Representing Linear Algebra Algorithms in Code: The FLAME API

Robert A. van de Geijn
The University of Texas at Austin

The Formal Linear Algebra Methods Environment (FLAME) encompasses a methodology for deriving provably correct algorithms for dense linear algebra operations as well as an approach to representing (coding) the resulting algorithms. Central to the philosophy underlying FLAME are the observations that it is at a high level of abstraction that one best reasons about the correctness of algorithms, that therefore algorithms should themselves be expressed at a high level of abstraction, and that codes that implement such algorithms should themselves use an Application Programming Interface (API) that captures this high level of abstraction. A key observation is that in reasoning about algorithms intricate indexing is typically avoided and it is with the introduction of intricate indexing that programming errors are often encountered and confidence in code is deminished. Thus a carefully designed API avoids explicit indexing whenever possible.

In this paper, we demonstrate how to construct one such API for coding linear algebra libraries in the C programming language. The emphasis is on properties that such APIs should embrace rather than the details of the particular API. Indeed, it should be obvious how similar interfaces can be defined for other languages, including Fortran, C++, and MATLAB M-script.

Categories and Subject Descriptors: []: —

General Terms:

Additional Key Words and Phrases:

1. INTRODUCTION

This paper is the fourth in a series that illustrate to the high-performance linear algebra library community the benefits of the formal derivation of algorithms.

—The first paper [Gunnels et al. 2001] gave a broad outline of the approach, introducing the concept of formal derivation and its application to dense linear algebra algorithms. In that paper we also showed that by introducing an Application Programming Interface (API) for coding the provably correct algorithms, claims about the correctness of the algorithms allow claims about the correctness of the implementation to be made. Finally, we showed that excellent performance can be attained. The primary vehicle for illustrating the techniques in that paper was the LU factorization.

Author's address: Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

—We showed that the method applies to more complex operations in the second paper [Quintana-Ortí and van de Geijn]. In that paper we showed how a large number of new high-performance algorithms for the solution of the triangular Sylvester equation can be derived using the methodology.

—The third paper focused primarily on the derivation method [Bientinesi et al.]. In particular, that paper contains a step-by-step "recipe" that novice and veteran alike can use to rapidly derive correct algorithms.

In a number of less-detailed workshop papers we also presented some of the above mentioned material [Gunnels and van de Geijn 2001b; Bientinesi et al. 2002].

This paper makes the following contributions:

—While in the previous papers we alluded at an API that allows code to reflect algorithms that have been derived to be correct, in this paper we explicitly give this API. By allowing the code to closely mirror the algorithms as they are naturally presented as a result of the derivation process, the proven correctness of the algorithms provides a high degree of confidence in the correctness of the code.

—We show that intricate indexing that is avoided when reasoning about the correctness of algorithms can also be avoided when coding.

—We show that by discarding conventional wisdom related to the order in which input and output parameters should appear in the calling sequence of a routine, formating can be used to further allow the code to mirror the algorithm.

We purposely emphasize how to capture the high level of abstraction used in presenting the algorithms rather than the particular details of the interface itself. This will allow one to define similar interfaces for other languages, such as Fortran, C++, and Matlab M-script [Moler et al. 1987]. Indeed, we have also defined similar interfaces for these languages. **Nonetheless, while this paper illustrates one possible interface, the FLAME interface, as presented, has been an invaluable teaching tool for undergraduate and graduate courses that include the topic of high-performance computing [Gunnels and van de Geijn 2001a; Bientinesi and van de Geijn 2002].**

Those familiar with our Parallel Linear Algebra Package (PLAPACK) [Alpatov et al. 1997; van de Geijn 1997] will recognize that many of the same observations were incorporated into the API for that package a half decade ago. In particular, the PLAPACK API avoids indexing much like FLAME does. However, the PLAPACK approach is extended in FLAME to capitalize on the very rigid structure that algorithms exhibit when developed using our derivation methodology. The latest release of the PLAPACK API itself now also incorporates those extensions so that a parallel PLAPACK code is almost identical to a sequential FLAME code.

This paper is organized as follows: In Section 2, we present an example of how we represent a broad class of linear algebra algorithms in our other papers. The most important components of the API are presented in Section 3. Performance related issues are discussed in Section 4 followed by a few concluding remarks in Section 5.

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$

where B_T has 0 rows and L_{TL} is 0×0

while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**

Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline\hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where b^T is a row and λ_{11} is a scalar

$b_1^T := b_1^T - l_{10}^T B_0$

$b_1^T := L_{11}^{-1} b_1^T$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline\hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

enddo

Fig. 1. Unblocked algorithm for the TRSM example.

2. A TYPICAL DENSE LINEAR ALGEBRA ALGORITHM

In [Bientinesi et al.] we introduced a methodology for the systematic derivation of provably correct algorithms for dense linear algebra algorithms. It is highly recommended that the reader become familiar with that paper before proceeding with the remainder of this paper. This section gives the minimal background in an attempt to make the present paper self-contained.

The algorithms that result from the derivation process present themselves in a very rigid format. We illustrate this format in Fig. 1 which gives an (unblocked) algorithm for the computation of $B := L^{-1}B$, where B is an $m \times n$ matrix and L is an $m \times m$ lower triangular matrix. This operation is often referred to as triangular solve with multiple right-hand sides (TRSM). Notice that the presented algorithm was derived in [Bientinesi et al.].

At the top of the loop-body, it is assumed that different regions of the operands L and B have been used and/or updated in a consistent fashion. These regions are initialized by

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$

where B_T has 0 rows and L_{TL} is 0×0

Here T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

NOTE 1. *Of particular importance in the algorithm are the the single and double lines used to partition and repartition the matrices. Double lines are used to demark regions in the matrices that have been used and/or updated in a consistent fashion. Another way of interpreting double lines is that they keep track of how far into the matrices the computation has progressed.*

Let $\hat{B}$ equal the original contents of B and assume that $\hat{B}$ is partitioned like B.

