; see SLATE code for details. Continued in Algorithm 2.8.

```cpp
// C = alpha AB + beta C; A is one block col, B is one block row
template <typename scalar_t>
void gemm(internal::TargetType<Target::HostTask>,
  scalar_t alpha, Matrix<scalar_t>& A,
  scalar_t beta, Matrix<scalar_t>& C,
  Layout layout , int priority)
{
  LayoutConvert convert( layout );
  int err = 0;
  for (int device = 0; device < C.num_devices(); ++device) {
    #pragma omp task shared (A, B, C, err ) priority(priority)
    {
      Op opA = A.op();
      Op opB = B.op();
      // Get tiles involved with updating C's local tiles.
      std::set<ij_tuple> A_tiles_set , B_tiles_set , C_tiles_set ;
      for (int64_t i = 0; i < C.mt(); ++i) {
        for (int64_t j = 0; j < C.nt(); ++j) {
          if (C.tileIsLocal(i, j) && device == C.tileDevice(i, j)) {
            A_tiles_set.insert({i, 0});
            B_tiles_set.insert({0, j});
            C_tiles_set.insert({i, j});
          }
        }
      }
      // Copy tiles to GPU device as needed.
      #pragma omp task default(shared)
      {
        A.tileGetForReading( A_tiles_set , device , convert );
      }
      #pragma omp task default(shared)
      {
        B.tileGetForReading( B_tiles_set , device , convert );
      }
      #pragma omp task default(shared)
      {
        C.tileGetForWriting( C_tiles_set , device , convert );
      }
    }
  }
}
```
Algorithm 2.8 Batched GPU device implementation, continued from Algorithm 2.7, continued in Algorithm 2.9.

```c
// Build batches for 4 regions.
// Assumes uniform tile sizes in each region!
int64_t batch_size = C_tiles_set.size();
scalar_t** a_array_host = C.array_host(device);
scalar_t** b_array_host = a_array_host + batch_size;
scalar_t** c_array_host = b_array_host + batch_size;

scalar_t** a_array_dev = C.array_device(device);
scalar_t** b_array_dev = a_array_dev + batch_size;
scalar_t** c_array_dev = b_array_dev + batch_size;

int64_t batch_count = 0;
int64_t batch_count_00 = 0;
int64_t lda00 = 0;
int64_t ldb00 = 0;
int64_t ldc00 = 0;
int64_t mb00 = C.tileMb(0);
int64_t nb00 = C.tileNb(0);
int64_t kb = A.tileNb(0); // == A.tileMb(0)

for (int64_t i = 0; i < C.mt()-1; ++i) {
    for (int64_t j = 0; j < C.nt()-1; ++j) {
        if (C.tileIsLocal(i, j)) {
            if (device == C.tileDevice(i, j)) {
                a_array_host[batch_count] = A(i, 0, device).data();
                b_array_host[batch_count] = B(0, j, device).data();
                c_array_host[batch_count] = C(i, j, device).data();
                lda00 = A(i, 0, device).stride();
                ldb00 = B(0, j, device).stride();
                ldc00 = C(i, j, device).stride();
                ++batch_count_00;
                ++batch_count;
            }
        }
    }
}
// ... build other 3 batches.

// cublas_handle uses this stream
cudaStream_t stream = C.compute_stream(device);
cublasHandle_t cublas_handle = C.cublas_handle(device);

// Copy batch arrays to device,
// which contains batch arrays for A, B, and C.
slate_cuda_call(
    cudaMemcpyAsync(C.array_device(device), C.array_host(device),
    sizeof(scalar_t*)*batch_count*3,
    cudaMemcpyHostToDevice,
    cubaMemcpyHostToDevice,
    stream));
```
if (batch_count_00 > 0) {
    slate_cublas_call(
        cublasGemmBatched(
            cublas_handle, // uses stream
            cublas_op_const(opA), cublas_op_const(opB),
            mb00, nb00, kb,
            &alpha, (const scalar_t**) a_array_dev, lda00,
            (const scalar_t**) b_array_dev, ldb00,
            &beta, c_array_dev, ldc00,
            batch_count_00));
    a_array_dev += batch_count_00;
    b_array_dev += batch_count_00;
    c_array_dev += batch_count_00;
} // ... launch other 3 batches.
slate_cuda_call(
    cudaStreamSynchronize(stream));

// Cleanup workspace tiles.
for (int64_t i = 0; i < C.mt(); ++i) {
    for (int64_t j = 0; j < C.nt(); ++j) {
        if (C.tileIsLocal(i, j) && device == C.tileDevice(i, j)) {
            // erase tmp local and remote device tiles;
            A.tileRelease(i, 0, device);
            B.tileRelease(0, j, device);
            // decrement life for remote tiles
            A.tileTick(i, 0);
            B.tileTick(0, j);
        }
    }
}
#pragma omp taskwait
if (err)
    throw std::exception();
2.3. **INTERNAL ROUTINES FOR MAJOR, PARALLEL TASKS**

CHAPTER 2. **API LAYERS**

**Executing multiple internal routines on devices**

Executing multiple internal routines simultaneously on a GPU causes two critical issues: data hazards and maintaining memory consistency. A data hazard happens when multiple routines try to use the same batch array, causing read after write (RAW) and write after read (WAR) data hazards. This issue can be overcome by allocating multiple GPU batch arrays in the driver routine. Note that the function call `allocateBatchArrays` can take two arguments: (1) batch size, and (2) number of workspace arrays. The default batch size value is zero, which instructs SLATE to figure out the right batch size at the runtime—whereas the default number of workspace arrays is set to 1, which means that SLATE will allocate one set of batch arrays. Thus, at the driver routine, the user must determine the number of required workspaces when calling `allocateBatchArrays` to avoid anticipated data hazards. The very basic formula to use is: Number of internal routines without lookahead + lookahead; see `slate::potrf` device implementation in Algorithm 2.10. Upon the function call, SLATE implicitly wraps the batch pointer arrays inside a C++ `std::vector<std::vector<scalar_t**>>`, and then the C++ container is resized upon request by the driver routine. Furthermore, every device internal routine in SLATE takes an additional workspace index argument to be used by the OpenMP task. The default value is zero, which means every internal routine is set to use the first set of batch arrays, and if a user doesn't send the right index, then the data hazard would occur. A good practice to follow is numbering the internal routines without the lookahead; the lookahead routine(s) must take a variable index starting from the number of internal routines without lookahead as a base index. For example, refer to `slate::potrf` device implementation in Algorithm 2.10.

