SLATE Developers' Guide

Ali Charara
Mark Gates
Jakub Kurzak
Asim YarKhan
Mohammed Al Farhan
Dalal Sukkari
Treece Burgess
Neil Lindquist
Jack Dongarra

Innovative Computing Laboratory

November 5, 2023

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

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

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

Revision	Notes
12-2019	first publication
02-2020	executing multiple internal routines on devices subsection
04-2020	add work routines
08-2020	copy editing
11-2023	major revision

```
@techreport{charara2019slate,
  author={Charara, Ali and Gates, Mark and Kurzak, Jakub and YarKhan, Asim and
          Al Farhan, Mohammed and Sukkari, Dalal and Burgess, Treece and
          Lindquist, Neil 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 2023-11},
  url={https://www.icl.utk.edu/publications/swan-011},
}
```

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

SLATE (Software for Linear Algebra Targeting Exascale)[1] has been 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 revisions noted in the front matter notes and BibTeX.

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

CHAPTER 2

API Layers

SLATE's API is composed of several layers, depicted in Figure 2.1. The drivers and computational routines are the primary public API; the internal task and tile routines implement major (thread parallel) and minor (sequential) tasks, respectively. The LAPACK++ and BLAS++ packages, including Batched BLAS++, are independent packages developed for SLATE that provide a portability layer over vendor-optimized CPU and GPU LAPACK and BLAS routines.

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 also developed simplified names using overloaded functions, using the Matrix types to identify the operation to be performed. For instance, `multiply( A, B, C )` maps 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. For more information on the SLATE API, see the *SLATE Users' Guide*, chapter 6: SLATE API.

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 $Ax \cong b$ (`gels`), or a singular-value decomposition $A = U\Sigma V^H$ (`svd`). Drivers in turn call computational routines to solve sub-problems. 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,

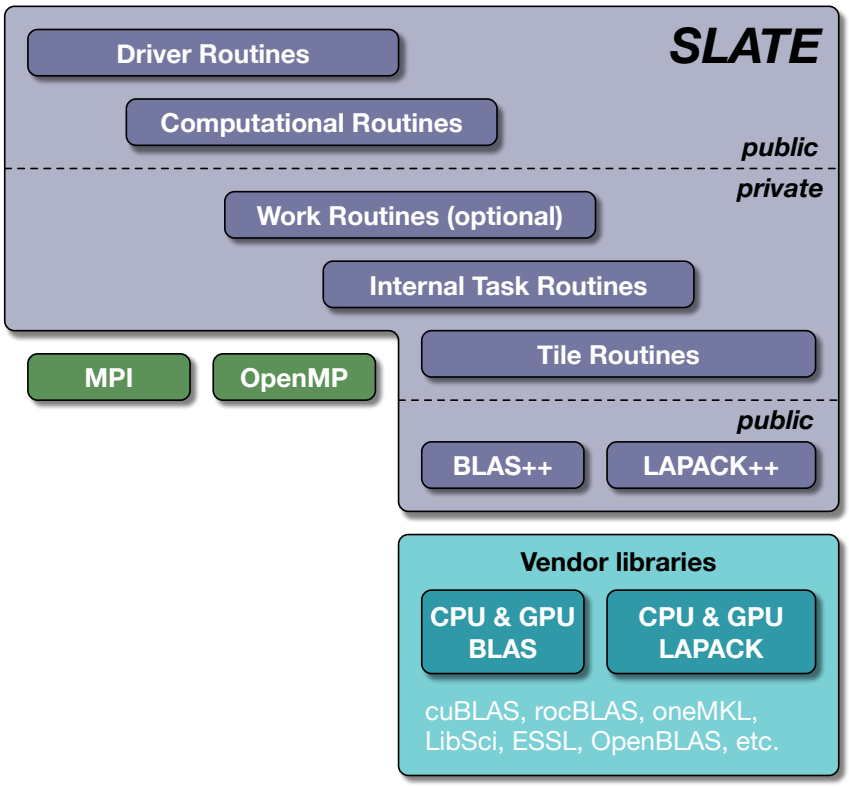


Figure 2.1: Software layers in SLATE.

as the computational routines will be. Nor do we need to unpack the opts argument; simply pass it along to the computational routines.

2.2 Computational Routines

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. If needed, code can be optimized for a specific target by providing an overloaded version, but this is discouraged. 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 (`impl::potrf`), used by the Cholesky driver. This routine is the implementation details; there is a public wrapper, `slate::potrf`, described in Section 2.2.2.

SLATE's `potrf` routine is approximately the same length as the ScaLAPACK `pzpotrf` and MAGMA `zpotrf` code (all excluding comments). Yet SLATE's code handles all precisions, multiple targets, distributed-memory and shared-memory parallelism, a variable lookahead to overlap communication and computation, and GPU acceleration. Of course, there is significant

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

```
2  namespace slate {
3
4  // Distributed parallel Cholesky factorization and solve.
5  // Solves AX = B.
6  // A: On input, matrix A to factor; on output, overwritten by L.
7  // B: On input, matrix B; on output, overwritten by X.
8  // opts: User options such as Target and Lookahead.
9  // scalar_t: Datatype: float, double, std::complex, etc.
10 // return: 0: success; i > 0: matrix not positive definite.
11 template <typename scalar_t>
12 int64_t posv(
13     HermitianMatrix<scalar_t>& A,
14     Matrix<scalar_t>& B,
15     Options const& opts)
16 {
17     // Factor A = LL^H.
18     int64_t info = potrf( A, opts );
19
20     // Solve AX = B using factorization.
21     if (info == 0) {
22         potrs( A, B, opts );
23     }
24     return info;
25 }
26
27 }  // namespace slate
```

code in lower levels, but this demonstrates that writing driver and computational routines can be simplified by delegating code complexity to lower-level abstractions. Comparing the whole library, for similar functionality, SLATE is 61k lines in 283 files, while ScaLAPACK is 238k lines in 1257 files, reflecting the roughly 4× code duplication for 4 precisions.

2.2.1 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 to handle the uplo = Upper case, so it must be passed by value; see Chapter 4.

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 (i.e., it is destructed if an exception is thrown, avoiding memory leaks), 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.

The tile life and tile tick mechanism is deprecated, to be replaced with an explicit release task. It has already been replaced in this Cholesky code, but `TileReleaseStrategy` is temporarily used to disable the tile tick mechanism in lower internal levels.

Algorithm 2.2 Cholesky factorization initialization. Continued in Algorithms 2.3 to 2.7.

```
21  namespace slate::impl {
22
23  /// Distributed parallel Cholesky factorization.
24  /// Generic implementation for any target.
25  template <Target target, typename scalar_t>
26  int64_t potrf(
27      slate::internal::TargetType<target>,
28      HermitianMatrix<scalar_t> A,
29      Options const& opts )
30  {
31      using real_t = blas::real_type<scalar_t>;
32      using BcastListTag = typename Matrix<scalar_t>::BcastListTag;
33      using lapack::device_info_int;
34
35      // Constants
36      const scalar_t one = 1.0;
37      const int priority_0 = 0;
38      const int queue_0 = 0;  // panel
39      const int queue_1 = 1;  // update
40      // Assumes column major
41      const Layout layout = Layout::ColMajor;
42
43      // Options
44      int64_t lookahead = get_option<int64_t>( opts, Option::Lookahead, 1 );
45
46      // Use only TileReleaseStrategy::Slate for potrf.
47      // Internal routines (trsm, herk, gemm) called in
48      // potrf won't release any tiles. Potrf will
49      // clean up tiles.
50      Options opts2 = Options( opts );
51      opts2[ Option::TileReleaseStrategy ] = TileReleaseStrategy::Slate;
52
53      bool hold_local_workspace = get_option<bool>(
54              opts2, Option::HoldLocalWorkspace, 0 );
55
56      // if upper, change to lower
57      if (A.uplo() == Uplo::Upper) {
58          A = conj_transpose( A );
59      }
60
61      int64_t info = 0;
62      int64_t A_nt = A.nt();
63
64      // OpenMP needs pointer types, but vectors are exception safe
65      std::vector< uint8_t > column_vector(A_nt);
66      uint8_t* column = column_vector.data();
67      SLATE_UNUSED( column ); // Used only by OpenMP
70
71      std::vector< device_info_int* > device_info_array( A.num_devices(), nullptr );
72
73      if (target == Target::Devices) {
74          // Allocate queues and batch arrays for the number of simultaneous tasks.
75          const int64_t batch_size_default = 0;
76          int num_queues = 2 + lookahead;
77          A.allocateBatchArrays( batch_size_default, num_queues );
78          A.reserveDeviceWorkspace();
79
80          for (int64_t dev = 0; dev < A.num_devices(); ++dev) {
81              blas::Queue* queue = A.comm_queue(dev);
82              device_info_array[ dev ] = blas::device_malloc<device_info_int>( 1, *queue );
83          }
84      }
```