At the top of the loop it will be assumed that B_B contains the original contents $\hat{B}_B$ while B_T has been updated with the contents $L_{TL}^{-1}\hat{B}_T$. As part of the loop, the boundaries between these regions are moved one row and/or column at a time so that progress towards completion is made. This is accomplished by

Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where b^T is a row and λ_{11} is a scalar

$\vdots$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

NOTE 2. *Single lines are introduced in addition to the double lines to demark regions that are to be updated and/or used in the next step of the algorithm. Upon completion of the update, the regions defined by the double lines are updated to reflect that the computation has moved forward.*

NOTE 3. *We adopt the often-used convention where matrices, vectors, and scalars are denoted by upper-case, lower-case, and greek letters, respectively.*

NOTE 4. *A row vector is indicated by adding a transpose to a vector, e.g.* b_1^T *and* l_{10}^T.

The repartitioning exposes submatrices that must be updated before the boundaries can be moved. That update is given by

$$\begin{array}{l} b_1^T := b_1^T - l_{10}^T B_0 \\ b_1^T := L_{11}^{-1} b_1^T \end{array}$$

Finally, the desired result has been computed when L_{TL} encompasses all of L so that the loop continues until $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ becomes *false*. Here $m(X)$ returns the row dimension of X.

NOTE 5. *We would like to claim that the algorithm in Fig. 1 captures how one might naturally explain a particular algorithmic variant for computing the solution to a triangular linear system with multiple right-hand sides.*

NOTE 6. *The presented algorithm only requires one to use indices from the sets* $\{T, B\}$, $\{L, R\}$, *and* $\{0, 1, 2\}$.

For performance reasons it is often necessary to formulate the algorithm as a *blocked* algorithm as illustrated in Fig. 2. The performance benefit comes from the fact that the algorithm is rich in matrix multiplication which allows processors with multi-level memories to achieve high performance [Dongarra et al. 1991; Anderson et al. 1992; Gunnels et al. 2001; Dongarra et al. 1990].

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$

where B_T has 0 rows and L_{TL} is 0×0

while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

where $\mathrm{m}(B_1) = b$ and $\mathrm{n}(L_{11}) = b$

$B_1 := B_1 - L_{10} B_0$

$B_1 := L_{11}^{-1} B_1$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

enddo

Fig. 2. Blocked algorithm for the TRSM example.

```
[ m, n ] = size( B );
for i=1:nb:m
  b = min( nb, m-b+1 );
  B( i:i+b-1, : ) = B( i:i+b-1, : ) - ...
     L( i:i+b-1, 1:i-1 ) * B( 1:i-1, : );
  B( i:i+b-1, : ) = inv( L( i:i+b-1, i:i+b-1 ) ) * B( i:i+b-1, : )
end
```

Fig. 3. Matlab implementation of blocked TRSM algorithm in Fig. 2

NOTE 7. *The algorithm in Fig. 2 is implemented by the Matlab code given in Fig. 3. We would like to claim that the introduction of indices to explicitly indicate the regions involved in the update complicates readability and reduces confidence in the correctness of the Matlab implementation. Indeed, an explanation of the code will inherently require the drawing of a picture that captures the repartitioned matrices in Fig. 2. In other words, someone experienced with Matlab can easily translate the algorithm in Fig. 2 into the implementation in Fig. 3. The converse is considerably more difficult.*

(We realize that the use of `inv( L( i:i+nb-1, i:i+nb-1 ) )` *can introduce numerical instability and that therefore one in practice would actually code this as the solution of a triangular system with multiple right-hand sides.)*

3. AN INTERFACE FOR CODING LINEAR ALGEBRA ALGORITHMS

In this section we introduce a set of library routines that will allow us to capture in code linear algebra algorithms presented in the format illustrated in the previous section. The idea is that by making the code look like the algorithms in Figs. 1 and 2 the opportunity for the introduction of coding errors is reduced. Readers familiar

with MPI [Gropp et al. 1994; Snir et al. 1996] and/or our own PLAPACK will recognize the programming style, *object-based programming*, as being very similar to that used by those (and other) interfaces.

3.1 Initializing and finalizing FLAME

Before using the FLAME environment one must initialize with a call to

```
void FLA_Init( )
```

Purpose: Initialize FLAME.

If no more FLAME calls are to be made, the environment is exited by calling

```
void FLA_Finalize( )
```

Purpose: Finalize FLAME.

3.2 Linear algebra objects

Notice that the following attributes describe a matrix as it is stored in the memory of a computer:

—the datatype of the entries in the matrix, e.g., `double` or `float`,

—m and n, the row and column dimensions of the matrix,

—the address where the data is stored, and

—the mapping that describes how the two dimensional array is mapped to one dimensional memory.

The following call creates an object (*descriptor*) that describes a matrix and creates space to store the entries in the matrix:

```
void FLA_Obj_create( int datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: Create an object that describes an $m \times n$ matrix and create the associated storage array.

Valid datatype values include

`FLA_INT`, `FLA_DOUBLE`, `FLA_FLOAT`, `FLA_DOUBLE_COMPLEX`, and `FLA_COMPLEX`

for the obvious datatypes that are commonly encountered. The leading dimension of the array that is used to store the matrix is itself determined inside of this call.

NOTE 8. *For simplicity, we chose to limit the storage of matrices to use column-major storage. The leading dimension of a matrix can be thought of as the dimension of the array in which the matrix is embedded (which is often larger than the row-dimension of the matrix) or as the increment (in elements) required to address consecutive elements in a row of the matrix. Column-major storage is chosen to be consistent with Fortran which is often still the choice of language for linear algebra applications.*

Sometimes it will be useful to create a descriptor without storage for the array. This allows a matrix that has already been stored in a conventional two-dimensional array to be attached to an object. The following call creates such a descriptor:

```
void FLA_Obj_create_without_buffer
          ( int datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: Create an object that describes an $m \times n$ matrix without creating the associated storage array.

Once an object has been created without an attached storage array, an existing two-dimensional array can be attached by calling

```
void FLA_Obj_attach_buffer( void *buff, int ldim, FLA_Obj *matrix )
```

Purpose: Attach an existing buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.

FLAME treats vectors as special cases of matrices: an $n \times 1$ matrix or a $1 \times n$ matrix. Thus, to create an object for a vector x of length n either of the following calls will suffice:

```
FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
FLA_Obj_create( FLA_DOUBLE, 1, n, &x );
```

`n` is an integer variable with value n.

