FLAG/C Users' Guide

María J. Zafont, Alberto Martín, Francisco D. Igual, and Enrique S. Quintana-Ortí
Universidad Jaime I (Spain)
al051631@alumail.uji.es, {martina,figual,quintana}@icc.uji.es

In this short users' guide we will describe the routines in the current implementation of FLAG/C, an Application Programming Interface (API) to interact with NVIDIA CUBLAS and CUDA using the FLAME framework for dense linear algebra operations. We strongly recommend you become familiar with FLAME first by visiting the web site for the project at http://www.cs.utexas.edu/users/flame. In particular, we urge you to read the related paper

Paolo Bientinesi, Enrique S. Quintana-Orti, and Robert van de Geijn.
"Representing Linear Algebra Algorithms in Code: The FLAME APIs".
ACM Transactions on Mathematical Software, 31(1):27-59, March 2005.

or its draft version available at http://www.cs.utexas.edu/users/flame/pubs/fire.ps.

1 Notation

In the description of the routines, we employ lowercase Greek letters and Roman letters to denote scalars and vectors, respectively. Uppercase Roman letters will be used for matrices. The only exceptions to these rules are matrix and vector dimensions (including block sizes), which will be denoted by Roman letters.

Assignment is denoted by operator ":=".

Some of the arguments in FLAG/C allow the user to specify options for the operation. A list of these (mode) arguments and possible values is offered in Table 1.

2 Initialization and finalization

FLAG_Bool FLAG_Init(void)
Purpose: Initialize the library.

FLAG_Bool FLAG_Finalize(void)
Purpose: Release all internal library resources. After FLAG_Finalize() returns, the library functions should not be used until FLAG_Init() is called again.

3 Object creation and destruction

The key element in the FLAG/C API is the FLAG_Obj which corresponds to an object with its contents (entries) in the device (GPU) memory. This element plays an analogous role to that of the FLA_Obj in the FLAME/C API, which stores its contents in the host (main) memory.

FLAG_Error FLAG_Obj_create(FLAG_Datatype datatype, int m, int n, FLAG_Obj *A)
Purpose: Create a new object A from an uninitialized FLAG_Obj structure. Upon returning, A points to a valid m×n object allocated in the device memory whose elements are of numerical type datatype.

FLAG_Error FLAG_Obj_create_conf_to(FLAG_Trans trans, FLAG_Obj B,
FLAG_Obj *A)
Purpose: Create a new object A with the same data type and dimensions as an existing object B. The user may optionally create A with the m and n dimensions transposed by specifying FLA_TRANSPOSE for the trans argument.

`FLAG_Error FLAG_Obj_create_without_buffer( FLAG_Datatype datatype, int m, int n, FLAG_Obj *A )`
Purpose: Create a new object, except without any internal numerical data buffer in the device memory. Before using the object, the user must attach a valid buffer with `FLAG_Obj_attach_buffer()`.

`FLAG_Error FLAG_Obj_attach_buffer( void *buffer, int ldim, FLAG_Obj *obj )`
Purpose: Attach a user-allocated region of memory in the device memory to an object that was created with `FLAG_Obj_create_without_buffer()`.

`FLAG_Error FLAG_Obj_free( FLAG_Obj *A )`
Purpose: Release the space allocated for object `A` in the device memory.

4 General query functions

`FLAG_Datatype FLAG_Obj_datatype( FLAG_Obj A )`
Purpose: Query the numerical data type of an object.

`int FLAG_Obj_length( FLAG_Obj A )`
Purpose: Query the number of rows in an object.

`int FLAG_Obj_width( FLAG_Obj A )`
Purpose: Query the number of columns in an object.

`void *FLAG_Obj_buffer( FLAG_Obj A )`
Purpose: Query the starting address of an objects's underlying element data buffer in the device memory.

`int FLAG_Obj_ldim( FLAG_Obj A )`
Purpose: Query the leading dimension associated with the object's underlying element data buffer in the device memory.

`FLAG_Error FLAG_Obj_show( char *s1, FLAG_Obj A, char *format, char *s2 )`
Purpose: Display the numerical values inside an `FLAG_Obj`.