Algorithm 2.3 Cholesky OpenMP task structure. Continued from Algorithm 2.2.

```
88      // set min number for omp nested active parallel regions
89      slate::OmpSetMaxActiveLevels set_active_levels( MinOmpActiveLevels );
90
91      #pragma omp parallel
92      #pragma omp master
93      {
94          int64_t kk = 0;  // column index (not block-column)
95          for (int64_t k = 0; k < A_nt; ++k) {
96              // Panel, normal priority
97              #pragma omp task depend( inout:column[ k ] ) priority( priority_0 ) \
98                  shared( info )
99              {
100                 // ... panel task, see Algorithm 2.4  ...
145             }
146
147             // update trailing submatrix, normal priority
148             if (k+1+lookahead < A_nt) {
149                 #pragma omp task depend( in:column[ k ] ) \
150                                  depend( inout:column[ k+1+lookahead ] ) \
151                                  depend( inout:column[ A_nt-1 ] )
152                 {
153                     // ... trailing matrix update task, see Algorithm 2.5 ...
165                 }
166             }
167
168             // update lookahead column(s), normal priority
169             // the batch_arrays_index_la must be initialized to the
170             // lookahead base index (i.e, number of kernels without lookahead),
171             // which is equal to "2" for slate::potrf, and then the variable is
172             // incremented with every lookahead column "j" ( j-k+1 = 2+j-(k+1) )
173             for (int64_t j = k+1; j < k+1+lookahead && j < A_nt; ++j) {
174                 #pragma omp task depend( in:column[ k ] ) \
175                                  depend( inout:column[ j ] )
176                 {
177                     // ... lookahead update task, see Algorithm 2.6 ...
198                 }
199             }
200
201             #pragma omp task depend( inout:column[ k ] )
202             {
203                 // ... release task, see Algorithm 2.7 ...
219             }
220             kk += A.tileNb( k );
221         }
222     }
223     A.tileUpdateAllOrigin();
224
225     if (hold_local_workspace == false) {
226         A.releaseWorkspace();
227     }
228     if (target == Target::Devices) {
229         for (int64_t dev = 0; dev < A.num_devices(); ++dev) {
230             blas::Queue* queue = A.comm_queue(dev);
231             blas::device_free( device_info_array[ dev ], *queue );
232         }
233     }
234
235     internal::reduce_info( &info, A.mpiComm() );
236     return info;
237 }
238
239 } // namespace slate::impl
```

Algorithm 2.4 Cholesky panel task in Algorithm 2.3.

```
103                 // factor A(k, k)
104                 int64_t iinfo;
105                 if (target == Target::Devices) {
106                     iinfo = internal::potrf<target>(
107                         A.sub(k, k), priority_0, queue_0,
108                         device_info_array[ A.tileDevice( k, k ) ] );
109                 }
110                 else {
111                     iinfo = internal::potrf<target>(
112                         A.sub(k, k), priority_0, queue_0 );
113                 }
114                 if (iinfo != 0 && info == 0)
115                     info = kk + iinfo;
116
117                 // send A(k, k) down col A(k+1:nt-1, k)
118                 if (k+1 <= A_nt-1)
119                     A.tileBcast(k, k, A.sub(k+1, A_nt-1, k, k), layout);
120
121                 // A(k+1:nt-1, k) * A(k, k)^{-H}
122                 if (k+1 <= A_nt-1) {
123                     auto Akk = A.sub(k, k);
124                     auto Tkk = TriangularMatrix< scalar_t >(Diag::NonUnit, Akk);
125                     internal::trsm<target>(
126                         Side::Right,
127                         one, conj_transpose( Tkk ),
128                         A.sub(k+1, A_nt-1, k, k),
129                         priority_0, layout, queue_0, opts2 );
130                 }
131
132                 BcastListTag bcast_list_A;
133                 for (int64_t i = k+1; i < A_nt; ++i) {
134                     // send A(i, k) across row A(i, k+1:i) and
135                     //                down col A(i:nt-1, i) with msg tag i
136                     bcast_list_A.push_back({i, k, {A.sub(i, i, k+1, i),
137                                                    A.sub(i, A_nt-1, i, i)},
138                                             i});
139                 }
140
141                 A.template listBcastMT<target>(
142                   bcast_list_A, layout);
```

Algorithm 2.5 Cholesky trailing matrix update task in Algorithm 2.3.

```
156                     // A(kl+1:nt-1, kl+1:nt-1) -=
157                     //     A(kl+1:nt-1, k) * A(kl+1:nt-1, k)^H
158                     // where kl = k + lookahead
159                     internal::herk<target>(
160                         real_t(-1.0), A.sub(k+1+lookahead, A_nt-1, k, k),
161                         real_t( 1.0), A.sub(k+1+lookahead, A_nt-1),
162                         priority_0, queue_1, layout, opts2 );
```

Algorithm 2.6 Cholesky lookahead update task in Algorithm 2.3.

```
180                     // A(j, j) -= A(j, k) * A(j, k)^H
181                     int queue_jk1 = j - k + 1;
182                     internal::herk<target>(
183                         real_t(-1.0), A.sub(j, j, k, k),
184                         real_t( 1.0), A.sub(j, j),
185                         priority_0, queue_jk1, layout, opts2 );
186
187                     // A(j+1:nt, j) -= A(j+1:nt-1, k) * A(j, k)^H
188                     if (j+1 <= A_nt-1) {
189                         auto Ajk = A.sub(j, j, k, k);
190                         internal::gemm<target>(
191                             -one, A.sub(j+1, A_nt-1, k, k),
192                                   conj_transpose( Ajk ),
193                             one,  A.sub(j+1, A_nt-1, j, j),
194                             layout, priority_0, queue_jk1, opts2 );
195                     }
```

Algorithm 2.7 Cholesky release task in Algorithm 2.3.

```
206                 auto panel = A.sub( k, A_nt-1, k, k );
207
208                 // Erase remote tiles on all devices including host
209                 panel.releaseRemoteWorkspace();
210
211                 // Update the origin tiles before their
212                 // workspace copies on devices are erased.
213                 panel.tileUpdateAllOrigin();
214
215                 // Erase local workspace on devices.
216                 panel.releaseLocalWorkspace();
```

2.2.2 Template dispatch

The public routine that the user actually calls (e.g., `slate::potrf`) dispatches to the target-specific versions, as shown in Algorithm 2.8. The user can specify the target as `HostTask`, `HostNest`, `HostBatch`, or `Devices` via the `opts` parameter. Note in this routine the matrix A is passed by reference, unlike the internal implementation where it is passed by value (Algorithm 2.2).