Similarly, FLAME treats scalars as a 1×1 matrix. Thus, to create a object for a scalar α the following call is made:

```
FLA_Obj_create( FLA_DOUBLE, 1, 1, &alpha )
```

A number of scalars occur frequently and are therefore predefined by FLAME:

`MINUS_ONE`, `ZERO`, and `ONE`.

Often it is useful to create a matrix that has the same datatype and dimensions as a given matrix. For this we provide the call

```
void FLA_Obj_create_conf_to( int trans, FLA_Obj old,
                                        FLA_Obj *matrix )
```

Purpose: Like `FLA_Obj_create` except that it creates an object with same datatype and dimensions as `old`, transposing if desired.

Valid values for `trans` include

`FLA_NO_TRANSPOSE` and `FLA_TRANSPOSE`.

If `trans` equals `FLA_NO_TRANSPOSE`, the new object has the same dimensions as `old`. Otherwise, it has the same dimensions as the transpose of `old`.

If an object was created with `FLA_Obj_create`, `FLA_Obj_create_without_buffer`, or `FLA_Obj_create_conf_to`, a call to `FLA_Obj_free` is required to ensure that all space associated with the object is properly released:

```
void FLA_Obj_free( FLA_Obj *obj )
```

Purpose: Free all space allocated to store data associated with `obj`.

3.3 Inquiry routines

In order to be able to work with the raw data, a number of inquiry routines can be used to access information about a matrix (or vector or scalar). The datatype and row and column dimensions of the matrix can be extracted by calling

```
int FLA_Obj_datatype( FLA_Obj matrix )
int FLA_Obj_length  ( FLA_Obj matrix )
int FLA_Obj_width   ( FLA_Obj matrix )
```

Purpose: Extract datatype, row, or column dimension of matrix, respectively.

The address of the array that stores the matrix and its leading dimension can be retrieved by calling

```
void *FLA_Obj_buffer( FLA_Obj matrix )
int FLA_Obj_ldim    ( FLA_Obj matrix )
```

Purpose: Extract address and leading dimension of the matrix, respectively.

3.4 A most useful utility routine

Our approach to the implementation of algorithms for linear algebra operations starts with the careful derivation of provably correct algorithms. The stated philosophy is that if the algorithms are correct, and the API allows the algorithms to be coded so that the code reflects the algorithms, then the code will be correct as well.

Nonetheless, we single out one of the more useful routines in the FLAME library, which is particularly helpful for debugging:

```
void FLA_Obj_show( char *string1, FLA_Obj A, char *format,
                   char *string2 )
```

Purpose: Print the contents of A.

In particular, the result of

```
FLA_Obj_show( "A =[", A, "%lf ", "];" );
```

is something like

```
A = [
< entries >
];
```

which can then be fed to Matlab. This becomes useful when checking results against a Matlab implementation of an operation.

3.5 An example: matrix-vector multiplication

We now give an example of how to use the calls introduced so far to write a simple driver routine that calls a routine that performs the matrix-vector multiplication $y = Ax$.

In Fig. 4 we give the driver routine.

—**line 1:** FLAME program files start by including the `FLAME.h` header file.

```
1   #include "FLAME.h"
2
3   main()
4   {
5     FLA_Obj
6       A, x, y;
7     int
8       m, n;
9
10    FLA_Init( );
11
12    printf( "enter matrix dimensions m and n:" );
13    scanf( "%d%d", &m, &n );
14
15    FLA_Obj_create( FLA_DOUBLE, m, n, &A );
16    FLA_Obj_create( FLA_DOUBLE, m, 1, &y );
17    FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
18
19    fill_matrix( A );
20    fill_matrix( x );
21
22    mv_mult( A, x, y );
23
24    FLA_Obj_show( "A = [", A, "%lf ", "]" );
25    FLA_Obj_show( "x = [", x, "%lf ", "]" );
26    FLA_Obj_show( "y = [", y, "%lf ", "]" );
27
28    FLA_Obj_free( &A );
29    FLA_Obj_free( &y );
30    FLA_Obj_free( &x );
31
32    FLA_Finalize( );
33  }
```

Fig. 4. A simple C driver for matrix-vector multiplication.

—**line 5–6:** FLAME objects `A`, `x`, and `y`, which will hold matrix A and vectors x and y, are declared to be of type `FLA_Obj`.

—**line 10:** Before any calls to FLAME routines can be made, the environment must be initialized by a call to `FLA_Init`.

—**line 12–13:** In our example, the user inputs the row and column dimension of matrix A.

—**line 15–17:** Descriptors are created for A, x, and y.

—**line 19–20:** The routine in Fig. 5, described below, is used to fill A and x with values.

—**line 22:** Compute $y = Ax$ using the routine for performing that operation given in Fig. 6.

—**line 24–26:** Print out the contents of A, x, and y.

—**line 28–30:** Free the created objects.

—**line 32:** Finalize FLAME.

```
1  #include "FLAME.h"
2
3  #define BUFFER( i, j ) buff[ (j)*lda + (i) ]
4
5  void fill_matrix( FLA_Obj A )
6  {
7    int datatype, m, n, lda;
8
9    datatype = FLA_Obj_datatype( A );
10   m        = FLA_Obj_length( A );
11   n        = FLA_Obj_width ( A );
12   lda      = FLA_Obj_ldim  ( A );
13
14   if ( datatype == FLA_DOUBLE ){
15     double *buff;
16     int    i, j;
17
18     buff = ( double * ) FLA_Obj_buffer( A );
19
20     for ( j=0; j<n; j++ )
21       for ( i=0; i<m; i++ )
22         BUFFER( i,j ) = i+j*0.01;
23   }
24   else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
25 }
```

Fig. 5. A simple routine for filling a matrix

A sample routine for filling `A` and `x` with data is given in Fig. 5. Notice that the macro definition in line 3 is used to access the matrix A stored in array `A` using column-major ordering.

The routine in Fig. 6 is itself a wrapper to the level 2 BLAS routine `cblas_dgemv`, a commonly available kernel for computing a matrix-vector multiplication [Dongarra et al. 1988]. Notice that in order to call this routine, which requires parameters describing the matrix, vectors, and scalars to be explicitly passed, requires all of the inquiry routines.