5 Interfacing with objects in the host memory

`FLAG_Error FLAG_Copy_host_to_device( FLA_Obj A, FLAG_Obj B )`
Purpose: Copy the numerical contents of A to B: $B := A$ where A and B are general matrices in the host and device memories, respectively.

`FLAG_Error FLAG_Obj_create_from_FLA_Obj( FLA_Obj A, FLAG_Obj *B )`
Purpose: Create a new object `B` with the same data type and dimensions as an existing object `A`, and copy the numerical contents of A to B: $B := A$ where A and B are general matrices in the host and device memories, respectively.

`FLAG_Error FLAG_Copy_device_to_host( FLAG_Obj A, FLA_Obj B )`
Purpose: Copy the numerical contents of A to B: $B := A$ where A and B are general matrices in the device and host memories, respectively.

6 Managing views

6.1 Vertical partitioning

```
FLAG_Error FLAG_Part_2x1 ( FLAG_Obj A,  FLAG_Obj *AT,
                                        FLAG_Obj *AB, int mb,  FLAG_Side side )
```
Purpose: Partition a matrix `A` into a top and bottom side where the side indicated by `side` has `mb` rows.

```
FLAG_Error FLAG_Repart_2x1_to_3x1( FLAG_Obj AT,  FLAG_Obj *A0,
                                                 FLAG_Obj *A1,
                                   FLAG_Obj AB,  FLAG_Obj *A2,
                                   int      mb,  FLAG_Side side )
```
Purpose: Repartition a 2×1 partitioning of matrix `A` into a 3×1 partitioning where `nb` rows are split from the side indicated by `side`.

```
FLAG_Error FLAG_Cont_with_3x1_to_2x1( FLAG_Obj *AT,  FLAG_Obj A0,
                                                     FLAG_Obj A1,
                                      FLAG_Obj *AB,  FLAG_Obj A2,  FLAG_Side side )
```
Purpose: Update the 2×1 partitioning of matrix `A` by moving the boundaries so that `A1` is shifted to the side indicated by `side`.

6.2 Horizontal partitioning

```
FLAG_Error FLAG_Part_1x2( FLAG_Obj A, FLAG_Obj *A1, FLAG_Obj *A2,
                          int     nb, FLAG_Side side )
```
Purpose: Partition a matrix `A` into a left and right side where the side indicated by `side` has `nb` columns.

```
FLAG_Error FLAG_Repart_1x2_to_1x3( FLAG_Obj  AL,               FLAG_Obj  AR,
                                   FLAG_Obj *A0, FLAG_Obj *A1, FLAG_Obj *A2,
                                   int nb, FLAG_Side side )
```
Purpose: Repartition a 1×2 partitioning of matrix `A` into a 1×3 partitioning where `nb` columns are split from the side indicated by `side`.

```
FLAG_Error FLAG_Cont_with_1x3_to_1x2( FLAG_Obj *AL,               FLAG_Obj *AR,
                                      FLAG_Obj  A0, FLAG_Obj A1, FLAG_Obj  A2,
                                      FLAG_Side side )
```
Purpose: Update the 1×2 partitioning of matrix `A` by moving the boundaries so that `A1` is shifted to the side indicated by `side`.

6.3 Bidimensional partitioning

```
FLAG_Error FLAG_Part_2x2( FLAG_Obj A, FLAG_Obj *ATL, FLAG_Obj *ATR,
                                      FLAG_Obj *ABL, FLAG_Obj *ABR,
                          int mb, int nb, FLAG_Quadrant quadrant )
```
Purpose: Partition a matrix `A` into four quadrants where the quadrant indicated by `quadrant` is `mb`×`nb`.

```
FLAG_Error FLAG_Repart_2x2_to_3x3
( FLAG_Obj ATL, FLAG_Obj ATR,FLAG_Obj *A00,FLAG_Obj *A01, FLAG_Obj *A02,
                             FLAG_Obj *A10,FLAG_Obj *A11, FLAG_Obj *A12,
  FLAG_Obj ABL, FLAG_Obj ABR,FLAG_Obj *A20,FLAG_Obj *A21, FLAG_Obj *A22,
  int       mb,  int      nb,FLAG_Quadrant quadrant )
```
Purpose: Repartition a 2×2 partitioning of matrix `A` into a 3×3 partitioning where `mb`×`nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

```
FLAG_Error FLAG_Cont_with_3x3_to_2x2
( FLAG_Obj *ATL, FLAG_Obj *ATR,  FLAG_Obj A00, FLAG_Obj A01, FLAG_Obj A02,
                                 FLAG_Obj A10, FLAG_Obj A11, FLAG_Obj A12,
  FLAG_Obj *ABL, FLAG_Obj *ABR,  FLAG_Obj A20, FLAG_Obj A21, FLAG_Obj A22,
  FLAG_Quadrant quadrant )
```
Purpose: Update the 2×2 partitioning of matrix `A` by moving the boundaries so that `A11` is shifted to the quadrant indicated by `quadrant`.