The second issue is maintaining memory consistency. The GPU internal routines of SLATE call `tileGetForReading` function on a set of tiles needed for the computation. This call copies these tiles from the host memory to the designated device memory—if and only if these tiles do not exist. At the end of the internal routine, `tileRelease` frees the device memory. When more than one internal routine uses the same set of tiles, a race condition exists calling `tileRelease`, which will remove the tiles from the memory, and the other internal routines are thereby no longer be able to read them. To overcome this issue, the `tileBcast` function call in the driver routine, which initiates the device memory by prefetching the required data for the subsequent OpenMP tasks, can take an additional `is_shared` argument that is set by default to `false`. If a user sets `is_shared` to `true`, the `tileBcast` function will call `tileGetAndHold` instead of `tileGetForReading` on the prefetched set of tiles to keep them on-hold on the device memory. This prevents any subsequent call to `tileRelease` from freeing the device memory. However, once a user flags tiles to be on-hold, the device memory is populated throughout the kernel execution cycle, and the memory space will no longer be used efficiently. Note that accumulating tiles in the device memory, where they are no longer in use by the internal routines, makes the memory footprint a limiting factor. To solve this issue, one can launch an independent OpenMP task to set these tiles un-hold and release them. For example, lines 18–30 of the device implementation of Cholesky in Algorithm 2.12 illustrate this solution, with additional demonstration of the tile release routine in Algorithm 2.13.
Algorithm 2.10 Device implementation of Cholesky factorization computational routine, \texttt{slate::potrf}. Continued in Algorithm 2.11.

```cpp
def potrf(slate::internal::TargetType<slate::Devices>,
          HermitianMatrix<scalar_t> A, int64_t lookahead)
{
    using real_t = blas::real_type<scalar_t>;
    using BcastList = typename Matrix<scalar_t>::BcastList;

    // Assumes column major
    const Layout layout = Layout::ColMajor;

    // if upper, change to lower
    if (A.uplo() == Uplo::Upper) {
        A = conj_transpose(A);
    }

    const int64_t A_nt = A.nt();

    // OpenMP needs pointer types, but vectors are exception safe
    std::vector<uint8_t> column_vector(A_nt);
    uint8_t* column = column_vector.data();

    const int priority_zero = 0;
    const int tag_zero = 0;
    const int life_factor_one = 1;
    const bool is_shared = lookahead > 0;
    const int batch_arrays_index_one = 1;
    const int64_t batch_size_zero = 0;
    const int64_t num_arrays_two = 2; // Number of kernels without lookahead

    // Allocate batch arrays = number of kernels without lookahead + lookahead
    // number of kernels without lookahead = 2 (internal::gemm & internal::trsm)
    // whereas internal::herk will be executed as many as lookaheads, thus
    // internal::herk needs batch arrays equal to the number of lookaheads
    // and the batch_arrays_index starts from
    // the number of kernels without lookahead, and then incremented by 1
    // for every execution for the internal::herk
    A.allocateBatchArrays(batch_size_zero, (num_arrays_two + lookahead));
    A.reserveDeviceWorkspace();

#pragma omp parallel
#pragma omp master
{
    omp_set_nested(1);
    for (int64_t k = 0; k < A_nt; ++k) {
        // Panel, normal priority
        #pragma omp task depend(inout:column[k])
        {
            // factor A(k, k)
            internal::potrf<slate::HostTask>(A.sub(k, k));

            // send A(k, k) down col A(k+1:nt-1, k)
            if (k+1 <= A_nt-1)
                A.tileBcast(k, k, A.sub(k+1, A_nt-1, k, k), layout);
```
Algorithm 2.11 Device implementation of Cholesky factorization. Continued from Algorithm 2.10. Continued in Algorithm 2.12.

```c
// A(k+1:nt-1, k) * A(k, k)'(-H)
if (k+1 <= A_nt-1) {
    auto Akk = A.sub(k, k);
    auto Tkk = TriangularMatrix<scalar_t>(Diag::NonUnit, Akk);
    internal::trsm<Target::Devices>(
        Side::Right,
        scalar_t(1.0), conj_transpose(Tkk),
        A.sub(k+1, A_nt-1, k, k),
        priority_zero, layout, batch_arrays_index_one);
}

BcastList bcast_list_A;
for (int64_t i = k+1; i < A_nt; ++i) {
    // send A(i, k) across row A(i, k+1:i) and
down col A(i:nt-1, i)
    bcast_list_A.push_back({i, k, {A.sub(i, i, k+1, i),
                                A.sub(i, A_nt-1, i, i)}});
}

// "is_shared" is to request copying the tiles to the devices,
// and set them on-hold, which avoids releasing them by either
// internal::gemm or internal::herk
// (avoiding possible race condition)
A.template listBcast<Target::Devices>(
    bcast_list_A, layout, tag_zero, life_factor_one, is_shared);

// update trailing submatrix, normal priority
if (k+1+lookahead < A_nt) {
    #pragma omp task depend(in:column[k])
    depend(inout:column[k+1+lookahead])
    depend(inout:column[A_nt-1])
    {
        // A(kl+1:nt-1, kl+1:nt-1) -=
        // A(kl+1:nt-1, k) * A(kl+1:nt-1, k)'H
        // where kl = k + lookahead
        internal::herk<Target::Devices>(
            real_t(-1.0), A.sub(k+1+lookahead, A_nt-1, k, k),
            real_t( 1.0), A.sub(k+1+lookahead, A_nt-1));
    }
}

// update lookahead column(s), normal priority
// the batch_arrays_index_1a must be initialized to the
// lookahead base index (i.e. number of kernels without lookahead),
// which is equal to "2" for slate::potrf, and then the variable is
// incremented with every lookahead column "j" ( j=k+1 = 2+j-(k+1) )
for (int64_t j = k+1; j < k+1+lookahead && j < A_nt; ++j) {
    #pragma omp task depend(in:column[k])
    depend(inout:column[j])
    {
```
Algorithm 2.12 Device implementation of Cholesky factorization. Continued from Algorithm 2.11.

```
1 // A(j, j) -= A(j, k) * A(j, k)\H
2 internal::herk<Target::Devices>(
3     real_t(-1.0), A.sub(j, j, k, k),
4     real_t( 1.0), A.sub(j, j));
5
6 // A(j+1:nt, j) -= A(j+1:nt-1, k) * A(j, k)\H
7 if (j+1 <= A_nt-1) {
8     auto Ajk = A.sub(j, j, k, k);
9     internal::gemm<Target::Devices>(
10         scalar_t(-1.0), A.sub(j+1, A_nt-1, k, k),
11         conj_transpose(Ajk),
12         scalar_t( 1.0), A.sub(j+1, A_nt-1, j, j),
13         layout, priority_zero, j-k+1);
14 }
15 }
16
17 // update the status of the on-hold tiles held by the invocation of
18 // the tileBcast routine, and then release them to free up memory
19 // the origin must be updated with the latest modified copy.
20 // for memory consistency
21 // TODO: find better solution to handle tile release, and
22 // investigate the correctness of the task dependency
23 if (lookahead > 0 && k >= lookahead) {
24     #pragma omp task depend(in: column[k]) \
25                 depend(inout: column[k+1])
26     {
27         potrfReleasePanel(A, k - lookahead);
28     }
29 }
30 }
31 #pragma omp taskwait
A.tileUpdateAllOrigin();
32 }
33 }
34 A.releaseWorkspace();
35 }
36 // namespace specialization
37 } // namespace internal
```
Algorithm 2.13 An auxiliary routine to release the panel tiles that are broadcast in \texttt{slate::potrf}.

```cpp
namespace internal {
namespace specialization {

// An auxiliary routine to release the panel tiles that are broadcast. Since
// the broadcast tiles are flagged to be OnHold on the devices memories to be
// accessed by multiple internal kernels while preventing the tileRelease call
// in these routine to release them before the others finish accessing
// them. Note: this function update the tiles origin to make sure that
// the origin memory is up-to-date and the coherency is kept consistent
// across multiple address spaces.

// A: Matrix to factor; overwritten by L
// k: Current column k of the input matrix A.

// template<typename scalar_t>
void potrfReleasePanel(HermitianMatrix<scalar_t> A, int64_t k) {
    const int64_t A_nt = A.nt();
    for (int64_t i = k + 1; i < A_nt; ++i) {
        if (A.tileIsLocal(i, k)) {
            A.tileUpdateOrigin(i, k);
            std::set<int> dev_set;
            A.sub(i, i, k + 1, i).getLocalDevices(&dev_set);
            A.sub(i, A.nt - 1, i, i).getLocalDevices(&dev_set);
            for (auto device : dev_set) {
                A.tileUnsetHold(i, k, device);
                A.tileRelease(i, k, device);
            }
        }
    }
}
```
2.4 Tile operations for small, sequential tasks

Tile routines update one or a small number of individual tiles, generally sequentially on a single CPU core. For instance, a tile gemm takes three tiles, \( A \), \( B \), and \( C \), and updates \( C \). Transposition of individual tiles is resolved at this level when calling optimized BLAS. This allows higher-level operations to ignore whether a matrix is transposed or not. Currently, all tile operations are CPU-only, since accelerators use only batch operations. Algorithm 2.14 gives an example of the tile gemm routine, used in the internal gemm routine (Algorithm 2.6).

Algorithm 2.14 Tile matrix multiply routine, `slate::gemm`. Cases for transposed \( C ( C^T \) and \( C^H \)) are omitted.

```cpp
1 // C = alpha AB + beta C
2 template <typename scalar_t>
3 void gemm(
4     scalar_t alpha, Tile<scalar_t> const& A,
5     Tile<scalar_t> const& B,
6     scalar_t beta, Tile<scalar_t>& C)
7 {
8     if (C.op() == Op::NoTrans) {
9         // C = opA(A) opB(B) + C
10         blas::gemm(blas::Layout::ColMajor,
11             A.op(), B.op(), // transpositions
12             C.mb(), C.nb(), A.nb(), // tile dimensions
13             alpha, A.data(), A.stride(),
14             B.data(), B.stride(),
15             beta, C.data(), C.stride());
16     } else { ... }
17 }
```

2.5 BLAS++, Batched BLAS++, and LAPACK++

At the lowest level, the BLAS++ and LAPACK++ packages provide thin, precision-independent, overloaded C++ wrappers around traditional BLAS, batched BLAS, and LAPACK routines, as discussed in Chapter 6. They use C++ calling conventions and enum values instead of character constants, but otherwise the calling sequence is similar to the standard BLAS and LAPACK routines. BLAS++ also includes batched BLAS, on both CPUs and GPUs.

A slightly higher-level interface taking arrays as `mdspan` objects may be developed as `mdspan` becomes standardized [3] and widespread in C++ standard library implementations. That would eliminate the separate dimension arguments, yielding, for instance,

```cpp
1 gemm( transA, transB, alpha, A, B, beta, C )
```

where \( A \), \( B \), and \( C \) are `mdspan` objects encapsulating their dimensions and column or row strides.
2.6 Work routines for actual OpenMP work

SLATE implements a middle “work” layer between the computational routines and the internal routines, which is called from within an OpenMP parallel region. This layer was introduced in early 2020, based on the needs of the generalized eigenvalue routine `slate::hegst`. Outside this context, it has not been widely used in SLATE. Some computational routines, e.g., `slate::hegst`, need to work on problem sizes larger than what internal routines can handle inside their parallel region. For instance, `slate::hegst` calls `slate::trsm` and `slate::trmm` that take a large triangular matrix, instead of a single tile triangular matrix. The `internal::trsm` and `internal::trmm` handle only the single tile triangular matrix case. Therefore, `slate::work` layer can allow `slate::hegst` to invoke a big triangular matrix solve and triangular matrix-matrix multiplication within `slate::hegst`, without having multiple OpenMP parallel regions. Having the `slate::work` layer avoids nested parallel regions (e.g., one computational routine calling another computational routine). In addition to the issues with nesting, there are also potential correctness and performance issues with initializing and cleaning up memory. Note that to avoid a segmentation fault with `#pragma omp taskwait` at the end of a `slate::work` routine, it must be invoked inside an `#pragma omp task`; we encountered this issue when running on the Summit supercomputer.