3.6 Views

As illustrated in Figs. 1 and 2, in stating a linear algorithm one may wish to partition matrices like

$$\textbf{Partition} \quad B \rightarrow \left(\frac{B_T}{B_B}\right) \text{ and } A \rightarrow \left(\frac{A_{TL} \| A_{TR}}{A_{BL} \| A_{BR}}\right)$$
$$\textbf{where} \quad B_T \text{ has } k \text{ rows and } A_{TL} \text{ is } k \times k$$

We hide complicated indexing by introducing the the notion of a *view*, which is a reference into an existing matrix or vector. Given that A is a descriptor of a matrix, the following call creates descriptors of the four quadrants:

```
1  #include "FLA.h"
2  #include "cblas.h"
3
4  void mv_mult( FLA_Obj A, FLA_Obj x, FLA_Obj y )
5  {
6    int
7      datatype_A,    m_A, n_A, ldim_A,    m_x, n_y, inc_x,    m_y, n_y, inc_y;
8
9    datatype_A = FLA_Obj_datatype( A );
10   m_A        = FLA_Obj_length( A );
11   n_A        = FLA_Obj_width ( A );
12   ldim_A     = FLA_Obj_ldim  ( A );
13
14   m_x        = FLA_Obj_length( x );       m_y        = FLA_Obj_length( y );
15   n_x        = FLA_Obj_width ( x );       n_y        = FLA_Obj_width ( y );
16
17
18   if ( m_x == 1 ) {
19     m_x  = n_x;
20     inc_x = FLA_Obj_ldim( x );
21   }
22   else inc_x = 1;
23
24   if ( m_y == 1 ) {
25     m_y  = n_y;
26     inc_y = FLA_Obj_ldim( y );
27   }
28   else inc_y = 1;
29
30   if ( datatype_A == FLA_DOUBLE ){
31     double *buff_A, *buff_x, *buff_y;
32
33     buff_A = ( double * ) FLA_Obj_buffer( A );
34     buff_x = ( double * ) FLA_Obj_buffer( x );
35     buff_y = ( double * ) FLA_Obj_buffer( y );
36
37     cblas_dgemv( CblasColMaj, CblasNoTrans,
38                  1.0, buff_A, ldim_A, buff_x, inc_x,
39                  1.0, buff_y, inc_y );
40   }
41   else {
42     printf( "datatype not yet supported\n" );
43     exit( 0 );
44   }
45 }
```

Fig. 6. A simple matrix-vector multiplication routine. This routine is implemented as a wrapper to the BLAS routine `cblas_dgemv` for matrix-vector multiplications.

Fig. 7. FLAME implementation of unblocked TRSM algorithm in Fig. 1

Fig. 8. FLAME implementation of blocked TRSM algorithm in Fig. 2

```
void FLA_Part_2x2( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                              FLA_Obj *ABL, FLA_Obj *ABR,
                   int mb, int nb, int quadrant )
```

Purpose: Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

Here `quadrant` can take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` and `nb` indicate the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively. The translation of the algorithm fragment on the left results in the code on the right

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is $m_b \times n_b$

```
FLA_Part_2x2( A, &ATL, /**/ &ATR,
                /* ************** */
                  &ABL, /**/ &ABR,
  mb, /* by */ nb, /* submatrix */ FLA_TL );
```

where parameters `mb` and `nb` have values m_b and n_b, respectively. Examples of the use of this routine can also be found in Figs. 7 and 8.

NOTE 9. *The above example stresses the fact that the formatting of the code as well as the careful introduction of comments can be used to help capture the algorithm in code. Clearly, much of the benefit of the API would be lost if in the example the code appeared as*

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

Also from Figs. 1 and 2, we notice that it is useful to be able to take a 2×2 partitioning of a given matrix A and repartition this into a 3×3 partitioning so that submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the call

```
void FLA_Repart_from_2x2_to_3x3
    ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                  FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
      FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
      int mb, int nb, int quadrant )
```

Purpose: Repartition a 2×2 partitioning of matrix A into a 3×3 partitioning where `mb` $\times$ `nb` submatrix A_{11} is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` and `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus

Repartition

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

where A_{11} is $m_b \times n_b$

translates to the code

```
FLA_Repart_from_2x2_to_3x3(
              ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
           /* ************* */  /* ********************* */
                                   &A10, /**/ &A11, &A12,
              ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
              mb, nb, FLA_BR );
```

where parameters `mb` and `nb` have values m_b and n_b, respectively. Others examples of the use of this routine can also be found in Figs. 7 and 8.

NOTE 10. *The calling sequence of* `FLA_Repart_from_2x2_to_3x3` *and related calls is a testimony to throwing out the convention that input parameters should be listed before output parameters or vice versa. Notice that is it specifically by mixing input and output parameters in what on the surface may appear to be complete chaos that the repartitioning in the algorithm can be elegantly captured in code.*

NOTE 11. *Chosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right) \quad \textit{and} \quad \begin{array}{lll} \texttt{L00}, & \texttt{l01}, & \texttt{L02}, \\ \texttt{l10t}, & \texttt{lambda11}, & \texttt{l12t}, \\ \texttt{L20}, & \texttt{l21}, & \texttt{L22}, \ \ldots \end{array}$$

in Figs. 1 and 7.

Once the contents of the so identified submatrices have been updated, the descriptions of A_{TL}, A_{TR}, A_{BL}, and A_{BR} must be updated to reflect that progress is being made, in terms of the regions identified by the double-lines. This moving of the double-lines is accomplished by a call to

```
void FLA_Cont_with_3x3_to_2x2
     ( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                     FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
       FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
       int quadrant )
```

Purpose: Update the 2×2 partitioning of matrix A by moving the boundaries so that A_{11} is added to the quadrant indicated by `quadrant`.

This time the value of `quadrant` (`FLA_TL`, `FLA_TR`, `FLA_BL`, or `FLA_BR`) indicates to which quadrant submatrix `A11` is to be added.

For example,

Continue with

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

translates to the code

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

Further examples of the use of this routine can again be found in Figs. 7 and 8.

Similarly, a matrix can be partitioned horizontally into two submatrices with the call

```
void FLA_Part_2x1( FLA_Obj A, FLA_Obj *AT,
                              FLA_Obj *AB, int mb, int side )
```

Purpose: Partition matrix A into a top and bottom side where the side indicated by `side` has `mb` rows.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` indicates the row dimension of A_T or A_B, respectively.