Parameter	Data type	Valid inputs	Meaning
datatype	FLAG_Datatype	FLAG_INT FLAG_FLOAT FLAG_COMPLEX	Data type of the elements of a vector or matrix
side	FLAG_Side	FLAG_LEFT FLAG_RIGHT	Indicates whether a matrix with special structure appears to the left or right of an operator
		FLAG_LEFT FLAG_RIGHT FLAG_TOP FLAG_BOTTOM	Also used to indicate that block 1 is split from the indicated side
uplo	FLAG_Uplo	FLAG_LOWER_TRIANGULAR FLAG_UPPER_TRIANGULAR	Only the elements from the indicated triangle of a symmetric or triangular matrix are referenced
trans	FLAG_Trans	FLAG_NO_TRANSPOSE	$op_{trans}(A) := A$
		FLAG_TRANSPOSE	$op_{trans}(A) := A^T$
		FLAG_CONJ_TRANSPOSE	$op_{trans}(A) := A^H = \bar{A}^T$
		FLAG_CONJ_NO_TRANSPOSE	$op_{trans}(A) := \bar{A}$
diag	FLAG_Diag	FLAG_NONUNIT_DIAG	Operate with diagonal elements
		FLAG_UNIT_DIAG	All diagonal elements are implicitly assumed to equal 1
		FLAG_ZERO_DIAG	All diagonal elements are implicitly assumed to equal 0
quadrant	FLAG_Quadrant	FLAG_TL FLAG_TR FLAG_BL FLAG_BR	Submatrix (1,1) is split from the indicated quadrant

Table 1: Description of the mode arguments for the routines in the FLAG/C API.

6.4 Merging views

```
FLAG_Error FLAG_Merge_2x1( FLAG_Obj AT,
                           FLAG_Obj AB,  FLAG_Obj *A )
```
Purpose: Merge a 2×1 set of adjacent matrix views into a single view.

```
FLAG_Error FLAG_Merge_1x2( FLAG_Obj AL, FLAG_Obj  AR,  FLAG_Obj *A )
```
Purpose: Merge a 1×2 set of adjacent matrix views into a single view.

```
FLAG_Error FLAG_Merge_2x2( FLAG_Obj ATL, FLAG_Obj ATR,
                           FLAG_Obj ABL, FLAG_Obj ABR,  FLAG_Obj *A )
```
Purpose: Merge a 2×2 set of adjacent matrix views into a single view.

7 BLAS

7.1 Level-1 BLAS

`FLAG_Error FLAG_Axpy( FLA_Obj alpha, FLAG_Obj A, FLAG_Obj B )`

Purpose: Perform an axpy update operation: $B := \alpha A + B$ where α is a scalar, and A and B are general matrices.

`FLAG_Error FLAG_Copy( FLAG_Obj A, FLAG_Obj B )`

Purpose: Copy the numerical contents of A to B: $B := A$ where A and B are general matrices.

`FLAG_Error FLAG_Copy_x( FLAG_Trans trans, FLAG_Obj A, FLAG_Obj B )`

Purpose: Copy the numerical contents of A to B: $B := op_{trans}(A)$ where A and B are general matrices. The `trans` argument allows the computation to proceed as if A were transposed.