Algorithm 2.15 gives an example of the triangular matrix solve computational routine, and Algorithm 2.16 gives an example of a triangular matrix solve work routine. Algorithm 2.18 gives an example of the computational routine for the reduction of a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.
Algorithm 2.15 Triangular matrix solve computational routine, slate::trsm.

namespace slate {

template <Target target, typename scalar_t>
void trsm(slate::internal::TargetType<target>,
          Side side,
          scalar_t alpha, TriangularMatrix<scalar_t> A,
          Matrix<scalar_t> B,
          int64_t lookahead)
{
    if (target == Target::Devices) {
        const int64_t batch_size_zero = 0;
        const int64_t num_arrays_two = 2; // Number of kernels without lookahead
        // Allocate batch arrays = number of kernels without
        // lookahead + lookahead
        // number of kernels without lookahead = 2
        // (internal::gemm & internal::trsm)
        // TODO
        // whereas internal::gemm with lookahead will be executed as many as
        // lookaheads, thus
        // internal::gemm with lookahead needs batch arrays equal to the
        // number of lookaheads
        // and the batch_arrays_index starts from
        // the number of kernels without lookahead, and then incremented by 1
        // for every execution for the internal::gemm with lookahead
        B.allocateBatchArrays(batch_size_zero, num_arrays_two);
        B.reserveDeviceWorkspace();
    }
    B.tileUpdateAllOrigin();
    B.releaseWorkspace();
}

// namespace slate
Algorithm 2.16 Triangular matrix solve work routine, slate::work::trsm. Handling upper triangular (uplo = Uplo::Upper) is omitted here; see SLATE code for details.

```cpp
namespace slate {
namespace work {

    template <Target target, typename scalar_t>
    void trsm(Side side, scalar_t alpha, TriangularMatrix<scalar_t> A,
              Matrix<scalar_t> B,
              uint8_t* row, int64_t lookahead) {
        using blas::conj;
        using BcastList = typename Matrix<scalar_t>::BcastList;
        // Assumes column major
        const Layout layout = Layout::ColMajor;
        // if on right, change to left by (conj)-transposing A and B to get
        // op(B) = op(A)^(-1) * op(B)
        if (side == Side::Right) {
            if (A.op() == Op::ConjTrans || B.op() == Op::ConjTrans) {
                A = conj_transpose(A);
                B = conj_transpose(B);
                alpha = conj(alpha);
            } else {
                A = transpose(A);
                B = transpose(B);
            }
        }
        // B is mt-by-nt, A is mt-by-mt (assuming side = left)
        assert(A.mt() == B.mt());
        assert(A.nt() == B.mt());
        const int64_t mt = B.mt();
        const int64_t nt = B.nt();
        const int priority_one = 1;
        const int priority_zero = 0;
        const int64_t batch_arrays_index_zero = 0;
        const int64_t batch_arrays_index_one = 1;

        if (A.uplo() == Uplo::Lower) {
            // ----------------------------
            // Lower/NoTrans or Upper/Trans, Left case
            // Forward sweep
            for (int64_t k = 0; k < mt; ++k) {
                scalar_t alph = k == 0 ? alpha : scalar_t(1.0);
                // panel (Akk tile)
                #pragma omp task depend(inout:row[k]) priority(1)
                {
                    // send A(k, k) to ranks owning block row B(k, :)
                    A.template tileBcast(k, k, B.sub(k, k, 0, nt-1).layout);
                    // solve A(k, k) B(k, :) = alpha B(k, :)
                    internal::trsm<target>(
                        Side::Left,
                        alph, A.sub(k, k),
                        B.sub(k, k, 0, nt-1),
                        priority_one, layout, batch_arrays_index_one);
                    // send A(i=k+1:mt-1, k) to ranks owning block row B(i, :)
                    BcastList bcast_list_A;
                    for (int64_t i = k+1; i < mt; ++i)
                        bcast_list_A.push_back((i, k, B.sub(i, k, 0, nt-1)));
                    A.template listBcast<target>(bcast_list_A, layout);
                }
            }

            // ----------------------------------------
            // Lower/NoTrans or Upper/Trans, Left case
            // Forward sweep
            for (int64_t k = 0; k < mt; ++k) {
            }

        } else {
            // ----------------------------------------
            // Lower/NoTrans or Upper/Trans, Left case
            // Forward sweep
            for (int64_t k = 0; k < mt; ++k) {
                scalar_t alph = k == 0 ? alpha : scalar_t(1.0);
                // panel (Akk tile)
                #pragma omp task depend(inout:row[k]) priority(1)
                {
                    // send A(k, k) to ranks owning block row B(k, :)
                    A.template tileBcast(k, k, B.sub(k, k, 0, nt-1).layout);
                    // solve A(k, k) B(k, :) = alpha B(k, :)
                    internal::trsm<target>(
                        Side::Left,
                        alph, A.sub(k, k),
                        B.sub(k, k, 0, nt-1),
                        priority_one, layout, batch_arrays_index_one);
                    // send A(i=k+1:mt-1, k) to ranks owning block row B(i, :)
                    BcastList bcast_list_A;
                    for (int64_t i = k+1; i < mt; ++i)
                        bcast_list_A.push_back((i, k, B.sub(i, k, 0, nt-1)));
                    A.template listBcast<target>(bcast_list_A, layout);
                }
            }
        }
    }
}
}
```
Algorithm 2.17 Triangular matrix solve work routine, slate::work::trsm. Handling upper triangular (uplo = Uplo::Upper) is omitted here; see SLATE code for details. Continued from Algorithm 2.16.

```cpp
// send B(k, j=0:nt-1) to ranks owning block col B(k+1:mt-1, j)
BcastList bcast_list_B;
for (int64_t j = 0; j < nt; ++j) {
    bcast_list_B.push_back((k, j, {B.sub(k+1, mt-1, j, j)}));
}
B.template listBcast<target>(bcast_list_B, layout);

// lookahead update, B(k+1:k+la, :) -= A(k+1:k+la, k) B(k, :)
for (int64_t i = k+1; i < k+1+lookahead && i < mt; ++i) {
    #pragma omp task depend(in: row[k])
    #pragma omp task depend inout: row[i] priority (1)
    {
        // TODO: execute lookahead on devices
        internal::gemm<Target::HostTask>(
            scalar_t(-1.0), A.sub(i, i, k, k),
            B.sub(k, k, 0, nt-1),
            alph, B.sub(i, i, 0, nt-1),
            layout, priority_one);
    }
}

// trailing update, B(k+1+lookahead:mt-1, :) -= A(k+1+lookahead:mt-1, k) B(k, :)
if (k+1+lookahead < mt) {
    #pragma omp task depend(in: row[k])
    #pragma omp task depend inout: row[k+1+lookahead]) \ 
    #pragma omp task depend inout: row[mt-1])
    {
        internal::gemm<Target>(
            scalar_t(-1.0),
            A.sub(k+1+lookahead, mt-1, k, k),
            B.sub(k, k, 0, nt-1),
            alph, B.sub(k+1+lookahead, mt-1, 0, nt-1),
            layout, priority_zero, batch_arrays_index_zero);
    }
    #pragma omp taskwait
}
```
2.6. WORK ROUTINES FOR ACTUAL OPENMP WORK  
CHAPTER 2. API LAYERS

Algorithm 2.18 Computational routine for the reduction of a complex Hermitian positive-definite generalized eigenvalue problem to the standard form, \texttt{slate::hegst}. Handling \texttt{itype = 2} and \texttt{itype = 2} is omitted here; see SLATE code for details.

```cpp
namespace slate {

  template <Target target, typename scalar_t>
  void hegst(slate::internal::TargetType<target>,
             int64_t itype, HermitianMatrix<scalar_t> A,
             HermitianMatrix<scalar_t> B,
             int64_t lookahead)
  {
    using BcastList = typename Matrix<scalar_t>::BcastList;
    if (itype != 1 && itype != 2 && itype != 3) {
      throw Exception("itype must be: 1, 2, or 3");
    }
    slate_assert(A.uplo() == B.uplo());
    slate_assert(A.nt() == B.nt());
    if (A.uplo() == Uplo::Upper) {
      A = conj_transpose(A);
      B = conj_transpose(B);
    }

    const int64_t nt = A.nt();
    const scalar_t half = 0.5;
    const scalar_t one = 1.0;
    const int tag_zero = 0;
    const int life_factor_two = 2;
    const Layout layout = Layout::ColMajor;
    // OpenMP needs pointer types, but vectors are exception safe
    uint8_t* column = column_vector.data();
    if (target == Target::Devices) {
      const int64_t batch_size_zero = 0;
      const int64_t num_arrays_two = 2;
      A.allocateBatchArrays(batch_size_zero, num_arrays_two);
      A.reserveDeviceWorkspace();
    }

    #pragma omp parallel
    #pragma omp master
    {  
      omp_set_nested(1);
      for (int64_t k = 0; k < nt; ++k) {
        auto Akk = A.sub(k, k);
        auto Bkk = B.sub(k, k);
        auto TBkk = TriangularMatrix<scalar_t>(Diag::NonUnit, Bkk);
        if (itype == 1) {
          #pragma omp task depend(inout:column[k])
          {
            internal::hegst<Target::HostTask>(
              itype, std::move(Akk),
              std::move(Bkk));
          }
        }
        if (k+1 <= nt-1) {
          auto Asub = A.sub(k+1, nt-1, k, k);
          auto Bsub = B.sub(k+1, nt-1, k, k);
          #pragma omp task depend(inout:column[k])
          {
            B.template tileBcast<Target>(k, k, Asub, layout);
            internal::trsm<Target>(
              Side::Right, one, conj_transpose(TBkk),
              std::move(Asub));
          }
        }
      }
    }

  }

}
```
Algorithm 2.19 Triangular matrix solve work routine, \texttt{slate::work::trsm}. Handling upper triangular (\texttt{uplo = Uplo::Upper}) is omitted here; see SLATE code for details. Continued from Algorithm 2.18.

```cpp
#pragma omp task depend(inout: column[k])
{
    A.tileBcast(
    k, k, Asub, layout, tag_zero, life_factor_two);

    BcastList bcast_list;
    for (int64_t i = k+1; i < nt; ++i) {
        bcast_list.push_back({i, k, (A.sub(i, i, k+1, i),
            A.sub(i, nt-1, i, i))});
    }

    B.template listBcast<target>(
        bcast_list, layout, tag_zero, life_factor_two);
}

#pragma omp task depend(in: column[k]) \ 
    depend(inout: column[k+1]) \ 
    depend(inout: column[nt-1])
{
    internal::hemm<Target::HostTask>(
        Side::Right, -half, std::move(Akk),
        std::move(Bsub),
        one, std::move(Asub));

    BcastList bcast_list;
    for (int64_t i = k+1; i < nt; ++i) {
        bcast_list.push_back({i, k, (A.sub(i, i, k+1, i),
            A.sub(i, nt-1, i, i))});
    }

    A.template listBcast<target>(bcast_list, layout);

    internal::her2k<target>(
        -one, std::move(Asub),
        std::move(Bsub),
        rone, A.sub(k+1, nt-1));

    internal::hemm<Target::HostTask>(
        Side::Right, -half, std::move(Akk),
        std::move(Bsub),
        one, std::move(Asub));

    auto Bk1 = B.sub(k+1, nt-1);
    auto TBk1 = TriangularMatrix<scalar_t>(Diag::NonUnit, Bk1);
    work::trsm<target, scalar_t>({ 
        Side::Left, one, TBk1,
        Asub, column, lookahead});
}
A.tileUpdateAllOrigin();
A.releaseWorkspace();
}
```