Given that matrix A is already partitioned horizontally it can be repartitioned into three submatrices with the call

```
void FLA_Repart_from_2x1_to_3x1( FLA_Obj AT,   FLA_Obj *A0,
                                               FLA_Obj *A1,
                                 FLA_Obj AB,   FLA_Obj *A2,
                                 int mb, int side )
```

Purpose: Repartition a 2×1 partitioning of matrix A into a 3×1 partitioning where submatrix A_1 with `mb` rows is split from the side indicated by `side`.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOMn` to indicate that `mb` submatrix A_1 is partitioned from A_T or A_B, respectively.

Given a 3×1 partitioning of a given matrix A, the middle submatrix can be appended to either the first or last submatrix with the call

```
void FLA_Cont_with_3x1_to_2x1( FLA_Obj *AT,  FLA_Obj A0,
                                             FLA_Obj A1,
                               FLA_Obj *AB,  FLA_Obj A2,
                               int side )
```

Purpose: Update the 2×1 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

Examples of the use of the routine that deals with the horizontal partitioning of matrices can be found in Figs. 7 and 8.

Finally, a matrix can be partitioned and repartitioned vertically with the calls

```
void FLA_Part_1x2( FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR,
                   int nb, int side )
```

Purpose: Partition matrix A into a left and right side where the side indicated by `side` has `nb` columns.

and

```
void FLA_Repart_from_1x2_to_1x3
       ( FLA_Obj AL,               FLA_Obj AR,
         FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
         int nb, int side )
```

Purpose: Repartition a 1×2 partitioning of matrix A into a 1×3 partitioning where submatrix A_1 with `nb` columns is split from the side indicated by `side`.

Here `side` can take on the values `FLA_LEFT` or `FLA_RIGHT`. Adding the middle submatrix to the first or last is now accomplished by a call to

```
void FLA_Cont_with_1x3_to_1x2
       ( FLA_Obj *AL,            FLA_Obj *AR,
         FLA_Obj A0, FLA_Obj A1, FLA_Obj A2,
         int side )
```

Purpose: Update the 1×2 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

3.7 Computational kernels

All operations described in the last subsection hide the details of indexing in the linear algebra objects. To compute with and/or update data associated with a linear algebra object and/or a view, one calls subroutines that perform the desired operations.

Such subroutines will typically take one of three forms:

—subroutines coded using the FLAME interface, or

—subroutines coded using a more traditional coding style.

—wrappers to highly optimized kernels,

Naturally these are actually three points on a spectrum of possibilities, since one can mix these techniques.

A subset of currently supported operations is given in Appendix A.6. Here, we discuss how to create subroutines that compute these operations. For additional information on supported functionality, please visit the webpage given at the end of this paper or [Gunnels and van de Geijn 2001a].

3.7.1 *Subroutines coded using the FLAME interface.* The subroutine itself could be coded using the FLAME approach to deriving algorithms [Bientinesi et al.] and the interface described in this paper.

For example, the implementation in Fig. 8 of the blocked algorithm given in Fig. 2 requires the update $B_1 := L_{11}^{-1} B_1$ which can be implemented by a call to the unblocked algorithm in Fig. 7.

```
1   #include "FLA.h"
2
3   void FLA_Inv_scal( FLA_Obj alpha, FLA_Obj x )
4   {
5     int datatype_alpha, datatype_x, n_x, inc_x, i;
6     double *buffer_alpha, *buffer_x, inv_alpha;
7
8     datatype_alpha = FLA_Obj_datatype( alpha );
9     datatype_x     = FLA_Obj_datatype( x );
10
11    if ( datatype_alpha != FLA_DOUBLE ||
12         datatype_x     != FLA_DOUBLE ){
13      printf( "datatype not yet supported\n" );
14      exit( 0 );
15    }
16
17    n_x   = FLA_Obj_length( x );
18    inc_x = 1;
19
20    if ( n_x == 1 ){
21      n_x = FLA_Obj_width( x );
22      inc_x = FLA_Obj_ldim( x );
23    }
24
25    buffer_alpha = ( double * ) FLA_Obj_buffer( alpha );
26    buffer_x     = ( double * ) FLA_Obj_buffer( x );
27
28    inv_alpha = 1.0 / *buffer_alpha;
29
30    for ( i=0; i<n_x; i++ )
31      *buffer_x++ *= inv_alpha;
32
33    /* For BLAS based implementation, comment out above loop
34       and uncomment the below call to cblas_dscal */
35
36    /* cblas_dscal( n_x, inv_alpha, buffer_x, inc_x ); */
37  }
```

Fig. 9. Sample implementation of `FLA_Inv_scal`.

3.7.2 *Subroutine coded using a more traditional coding style.* Notice that there is an overhead for the abstractions that we introduce to hide indexing. For implementations of blocked algorithms, this overhead is amortized over a sufficient amount of computation that it is typically not of much consequence. (In the case of the algorithm in Fig. 2 when B is $m \times n$ the indexing overhead is $O(m/b)$ while the useful computation is $O(m^2 n)$.) However, for unblocked algorithms or algorithms that operate on vectors, the relative cost is more substantial. In this case, it may become beneficial to code the subroutine using a more traditional style that exposes indices. For example, the operation

```
FLA_Inv_scal( lambda11, b1t );
```

can be implemented by the subroutine in Fig. 9. (Note that it is probably more efficient to instead implement it by calling `cblas_dscal` or the equivalent BLAS

routine for the appropriate datatype.)

3.7.3 *Wrappers to highly optimized kernels.* A number of matrix and/or vector operations have been identified to be frequently used by the linear algebra community. These are generally referred to as the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990]. Since highly optimized implementations of these operations are supported by widely available library implementations, it makes sense to provide a set of subroutines that are simply wrappers to the BLAS. An example of this is given in Fig. 6.

4. PERFORMANCE ISSUES

In a number of papers that were already mentioned in the introduction we have shown that the presented API can be used to attain high performance for implementations of a broad range of linear algebra operations. Thus, we do not include a traditional performance section. Instead, we discuss some of the issues.