In this case, `HostNest` and `HostBatch` are dispatched to `HostTask` because they lack implementations of some lower level internal routines. We may consider removing them altogether.

Previous implementations unpacked the `opts` in the dispatch routine or another intermediate wrapper, but it's easier to just pass `opts` and unpack it in `impl::potrf`.

Algorithm 2.8 Dispatch to target implementations.

```
313 namespace slate {
314
315 template <typename scalar_t>
316 int64_t potrf(
317     HermitianMatrix<scalar_t>& A,
318     Options const& opts)
319 {
320     using internal::TargetType;
321
322     Target target = get_option( opts, Option::Target, Target::HostTask );
323
324     switch (target) {
325         case Target::Host:
326         case Target::HostNest:
327         case Target::HostBatch:
328         case Target::HostTask:
329             return impl::potrf( TargetType<Target::HostTask>(), A, opts );
330
331         case Target::Devices:
332             return impl::potrf( TargetType<Target::Devices>(), A, opts );
333     }
334     return -2;  // shouldn't happen
335 }
336
337 }  // namespace slate
```

The routine in Algorithm 2.2 is the target-specific implementation in the `impl` namespace, templated on the target. It takes a dummy `TargetType` argument, which is the C++ idiom for partial specialization of a function. In Algorithm 2.2 it is templated on the target, whereas in Algorithm 2.9 it is not templated on the target but is specialized for a specific target.

Not recommended: An overload can be given for a specific target type, as shown in Algorithm 2.9; however, this is discouraged as it creates divergent implementations. Previously, Cholesky (`impl::potrf`) had two implementations, the generic one for any target and a specialization for Devices, which diverged over time. These two were merged into a single implementation with appropriate `if (target == Target::Devices)` blocks in Algorithm 2.2.

Algorithm 2.9 (Old) Cholesky overload specialization for Target::Devices.

```
249 namespace slate::impl {
250
251 /// Distributed parallel Cholesky factorization, for Target = Devices.
252 template <typename scalar_t>
253 int64_t potrf(
254     slate::internal::TargetType<Target::Devices>,
255     HermitianMatrix<scalar_t> A,
256     Options const& opts )
257 {
258     // ... code specific to GPU Devices implementation ...
259 }
260
261 } // namespace slate::impl
```

2.2.3 Executing multiple internal routines on devices

Care must be taken when executing multiple internal routines simultaneously on a GPU to avoid data hazards and race conditions. Data hazards occur when two routines write to the same memory buffer, for instance the GPU batch array. To avoid data hazards in SLATE, we allocate multiple GPU queues and batch arrays, one for each task that will run simultaneously on the GPU. Each task is assigned a different queue, and each queue has its own batch array. In Cholesky, this is the panel task (queue 0), trailing matrix update (queue 1), and lookahead updates (queues $2, \ldots, 2 + \text{lookahead} - 1$), for a total of `num_queues = 2 + lookahead`. Every internal routine takes a queue index. They don't take a queue itself is because in a multi-GPU setting, each GPU device has its own set of queues and batch arrays.

Deprecated: *We are working to remove the implicit tile release, as it is error prone, in favor of an explicit release task. Cholesky no longer uses it.* In SLATE, race conditions occur when internal routines do `tileRelease` to release local workspace tiles. The trailing matrix update task and the lookahead update task can both release the same tile. Whichever does so first removes the tile from memory, so the other routine is no longer able to access it. Setting the `OnHold` status in MOSI will disable `tileRelease`, then the tile must be manually released later on.

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 6 for how algorithms are implemented as tasks.) For instance, in the outer k loop, `slate::gemmC` calls a sequence of `slate::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 operation 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 2.10 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.

There is a potential to reduce overheads by 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. However, we have not yet implemented this.

In `impl::potrf`, the `internal::gemm` call is an OpenMP task. Within `internal::gemm`, each individual tile gemm call is a nested OpenMP task, with no dependencies. Before each tile gemm, `tileGetForReading` and `tileGetForWriting` ensure that the tiles are in CPU memory, initiating a transfer from accelerator memory if necessary.

Deprecated: *We are working to remove tile life, as it is error prone, in favor of an explicit release task. Cholesky no longer uses it.* 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.

LU and QR panel operations also exist as internal routines. However, unlike trailing matrix updates that have independent tasks, multi-threaded panel operations for Host targets create a set of interdependent tasks. This is problematic since OpenMP has no effective way to express such multi-threaded tasks. For CALU and CAQR with target Devices, we push the panel to the GPU, eliminating issues with the multi-threaded panel.

Algorithm 2.10 Host task implementation of internal matrix multiply routine, `slate::internal::gemm`, corresponding to a single block outer product.

```
67  namespace slate::internal {
68
69  template <typename scalar_t>
70  void gemm(
71      internal::TargetType<Target::HostTask>,
72      scalar_t alpha, Matrix<scalar_t>& A,
73                      Matrix<scalar_t>& B,
74      scalar_t beta,  Matrix<scalar_t>& C,
75      Layout layout, int priority, int64_t queue_index,
76      Options const& opts )
77  {
78      using ij_tuple = typename BaseMatrix<scalar_t>::ij_tuple;
79
80      // tile life, release, and tick are deprecated.
81      TileReleaseStrategy tile_release_strategy = get_option(
82              opts, Option::TileReleaseStrategy, TileReleaseStrategy::All );
83      bool call_tile_tick = tile_release_strategy == TileReleaseStrategy::Internal
84                            || tile_release_strategy == TileReleaseStrategy::All;
85
86      std::set<ij_tuple> A_tiles_set, B_tiles_set;
87      for (int64_t i = 0; i < C.mt(); ++i) {
88          for (int64_t j = 0; j < C.nt(); ++j) {
89              if (C.tileIsLocal( i, j )) {
90                  A_tiles_set.insert( { i, 0 } );
91                  B_tiles_set.insert( { 0, j } );
92              }
93          }
94      }
95      A.tileGetForReading( A_tiles_set, LayoutConvert( layout ) );
96      B.tileGetForReading( B_tiles_set, LayoutConvert( layout ) );
97
98      #pragma omp taskgroup
99      for (int64_t i = 0; i < C.mt(); ++i) {
100         for (int64_t j = 0; j < C.nt(); ++j) {
101             if (C.tileIsLocal( i, j )) {
102                 #pragma omp task slate_omp_default_none \
103                     shared( A, B, C ) \
104                     firstprivate( i, j, layout, alpha, beta, call_tile_tick ) \
105                     priority( priority )
106                 {
107                     C.tileGetForWriting( i, j, LayoutConvert( layout ) );
108                     tile::gemm(
109                         alpha, A( i, 0 ), B( 0, j ),
110                         beta,  C( i, j ) );
111
112                     // tile life, release, and tick are deprecated.
113                     if (call_tile_tick) {
114                         A.tileTick( i, 0 );
115                         B.tileTick( 0, j );
116                     }
117                 }
118             }
119         }
120     }
121 }
122
123 } // namespace slate::internal
```

2.3.1 Batched GPU tasks

Compared to the CPU implementation in Algorithm 2.10, the batched GPU implementation is significantly more complicated. Each device is handled by a separate task in parallel (Algorithm 2.11). After some initialization to deal with transposed and conjugate-transposed matrices (Algorithm 2.12), it loops over all the relevant tiles to copy them to the GPU device if they aren't already resident (Algorithm 2.13). This uses the MOSI tile coherency API (Chapter 8) to determine which tiles need to be transferred, then transfers them with a single call. Copying of the sets is launched as nested tasks for increased parallelism.