// namespace slate
SLATE makes tiles first-class objects that can be individually allocated and passed to low-level tile routines. 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 storage, easing an application’s transition from ScaLAPACK to SLATE. Compared to other distributed dense linear algebra formats, SLATE’s matrix structure offers numerous advantages:

First, the same structure can be used for holding many different matrix types: general, symmetric, triangular, band, symmetric band, etc., as shown in Figure 3.1. Little memory is wasted for storing parts of the matrix that hold no useful data (e.g., the upper triangle of a lower triangular matrix). Instead of wasting $O(n^2)$ memory as ScaLAPACK does, only $O(nn_b)$ memory is wasted in the diagonal tiles for a block size $n_b$; all unused off-diagonal tiles are simply never allocated. There is no need for using complex matrix storage schemes such as the Recursive Packed Format (RPF) \cite{4} or Rectangular Full Packed (RFP) \cite{5} in order to save space.

Second, the matrix can be easily converted, in parallel, from one layout to another with $O(P)$ memory overhead for $P$ processors (cores/threads). Possible conversions include: changing tile layout from column-major to row-major, “packing” of tiles for efficient BLAS execution \cite{6}, and low-rank compression of tiles. Notably, transposition of the matrix can be accomplished by transposition of each tile and remapping of the indices. There is no need for complex in-place layout translation and transposition algorithms \cite{7}.

Also, tiles can be easily allocated and copied among different memory spaces. Both inter-node communication and intra-node communication are vastly simplified. Tiles can be easily and efficiently transferred between nodes using MPI. Tiles can be easily moved in and out of fast memory, such as the MCDRAM in Xeon Phi processors. Tiles can also be copied to one or more device memories in the case of GPU acceleration.
Figure 3.1: General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored.

Figure 3.2: View of symmetric matrix on process (0, 0) in $2 \times 2$ process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1).

Figure 3.3: Block sizes can vary. Most algorithms require square diagonal tiles.
In practical terms, a SLATE matrix is implemented using the `std::map` container from the C++ standard library as:

```cpp
std::map< std::tuple<int64_t, int64_t>, TileNode<scalar_t>* >
```

The map's key is a tuple consisting of the tile's \((i, j)\) block row and column indices in the matrix. The `TileNode` can then be indexed by the host or accelerator device ID to retrieve a `TileInstance`, which is a simple structure containing the `Tile` itself, its MOSI state (see Chapter 9), and a lock. SLATE relies on global indexing of tiles, meaning that each tile is identified by the same unique tuple across all processes. The lightweight `Tile` object stores a tile's data and properties such as dimensions, uplo, and transposition operation.

In addition to facilitating the storage of different types of matrices, this structure also readily accommodates partitioning of the matrix to the nodes of a distributed-memory system. Each node stores only its local subset of tiles, as shown in Figure 3.2. Mapping of tiles to nodes is defined by a C++ lambda function, and set to 2D block cyclic mapping by default, but the user can supply an arbitrary mapping function. Similarly, distribution to accelerators within each node is ID block cyclic by default, but the user can substitute an arbitrary function.

Remote access is realized by replicating remote tiles in the local matrix for the duration of the operation. This is shown in Figure 3.2 for the trailing matrix update in Cholesky, where portions of the remote panel (yellow) have been copied locally.

Finally, SLATE can support non-uniform tile sizes (Figure 3.3). Most factorizations require that the diagonal tiles are square, but the block row heights and block column widths can, in principle, be arbitrary. This will facilitate applications where the block structure is significant, for instance in Adaptive Cross Approximation (ACA) linear solvers [8].

### 3.0.1 Tile management

A `Tile` can be one of three types, as denoted by the enum `TileKind`:

```cpp
global enum class TileKind
{
  Workspace,
  SlateOwned,
  UserOwned,
};
```

defined by:

**UserOwned**: User allocated origin tile. This is the original instance of a tile initialized upon matrix creation. The tile's memory is managed by the user, not by SLATE. The tile has been initialized with a pre-existing data buffer. The tile's memory should not be freed by SLATE.

**SlateOwned**: SLATE-allocated origin tile. This is the original instance of the tile received upon matrix creation or by `tileInsert()`. The tile's memory is managed by SLATE, and is freed when the matrix is destructed.
**Workspace:** SLATE-allocated workspace tile. This is an instance of the tile that is used as temporary workspace in a memory space different from that of the corresponding origin tile. The tile is created with `tileInsertWorkspace()` for receiving a remote tile copy or for computation on a different device (CPU or accelerator) than the origin. It should be released back to the matrix’s memory pool after being used.

It is important to note that at most one instance of a tile per memory space (i.e., per CPU or accelerator device) is allowed.

An operation computing on a device needs to create copies of the involved tiles on the device as workspace tiles and purge them after usage in order to minimize memory consumption. On the other hand, certain algorithms may need to hold a set of tiles on the device for the duration of the algorithm to allow multiple accesses to these tiles and minimize the data traffic from/to host memory to/from device memory. These requirements necessitate the adoption of a coherency protocol that seamlessly manages the tile copies on various memory spaces, as described in Chapter 9.
The design of SLATE revolves around the Tile class and the Matrix class hierarchy listed below. 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.

**BaseMatrix** Abstract base class for all matrices.

**Matrix** General, $m \times n$ matrix.

**BaseTrapezoidMatrix** Abstract base class for all upper or lower trapezoid storage, $m \times n$ matrices. For upper, tiles $A(i, j)$ for $i \leq j$ are stored; for lower, tiles $A(i, j)$ for $i \geq j$ are stored.

**TrapezoidMatrix** Upper or lower trapezoid, $m \times n$ matrix; the opposite triangle is implicitly zero.

**TriangularMatrix** Upper or lower triangular, $n \times n$ matrix.

**SymmetricMatrix** Symmetric, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = a_{i,j}$).

**HermitianMatrix** Hermitian, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = \overline{a_{i,j}}$).

**BaseBandMatrix** Abstract base class for band matrices, with a lower bandwidth $k_l$ (number of sub-diagonals) and upper bandwidth $k_u$ (number of super-diagonals).

**BandMatrix** General, $m \times n$ band matrix. All tiles within the band exist, e.g., $A(i, j)$ for $j = i - k_l, \ldots, i + k_u$. 
BaseTriangularBandMatrix  Abstract base class for all upper or lower triangular storage, \( n \times n \) band matrices. For upper, tiles within the band in the upper triangle exist; for lower, tiles within the band in the lower triangle exist.

TriangularBandMatrix  Upper or lower triangular, \( n \times n \) band matrix; the opposite triangle is implicitly zero.

SymmetricBandMatrix  Symmetric, \( n \times n \) band matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry \( (a_{j,i} = a_{i,j}) \).

HermitianBandMatrix  Hermitian, \( n \times n \) band matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry \( (a_{j,i} = \overline{a_{i,j}}) \).

The BaseMatrix class stores the matrix dimensions; whether the matrix is upper, lower, or general; whether it is non-transposed, transposed, or conjugate-transposed; how the matrix is distributed; and the set of tiles—both local tiles and temporary workspace tiles as needed during the computation. It also stores the distribution parameters and MPI communicators that would traditionally be stored in a ScaLAPACK context. As such, there is no separate structure to maintain state, nor any need to initialize or finalize the SLATE library.

Currently in the band matrix hierarchy there is no TrapezoidBandMatrix. This is simply because we haven’t found a need for it; if a need arises, it can be added.

SLATE routines require the correct matrix types for their arguments, which helps to ensure correctness, while inexpensive shallow copy conversions exist between the various matrix types. For instance, a general Matrix can be converted to a TriangularMatrix for doing a triangular solve (trsm), without copying. The two matrices have a reference-counted C++ shared pointer to the same underlying data (std::map of tiles).