Conventional wisdom used to dictate that raising the level of abstraction at which one codes will adversely impact the performance of the implementation. We, like others, disagree for a number of reasons:

—By raising the level of abstraction, more ambitious algorithms can be implemented which can achieve higher performance [Gunnels et al. 2001; Quintana-Ortí and van de Geijn ; Gunnels and van de Geijn 2001b; Bientinesi et al. 2002; Alpatov et al. 1997; van de Geijn 1997].
One can, of course, argue that these same algorithms can also be implemented at a lower level of abstraction. While this is true for individual operations, implementing entire libraries at a low level of abstraction greatly increases the effort required to implement, maintain, and verify correctness.

—Once implementations are implemented with an API at a high level of abstraction, components can be selectively optimized at a low level of abstraction. We learn from this that the API must be designed to easily accommodate this kind of optimization, as is also discussed in Section 3.7.

—Recent compiler technology (e.g., [Guyer and Lin 1999; 2000b; 2000a; Guyer et al. 2001]) allows library developers to specify dependencies between routines at a high level of abstraction which allows compilers to optimizes between layers. of libraries, automatically achieving the kinds of optimizations that would otherwise be performed by hand.

—Other situations in which abstraction offers the opportunity for higher performance include several mathematical libraries and C++ optimization techniques as well. For example, PMLP [Birov et al. 1998] uses C++ templates to support many different storage formats, thereby decoupling storage format from algorithmic correctness in classes of sparse linear algebra, thus allowing this degree of freedom to be explored for optimizing performance. Also, PMLP features operation sequences and non-blocking operations in order to allow scheduling of mathematical operations asynchronously from user threads. Template metaprogramming and expression templates support concepts including compile-time optimizations involving loop fusion, expression simplification, and removal of unnecessary temporaries; these allow C++ to utilize fast kernels while removing

abstraction barriers between kernels, and further abstraction barriers between sequences of user operations (systems include Blitz++ [Veldhuizen et al.] and PETE [PET]). These techniques, in conjunction with an appropriate FLAME-like API for C++, should allow our algorithms to be expressed at a high level of abstraction without compromising performance.

NOTE 12. *The lesson to be learned is that by raising the level of abstraction, a high degree of confidence in the correctness of the implementation can be achieved while more aggressive optimizations, by hand or by a compiler, can simultaneously be facilitated.*

5. CONCLUSION

In this paper, we have presented a simple interface for implementing linear algebra algorithms. In isolation, the interface illustrates how raising the level of abstraction at which one codes allows one to avoid intricate indexing in the code, which reduces the opportunity for the introduction of errors and raises the confidence of the correctness of the code. In combination with our formal derivation methodology, the API can be used to implement algorithms derived using that methodology so that the proven correctness of those algorithms translates to a high degree of confidence in the implementation.

We again emphasize that the presented API is merely a very simple one that illustrates the issues. Similar interfaces for the Fortran, C++, Matlab M-script, and other languages are easily defined, allowing special features of those languages to be used to even further raise the level of abstraction at which one codes. In addition, the API can be extended to incorporate different datastructures for storing matrices or to allow hierarchical matrices to be defined. (The latter can be achieved by simply allowing `FLA_Obj` as a datatype, which would indicate that each entry in the matrix is itself a matrix object.)

Further Information

Please visit `http://www.cs.utexas.edu/users/flame/`.

Acknowledgments

An ever-growing number of people have contributed to date to the methodology that underlies the Formal Linear Algebra Methods Environment. These include

—UT-Austin: Paolo Bientinesi, Mark Hinga, Dr. Margaret Myers, Vinod Valsalam, and Thierry Joffrain.

—IBM's T.J. Watson Research Center: Dr. John Gunnels and Dr. Fred Gustavson.

—University of Jaume I, Spain: Prof. Enrique Quintana Ortí.

—Intel: Dr. Greg Henry.

—Mississippi State University: Prof. Anthony Skjellum and Wenhao Wu.

In addition, numerous students in undergraduate and graduate courses on high-performance computing at UT-Austin have provided valuable feedback.

REFERENCES

PETE, the portable expression template engine. http://www.acl.lanl.gov/pete Date of access: Oct 24th, 2002.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. 1997. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*.

Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

Bientinesi, P., Gunnels, J. A., Gustavson, F. G., Henry, G. M., Myers, M. E., Quintana-Orti, E. S., and van de Geijn, R. A. 2002. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02) , a workshop in conjunction with the 16th Annual ACM International Conference on Supercomputing (ICS'02)*.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*. submitted.

Bientinesi, P. and van de Geijn, R. A. 2002. Developing linear algebra algorithms: Class projects Spring 2002. Tech. Rep. CS-TR-02-??, Department of Computer Sciences, The University of Texas at Austin. June. In preparation. `http://www.cs.utexas.edu/users/flame/pubs/`.

Birov, L., Purkayastha, A., Skjellum, A., Dandass, Y., and Bangalore, P. V. 1998. PMLP home page. http://www.erc.msstate.edu/labs/hpcl/pmlp.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1 (March), 1–17.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1 (March), 1–17.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

Gropp, W., Lusk, E., and Skjellum, A. 1994. *Using MPI*. The MIT Press.

Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. 2001. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft. 27,* 4 (December), 422–455.

Gunnels, J. A. and van de Geijn, R. A. 2001a. Developing linear algebra algorithms: A collection of class projects. Tech. Rep. CS-TR-01-19, Department of Computer Sciences, The University of Texas at Austin. May. `http://www.cs.utexas.edu/users/flame/`.

Gunnels, J. A. and van de Geijn, R. A. 2001b. Formal methods for high-performance linear algebra libraries. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press, 193–210.

Guyer, S. Z., Berger, E., and Lin, C. 2001. Customizing software libraries for performance portability. In *10^{th} SIAM Conference on Parallel Processing for Scientific Computing*.

Guyer, S. Z. and Lin, C. 1999. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*. 39–52.

Guyer, S. Z. and Lin, C. 2000a. *Broadway: A Software Architecture for Scientific Computing*. Kluwer Academic Press, 175–192.

Guyer, S. Z. and Lin, C. 2000b. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*. 221–238.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5,* 3 (Sept.), 308–323.

Moler, C., Little, J., and Bangert, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

Quintana-Ortí, E. S. and van de Geijn, R. A. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft.*. conditionally accepted.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. 1996. *MPI: The Complete Reference*. The MIT Press.

van de Geijn, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

Veldhuizen, T. et al. Blitz++. URL:http://monet.uwaterloo.ca/blitz/.