Then it constructs the batch arrays of pointers to tiles on the GPU and calls a batched BLAS++ routine (Algorithm 2.14). Each region with a different tile size is handled by a separate batch call. Originally, 4 regions were used: the bulk of the tiles in the top-left area, and cleanup in the top-right column, bottom-left row, and bottom-right tile. This was generalized for an arbitrary number of regions, to handle sliced arrays that may require 9 regions ({ top, middle, bottom } × { left, center, right }), and non-uniform tile sizes that may require an arbitrary number of regions.

The deprecated tile release and tile tick mechanism is omitted here, and will be removed in the future.

Past work [2] investigated splitting this `internal::gemm` into two pieces: a prep step to copy tiles to the GPU and prepare the batch arrays, and an exec step to execute the batched gemm. This was helpful on single node, multi-GPU machines like an NVIDIA DGX, but didn't show benefits on distributed machines like Summit. It was incompatible with the BLAS++ batched implementation, which takes `std::vector` on host, so was removed to `slate/old/src/gemm.cc` and `slate/old/src/internal/internal_gemm_split.cc`. However, the idea could be revisited, for instance if BLAS++ could take device arrays instead of `std::vector`.

Algorithm 2.11 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 Algorithms 2.12 to 2.14.

```
381 namespace slate::internal {
382
383 template <typename scalar_t>
384 void gemm(
385     internal::TargetType<Target::Devices>,
386     scalar_t alpha, Matrix< scalar_t >& A,
387                     Matrix< scalar_t >& B,
388     scalar_t beta,  Matrix< scalar_t >& C,
389     Layout layout, int priority, int64_t queue_index,
390     Options const& opts )
391 {
392     using blas::conj;
393     using std::swap;
394     using ij_tuple = typename BaseMatrix<scalar_t>::ij_tuple;
395
396     // tile life, release, tick are deprecated.
397     TileReleaseStrategy tile_release_strategy = get_option(
398             opts, Option::TileReleaseStrategy, TileReleaseStrategy::All );
399
400     #pragma omp taskgroup
401     for (int device = 0; device < C.num_devices(); ++device) {
402         #pragma omp task shared( A, B, C ) priority( priority ) \
403             firstprivate( alpha, beta, layout, queue_index, device, tile_release_strategy )
404         {
405             // ... gemm task on each device, see Algorithms 2.12 to 2.14 ...
567         }
568     }
569 }
570
571 }  // namespace slate::internal
```

Algorithm 2.12 gemm task on one device in Algorithm 2.11, step 1, initialization.

```
409             // if op(C) is NoTrans, invert opA, opB if possible
410             Op opA = A.op();
411             if (C.op() != Op::NoTrans) {
412                 if (opA == Op::NoTrans)
413                     opA = C.op();
414                 else if (A.op() == C.op() || C.is_real) {
415                     // A and C are both Trans or both ConjTrans;
416                     // Trans == ConjTrans if real
417                     opA = Op::NoTrans;
418                 }
419                 else {
420                     throw;  // ConjNoTrans not supported
421                 }
422             }
423
424             Op opB = B.op();
425             if (C.op() != Op::NoTrans) {
426                 if (opB == Op::NoTrans)
427                     opB = C.op();
428                 else if (opB == C.op() || C.is_real) {
429                     // B and C are both Trans or both ConjTrans;
430                     // Trans == ConjTrans if real
431                     opB = Op::NoTrans;
432                 }
433                 else {
434                     throw;  // ConjNoTrans not supported
435                 }
436             }
437
438             if (C.op() == Op::ConjTrans) {
439                 alpha = conj( alpha );
440                 beta  = conj( beta );
441             }
```

Algorithm 2.13 gemm task on one device in Algorithm 2.11, step 2, copying tiles.

```
445             std::set<ij_tuple> A_tiles_set, B_tiles_set, C_tiles_set;
446             for (int64_t i = 0; i < C.mt(); ++i) {
447                 for (int64_t j = 0; j < C.nt(); ++j) {
448                     if (C.tileIsLocal( i, j )) {
449                         if (device == C.tileDevice( i, j )) {
450                             A_tiles_set.insert( { i, 0 } );
451                             B_tiles_set.insert( { 0, j } );
452                             C_tiles_set.insert( { i, j } );
453                         }
454                     }
455                 }
456             }
457
458             #pragma omp taskgroup
459             {
460                 #pragma omp task slate_omp_default_none \
461                     shared( A, A_tiles_set ) firstprivate( layout, device )
462                 {
463                     A.tileGetForReading( A_tiles_set, device, LayoutConvert( layout ) );
464                 }
465                 #pragma omp task slate_omp_default_none \
466                     shared( B, B_tiles_set ) firstprivate( layout, device )
467                 {
468                     B.tileGetForReading( B_tiles_set, device, LayoutConvert( layout ) );
469                 }
470                 #pragma omp task slate_omp_default_none \
471                     shared( C, C_tiles_set ) firstprivate( layout, device )
472                 {
473                     C.tileGetForWriting( C_tiles_set, device, LayoutConvert( layout ) );
474                 }
475             }
```

Algorithm 2.14 gemm task on one device in Algorithm 2.11, step 3, batched gemm calls.

```
479             int64_t batch_size = C_tiles_set.size();
480
481             scalar_t** a_array_host = C.array_host( device, queue_index );
482             scalar_t** b_array_host = a_array_host + batch_size;
483             scalar_t** c_array_host = b_array_host + batch_size;
484
485             // C comes first since we do computation for a local C
486             auto group_params = device_regions_build<false, 3, scalar_t>(
487                 {C, A, B},
488                 {c_array_host, a_array_host, b_array_host},
489                 device );
490
491             if (C.op() != Op::NoTrans) {
492                 swap( opA, opB );
493             }
494
495             {
496                 trace::Block trace_block( "blas::batch::gemm" );
497
498                 std::vector<Op> opA_( 1, opA );
499                 std::vector<Op> opB_( 1, opB );
500                 std::vector<scalar_t> alpha_( 1, alpha );
501                 std::vector<scalar_t> beta_( 1, beta );
502                 std::vector<int64_t> k( 1, A.tileNb( 0 ) );
503                 // info size 0 disables slow checks in batched BLAS++.
504                 std::vector<int64_t> info;
505
506                 blas::Queue* queue = C.compute_queue( device, queue_index );
507                 assert( queue != nullptr );
508
509                 for (size_t g = 0; g < group_params.size(); ++g) {
510
511                     int64_t group_count = group_params[ g ].count;
512
513                     std::vector<int64_t>    m( 1, group_params[ g ].mb );
514                     std::vector<int64_t>    n( 1, group_params[ g ].nb );
515                     std::vector<int64_t> ldda( 1, group_params[ g ].ld[1] );
516                     std::vector<int64_t> lddb( 1, group_params[ g ].ld[2] );
517                     std::vector<int64_t> lddc( 1, group_params[ g ].ld[0] );
518
519                     std::vector<scalar_t*> a_array( a_array_host, a_array_host+group_count );
520                     std::vector<scalar_t*> b_array( b_array_host, b_array_host+group_count );
521                     std::vector<scalar_t*> c_array( c_array_host, c_array_host+group_count );
522
523                     if (C.op() != Op::NoTrans) {
524                         swap( m, n );
525                         swap( a_array, b_array );
526                         swap( ldda, lddb );
527                     }
528
529                     blas::batch::gemm(
530                         layout, opA_, opB_,
531                         m, n, k,
532                         alpha_, a_array, ldda,
533                                 b_array, lddb,
534                         beta_,  c_array, lddc,
535                         group_count, info, *queue);
536
537                     a_array_host += group_count;
538                     b_array_host += group_count;
539                     c_array_host += group_count;
540                 }
541
542                 queue->sync();
543             }
```

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 2.15 gives an example of the tile gemm routine, which is used in the internal gemm routine (Algorithm 2.10).