Likewise, copying a matrix object is an inexpensive shallow copy, using a C++ shared pointer. Sub-matrices are also implemented by creating an inexpensive shallow copy, with the matrix object storing the offset from the top-left of the original matrix and the transposition operation with respect to the original matrix.

Transpose and conjugate-transpose are supported by creating an inexpensive shallow copy and changing the transposition operation flag stored in the new matrix object. For a matrix \( A \) that is a possibly transposed copy of an original matrix \( A^0 \), the function \( A . \text{op()} \) returns \( \text{Op}::\text{NoTrans}, \text{Op}::\text{Trans}, \) or \( \text{Op}::\text{ConjTrans} \), indicating whether \( A \) is non-transposed, transposed, or conjugate-transposed, respectively. The functions \( A = \text{transpose}(A^0) \) and \( A = \text{conj_transpose}(A^0) \) return new matrices with the operation flag set appropriately. Querying properties of a matrix object takes the transposition and sub-matrix offsets into account. For instance, \( A . \text{mt}() \) is the number of block rows of \( \text{op}(A_0) \), where \( A = \text{op}(A_0) = A_0, A^T_0, \) or \( A^H_0 \). The function \( A(i, j) \) returns the \( i, j \)-th tile of \( \text{op}(A_0) \), with the tile’s operation flag set to match the \( A \) matrix.

SLATE supports upper and lower storage with \( A . \text{uplo}() \) returning \( \text{Uplo}::\text{Upper} \) or \( \text{Uplo}::\text{Lower} \). Tiles likewise have a flag indicating upper or lower storage, accessed by \( A(i, j) . \text{uplo}() \). For tiles on the matrix diagonal, the uplo flag is set to match the matrix, while for off-diagonal tiles it is set to \( \text{Uplo}::\text{General} \).
The classical BLAS take 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 [9] 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, at the high level, gemm can ignore the operations on \( A \) and \( B \). If transposed, the matrix object itself handles swapping indices to obtain the correct tiles during the algorithm. At the low level, the transposition operation is set on the tiles, and is passed on to the underlying node-level BLAS gemm routine.

Similarly, the Cholesky factorization, shown in Algorithm 2.2, implements only the lower case; the upper case is handled by a shallow copy transposition to map it to the lower case. The data is not physically transposed in memory; only the transpose op flag is set so that the matrix is logically lower.

Note that for the shallow copy to work correctly, matrices must be passed by value, rather than by reference. For instance, if \( \text{potrf} \) used pass-by-reference (Algorithm 5.1), when called by a user as in:

```
1 A = HermitianMatrix( Uplo::Upper, n, ... );
2 printf( "before: op %s, uplo %s\n", op2str( A.op() ), uplo2str( A.uplo() ) );
3 potrf( A );
4 printf( "after: op %s, uplo %s\n", op2str( A.op() ), uplo2str( A.uplo() ) );
```

\( \text{potrf} \) would have the unintended side effect of transposing the matrix \( A \) in the user’s code:

```
1 before: op notrans, uplo upper
```
Instead, the matrix $A$ is passed by value into $\text{potrf}$ (Algorithm 5.2), so transposition within the computational routine doesn’t affect transposition in the user’s code. (Though some wrappers may pass it by reference.) This results in no unintended side effects:

Algorithm 5.1 Erroneous code: passing $A$ by reference and transposing it, unintentionally transposing it in caller’s code.

```cpp
template <Target target, typename scalar_t>
void potrf(slate::internal::TargetType<target>,
           HermitianMatrix<scalar_t>& A, int64_t lookahead )
{
    // If upper, change to lower.
    // Since $A$ is passed by reference (HermitianMatrix<scalar_t>& $A$),
    // this inadvertently transposes the matrix in the user’s code -- a bug!
    if (A.uplo() == Uplo::Upper) {
        A = conj_transpose(A);
    }
    // Continue with code that assumes $A$ is logically lower...
}
```

Algorithm 5.2 Correct code: passing $A$ by value and transposing it, without transposing it in caller’s code.

```cpp
template <Target target, typename scalar_t>
void potrf(slate::internal::TargetType<target>,
           HermitianMatrix<scalar_t> A, int64_t lookahead )
{
    // If upper, change to lower.
    // Since $A$ is passed by value (HermitianMatrix<scalar_t> $A$),
    // with shallow-copy semantics,
    // this doesn’t transpose the matrix $A$ in the user’s code.
    if (A.uplo() == Uplo::Upper) {
        A = conj_transpose(A);
    }
    // Continue with code that assumes $A$ is logically lower...
}
```
CHAPTER 6

Handling of Precisions

SLATE handles multiple precisions with C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. Operations are defined to apply consistently across all precisions. For instance, `blas::conj` extends `std::conj` to apply to real precisions (float, double), where it is a no-op. SLATE’s BLAS++ component [10] provides overloaded, precision-independent wrappers for all the underlying node-level BLAS, which SLATE’s PBLAS are built on top of. For instance, `blas::gemm` in BLAS++ maps to the classical `sgemm`, `dgemm`, `cgemm`, or `zgemm` BLAS, depending on the precision of its arguments. For real arithmetic, symmetric and Hermitian matrices are considered interchangeable, so `hemm` maps to `symm`, `herk` to `syrk`, and `her2k` to `syr2k`. This mapping aids in templating higher-level routines, such as Cholesky, which does a `herk` (mapped to `syrk` in real) to update the trailing matrix.

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 accommodate other data types, such as half, double-double, or quad precision, given appropriate underlying node-level BLAS. For instance, Intel MKL and NVIDIA cuBLAS provide half-precision `gemm` operations.

SLATE also implements mixed-precision algorithms [11] that factor a matrix in low precision, then use iterative refinement to attain a high-precision final result. These exploit the faster processing in low precision for the $O(n^3)$ factorization work, while refinement in the slower high precision is only $O(n^2)$ work. In SLATE, the low and high precisions are independently templated; currently we use the traditional single and double combination. However, recent interest in half precision has led to algorithms using it with either single or double [12, 13]. One could also go to higher precisions, using double-double [14] or quad for the high precision. By adding the relevant underlying node-level BLAS operations in the desired precisions to
BLAS++, the templated nature of SLATE greatly simplifies instantiating different combinations of precisions.
SLATE utilizes three or four levels of parallelism: distributed parallelism between nodes using MPI, explicit thread parallelism using OpenMP, implicit thread parallelism within the vendor’s node-level BLAS, and, at the lowest level, vector parallelism for the processor’s single instruction, multiple data (SIMD) vector instructions. For multi core, SLATE typically uses all the threads explicitly, and uses the vendor's BLAS in sequential mode. For GPU accelerators, SLATE uses a batch BLAS call, utilizing the thread block parallelism built into the accelerator’s BLAS.

The cornerstones of SLATE are (1) the SPMD programming model for productivity and maintainability, (2) dynamic task scheduling using OpenMP for maximum node-level parallelism and portability, (3) the lookahead technique for prioritizing the critical path, (4) primary reliance on the 2D block cyclic distribution for scalability, (5) reliance on the gemm operation, specifically its batch rendition, for maximum hardware utilization.

The Cholesky factorization demonstrates the basic framework, with its task graph shown in Figure 7.1 and code shown in Algorithm 2.2. Dataflow tasking (omp task depend, Algorithm 2.2 lines 32, 64, 85) is used for scheduling operations with dependencies on large blocks of the matrix. Dependencies are performed on a dummy vector, representing each block column in the factorization, rather than on the matrix data itself. Within each large block, either nested tasking (omp task, Algorithm 2.6 line 14) or batch operations of independent tile operations are used for scheduling individual tile operations to individual cores, without dependencies. For accelerators, batched BLAS calls are used for fast processing of large blocks of the matrix, using accelerators.

Compared to pure tile-by-tile dataflow scheduling, as used by DPLASMA and Chameleon, this approach minimizes the size of the task graph and number of dependencies to track. For a matrix of $N \times N$ tiles, tile-by-tile scheduling creates $O(N^3)$ tasks and dependencies, which can lead to significant scheduling overheads. This is one of the main performance handicaps of the
CHAPTER 7. PARALLELISM MODEL

OpenMP version of the PLASMA library [9], in the case of many-core processors such as the Xeon Phi family. In contrast, the SLATE approach creates $O(N)$ dependencies, eliminating the issue of scheduling overheads. At the same time, this approach is a necessity for scheduling a large set of independent tasks to accelerators, in order to fully occupy their massive compute resources. It also eliminates the need to use a hierarchical task graph to satisfy the vastly different levels of parallelism on CPUs vs. on accelerators [15].

At each step of Cholesky, one or more columns of the trailing submatrix are prioritized for processing, using the OpenMP priority clause, to facilitate faster advancement along the critical path, implementing a lookahead. At the same time, the lookahead depth needs to be limited, as it is proportional to the amount of extra memory required for storing temporary tiles. Deep lookahead translates to depth-first processing of the task graph, synonymous with left-looking algorithms, but can also lead to catastrophic memory overheads in distributed-memory environments [16].

Distributed-memory computing is implemented by filtering operations based on the matrix distribution function (Algorithm 2.6 line 13); in most cases, the owner of the output tile performs the computation to update the tile. Appropriate communication calls are issued to send tiles to where the computation will occur. Management of multiple accelerators is handled by a node-level memory consistency protocol.