A. SUMMARY OF FLAME ROUTINES

In this appendix, we list a number of routines supported as part of the current implementation of the FLAME library. Those experienced with linear algebra libraries will recognize most routines as implementing the functionality of the BLAS.

A.1 Notation

In the descriptions of the below discussed routine, we use the following conventions:

—**Matrices, vectors, and scalar** are denoted by upper-case, lower-case, and lower-case Greek letters, respectively.

—**Transposition:**

$$\mathrm{op}_{\texttt{trans}}(X) = \begin{cases} X & \text{if } \texttt{trans == FLA_NO_TRANSPOSE} \\ X^T & \text{if } \texttt{trans == FLA_TRANSPOSE} \\ X^C & \text{if } \texttt{trans == FLA_CONJ_TRANSPOSE} \\ \bar{X} & \text{if } \texttt{trans == FLA_CONJUGATE} \end{cases}.$$

—**Triangular matrices:** Parameter `uplo` is used to indicate whether a triangular matrix is stored in the lower or upper triangular part of array `A`. This parameter can take on the values `FLA_LOWER_TRIANGULAR` and `FLA_UPPER_TRIANGULAR`. Parameter `diag` is used to indicate the values of the diagonal elements of matrix A:

`diag`	Values on the diagonal of A
`FLA_NONUNIT_DIAG`	As stored in `A`.
`FLA_UNIT_DIAG`	Implicitly take all diagonal elements to equal one.
`FLA_ZERO_DIAG`	Implicitly take all diagonal elements to equal zero.

—**Symmetric matrices:** Only the upper or lower triangular part of a symmetric matrix is stored, as indicated by parameter `uplo`:

`uplo`	Storage of matrix
`FLA_LOWER_TRIANGULAR`	Only lower triangular part of matrix is stored.
`FLA_UPPER_TRIANGULAR`	Only upper triangular part of matrix is stored.

Notice that the values in the other part of the matrix cannot be disturbed and/or used.

—Unless specified otherwise, all routines are of type `void`.

A.2 Initializing and finalizing FLAME

`FLA_Init()` Initialize FLAME.

`FLA_Finalize()` Finalize FLAME.

A.3 Linear algebra objects

```
FLA_Obj_create(int datatype, int m, int n, FLA_Obj *matrix)
```
Create an object that describes an $m \times n$ matrix and create the associated storage array.

```
FLA_Obj_create_without_buffer(int datatype, int m, int n, FLA_Obj *matrix)
```
Create an object that describes an $m \times n$ matrix without creating the associated storage array.

```
FLA_Obj_attach_buffer(*buff, int ldim, FLA_Obj *matrix)
```
Attach an existing buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.

```
FLA_Obj_create_conf_to(int trans, FLA_Obj old, FLA_Obj *matrix)
```
Like `FLA_Obj_create` except that it creates an object with same datatype and dimensions as `old`, transposing if desired.

```
FLA_Obj_free(FLA_Obj *obj)
```
Free all space allocated to store data associated with `obj`.

```
int FLA_Obj_datatype(FLA_Obj matrix)
```
Extract datatype of matrix.

```
int FLA_Obj_length(FLA_Obj matrix)
```
Extract row dimension of matrix.

```
int FLA_Obj_width(FLA_Obj matrix)
```
Extract column dimension of matrix.

```
void *FLA_Obj_buffer(FLA_Obj matrix)
```
Extract the address where the matrix is stored.

```
int FLA_Obj_ldim(FLA_Obj matrix)
```
Extract the leading dimension for the array in which the matrix is stored.

A.4 Views

```
FLA_Part_2x2(FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                        FLA_Obj *ABL, FLA_Obj *ABR, int mb, int nb, int quadrant)
```
Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

```
FLA_Repart_from_2x2_to_3x3
     (FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                  FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
      FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
      int mb, int nb, int quadrant)
```
Repartition a 2×2 partitioning of matrix A into a 3×3 partitioning where `mb` $\times$ `nb` submatrix A_{11} is split from the quadrant indicated by `quadrant`.

```
FLA_Cont_with_3x3_to_2x2
      (FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                     FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
       FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
       int quadrant)
```
Update the 2×2 partitioning of matrix A by moving the boundaries so that A_{11} is added to the quadrant indicated by `quadrant`.

```
FLA_Part_2x1(FLA_Obj A, FLA_Obj *AT,
                        FLA_Obj *AB, int mb, int side)
```
Partition matrix A into a top and bottom side where the side indicated by `side` has `mb` rows.

```
FLA_Repart_from_2x1_to_3x1(FLA_Obj AT,   FLA_Obj *A0,
                                         FLA_Obj *A1,
                           FLA_Obj AB,   FLA_Obj *A2, int mb, int side)
```
Repartition a 2×1 partitioning of matrix A into a 3×1 partitioning where submatrix A_1 with `mb` rows is split from the side indicated by `side`.

```
FLA_Cont_with_3x1_to_2x1(FLA_Obj *AT,  FLA_Obj A0,
                                       FLA_Obj A1,
                         FLA_Obj *AB,  FLA_Obj A2, int side)
```
Update the 2×1 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

```
FLA_Part_1x2(FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR, int nb, int side)
```
Partition matrix A into a left and right side where the side indicated by `side` has `nb` columns

```
FLA_Repart_from_1x2_to_1x3(FLA_Obj AL,               FLA_Obj AR,
                           FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
                           int nb, int side)
```
Repartition a 1×2 partitioning of matrix A into a 1×3 partitioning where submatrix A_1 with `nb` columns is split from the side indicated by `side`.

```
FLA_Cont_with_1x3_to_1x2(FLA_Obj *AL,            FLA_Obj *AR,
                         FLA_Obj A0, FLA_Obj A1, FLA_Obj A2, int side)
```
Update the 1×2 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

A.5 Printing the contents of an object

```
FLA_Obj_show(char *string1, FLA_Obj A, char *format, char *string2)
```
Print the contents of A.