`FLAG_Error FLAG_Dot( FLAG_Obj x, FLAG_Obj y, FLA_Obj rho )`

Purpose: Perform a dot (inner) product operation between two vectors: $\rho := x^T y$ where ρ is a scalar and x and y are vectors.

`FLAG_Error FLAG_Iamax( FLAG_Obj x, FLA_Obj index )`

Purpose: Find the index i of the element of x whic has the maximum absolute value, where x is a vector and i is a scalar. If the maximum absolute value is shared by more than one element, then the element whose index is highest is chosen.

`FLAG_Error FLAG_Inv_scal( FLA_Obj alpha, FLAG_Obj A )`

Purpose: Perform the inverse scaling operation: $A := \frac{1}{\alpha} A$ where α is a scalar and A is a general matrix.

`FLAG_Error FLA_Nrm1(FLA_Obj A, FLA_Obj alpha)`

Purpose: Compute the 1-norm: $\alpha := \|A\|_1$ where A is a matrix.

`FLAG_Error FLAG_Nrm2( FLAG_Obj x, FLA_Obj nrm_x )`

Purpose: Compute the 2-norm of a vector: $\alpha := \|x\|_2$.

`FLAG_Error FLAG_Scal( FLA_Obj alpha, FLAG_Obj A )`

Purpose: Perform a scaling operation: $A := \alpha A$ where α is a scalar and A is a general matrix.

`FLAG_Error FLAG_Swap( FLAG_Obj A, FLAG_Obj B`

Purpose: Swap the contents of two general matrices A and B.

7.2 Level-2 BLAS

`FLAG_Error FLAG_Gemv( FLAG_Trans trans, FLA_Obj alpha, FLAG_Obj A,`
`FLAG_Obj x, FLA_Obj beta, FLAG_Obj y )`

Purpose: Perform the general matrix-vector product operation: $y := \alpha op_{transa}(A)x + \beta y$ where α and β are scalars, A is a general matrix, and x and y are vectors. The `trans` argument allows the computation to proceed as if A were transposed.

`FLAG_Error FLAG_Ger( FLA_Obj alpha, FLAG_Obj x, FLAG_Obj y, FLAG_Obj A )`

Purpose: Perform the general rank-1 update operation: $A := \alpha x y^T + A$ where α is a scalar, A is a general matrix, and x and y are vectors.

`FLAG_Error FLAG_Symv( FLAG_Uplo uplo, FLA_Obj alpha, FLAG_Obj A,`
`FLAG_Obj x, FLA_Obj beta, FLAG_Obj y )`

Purpose: Perform the symmetric matrix-vector multiplication operation: $y := \alpha A x + \beta y$, where α and β are scalars, A is a symmetric matrix, and x and y are vectors. The `uplo` argument indicates whether the lower or upper triangule of A is referenced by the operation.

```
FLAG_Error FLAG_Syr( FLAG_Uplo uplo, FLA_Obj alpha, FLAG_Obj x,
                     FLAG_Obj A )
```
Purpose: Perform the symmetric rank-1 update operation: $A := \alpha x x^T + A$, where α is a scalar, A is a symmetric matrix, and x is a vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

```
FLAG_Error FLAG_Syr2( FLAG_Uplo uplo, FLA_Obj alpha, FLAG_Obj x,
                      FLAG_Obj y, FLAG_Obj A )
```
Purpose: Perform the symmetric rank-2 operation: $A := \alpha x y^T + \alpha y x^T + A$ where α is a scalar, A is a symmetric matrix, and x and y are vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

```
FLAG_Error FLAG_Trmv( FLAG_Uplo uplo, FLAG_Trans trans,
                      FLAG_Diag diag, FLAG_Obj A, FLAG_Obj x )
```
Purpose: Perform the triangular matrix-vector product operation: $x := \alpha op_{trans}(A)x$ where α is a scalar, A is a triangular matrix, and x is a vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `trans` argument allows the computation to proceed as if A were transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

```
FLAG_Error FLAG_Trsv( FLAG_Uplo uplo, FLAG_Trans trans, FLAG_Diag diag,
                      FLAG_Obj A, FLAG_Obj x )
```
Purpose: Perform the triangular solve with a single right-hand side: $x := \alpha op_{trans}(A)^{-1}x$ where α is a scalar, A is a triangular matrix, and x is a vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `trans` argument allows the computation to proceed as if A were transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