The user can choose among various target implementations. In the case of accelerated execution, the updates are executed as calls to batched gemm (Target::Devices). In the case of multi-core execution, the updates can be executed as:

Figure 7.1: Tasks in Cholesky factorization. Arrows depict dependencies.
CHAPTER 7. PARALLELISM MODEL

Figure 7.2: Performance of square dgemm, as fraction of maximum single-core ESSL performance (23.6 gigaFLOP/s) and cuBLAS performance (4560 gigaFLOP/s), respectively.

- a set of OpenMP tasks (Target::HostTask),
- a nested parallel for loop (Target::HostNest), or
- a call to batch gemm (Target::HostBatch).

To motivate our choices of CPU tasks on individual tiles and GPU tasks using batches of tiles, we examine the performance of dgemm. Libraries such as DPLASMA and Chameleon have demonstrated that doing operations on a tile-by-tile basis can achieve excellent CPU performance. For instance, as shown in Figure 7.2, for tile sizes $\geq 160$, IBM Engineering and Scientific Subroutine Library (ESSL) dgemm achieves over 90% of its maximum performance. In contrast, accelerators would take much larger tiles to reach their maximum performance. On an NVIDIA P100, cuBLAS dgemm would require an unreasonably large tile size $\geq 3136$ to achieve 90% of its maximum performance. DPLASMA dealt with this disparity in tile sizes between the CPU and GPU by using a hierarchical directed acyclic graph (DAG), in which the CPU has small tiles and the GPU has large tiles [15].

Instead, in SLATE we observe that most gemm operations are block outer products, where $A$ is a block column and $B$ is a block row (e.g., the Schur complement in LU factorization), and that these can be implemented using a batch gemm. In Figure 7.3, the regular cuBLAS dgemm uses standard LAPACK column-major layout, while the tiled / batch dgemm uses a tiled layout with $k \times k$ tiles and multiplies all tiles simultaneously using cuBLAS batch dgemm. This demonstrates that at specific sizes (192, 256, ...), the batched dgemm matches the performance of a regular dgemm. Thus, with an appropriately chosen, modest block size, SLATE can achieve the maximum performance from accelerators.

SLATE intentionally relies on standards in MPI, OpenMP, and BLAS to maintain easy portability. Any CPU platform with good implementations of these standards should work well for SLATE. For accelerators, any platform that implements batched gemm, on which SLATE relies, is a good target. Differences between vendors’ BLAS implementations will be abstracted at a low level in the BLAS++ library to ease porting. There are very few accelerator (e.g., CUDA) kernels in SLATE—currently just matrix norms and transposition—so porting should be a lightweight task.
Figure 7.3: Block outer-product dgemm, $C = C - AB$, where $C$ is $40,000 \times 40,000$, $A$ is $40,000 \times k$, $B$ is $k \times 40,000$. 
Communication in SLATE relies on explicit dataflow information. When a tile will be needed for computation, it is broadcast to all the processes where it is required, as shown in Figure 8.1 for broadcasting a single tile from the Cholesky panel to its trailing matrix update. Rather than explicitly listing MPI ranks, the broadcast is expressed in terms of the destination tiles to be updated. tileBcast takes a tile’s \((i,j)\) indices and a sub-matrix that the tile will update; the tile is sent to all processes owning that sub-matrix (Algorithm 2.2 lines 39 and 60). To optimize communication, listBcast aggregates a list of these tile broadcasts and pipelines the MPI and CPU-to-accelerator communication. As the set of processes involved is dynamically determined from the sub-matrix, using an MPI broadcast would require setting up a new MPI communicator, which is an expensive global blocking operation. Instead, SLATE uses point-to-point MPI communication in a hypercube tree fashion to broadcast the data.

Figure 8.1: Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix.
9.1 Coherency control

We describe here the protocol used in SLATE to maintain coherency of tiles’ instances among memory spaces (host memory, device memories). The protocol described here is inspired by known cache coherency protocols, but adapted to serve the needs of SLATE algorithms; specifically, no other memory exists as a backing store (as is the main memory in relation to a cache), nor auto eviction.

Concretely, this “coherency protocol” is used to maintain coherency between multiple copies of a tile in different memory spaces within one node (CPU memory, multiple GPU memories). Further, in this document, we will refer to this coherency protocol by the name MOSI (an acronym of the states we assign to the tiles: Modified, OnHold, Shared, Invalid).

The governing principles and requirements in MOSI protocol, besides maintaining tiles coherency, are:

- Tile data can originate in either CPU or GPU memory.
- Minimal memory occupation: workspace data to be purged when not in use.
- Data can be held in a memory space for multiple accesses.
- Minimal data transfers should be incurred across memory spaces.
- Coherent states are to be maintained at any time, i.e., any function would assume a coherent state upon entry, and will maintain that coherency upon exit. Consequently,
9.1. COHERENCY CONTROL

routines need not fix an incoherent state due to previous calls, but will make necessary
and minimal validation to ensure it is being called without violating coherency.

- The user/programmer shall be relieved, as mush as possible, from thinking about tile
state management (i.e., tile state management should be implicit).

### 9.1.1 Tile States

```c
<slate_Storage.h>:
enum MOSI
{
    Modified = 0x0100,
    OnHold = 0x1000,
    Shared = 0x0010,
    Invalid = 0x0001,
};
```

(Note this is not `enum class` because we do bitwise OR of states.)

A tile’s instance can be in one of three states: Modified, Shared, or Invalid. An additional OnHold flag can be set with any state. The states have the following meanings:

- **Modified (M)**: tile’s data is modified, other instances should be I; instance cannot be purged.
- **Shared (S)**: tile’s data is up to date, other instances may be in S or I; instance may be purged unless on hold.
- **Invalid (I)**: tile’s data is obsolete, other instances may be M, S, or I; instance may be purged unless on hold.
- **OnHold (O)**: a flag orthogonal to the three states above, indicating that a hold is set on this tile instance, thus it cannot be purged until the hold is unset.

The state of a tile instance is associated with its pointer in the TilesMap of the MatrixStorage class. Recall that a map entry holds a key being a tuple of the tile’s (row, col) position in the matrix and the device number, and a value being a struct of tile’s instance pointer and its MOSI state.

Two instances of the same tile can be in any of (I,S), (I,M), (I,I), or (S,S), as illustrated in Table 9.1. Coherence is maintained by enforcing these restrictions.

Getting and setting this state, as well as copying tiles across memory spaces, is facilitated in the MOSI API as explained next.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X ✓</td>
</tr>
<tr>
<td>S</td>
<td>X ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>I</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

**Table 9.1:** Valid state combinations of two instances of same tile.
9.1.2 MOSI API

The routines that control the tile state are the following member functions of the BaseMatrix class:

- tileState(...)  
- tileGetForReading(...)  
- tileGetForWriting(...)  
- tileModified(...)  
- tileGetAndHold(...)  
- tileUnsetHold(...)  
- tileOnHold(...)  
- tileRelease(...)

Here are the signatures of these routines and an explanation of their behavior:

```cpp
class BaseMatrix {
  ...
  // Returns tile(i, j)'s state on device (defaults to host).
  MOSI tileState(int64_t i, int64_t j, int device=host_num_);
  // Returns whether tile(i, j) is on hold on device (defaults to host).
  bool tileOnHold(int64_t i, int64_t j, int device=host_num_);
  // Sets tile(i, j)'s state on device.
  void tileState(int64_t i, int64_t j, int device, MOSI mosi);
  // Sets tile(i, j)'s state on host.
  void tileState(int64_t i, int64_t j, MOSI mosi);
  // Gets tile(i, j) for reading on device.
  // Will copy-in the tile if it does not exist or its state is Invalid.
  // Other instances will be invalidated.
  void tileGetForReading(int64_t i, int64_t j, int device=host_num_);
  // Gets all local tiles for reading on device.
  void tileGetAllForReading(int device=host_num_);
  // Gets all local tiles for reading on corresponding devices.
  void tileGetAllForReadingOnDevices();
  // Gets tile(i, j) for writing on device.
  // Sets state to Modified.
  // Will copy tile in if not exists or state is Invalid.
  // Other instances will be invalidated.
  void tileGetForWriting(int64_t i, int64_t j, int device=host_num_);
  // Gets all local tiles for writing on device.
  void tileGetAllForWriting(int device=host_num_);
  // Gets all local tiles for writing on corresponding devices.
}
```
9.1. COHERENCY CONTROL

CHAPTER 9. MOSI COHERENCY PROTOCOL

```cpp
void tileGetAllForWritingOnDevices();
// Marks tile(i, j) as Modified on device.
// Other instances will be invalidated.
// Unless permissive, asserts if other instances are in Modified state.
void tileModified(int64_t i, int64_t j, bool device=host_num_, bool permissive=false);
// Gets tile(i, j) on device and marks it as OnHold.
// Will copy tile in if it does not exist or its state is Invalid.
// Updates the source tile's state to Shared if copied-in.
void tileGetAndHold(int64_t i, int64_t j, bool device=host_num_);
// Gets all local tiles on device and marks them as OnHold.
void tileGetAndHoldAll(bool device=host_num_);
// Gets all local tiles on corresponding devices and marks them as OnHold.
void tileGetAndHoldAllOnDevices();
// Unsets tile(i, j)'s hold on device
void tileUnsetHold(int64_t i, int64_t j, bool device=host_num_);
// Deletes the tile(i, j)'s instance on device if it is a workspace tile
// that is not modified and no hold is set on it.
void tileRelease(int64_t i, int64_t j, bool device=host_num_);
// Updates the origin instance of tile(i, j) if not MOSI::Shared
void tileUpdateOrigin(int64_t i, int64_t j);
// Updates all origin instances of tiles if not MOSI::Shared
void tileUpdateAllOrigin();
// Debugging routine:
void checkTileStates();
...
```

9.1.3 Data transfer

tileGetForReading(), tileGetForWriting(), and tileGetAndHold() may initiate a data copy from a source memory space to the destination memory space. While the destination memory space is identified by the device id passed in as a parameter (could be host or GPU device), the source is automatically detected from existing instances of the same tile. In the process of finding the source, if an M instance is found it will be used; otherwise, the closest valid S instance is used (not implemented yet; currently, any valid instance is used). The closest is defined as the source instance that shares the highest bandwidth and costs least hops, considering the possibility of using peer-to-peer copy between devices.