A.6 Subset of supported operations

```
FLA_Axpy(FLA_Obj alpha, FLA_Obj A, FLA_Obj B)
```
$B := \alpha A + B$.

```
FLA_Axpy_x(int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B)
```
$B := \alpha \mathrm{op}_{\mathtt{trans}}(A) + B$.

```
FLA_Copy(FLA_Obj A, FLA_Obj B)
```
$B := A$.

```
FLA_Copy_x(int trans, FLA_Obj A, FLA_Obj B)
```
$B := \mathrm{op}_{\mathtt{trans}}(A)$.

```
FLA_Dot(FLA_Obj x, FLA_Obj y, FLA_Obj rho)
```
$\rho := x^T y$.

```
FLA_Dot_x(FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho)
```
$\rho := \alpha x^T y + \beta \rho$.

```
FLA_Gemm(int transa, int transb, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                                 FLA_Obj beta,  FLA_Obj C)
```
$C := \alpha \mathrm{op}_{\mathtt{transa}}(A) \mathrm{op}_{\mathtt{transb}}(B) + \beta C$.

`FLA_Gemv(int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y)`
$y := \alpha \text{op}_{\texttt{trans}}(A)x + \beta y$.

`FLA_Ger(FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A)`
$A := \alpha x y^T + A$.

`FLA_Iamax(FLA_Obj x, FLA_Obj k)`
Compute index k such that $|\chi_k| = \|x\|_\infty$. **Note:** This operation only works when x has unit row or column dimension.

`FLA_Invert(FLA_Obj alpha)`
$\alpha := 1/\alpha$.

`FLA_Inv_scal(FLA_Obj alpha, FLA_Obj A)`
$A := \frac{1}{\alpha} A$.

`FLA_Negate(FLA_Obj A)`
$A := -A$.

`FLA_Nrm1(FLA_Obj A, FLA_Obj alpha)`
$\alpha := \|A\|_1$.

`FLA_Nrm2(FLA_Obj x, FLA_Obj alpha)`
$\alpha := \|x\|_2$. **Note:** This operation only works when x has unit row or column dimension.

`FLA_Nrm_inf(FLA_Obj A, FLA_Obj alpha)`
$\alpha := \|A\|_\infty$.

`FLA_Obj_set_to_one(FLA_Obj A)`
Set all elements of A to one.

`FLA_Obj_set_to_zero(FLA_Obj A)`
Set all elements of A to zero.

`FLA_Random_matrix(FLA_Obj A)`
Fill A with random values in the range $(-1, 1)$.

`FLA_Scal(FLA_Obj alpha, FLA_Obj A)`
$A := \alpha A$.

`FLA_Set_diagonal(FLA_Obj sigma, FLA_Obj A)`
Set the diagonal of A to σI. All other values in A are unaffected.

`FLA_Shift_spectrum(FLA_Obj alpha, FLA_Obj sigma, FLA_Obj A)`
$A := A + \alpha\sigma I$.

`FLA_Sqrt(FLA_Obj alpha)`
$\alpha := \sqrt{\alpha}$. **Note:** `A` must describe a scalar.

`FLA_Swap(FLA_Obj A, FLA_Obj B)`
$A, B := B, A$.

`FLA_Swap_x(int trans, FLA_Obj A, FLA_Obj B)`
$A, B := \text{op}_{trans}(A), \text{op}_{trans}(B)$.

`FLA_Symm(int side, int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$, where A is symmetric, `side` indicates the side from which A multiplies B, `uplo` indicates whether A is stored in the upper or lower triangular part of `A`.

`FLA_Symmetrize(int uplo, int conj, FLA_Obj A)`
$A := \text{symm}(A)$ or $A := \text{herm}(A)$, where `uplo` indicates whether A is originally stored only in the upper or lower triangular part of `A`.

`FLA_Symv(int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y)`
$y := \alpha Ax + \beta y$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

`FLA_Syr(int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A)`
$A := \alpha xx^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

`FLA_Syr2(int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A)`
$A := \alpha xy^T + \alpha yx^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

`FLA_Syr2k(int uplo, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha(\text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(B)^T + \text{op}_{\texttt{trans}}(B)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

`FLA_Syrk(int uplo, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj beta, FLA_Obj C)`
$C := \alpha\text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

`FLA_Triangularize(int uplo, int diag, FLA_Obj A)`
$A := \text{lower}(A)$ or $A := \text{upper}(A)$.

`FLA_Trmm(int side, int uplo, int trans, int diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B)`
$B := \alpha\text{op}_{\texttt{trans}}(A)B$ (`side == FLA_LEFT`) or $B := \alpha B\text{op}_{\texttt{trans}}(A)$ (`side == FLA_RIGHT`). where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmm_x(int side, int uplo, int transa, int transb, int diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha\text{op}_{\texttt{transa}}(A)\text{op}_{\texttt{transb}}(B) + \beta C$ (`side == FLA_LEFT`) or $C := \alpha\text{op}_{\texttt{transb}}(B)\text{op}_{\texttt{transa}}(A) + \beta C$ (`side == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmv(int uplo, int trans, int diag, FLA_Obj A, FLA_Obj x)`
$x := \text{op}_{\texttt{trans}}(A)x$, where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmv_x(int uplo, int trans, int diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y)`
Update $y := \alpha\text{op}_{\texttt{trans}}(A)x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trsm(int side, int uplo, int trans, int diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B)`
$B := \alpha\text{op}_{\texttt{trans}}(A)^{-1}B$ (`SIDE == FLA_LEFT`) or $B := \alpha B\text{op}_{\texttt{trans}}(A)^{-1}$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trsm_x(int side, int uplo, int transa, int transb, int diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha\text{op}_{\texttt{transa}}(A)^{-1}\text{op}_{\texttt{transb}}(B) + \beta C$ (`SIDE == FLA_LEFT`) or $C := \alpha\text{op}_{\texttt{transb}}(B)\text{op}_{\texttt{transa}}(A)^{-1} + \beta C$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trsv(int uplo, int trans, int diag, FLA_Obj A, FLA_Obj x)`
$x := \text{op}_{\texttt{trans}}(A)^{-1}x$, where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsv_x(int uplo, int trans, int diag, FLA_Obj alpha,
           FLA_Obj A, FLA_Obj x, FLA_Obj beta,  FLA_Obj y)
```
$y := \alpha \text{op}_{\texttt{trans}}(A)^{-1}x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

Received Month Year; revised Month Year; accepted Month Year