7.3 Level-3 BLAS

```
FLAG_Error FLAG_Gemm( FLAG_Trans transa, FLAG_Trans transb, FLA_Obj alpha,
                      FLAG_Obj A, FLAG_Obj B, FLA_Obj beta, FLAG_Obj C )
```
Purpose: Perform the matrix-matrix multiplication operation: $C := \alpha op_{transa}(A) op_{transb}(B) + \beta C$ where α and β are scalars, and A, B, and C are general matrices. The `transa` and `transb` arguments allow the computation to proceed as if A and/or B were transposed.

```
FLAG_Error FLAG_Symm( FLAG_Side side, FLAG_Uplo uplo, FLA_Obj alpha,
                      FLAG_Obj A, FLAG_Obj B, FLA_Obj beta, FLAG_Obj C)
```
Purpose: Perform one of the following symmetric matrix-matrix multiplication operations: $C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$ where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. The `side` argument indicates whether the symmetric matrix A is multiplied on the left or the right side of B. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

```
FLAG_Error FLAG_Syrk( FLAG_Uplo uplo, FLAG_Trans trans, FLA_Obj alpha,
                      FLAG_Obj A, FLA_Obj beta, FLAG_Obj C )
```
Purpose: Perform the symmetric rank-k update operation: $C := \alpha op_{trans}(A) op_{trans}(A)^T + \beta C$ where α and β are scalars, C is a symmetric matrix, and A is a general matrix. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A were transposed.

```
FLAG_Error FLAG_Syr2k ( FLAG_Uplo uplo, FLAG_Trans trans, FLA_Obj alpha,
                        FLAG_Obj A, FLAG_Obj B, FLA_Obj beta, FLAG_Obj C )
```
Purpose: Perform the symmetric rank-$2k$ update operation: $C := \alpha(op_{trans}(A) op_{trans}(B)^T + op_{trans}(B) op_{trans}(A)^T) + \beta C$ where α and β are scalars, C is a symmetric matrix, and A is a general matrix. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A and B were transposed.

```
FLAG_Error FLAG_Trmm( FLAG_Side side, FLAG_Uplo uplo, FLAG_Trans trans,
                      FLAG_Diag diag, FLA_Obj alpha, FLAG_Obj A,
                      FLAG_Obj B)
```

Purpose: Perform one of the following triangular matrix-matrix multiplication operations: $B := \alpha op_{trans}(A)B$ or $B := \alpha Bop_{trans}(A)$ where α is a scalar, A is a triangular matrix, and B is a general matrix. The `side` argument indicates whether the triangular matrix A is multiplied on the left or the right side of B. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `trans` argument allows the computation to proceed as if A were transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

```
FLAG_Error FLAG_Trsm( FLAG_Side side, FLAG_Uplo uplo, FLAG_Trans trans,
                      FLAG_Diag diag, FLA_Obj alpha, FLAG_Obj A,
                      FLAG_Obj B)
```

Purpose: Perform one of the following triangular solves with multiple right-hand sides: $B := \alpha op_{trans}(A)^{-1}B$ or $B := \alpha Bop_{trans}(A)^{-1}$ where α is a scalar, A is a triangular matrix, and B is a general matrix. The `side` argument indicates whether the triangular matrix A is multiplied on the left or the right side of B. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `trans` argument allows the computation to proceed as if A were transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

8 FLAG/C example: the Cholesky factorization

To illustrate the functionality and usage of the FLAG/C API, we offer a simple code example which prepares the environment for the Cholesky factorization of a SPD matrix. Figure 1 shows a sequence of steps which perform the following operations:

- line 10: before any FLAG/C invocation the environment is initialized with routine FLAG_Init.
- lines 13-14: the objects A and A_cpu that will hold the matrix are created in the GPU and CPU memory, respectively.
- line 17: the contents of the matrix in the host memory space are initialized to hold a symmetric positive definite matrix.
- line 20: data stored in a local buffer in the CPU memory is transferred to the GPU memory using routine FLAG_set_from_buffer.
- line 23: the contents of the created matrix are shown.
- line 26: calculation of the Cholesky factorization with the corresponding routine. The code of this routine will be shown later.
- line 29: (optionally) transfer result back to CPU memory by the invocation of the routine FLAG_Copy_device_to_host.
- line 32-33: memory space allocated for the objects in GPU and CPU memory is released with routine FLAG_Obj_free.
- line 36: close the environment with routine FLAG_Finalize.