9.1.4 State diagrams

Tile instances may change state following the operations that read or update them. Diagrams in Figures 9.1 and 9.2 illustrate the state transitions that each routine causes, while Figure 9.3 illustrates the same state transitions from the perspective of tiles.
9.1. COHERENCY CONTROL

CHAPTER 9. MOSI COHERENCY PROTOCOL

tileGetForReading()
- dst: the tile instance being acquired/updated.
- src: the tile instance read from, in case a copy is involved.
- other: all other instances of the same tile.
- A src instance cannot be Invalid, and other instances would not be in Modified state if coherence is maintained.

Figure 9.1: MOSI state transitions with tileGetForReading(), tileGetForWriting(), and tileModified() routines. Circled are the MOSI states. Arrows represent the state transition of the labeled tile instance: dst, src, and other. X+O denotes a tile instance in state X and has a hold on it. T+copy denotes a copy of data is carried to update instance T.

tileModified()
- dst: the tile instance being marked Modified.
- other: all other instances of the same tile.
- An instance, other than dst, that is in a Modified state will issue an error unless the permissive flag is true.

tileGetForWriting()
- Is a tileGetForReading() followed by a tileModified().
9.1. COHERENCY CONTROL

CHAPTER 9. MOSI COHERENCY PROTOCOL

tileGetAndHold()
- Is a tileGetForReading() followed by setting a hold on dst.

tileUnsetHold()
- Removes the hold on dst instance; other instances are not affected.

tileRelease()
- Deletes a tile instance if not in Modified state and no hold is set on it.

Figure 9.2: MOSI state transitions with tileGetAndHold(), tileUnsetHold(), and tileRelease() routines. Circled are the MOSI states. Arrows represent the state transition of the labeled tile instance: dst, src, and other. X+O denotes a tile instance in state X and has a hold on it. T+cpy denotes a copy of data is carried to update instance T.
9.1. COHERENCY CONTROL

CHAPTER 9. MOSI COHERENCY PROTOCOL

Destination tile

- A destination tile in state X is transitioned to state Y under operation OP.
- OP+cpy denotes a copy of data is carried to update the tile instance along the OP.

Source tile

- A source tile is involved in operation OP only when a data transfer/copy is needed.
- A tile cannot be in Invalid state when identified as the only source tile in a read-/write/hold operation.

Other tiles

- Other tiles are the tile instances that are not destination or source.
- Other tiles cannot be Modified in a tileModified operation.

Figure 9.3: Tile state transitions under MOSI API from a tile perspective. Circed are the tile instance states. Arrows represent the transition caused by the labeled operation: read, write, modified, hold, and unhold. X+O denotes a tile instance in state X and has a hold on it.
9.2 Developer hints

**Acquiring tiles**  An operation that consumes tiles for reading or writing should acquire the tiles first. Tiles to be read-only should be acquired using the `tileGetForReading()` routine at the operation start on the intended device, which will ensure that the most up-to-date tile instance is brought into the device. Tiles to be modified should be acquired using the `tileGetForWriting()` routine at the operation start on the intended device, which will ensure that the most up-to-date tile instance is brought in, then marks it “Modified” and invalidates other instances. An alternative is to acquire tiles to be modified using the `tileGetForReading()` routine at the operation start, then mark them modified after updating using the `tileModified()` routine.

**Tile purging**  Tiles acquired for reading, unless origin, are placed in a workspace tile instance, and should be purged after the operation is over to make room on the device’s memory. Purging is accomplished by calling tile `tileRelease()` routine, which will delete a tile instance only if it is a workspace with no hold on it and not modified. `tileErase()`, on the other hand, erases the indicated tile instance unconditionally, and should therefore be used carefully.

**Modified tiles**  A tile instance that is acquired by `tileGetForWriting()` is marked Modified. However, a newly inserted tile instance may get updated without using the `slate::internal` routines, for example, by issuing lapack calls on them, or by direct editing. In addition, tiles acquired for reading (or for writing followed by a copy to other devices) may be updated similarly. In such cases, it is necessary to call `tileModified()` in order to mark a tile as Modified and maintain coherency. `tileModified()` will invalidate other tile instances, thus forcing them to update subsequently. `tileModified()` will check if other tile instances are already in Modified state, as a coherency check, since two instances may not be modified concurrently. However, in some cases, other modified instances may need to be ignored, which can be relayed to `tileModified()` by setting the permissive parameter to true.

**Holding tiles in a memory space**  Some algorithms need to hold some tile instances with valid states in a certain memory space, and prevent them from being purged during workspace releasing. This can be accomplished using the `tileGetAndHold()`, which will put a hold on the tile until `tileUnsetHold()` is called, at which time a `tileRelease()` should generally be invoked (unless the algorithm requires otherwise).
A tile’s data can be stored in either column-major or row-major layout. In column-major layout, elements of a column have a memory stride of 1—that is, they are stored contiguously in memory, and elements of a row have a memory stride of at least the number of rows in the tile. In row-major layout, elements of a row have a memory stride of 1—that is, stored contiguously in memory, and elements of a column have a memory stride of at least the number of columns in the tile. Another representation where both the row and column strides are greater than one is possible; however, this later representation is not yet considered in SLATE.

SLATE supports converting tiles’ layout for performance considerations. Layout conversion is mainly motivated by the fact that some algorithms perform much faster when access to a tile’s element is contiguous in a row-major layout, or a column-major layout. The following sections explain the API and mechanisms used to establish layout conversion, especially tiles that cannot be transposed in-place.

10.1 Layout representation and API

The column-major or row-major layout (referred to as layout herein) is defined by the enum:

```c++
enum class Layout : char
{
    ColMajor = 'C',
    RowMajor = 'R'
};
```

The tile’s layout is stored at the tile instance (indicating the Col/Row major storage of a tile’s data) in the `Tile::layout_` member variable (Algorithm 10.1). Similarly, the matrix layout
10.2. Layout Conversion

Chapter 10. Column Major and Row Major Layout

(defaulting to ColMajor) is stored at the BaseMatrix::layout_ member variable (Algorithm 10.2). A MOSI operation (tileGetForReading(), tileGetForWriting(), etc...) specifies the layout of the destination tile instance using the following enum:

```cpp
enum class LayoutConvert : char {
    ColMajor = 'C',
    RowMajor = 'R',
    None = 'N'
};
```

Algorithms 10.1 to 10.3 show the function signatures of the API that manages tile layout conversions at the Tile, BaseMatrix, and MatrixStorage classes. The mechanisms by which tile conversion is established are explained in the next section.

10.2 Layout conversion

To foster high performance, algorithms in SLATE should operate in their preferred layout. For example, in LU factorization, row swapping during pivoting performs much better on devices when the tiles are in row-major. However, the panel factorization in the LU factorization prefers the col-major layout. As such, a runtime conversion between row-major and col-major layout is needed at the start of any computational or internal routine to ensure the tiles are in the needed layout. Obviously, the computational routine must reset the tiles layout when computations are done to the matrix original layout.

Layout conversion is implicitly handled at the MOSI calls by supplying the intended layout to the tileGet***() routines. As such, each computational routine sets a local variable indicating its preferred tile layout for computations, and passes this to any subroutine call. In turn, some internal routines can operate in both row-major or col-major tile layout, and receive a parameter to determine which layout to use, for example, internal:gemm. However, other internal routines can operate only in one of the col-major or row-major layouts, and enforce it through the tileGet***() call. It is a general and preferred practice in SLATE to fetch the set of tiles to operate on at the beginning of each internal routine using the tileGet***() calls, which receive a parameter instructing it to convert the tiles to one of the layouts (LayoutConvert::ColMajor or LayoutConvert::RowMajor), or not to convert at all (LayoutConvert::None) because the routine is layout indifferent.

The routines BaseMatrix::tileLayoutConvert**() are available to convert the layout of a tile or set of tiles into the intended layout on a certain device, possibly in batch mode. However, it is important to note that these routines should rarely be needed and are best avoided. All layout conversions should be achievable through the MOSI tileGet***() routines, which in turn call the tile conversion routines.

