Vertical integration of the linear and multilinear software stack

Devin Matthews

Robert van de Geijn

(building on decades of work involving dozens of collaborators)

Traditional software layering

Basic layering

- BLAS as a black box with an immutable interface
- LAPACK built on. BLAS +utility routines (some BLAS-like). SMP parallelization within the BLAS.
- ScaLAPACK built on BLAS, LAPACK, MPI, …

Variations on a theme

- DAG scheduling (MAGMA, etc.)
- …

This has served the community for the past 30+ years.

Decades of alternative R&D

- Efficient collective communication (early 1990s)
- PLAPACK (1996)
- The FLAME methodology (1999)
- libflame (2005)
- SuperMatrix (2009)
- BLAS-like Library Instantiation Software (BLIS) (2012)
- Elemental (2013)
- ROTE (2015)
- TBLIS (2018)

What opportunities does BLIS 2.0 facilitate?

Translating Cholesky into code

Algorithm: $[A] := \text{CHOL\_BLK}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ do

Determine block size b

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right)$$

Variant 1:	Variant 2:	Variant 3:
$A_{10} := A_{10} L_{00}^{-H}$		
$A_{11} := A_{11} - A_{10} A_{10}^{H}$	$A_{11} := A_{11} - A_{10} A_{10}^{H}$	
$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$
	$A_{21} := A_{21} - A_{20} A_{10}^{H}$	
	$A_{21} := A_{21} L_{11}^{-H}$	$A_{21} := A_{21} L_{11}^{-H}$
		$A_{22} := A_{22} - A_{21} A_{21}^{H}$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

libflame

Algorithm: $[A] := \text{CHOL\_BLK}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ do

Determine block size b

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right)$$

Variant 1:	Variant 2:	Variant 3:
$A_{10} := A_{10} L_{00}^{-H}$		
$A_{11} := A_{11} - A_{10} A_{10}^{H}$	$A_{11} := A_{11} - A_{10} A_{10}^{H}$	
$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$
	$A_{21} := A_{21} - A_{20} A_{10}^{H}$	
	$A_{21} := A_{21} L_{11}^{-H}$	$A_{21} := A_{21} L_{11}^{-H}$
		$A_{22} := A_{22} - A_{21} A_{21}^{H}$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
FLA_Part_2x2( A,    &ATL, &ATR,
                    &ABL, &ABR,     0, 0, FLA_TL );

while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){

  b = FLA_Determine_blocksize( ABR, FLA_BR, FLA_Cntl_blocksize( cntl ) );

  FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                       /* ************* */   /* ******************** */
                                              &A10, /**/ &A11, &A12,
                         ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
                         b, b, FLA_BR );

  /*-------------------------------------------------------------*/

  // A11 = chol( A11 )
  r_val = FLA_Chol_internal( FLA_LOWER_TRIANGULAR, A11,
                             FLA_Cntl_sub_chol( cntl ) );

  if ( r_val != FLA_SUCCESS )
    return ( FLA_Obj_length( A00 ) + r_val );

  // A21 = A21 * inv( tril( A11 )' )
  FLA_Trsm_internal( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                     FLA_CONJ_TRANSPOSE, FLA_NONUNIT_DIAG,
                     FLA_ONE, A11, A21,
                     FLA_Cntl_sub_trsm( cntl ) );

  // A22 = A22 - A21 * A21'
  FLA_Herk_internal( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                     FLA_MINUS_ONE, A21, FLA_ONE, A22,
                     FLA_Cntl_sub_herk( cntl ) );

  /*-------------------------------------------------------------*/

  FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                   A10, A11, /**/ A12,
                          /* ************** */  /* ****************** */
                            &ABL, /**/ &ABR,       A20, A21, /**/ A22,
                            FLA_TL );
}
```

libflame

Algorithm: $A := \text{CHOL}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ do

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{array}\right)$$

Variant 1:	Variant 2:	Variant 3:
$a_{10}^T := a_{10}^T L_{00}^{-H}$	$\alpha_{11} := \alpha_{11} - a_{10}^T (a_{10}^T)^H$	
$\alpha_{11} := \alpha_{11} - a_{10}^T (a_{10}^T)^H$	$\alpha_{11} := \sqrt{\alpha_{11}}$	$\alpha_{11} := \sqrt{\alpha_{11}}$
$\alpha_{11} := \sqrt{\alpha_{11}}$	$a_{21} := a_{21} - A_{20} (a_{10}^T)^H$	
	$a_{21} := a_{21} / \alpha_{11}$	$a_{21} := a_{21} / \alpha_{11}$
		$A_{22} := A_{22} - a_{21} a_{21}^H$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & a_{01} & A_{02} \\ a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