Figure 2 implements a blocked variant of the Cholesky factorization using the FLAG/C API. The equivalent algorithm expressed using the FLAME/C notation is shown

```
1  void FLAG_Cholesky_Driver ( int n, int nb_alg )
2  {
3
4    /* Object A will store the matrix in GPU memory */
5    FLAG_Obj A;
6    /* Object A_cpu will store the matrix in host memory */
7    FLA_Obj  A_cpu;
8
9    /* Initialize environment */
10   FLAG_Init( );
11
12   /* Create objects on CPU and GPU */
13   FLA_Obj_create( FLA_FLOAT, n, n, &A_cpu );
14   FLAG_Obj_create( FLAG_FLOAT, n, n, &A );
15
16   /* Initialize A_cpu as a SPD matrix */
17   fill_matrix_SPD( A_cpu );
18
19   /* Transfer matrix from CPU to GPU */
20   FLAG_Copy_host_to_device( A_cpu, A );
21
22   /* Show matrix contents */
23   FLAG_Obj_show( "A= ", A, "%f", "" );
24
25   /* CHOLESKY FACTORIZATION OF A */
26   result = FLAG_Cholesky( A, nb_alg );
27
28   /* Copy results back to host memory */
29   FLAG_Copy_device_to_host( A, A_cpu );
30
31   /* Free objects */
32   FLAG_Obj_free( &A );
33   FLA_Obj_free( &A_cpu );
34
35   /* Finalize environment */
36   FLAG_Finalize( );
37
38 }
```

Figure 1: Generic `FLAG/C` driver for the invocation of the Cholesky factorization routine

```
1  int FLAG_Cholesky( FLAG_Obj A, int nb_alg )
2  {
3    FLAG_Obj     ATL, ATR,   A00, A01, A02,
4                 ABL, ABR,   A10, A11, A12,
5                             A20, A21, A22;
6
7    int b, k=1, value;
8
9    FLAG_Part_2x2( A,  &ATL, /**/ &ATR,
10                  /* ************** */
11                     &ABL, /**/ &ABR,
12              /* with */ 0, /* by */ 0, /* submatrix */ FLAG_TL );
13
14   while ( FLAG_Obj_length( ATL ) < FLAG_Obj_length( A ) ){
15     b = min( min( FLAG_Obj_length( ABR ), FLAG_Obj_width( ABR ) ), nb_alg );
16
17     FLAG_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
18                           /* ************* */   /* ******************* */
19                                  /**/            &A10, /**/ &A11, &A12,
20                             ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
21                    /* with */ b, /* by */ b, /* A11 split from */ FLAG_BR );
22
23     /* ************************************************************************ */
24
25     FLAG_Trsm( FLAG_RIGHT, FLAG_LOWER_TRIANGULAR,
26                FLAG_TRANSPOSE, FLAG_NONUNIT_DIAG,
27                FLA_ONE, A00, A10 );
28
29     FLAG_Syrk( FLAG_LOWER_TRIANGULAR, FLAG_NO_TRANSPOSE,
30                FLA_MINUS_ONE, A10, FLA_ONE, A11 );
31
32     if( FLAG_Cholesky_Unblocked( A11 ) != FLA_SUCCESS )
33       return FLA_FAILURE;
34
35     /* ************************************************************************ */
36
37     FLAG_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
38                                      /**/            A10, A11, /**/ A12,
39                              /* ************** */ /* ****************** */
40                                 &ABL, /**/ &ABR,      A20, A21, /**/ A22,
41                              /* with A11 added to submatrix */ FLAG_TL );
42   }
43
44   return FLA_SUCCESS;
45 }
```

Figure 2: `FLAG/C` implementation of the Cholesky factorization algorithm shown in Figure 3.