Algorithm 2.15 Tile matrix multiply routine, `slate::tile::gemm`. Cases for transposed C (C^T and C^H) are omitted.

```
30 namespace slate::tile {
31
32 // C = alpha AB + beta C
33 template <typename scalar_t>
34 void gemm(
35     scalar_t alpha, Tile<scalar_t> const& A,
36                     Tile<scalar_t> const& B,
37     scalar_t beta,  Tile<scalar_t>& C )
38 {
39     using blas::conj;
40
41     if (C.op() == Op::NoTrans) {
42         // C = opA( A ) opB( B ) + C
43         blas::gemm( C.layout(),
44                     A.op(), B.op(),
45                     C.mb(), C.nb(), A.nb(),
46                     alpha, A.data(), A.stride(),
47                            B.data(), B.stride(),
48                     beta,  C.data(), C.stride() );
49     }
50     else {
51         // ... C^T and C^H case ...
92     }
93 }
94
95 }  // namespace slate::tile
```

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 5. 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.

Future work: 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:

```
blas::gemm( transA, transB, alpha, A, B, beta, C );
  or
blas::gemm( alpha, A, B, beta, C );
  or
blas::gemm( A, B, C );
```

where `A`, `B`, and `C` are `mdspan` objects encapsulating their dimensions and column or row strides. The `trans` and `alpha` or `beta` scaling may also be encapsulated in `mdspan`,

2.6 Work routines for actual OpenMP work

In a couple instances, 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 a `#pragma omp task`; we encountered this issue when running on the Summit supercomputer.

Algorithm 2.16 gives an example of the triangular matrix solve computational routine, and Algorithm 2.17 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 2.16 Triangular matrix solve computational routine, `slate::impl::trsmB`.

```
12  namespace slate::impl {
13
14  template <Target target, typename scalar_t>
15  void trsmB(
16      Side side,
17      scalar_t alpha, TriangularMatrix<scalar_t>& A,
18                                Matrix<scalar_t>& B,
19      Options const& opts )
20  {
21      // Options
22      int64_t lookahead = get_option<int64_t>( opts, Option::Lookahead, 1 );
23
24      if (target == Target::Devices) {
25          // Allocate batch arrays = number of kernels without
26          const int64_t batch_size_default = 0;
27          int num_queues = 2 + lookahead;
28          B.allocateBatchArrays( batch_size_default, num_queues );
29          B.reserveDeviceWorkspace();
30      }
31
32      // OpenMP needs pointer types, but vectors are exception safe
33      std::vector<uint8_t> row_vector( A.nt() );
34      uint8_t* row = row_vector.data();
35
36      // set min number for omp nested active parallel regions
37      slate::OmpSetMaxActiveLevels set_active_levels( MinOmpActiveLevels );
38
39      #pragma omp parallel
40      #pragma omp master
41      {
42          #pragma omp task
43          {
44              work::trsm<target, scalar_t>( side, alpha, A, B, row, opts );
45              B.tileUpdateAllOrigin();
46          }
47      }
48      B.releaseWorkspace();
49  }
50
51  } // namespace slate::impl
```

Algorithm 2.17 Triangular matrix solve work routine, `slate::work::trsm`.

```
13  namespace slate::work {
14
15  template <Target target, typename scalar_t>
16  void trsm(
17      Side side, scalar_t alpha,
18      TriangularMatrix<scalar_t> A,
19      Matrix<scalar_t> B,
20      uint8_t* row,
21      Options const& opts )
22  {
23      // ... call internal routines to implement trsm ...
241
242     #pragma omp taskwait
243 }
244
245 } // namespace slate::work
```

Algorithm 2.18 Computational routine for the reduction of a generalized positive-definite Hermitian eigenvalue problem to standard form, `slate::hegst`, showing setup and call to `work::trsm`.

```
13  namespace slate::impl {
14
15  template <Target target, typename scalar_t>
16  void hegst(
17      int64_t itype, HermitianMatrix<scalar_t> A,
18                     HermitianMatrix<scalar_t> B,
19      Options const& opts )
20  {
50      // OpenMP needs pointer types, but vectors are exception safe
51      std::vector<uint8_t> column_vector(nt);
52      uint8_t* column = column_vector.data();
53
54      if (target == Target::Devices) {
55          // Allocate batch arrays = number of kernels without
56          const int64_t batch_size_default = 0;
57          int num_queues = 2 + lookahead;
58          A.allocateBatchArrays( batch_size_default, num_queues );
59          A.reserveDeviceWorkspace();
60      }
139     // ...
140
141                         work::trsm<target>(
142                             Side::Left, one, TBk1, Asub, column, opts );
143
144     // ...
218 }
219
220 } // namespace slate::impl
```

CHAPTER 3

Matrix Storage

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 $O(n n_b)$ 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) [4] or *Rectangular Full Packed* (RFP) [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 [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 [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.

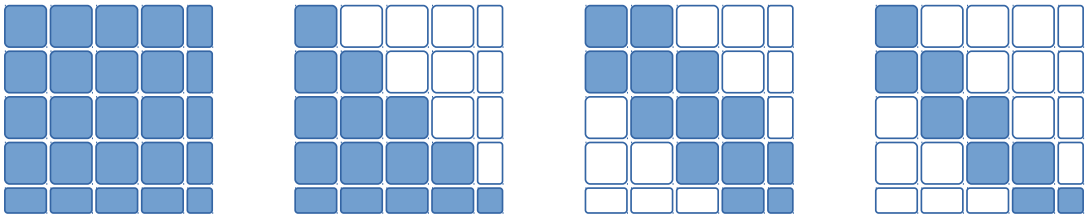


Figure 3.1: General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored.

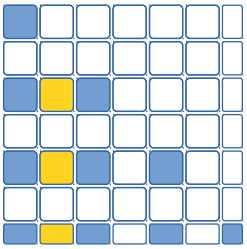


Figure 3.2: View of symmetric matrix on process (0, 0) in 2×2 process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1).

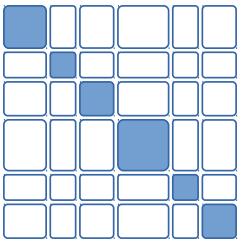


Figure 3.3: Block sizes can vary. Most algorithms require square diagonal tiles.

In practical terms, a SLATE matrix is implemented using the `std::map` container from the C++ standard library as:

```
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 the corresponding `Tile` instance. 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 MOSI state, dimensions, uplo, and transposition operation. Note that to prevent SLATE from getting into an invalid state, the canonical `Tile` instances held by the `TileNode`'s should never be directly accessible to users. Instead, modifications to a tile's properties should be done through the matrix object, and only copies should be returned.

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 1D 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`:

```
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 8.

CHAPTER 4

Handling of Side, Uplo, Trans, etc.

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 `potrf` used pass-by-reference (Algorithm 4.1), a user calling `potrf` would have the unintended side effect of transposing the matrix A in the user's code:

Code:

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

Incorrect output:

```
before: op notrans, uplo upper
after:  op conj,    uplo lower
```

Correct output:

```
before: op notrans, uplo upper
after:  op notrans, uplo upper
```

Instead, the matrix A is passed by value into `potrf` (Algorithm 4.2), 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 the correct output with no unintended side effects.