Keep in mind that, as a tile can have instances on any of the memory spaces available at the hardware computation node, a tile instance layout is independent of the layout of other instances of the same tile. Additionally, conversion of a tile instance’s layout does not change its MOSI state, i.e. a tile does not get MOSI::Modified by changing its layout since the data is still the same, only represented differently in memory.
Algorithm 10.1 Tile's layout member functions and member variables.

class Tile
{
    ...
    // return current layout of front buffer
    Layout layout() const;
    // set current layout flag of front buffer
    void layout(Layout in_layout);
    // return current layout of user-provided buffer
    Layout userLayout() const { return user_layout_; }
    // return Whether the front memory buffer is contiguous
    bool isContiguous() const
    {
        return (layout_ == Layout::ColMajor && stride_ == mb_) ||
            (layout_ == Layout::RowMajor && stride_ == nb_);
    }
    // Returns whether this tile can safely store its data in transposed form
    // based on its 'TileKind', buffer size, Layout, and stride.
    bool isTransposable();
    // Attaches the new_data buffer to this tile as an extended buffer
    void makeTransposable(scalar_t* data);
    // Resets the tile's member fields related to being extended.
    void layoutReset();
    // return Whether this tile has extended buffer
    bool extended() const;
    // return Pointer to the extended buffer
    scalar_t* extData();
    // return Pointer to the user allocated buffer
    scalar_t* userData();
    void layoutSetFrontDataExt(bool front = true);
    // return Pointer to the back buffer
    scalar_t* layoutBackData();
    // return Stride of the back buffer
    int64_t layoutBackStride() const;
    // Convert layout of this tile
    // CUDA stream must be provided if conversion is to happen on device
    void layoutConvert(
        scalar_t* work_data,
        cudaStream_t stream = nullptr,
        bool async = false);

protected:
    int64_t stride_; // Temporarily store user-provided-memory's stride
    int64_t user_stride_; // Temporarily store user-provided-memory's stride
    scalar_t* data_; // Temporarily point to user-provided memory buffer.
    scalar_t* user_data_; // Temporarily point to user-provided memory buffer.
    scalar_t* ext_data_; // Points to auxiliary buffer.

    /// layout_: The physical ordering of elements in the data buffer:
    /// - ColMajor: elements of a column are 1-strided
    /// - RowMajor: elements of a row are 1-strided
    Layout layout_; // Temporarily stores user-provided-memory's layout
    Layout user_layout_; // Temporarily stores user-provided-memory's layout
    ...
};
Algorithm 10.2 Matrix’s layout member functions and member variables.

class BaseMatrix
{
   ...
   public:
      // Returns matrix layout flag
      Layout layout() const;

      // Returns Layout of tile(i, j, device/host)
      Layout tileLayout(int64_t i, int64_t j, int device=host_num_);

      // Sets Layout of tile(i, j, device/host)
      void tileLayout(int64_t i, int64_t j, int device, Layout layout);

      // Returns whether tile(i, j, device/host) can be safely transposed.
      bool tileLayoutIsConvertible(int64_t i, int64_t j, int device=host_num_);

      // Converts tile(i, j, device) into 'layout'.
      void tileLayoutConvert(int64_t i, int64_t j, int device, Layout layout,
                             bool reset = false, bool async = false);

      // Converts a set of tiles on device into 'layout'.
      void tileLayoutConvert(std::set<ij_tuple>& tile_set, int device,
                              Layout layout, bool reset = false);

      void tileLayoutConvert(int device, Layout layout, bool reset = false);

      void tileLayoutConvertOnDevices(Layout layout, bool reset = false);

      void tileLayoutReset(int64_t i, int64_t j, int device, Layout layout);

      void tileLayoutReset(std::set<ij_tuple>& tile_set, int device, Layout layout);

      void tileLayoutReset();
   ...
   protected:
      // intended layout of the matrix. defaults to ColMajor.
      Layout layout_;
};

Algorithm 10.3 Matrix’s layout member functions and member variables.

class MatrixStorage
{
   ...
   public:
      void tileMakeTransposable(Tile<scalar_t>* tile);
      void tileLayoutReset(Tile<scalar_t>* tile);
   ...
10.2.1 Layout conversion of extended tiles

SLATE allocates and manages memory through the `Memory` class. At the construction of any matrix (`Matrix`, `TriangularMatrix`, etc.), the parent `BaseMatrix` constructor initiates a static object of the `MatrixStorage`, which, in addition to its functionality, acts as an interface to the static `Memory` object. Ideally, a large pool of memory is allocated at the matrix construction through the `Memory` object. Any shallow copy of the initiated matrix shares the same `MatrixStorage` and `Memory` objects.

The tiles inserted at the matrix object may occupy memory provided by the user upon construction of the matrix, or otherwise occupy memory blocks provided by the `Memory` object. Memory provided by the user for a tile may be contiguous, or may be strided, while memory provided by the `Memory` object is provided in square contiguous blocks.

For converting a layout in place, the tiles memory needs to be contiguous or square. Tiles whose memory is strided and is rectangular cannot be transposed in place. To facilitate a seamless layout conversion of all tiles, a mechanism of extending the tiles memory is developed. An extended tile has an extra memory buffer attached to it, which facilitates transposing the tiles data back and forth between the original memory buffer and the extended memory buffer. Auxiliary member variables of the `Tile` class help maintain consistent flags and memory buffer pointers of the extended tile, as shown in Algorithm 10.1. At any time, the front buffer of an extended tile (can be the original memory buffer referred to as `Tile::user_data_`, or the extended buffer referred to as `Tile::ext_data_`), holds the most up-to-date data and in the current layout.

10.3 Layout-aware MOSI

As discussed in previous sections, a call to fetch a tile through the MOSI API (`tileGet***()` routines) handles the layout conversion automatically, based on the layout conversion parameters. However, with the possibility of a tile being extended, many cases arise that can be invalid (and should be guarded against), simple to handle, or involve a more elaborate set of actions—possibly including extending a tile or allocating a temporary workspace buffer. Additionally, for performance purposes, data transposition is preferably executed on the devices whenever possible. Table 10.1 details the possible cases and summarizes the set of actions taken. These cases are implemented within the `BaseMatrix::tileCopyDataLayout()`, which is a private function callable only from the `tileGet()` routine.
### Table 10.1: Layout aware MOSI.

<table>
<thead>
<tr>
<th>Rect</th>
<th>Source</th>
<th>Destination</th>
<th>Layout</th>
<th>Device</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User</td>
<td>User</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ext</td>
<td>Ext</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rect</td>
<td>User</td>
<td>Ext</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rect</td>
<td>User</td>
<td>Ext</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rect</td>
<td>User</td>
<td>Ext</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table contains actions such as `Copy`, `Set front buffer`, `Copy to back buffer`, etc., with some actions being invalid. The layout-aware MOSI approach aims to optimize data movement and reduce overhead in software and hardware systems, ensuring efficient data processing and transfer.
In order to facilitate easy and quick adoption of SLATE, a set of compatibility APIs is provided for routines that will allow ScaLAPACK and LAPACK functions to execute using the matching SLATE routines. SLATE can support such compatibility because the flexible tile layout adopted by SLATE was purposely designed to match LAPACK and ScaLAPACK matrix layouts.

11.1 LAPACK Compatibility API

The SLATE-LAPACK compatibility API is parameter matched to standard LAPACK calls with the function names prefaced by `slate_`. The prefacing was necessary because SLATE uses standard LAPACK routines internally, and the function names would clash if the SLATE-LAPACK compatibility API used the standard names.

Each supported LAPACK routine (e.g. `gemm`) added to the compatibility library provides interfaces for all data types (single, double, single complex, double complex, mixed) that may be required. These interfaces (e.g. `slate_sgemm`, `slate_dgemm`) call a type-generic routine that sets up other SLATE requirements.

The LAPACK data is then mapped to a SLATE matrix type using a support routine `fromLAPACK`. SLATE requires a block/tile size (nb) because SLATE algorithms view matrices as composed of tiles of data. This tiling does not require the LAPACK data to be moved—it is a view on top of the pre-existing LAPACK data layout.

SLATE will attempt to manage the number of available threads so that SLATE uses the threads to generate and manage tasks and the internal lower-level BLAS calls all run single threaded. These settings may need to be altered to support different BLAS libraries since each library
may have its own methods for controlling the threads used for BLAS computations.

The SLATE execution target (e.g., HostTask, Devices) is not something available from the LAPACK function parameters (e.g., dgemm). The execution target information defaults to HostTask (running on the CPUs) but the user can specify the execution target to the compatibility routine using environment variables, allowing the LAPACK call (e.g., slate_dgemm) to execute on Device/GPU targets.

The compatibility library will then call the SLATE version of the routine (slate::gemm) and execute it on the selected target.

### 11.2 ScaLAPACK Compatibility API

The SLATE-ScaLAPACK compatibility API is intended to be link time-compatible with standard ScaLAPACK, matching both function names and parameters to the degree possible.

Each supported ScaLAPACK routine (e.g., gemm) has interfaces for all the supported data types (e.g., pdgemm, psgemm) and all the standard Fortran name manglings (i.e. uppercase, lowercase, added underscore). So, a call to a ScaLAPACK function will be intercepted using function name expected by the end user.

All the defined Fortran interface routines (e.g., pdgemm, PDGEMM, pdgemm_) call a single type-generic SLATE function that sets up the translation between the ScaLAPACK and SLATE parameters. The ScaLAPACK matrix data can be mapped to SLATE matrix types using a support function fromScaLAPACK provided by SLATE. This mapping does not move the ScaLAPACK data from its original locations. A SLATE matrix structure is defined that references the ScaLAPACK data using the ScaLAPACK blocking factor to define SLATE tiles. Note: SLATE algorithms tend to perform better at larger block sizes, especially on GPU devices, so it is preferable if ScaLAPACK uses a larger blocking factor.

The SLATE execution target (e.g., HostTask, Devices) defaults to HostTask (running on the CPUs) but the user can specify the execution target to the compatibility routine using environment variables. This allows an end user to use ScaLAPACK and SLATE within the same executable. ScaLAPACK functions that have an analogue in SLATE will benefit from any algorithmic or GPU speedup, and any functions that are not yet in SLATE will transparently fall through to the pre-existing ScaLAPACK implementations.