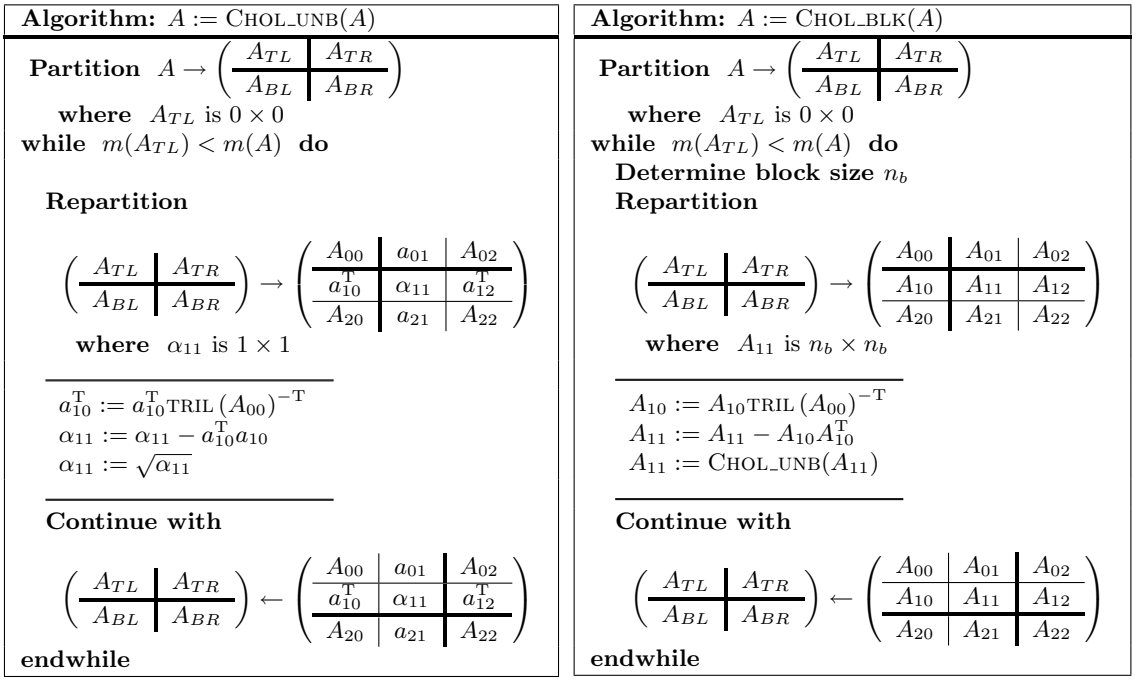

Figure 3: Algorithmic variants of the Cholesky factorization represented using the FLAME notation. Left: scalar implementations. Right: blocked implementations.

in Figure 3 (right). The notation $m(B)$ in the algorithm refers to the row dimension of B. We believe the rest of the notation is intuitive.

The code skeleton for the implementation shown in Figure 2 can be easily generated with the `Spark` tool[1]. Spark is a tool for mechanical creation of code skeletons. By filling out a menu, an outline for FLAME/C code (and other APIs) is generated. This greatly simplifies the generation of the rigid format advocated by the `FLAME/C` methodology.

`FLAG/C` has a complete set of routines to partition and repartition objects. A comparison of Figures 2 and 3 (right) illustrates how the routines in `FLAG/C` for the partitioning (`FLAG_Part_2x2`), repartitioning (`FLAG_Repart_2x2_to_3x3`), and redistribution of the partitions (`FLAG_Cont_with_3x3_to_2x2`) track the movement of the thick lines in the algorithm.

`FLAG/C` also provides a full interface for the three levels of the *Basic Linear Algebra Subprograms* (BLAS); see, e.g., lines 25–30 of Figure 2. These routines are implemented as wrappers that encapsulate the indexing and storage details of the matrices, working with objects as defined above, and using a tuned implementation of the BLAS for the target graphics processor (CUBLAS) as the underlying BLAS implementation.

[1] http://www.cs.utexas.edu/users/flame/Spark/