Algorithm 4.1 Erroneous code: passing A by reference and transposing it, unintentionally transposing it in caller's code.

```
1   template <Target target, typename scalar_t>
2   int64_t potrf(
3       slate::internal::TargetType<target>,
4       HermitianMatrix<scalar_t>& A, ... )
5   {
6       // If upper, change to lower.
7       // Since A is passed by reference (HermitianMatrix<scalar_t>& A),
8       // this inadvertently transposes the matrix in the user's code -- a bug!
9       if (A.uplo() == Uplo::Upper) {
10          A = conj_transpose( A );
11      }
12
13      // Continue with code that assumes A is logically lower...
14  }
```

Algorithm 4.2 Correct code: passing A by value and transposing it, without transposing it in caller's code.

```
1   template <Target target, typename scalar_t>
2   int64_t potrf(
3       slate::internal::TargetType<target>,
4       HermitianMatrix<scalar_t> A, ... )
5   {
6       // If upper, change to lower.
7       // Since A is passed by value (HermitianMatrix<scalar_t> A),
8       // with shallow-copy semantics,
9       // this doesn't transpose the matrix A in the user's code.
10      if (A.uplo() == Uplo::Upper) {
11          A = conj_transpose( A );
12      }
13
14      // Continue with code that assumes A is logically lower...
15  }
```

CHAPTER 5

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. Whereas `std::conj` applied to a real number returns `std::complex` with a zero imaginary part, which is not generally what is desired in linear algebra algorithms. For instance, `alpha = blas::conj(alpha);` works for real numbers, but `std::conj` would not work. 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 aides 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 oneMKL, NVIDIA cuBLAS, and AMD rocBLAS 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.

CHAPTER 6

Parallelism Model

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 6.1 and code shown in Algorithms 2.2 to 2.7. Dataflow tasking (`omp task depend`, Algorithm 2.2 lines 97, 149, 174, 201) 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 2.10 line 102) 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

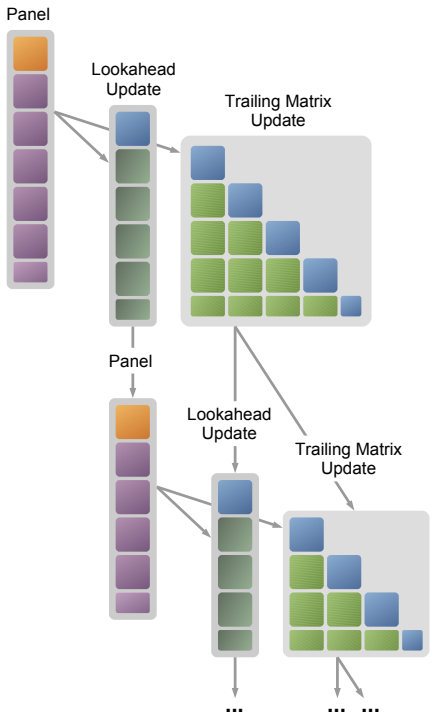


Figure 6.1: Tasks in Cholesky factorization. Arrows depict dependencies.

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 2.10 line 101); 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:

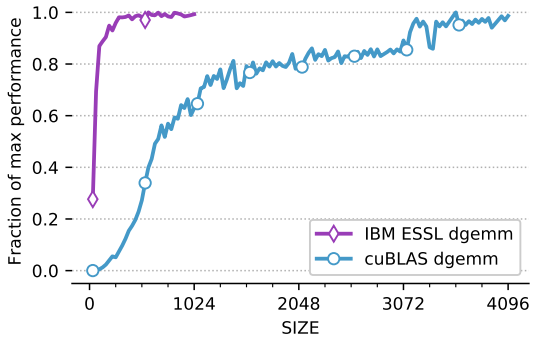


Figure 6.2: Performance of square `dgemm`, as fraction of maximum single-core ESSL performance (23.6 gigaFLOP/s) and cuBLAS performance (4560 gigaFLOP/s), respectively.

- 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 6.2, for tile sizes ≥ 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 ≥ 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 Figures 6.3 and 6.4, 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 $(192, 256, \ldots)$, which occur at multiples of 64, 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.

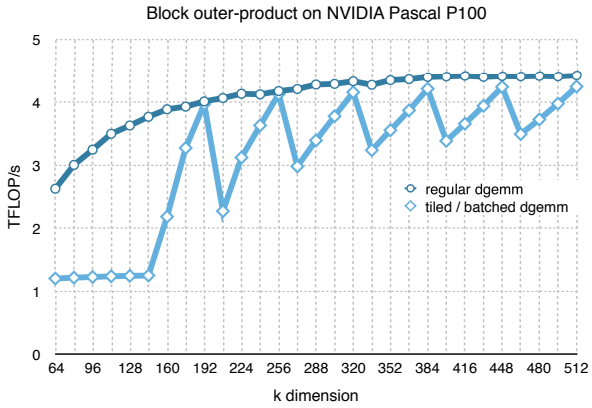


Figure 6.3: NVIDIA Pascal: block outer-product `dgemm`, $C = C - AB$, where C is $40{,}000 \times 40{,}000$, A is $40{,}000 \times k$, B is $k \times 40{,}000$.

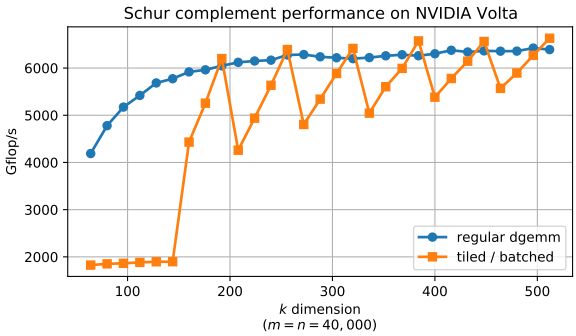


Figure 6.4: NVIDIA Volta: block outer-product `dgemm`, $C = C - AB$, where C is $40{,}000 \times 40{,}000$, A is $40{,}000 \times k$, B is $k \times 40{,}000$.

CHAPTER 7

Message Passing Communication

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 7.1 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 119 and 141). 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.

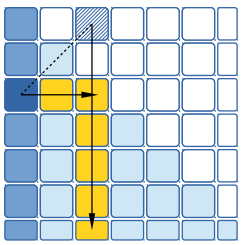


Figure 7.1: Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix.

CHAPTER 8

MOSI Coherency Protocol

8.1 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, 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 much as possible, from thinking about tile state management (i.e., tile state management should be implicit).

8.1.1 Tile States

```
<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 **Shared**or **I**; instance may be purged unless on hold.

Invalid (I): tile's data is obsolete, other instances may be **Modified**, **Shared**, 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 OnHold state is deprecated. It disables the tile release mechanism, which is deprecated.*

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 a *value* being a `tileNode` containing pointers to tiles on the host and GPU devices. The MOSI state is stored in each tile itself, which allows tile routines to verify the correct MOSI status. For instance, `tile::gemm` can verify that A and B are at least **Shared**(readable), and C is **Modified**(read/write).

Two instances of the same tile can be in any of (**Invalid**, **Shared**), (**Invalid**, **Modified**), (**Invalid**, **Invalid**), or (**Shared**, **Shared**), as illustrated in Table 8.1. 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.

	M	S	I
M	X	X	✓
S	X	✓	✓
I	✓	✓	✓

Table 8.1: Valid state combinations of two instances of same tile.

8.1.2 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:

```
1  class BaseMatrix {
2      ...
3
4      // Returns tile(i, j)'s state on device (defaults to host).
5      MOSI tileState(int64_t i, int64_t j, int device=host_num_);
6
7      // Returns whether tile(i, j) is on hold on device (defaults to host).
8      bool tileOnHold(int64_t i, int64_t j, int device=host_num_);
9
10     // Gets tile(i, j) for reading on device.
11     // Will copy-in the tile if it does not exist or its state is Invalid.
12     // Sets tile state to Shared if copied-in.
13     // Updates source tile's state to shared if copied-in.
14     void tileGetForReading(int64_t i, int64_t j, int device=host_num_);
15
16     // Gets all local tiles for reading on device.
17     void tileGetAllForReading(int device=host_num_);
18
19     // Gets all local tiles for reading on corresponding devices.
20     void tileGetAllForReadingOnDevices();
21
22     // Gets tile(i, j) for writing on device.
23     // Sets state to Modified.
24     // Will copy tile in if not exists or state is Invalid.
25     // Other instances will be invalidated.
26     void tileGetForWriting(int64_t i, int64_t j, int device=host_num_);
27
```

```
28      // Gets all local tiles for writing on device.
29      void tileGetAllForWriting(int device=host_num_);
30
31      // Gets all local tiles for writing on corresponding devices.
32      void tileGetAllForWritingOnDevices();
33
34      // Marks tile(i, j) as Modified on device.
35      // Other instances will be invalidated.
36      // Unless permissive, asserts if other instances are in Modified state.
37      void tileModified(int64_t i, int64_t j, int device=host_num_, bool permissive=false);
38
39      // Gets tile(i, j) on device and marks it as OnHold.
40      // Will copy tile in if it does not exist or its state is Invalid.
41      // Updates the source tile's state to Shared if copied-in.
42      void tileGetAndHold(int64_t i, int64_t j, int device=host_num_);
43
44      // Gets all local tiles on device and marks them as OnHold.
45      void tileGetAndHoldAll(int device=host_num_);
46
47      // Gets all local tiles on corresponding devices and marks them as OnHold.
48      void tileGetAndHoldAllOnDevices();
49
50      // Unsets tile(i, j)'s hold on device
51      void tileUnsetHold(int64_t i, int64_t j, int device=host_num_);
52
53      // Deletes the tile(i, j)'s instance on device if it is a workspace tile
54      // that is not modified and no hold is set on it.
55      void tileRelease(int64_t i, int64_t j, int device=host_num_);
56
57      /// Updates the origin instance of tile(i, j) if not MOSI::Shared
58      void tileUpdateOrigin(int64_t i, int64_t j);
59
60      /// Updates all origin instances of tiles if not MOSI::Shared
61      void tileUpdateAllOrigin();
62
63      // Debugging routine:
64      // Check state is coherent for all matrix tile instances
65      void checkTileStates();
66
67      ...
68  }
```

8.1.3 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. SLATE searches for the first **Modified**or **Shared**instance, searching devices first, then the host. This ordering ensures it will prefer device-to-device copies over host-to-device copies. However, it does not yet search for the closest instance based on the bus topology (PCIe, NVlink, Infinity Fabric, etc.). Typically we run SLATE with one MPI rank per GPU or one MPI rank per socket, which tends to lessen concerns about NUMA access between GPUs.

8.1.4 State diagrams

Tile instances may change state following the operations that read or update them. Diagrams in Figures 8.1 and 8.2 illustrate the state transitions that each routine causes, while Figure 8.3 illustrates the same state transitions from the perspective of tiles.

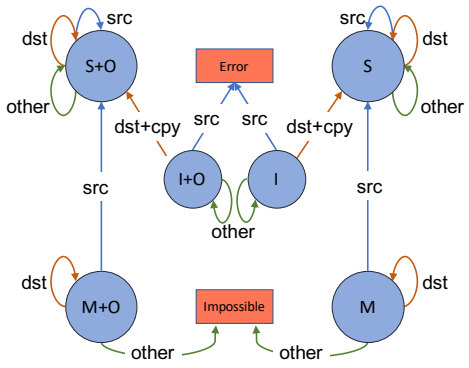
8.2 Developer hints

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.

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 the `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.

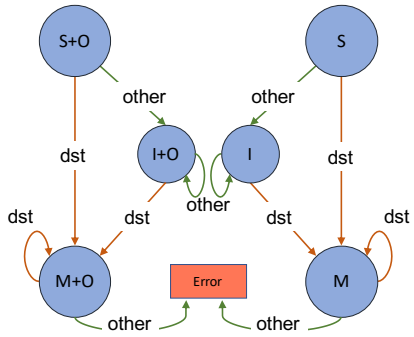
Modified tiles: A tile instance that is acquired by `tileGetForWriting()` is marked `Modified`. However, a newly inserted tile instance may get updated without using the `slate::internal` routines, for example, by issuing `lapack` calls on them, or by direct editing. In addition, tiles acquired for reading (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). *This use is deprecated.*



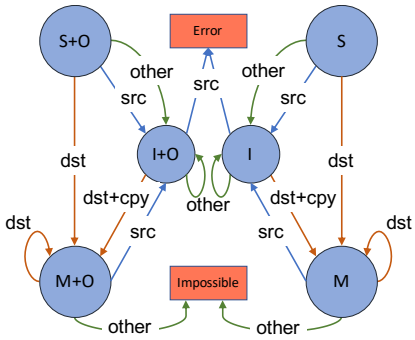
`tileGetForReading()`

- `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.



`tileModified()`

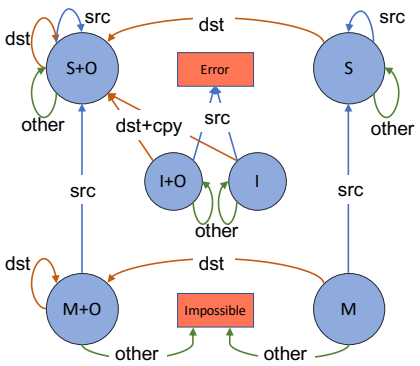
- `dst`: the tile instance being marked `Modified`.
- `other`: all other instances of the same tile.
- An instance, other than dst, that is in a `Modified` state will issue an error unless the `permissive` flag is true.



`tileGetForWriting()`

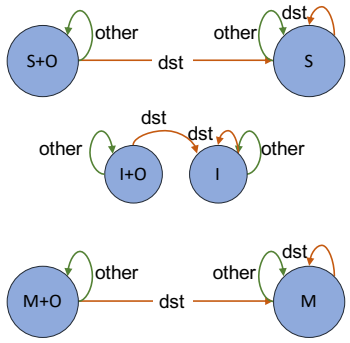
- Is a `tileGetForReading()` followed by a `tileModified()`.

Figure 8.1: 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+cpy denotes a copy of data is carried to update instance T.



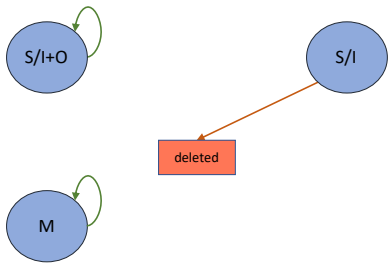
`tileGetAndHold()`

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



`tileUnsetHold()`

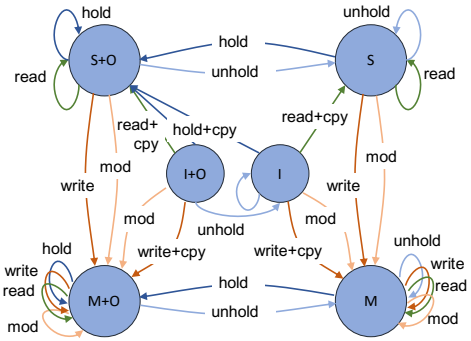
- Removes the hold on `dst` instance; other instances are not affected.



`tileRelease()`

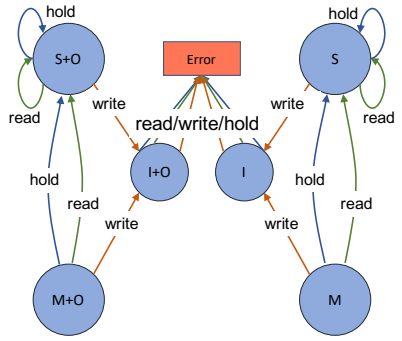
- Deletes a tile instance if not in `Modified` state and no hold is set on it.

Figure 8.2: 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.



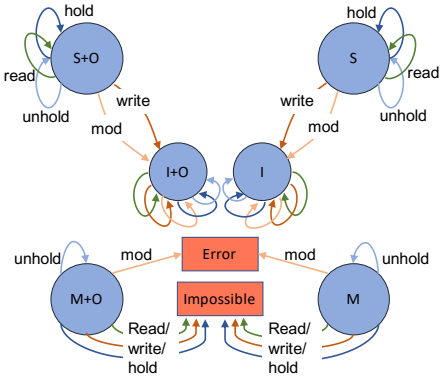
Destination tile

- 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 8.3: Tile state transitions under MOSI API from a tile perspective. Circled 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.

CHAPTER 9

Column Major and Row Major Layout

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, and is incompatible with the traditional BLAS.

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.

9.1 Layout representation and API

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