```
for ( i = 0; i < mn_A; ++i )
{
  double*   alpha11   = buff_A + (i  )*cs_A + (i  )*rs_A;
  double*   a21       = buff_A + (i  )*cs_A + (i+1)*rs_A;
  double*   A22       = buff_A + (i+1)*cs_A + (i+1)*rs_A;

  int       mn_ahead  = mn_A - i - 1;
  int       mn_behind = i;

  /*------------------------------------------------------------*/

  // r_val = FLA_Sqrt( alpha11 );
  // if ( r_val != FLA_SUCCESS )
  //   return ( FLA_Obj_length( A00 ) + 1 );
  bl1_dsqrte( alpha11, &e_val );
  if ( e_val != FLA_SUCCESS ) return mn_behind;

  // FLA_Inv_scal_external( alpha11, a21 );
  bl1_dinvscalv( BLIS1_NO_CONJUGATE,
                 mn_ahead,
                 alpha11,
                 a21, rs_A );

  // FLA_Her_external( FLA_LOWER_TRIANGULAR, FLA_MINUS_ONE, a21, A22 );
  bl1_dsyr( BLIS1_LOWER_TRIANGULAR,
            mn_ahead,
            buff_m1,
            a21, rs_A,
            A22, rs_A, cs_A );

  /*------------------------------------------------------------*/

}
```

Elemental

Algorithm: $[A] := \text{CHOL\_BLK}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ do

Determine block size b

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right)$$

Variant 1:	Variant 2:	Variant 3:
$A_{10} := A_{10} L_{00}^{-H}$		
$A_{11} := A_{11} - A_{10} A_{10}^{H}$	$A_{11} := A_{11} - A_{10} A_{10}^{H}$	
$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$	$A_{11} := \text{Chol\_unb}(A_{11})$
	$A_{21} := A_{21} - A_{20} A_{10}^{H}$	
	$A_{21} := A_{21} L_{11}^{-H}$	$A_{21} := A_{21} L_{11}^{-H}$
		$A_{22} := A_{22} - A_{21} A_{21}^{H}$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
PartitionDownDiagonal( A, ATL, ATR,
                           ABL, ABR, 0 );
while( ABR.Height() > 0 )
{
    RepartitionDownDiagonal( ATL, /**/ ATR,  A00, /**/ A01, A02,
                             /*************/ /******************/
                                   /**/       A10, /**/ A11, A12,
                              ABL, /**/ ABR,  A20, /**/ A21, A22 );

    A12_Star_MC.AlignWith( A22 );
    A12_Star_MR.AlignWith( A22 );
    A12_Star_VR.AlignWith( A22 );
    //------------------------------------------------------------//
    A11_Star_Star = A11;
    lapack::internal::LocalChol( Upper, A11_Star_Star );
    A11 = A11_Star_Star;

    A12_Star_VR = A12;
    blas::internal::LocalTrsm
      ( Left, Upper, ConjugateTranspose, NonUnit,
        (T)1, A11_Star_Star, A12_Star_VR );

    A12_Star_MC = A12_Star_VR;
    A12_Star_MR = A12_Star_VR;
    blas::internal::LocalTriangularRankK
      ( Upper, ConjugateTranspose,
        (T)-1, A12_Star_MC, A12_Star_MR, (T)1, A22 );
    A12 = A12_Star_MR;
    //------------------------------------------------------------//
    A12_Star_MC.FreeAlignments();
    A12_Star_MR.FreeAlignments();
    A12_Star_VR.FreeAlignments();

    SlidePartitionDownDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
                                     /**/       A10, A11, /**/ A12,
                               /*************/ /******************/
                               ABL, /**/ ABR,  A20, A21, /**/ A22 );
}
```

Some Observations

- These APIs (instantiations of the FLAME algorithm) have a lot of consonance across scales (unblocked [scalar/vector], blocked [matrix], distributed).

Some Observations

- These APIs (instantiations of the FLAME algorithm) have a lot of consonance across scales (unblocked [scalar/vector], blocked [matrix], distributed).
- But there are important differences, e.g.:
  - Unblocked code uses different BLAS functions, and uses explicit indices and pointers for performance. In general, we see a distinct trend towards more expressive but “heavier-weight” languages/code with increasing scale.
  - Distributed code has to deal with “additional” concerns of data layout. It also explicitly calls into a non-distributed layer below it.

Some Questions

- These APIs still adhere to a “black-box” interface layer at each level. How can the algorithm be improved by “reaching through” the layers?
- Each layer has many choices (block size, variant, recursion depth, etc.). How do we collect these choices at each layer together into an algorithm?
- How can we extend concepts like matrix partitioning and data distributions across scales?
- How do we extend concepts like matrix partitioning and data distributions to multi-dimensional arrays (tensors)?

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

Why ranges?

- Ranges are *expressive*: they convey precisely the information needed to partition vectors, matrices, and tensors, and nothing else. They are *generalized indices*.

Why ranges?