```
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 9.1). Similarly, the matrix layout (defaulting to `ColMajor`) is stored at the `BaseMatrix::layout_` member variable (Algorithm 9.2). A MOSI operation (`tileGetForReading()`, `tileGetForWriting()`, etc...) specifies the layout of the destination tile instance using the following enum:

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

Algorithms 9.1 to 9.3 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.

9.2 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.

Inside `tileGet***()`, the logic to copy and transpose is implemented within the `BaseMatrix::tileCopyDataLayout()` routine, which is a private function called only from the `tileGet()` routine. To avoid extra memory allocations, this routine checks if one if the tiles has a back buffer that can be used as workspace. Additionally, for performance purposes, out of place transposition is always done on device.

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.

Algorithm 9.1 Tile's layout member functions and member variables.

```
class Tile
{
    ...
    // Returns the current layout
    Layout layout() const;

    // Sets the current layout, stride, and front buffer
    void setLayout(Layout in_layout);

    // Returns the current layout of user-provided buffer
    Layout userLayout() const { return user_layout_; }

    // Returns whether the front memory buffer is contiguous
    bool isContiguous() const;

    // Returns whether the user's memory buffer is contiguous
    bool isUserContiguous() const;

    // 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();

    // Returns whether this tile has an extended buffer
    bool extended() const;

    // Returns the pointer to the user allocated buffer
    scalar_t* userData();

    // Returns the pointer to the extended buffer
    scalar_t* extData();

    // Returns the pointer to the back buffer
    scalar_t* layoutBackData();

    // Returns the stride of the back buffer
    int64_t layoutBackStride() const;

    // Convert layout of this tile
    // work_data must be provided if the tile is rectangular and unextended
    // queue must be provided if conversion is to happen on device
    void layoutConvert( scalar_t* work_data = nullptr );
    void layoutConvert( blas::Queue& queue, bool async = false );
    void layoutConvert( scalar_t* work_data, blas::Queue& queue, bool async = false );
protected:
    int64_t stride_;
    int64_t user_stride_; // Stores user-provided-memory's stride

    scalar_t* data_;
    scalar_t* user_data_; // Points 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_;
    Layout user_layout_; // Stores user-provided-memory's layout
...
};
```

Algorithm 9.2 Matrix's layout member functions and member variables.

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

    // Returns the layout of tile(i, j, device/host)
    Layout tileLayout(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 9.3 Matrix's layout member functions and member variables.

```
class MatrixStorage
{
...
public:

    void tileMakeTransposable(Tile<scalar_t>* tile);
    void tileLayoutReset(Tile<scalar_t>* tile);
...
};
```

Keep in mind that, as a tile can have instances in 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 become MOSI::Modified by changing its layout since the data is still the same, only represented differently in memory.

9.2.1 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 instantiates a `MatrixStorage` object, which acts as an interface to the `Memory` object. Ideally, a large pool of memory is allocated at the matrix construction through the `Memory` object. Shallow copies of the matrix share 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 into the same memory, the tile's memory needs to be contiguous or square. Tiles whose memory is strided and are rectangular cannot be transposed into the same memory. To facilitate a seamless layout conversion of all tiles, a mechanism of extending the tiles memory is used. 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 9.1. 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. The logic to manage the buffers and stride is contained in the `Tile::setLayout()` routine. In order to ensure that tiles remain in consistent states, this is the only routine that should change the front buffer and stride.

Bibliography

[1] Mark Gates, Ali Charara, Jakub Kurzak, Asim YarKhan, Mohammed Al Farhan, Dalal Sukkari, and Jack Dongarra. SLATE users' guide, SWAN no. 10. Technical Report ICL-UT-19-01, Innovative Computing Laboratory, University of Tennessee, July 2020. URL https://www.icl.utk.edu/publications/swan-010. revision 07-2020.

[2] Mark Gates, Ali Charara, Asim YarKhan, Dalal Sukkari, Mohammed Al Farhan, and Jack Dongarra. SLATE working note 14 performance tuning slate. Technical Report ICL-UT-20-01, Innovative Computing Laboratory, University of Tennessee, December 2019. URL https://www.icl.utk.edu/publications/swan-014. revision 12-2019.

[3] H. Carter Edwards, Bryce Adelstein Lelbach, Daniel Sunderland, David Hollman, Christian Trott, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, and Daniel Sunderland. *P0009r7 : mdspan: A Non-Owning Multidimensional Array Reference*. ISO, 2018. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0009r7.html.

[4] Fred Gustavson, André Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, 1541:195–206, 1998. doi:10.1007/BFb0095337.

[5] Fred G Gustavson, Jerzy Waśniewski, Jack J Dongarra, and Julien Langou. Rectangular full packed format for cholesky's algorithm: factorization, solution, and inversion. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):18, 2010. doi:10.1145/1731022.1731028.

[6] *Introducing the new Packed APIs for GEMM*. Intel Corp., 2016. URL https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm.

[7] Fred Gustavson, Lars Karlsson, and Bo Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3): 17, 2012. doi:10.1145/2168773.2168775.

[8] Stefan Kurz, Oliver Rain, and Sergej Rjasanow. The adaptive cross-approximation technique for the 3d boundary-element method. *IEEE Transactions on Magnetics*, 38(2):421–424, 2002. doi:10.1109/20.996112.

[9] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šíšístek, David Stevens, Mawussi Zounon, and Samuel d. Relton. Plasma: Parallel linear algebra software for multicore using openmp. *ACM Transactions on Mathematical Software (TOMS)*, 45:16:1–16:35, 2019. doi:10.1145/3264491.

[10] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. C++ api for blas and lapack. Technical Report ICL-UT-17-03, SLATE Working Note 2, Innovative Computing Laboratory, University of Tennessee, 06-2017 2017. URL https://www.icl.utk.edu/publications/swan-002.

[11] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 21(4):457–466, 2007. doi:10.1177/1094342007084026.

[12] Erin Carson and Nicholas J Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018. doi:10.1137/17M1140819.

[13] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 47. IEEE Press, 2018. doi:10.1109/SC.2018.00050.

[14] Yozo Hida, Xiaoye S Li, and David H Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pages 155–162. IEEE, 2001. doi:10.1109/ARITH.2001.930115.

[15] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. IEEE, 2015. doi:10.1109/IPDPS.2015.56.

[16] Jakub Kurzak, Piotr Luszczek, Ichitaro Yamazaki, Yves Robert, and Jack Dongarra. Design and implementation of the PULSAR programming system for large scale computing. *Supercomputing Frontiers and Innovations*, 4(1):4–26, 2017. doi:http://dx.doi.org/10.14529/jsfi170101.