- Ranges are *expressive*: they convey precisely the information needed to partition vectors, matrices, and tensors, and nothing else. They are *generalized indices*.
- Ranges are *extensible*: any number of dimensions can be partitioned independently.

Why ranges?

- Ranges are *expressive*: they convey precisely the information needed to partition vectors, matrices, and tensors, and nothing else. They are *generalized indices*.
- Ranges are *extensible*: any number of dimensions can be partitioned independently.
- Ranges are *efficient*: a range is very lightweight, only creates a sub-matrix when used to index a parent matrix, and can easily be fused to refer to groups of sub-matrices. Loop algorithms on ranges are compiler-friendly and optimized extremely well.

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

Where did unblocked go?

- Ranges are typed: a range of 1 column/row is a scalar.

Where did unblocked go?

- Ranges are typed: a range of 1 column/row is a scalar.
- Sub-matrices are types:
  - A[R1][C1] is a true sub-matrix
  - A[R1][c1] (c1 == scalar) is a sub-vector
  - A[r1][c1] is a scalar

Where did unblocked go?

- Ranges are typed: a range of 1 column/row is a scalar.
- Sub-matrices are types:
  - A[R1][C1] is a true sub-matrix
  - A[R1][c1] (c1 == scalar) is a sub-vector
  - A[r1][c1] is a scalar
- Function overload/duck typing/templating changes behavior based on operand types.

Where did unblocked go?

- Ranges are typed: a range of 1 column/row is a scalar.
- Sub-matrices are types:
  - A[R1][C1] is a true sub-matrix
  - A[R1][c1] (c1 == scalar) is a sub-vector
  - A[r1][c1] is a scalar
- Function overload/duck typing/templating changes behavior based on operand types.
- Different operations (based on type) could traverse different branches of the control tree.

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

Where's communication (and other stuff)?

- Relationships between matrices (indexing with the same range) imply various conditions:
  - Compatibility (layout, locality, alignment, etc.)
  - Connectedness ("these things go together", a transitive property e.g. important for fusion)

Where's communication (and other stuff)?

- Relationships between matrices (indexing with the same range) imply various conditions:
  - Compatibility (layout, locality, alignment, etc.)
  - Connectedness ("these things go together", a transitive property e.g. important for fusion)
- How these conditions translate to the algorithm depends on the logic:
  - E.g., communication *could be* implied
  - Algorithmic flexiblity: choice of algorithm encoded in the control tree

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            trsm(BLIS_RIGHT, BLIS_LOWER, BLIS_CONJ_TRANSPOSE, BLIS_NONUNIT_DIAG, 1, A[R1][R1], A[R2][R1]);

            // A22 = A22 - A21 * A21'
            herk(BLIS_LOWER,  -1, A[R2][R1], 1, A[R2][R2]);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

Breaking through layers

- The control tree ties togethers all the layers of abstraction.

Breaking through layers

- The control tree ties togethers all the layers of abstraction.
- Algorithmic choices are made “up front”: the control tree can directly call into kernels, skip packing/blocking/error checking, etc. if the algorithm calls for it.

Breaking through layers

- The control tree ties togethers all the layers of abstraction.
- Algorithmic choices are made “up front”: the control tree can directly call into kernels, skip packing/blocking/error checking, etc. if the algorithm calls for it.
- Can use well-defined interfaces or kernels, but not necessarily needed. JIT? Sure.

```
template <typename Type, int Variant, bool Blocked>
dim_t cholesky_var3_l_impl(const marray_view<Type, 2>& A, const control_tree* cntl)
{
    assert(A.length(0) == A.length(1));

    auto [T, B] = partition_columns(A, FORWARD);

    while (B)
    {
        auto [R0, R1, R2] = repartition<Blocked ? DYNAMIC : 1>(T, B, control->bsz, FORWARD);

            // A11 = chol( A11 )
            auto r_val = cntl->cholesky(A[R1][R1], cntl);
            if (r_val != BLIS_SUCCESS) return T.size() + r_val;

            // A21 = A21 * inv( tril( A11 )' )
            cntl->trsm(1, A[R1][R1], A[R2][R1], cntl);

            // A22 = A22 - A21 * A21'
            cntl->herk(-1, A[R2][R1], 1, A[R2][R2], cntl);

        std::tie(T, B) = continue_with(R0, R1, R2, FORWARD);
    }

    return BLIS_SUCCESS;
}
```

Breaking through layers

- The control tree ties togethers all the layers of abstraction.
- Algorithmic choices are made “up front”: the control tree can directly call into kernels, skip packing/blocking/error checking, etc. if the algorithm calls for it.
- Can use well-defined interfaces or kernels, but not necessarily needed. JIT? Sure.
- At a high level, code isn’t “the algorithm”. The control tree abstraction tells us what to do; logic, expert knowledge, and hardware properties (modeling, fine tuning) tell us what to put in the control tree.