Chapter 10

Symmetric Rank-2k Update

\[ C \leftarrow A^T B + B^T A + C \]

\( C \) symmetric, stored in lower triangle.

by

Robert M. Green
Jerry L. Slayton

In this chapter, we discuss the implementation of the Symmetric Rank-2k update matrix matrix multiplication:

\[ C \leftarrow A^T B + B^T A + C \]

where \( C \) is a \( n \times n \) symmetric matrix stored in the lower triangular portion of array \( C \). \( K \) and \( N \) are the dimensions of \( A \) and \( B \).

Let us consider

\[ D = A^T B + B^T A + C \]

keeping in mind that \( D \) will overwrite \( C \).

10.1 Algorithms that start by splitting \( C \)

Partition

\[ C \rightarrow \begin{pmatrix} C_{TL} & C_{BL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \]

where \( C_{TL} \) is \( k \times k \). Also, partition

\[ A \rightarrow ( A_L \parallel A_R ) , \quad B \rightarrow ( B_L \parallel B_R ) \quad \text{and} \quad D \rightarrow \begin{pmatrix} D_{TL} & D_{BL}^T \\ D_{BL} & D_{BR} \end{pmatrix} \]

where \( A_T \) and \( B_T \) have \( k \) rows.

Now, (10.1) becomes

\[ \begin{pmatrix} D_{TL} & D_{BL}^T \\ D_{BL} & D_{BR} \end{pmatrix} = ( A_L \parallel A_R )^T ( B_L \parallel B_R ) + ( B_L \parallel B_R )^T ( A_L \parallel A_R ) + \begin{pmatrix} C_{TL} & C_{BL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \]
Multiplying out the right-hand-side yields (note: the top right is the transpose of the bottom left and we will not be storing this in memory.)

\[
\begin{pmatrix}
D_{TL} & D_{BL}^T \\
D_{BL} & D_{BR}
\end{pmatrix}
= \begin{pmatrix}
A_L^T B_L + B_L^T A_L + C_{TL} \\
A_R^T B_L + B_R^T B_L + C_{BL}
\end{pmatrix}
\begin{pmatrix}
D_{TL}^T \\
A_R^T B_R + B_R^T A_R + C_{BR}
\end{pmatrix}
\]

This in turn exposes the equalities

\[
\begin{align*}
D_{TL} &= A_L^T B_L + B_L^T A_L + C_{TL} \\
D_{BL} &= A_R^T B_L + B_R^T B_L + C_{BL} \\
D_{BR} &= A_R^T B_R + B_R^T A_R + C_{BR}
\end{align*}
\]

which must hold.

Possible contents of \( D \) at this intermediate step are given in Fig. 10.1.

<table>
<thead>
<tr>
<th>( D ) contains</th>
<th>Comments</th>
</tr>
</thead>
</table>
| \[ \begin{pmatrix}
C_{TL} \\
C_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix} \] | This indicates no progress has been made. Clearly not a desirable state. |
| \[ \begin{pmatrix}
D_{TL} \\
C_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix} \] | This one is a viable option. With progress coming from the top left and heading down and right as you use and then overwrite the elements in \( C \) you no longer need those elements for further computations. |
| \[ \begin{pmatrix}
D_{TL} \\
D_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix} \] | This one is a viable option. This time as progress proceeds from the left to right you again use and then write over elements in \( C \) that you will not need later in computations. |
| \[ \begin{pmatrix}
D_{TL} \\
D_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
D_{BR}
\end{pmatrix} \] | Notice that this assumes that the computation has finished, which is not very interesting. |
| \[ \begin{pmatrix}
D_{TL} \\
D_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix} \] | This one is a viable option. This is just the reverse of the second possibility. With progress traveling from bottom right to top left you maintain the system of using and then overwriting elements that are not needed later on. |
| \[ \begin{pmatrix}
C_{TL} \\
D_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
D_{BR}
\end{pmatrix} \] | This one is a viable option. |
| \[ \begin{pmatrix}
C_{TL} \\
D_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
D_{BR}
\end{pmatrix} \] | This one is a viable option but doesn’t lend itself to division by the methods discussed in this chapter. |

Figure 10.1: Possible contents of \( D \) at an intermediate step.

### 10.1.1 Right-down moving lazy algorithm (w.r.t. \( C \))

Consider the condition that currently

\[
(10.2) \quad D \text{ contains } \begin{pmatrix}
D_{TL} \\
C_{BL}
\end{pmatrix}
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix}
\]

Notice that this indicates that \( A_L \), \( B_L \) and \( C_{TL} \) have been used to update \( D \), which will call an algorithm relative to operand \( D \).

In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved down and right. Thus, this algorithm naturally moves through matrices \( D \) in the “down right” direction.
Unblocked algorithm

Repartition: $D$

\[
\begin{pmatrix}
D_{TL} & D_{BL}^T & D_{BR}^T \\
D_{BL} & D_{B0} & D_{B1} \\
D_{BR} & D_{B2} & D_{B2}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
D_{00} & * & * \\
\gamma_{10} & \delta_{11} & * \\
\gamma_{12} & \delta_{21} & D_{22}
\end{pmatrix}
\]

where $d_{10}$ is a row and $\delta_{11}$ is a scalar. Similarly, we repartition $A, B,$

\[
\begin{pmatrix}
A_L & A_R \\
A_0 & a_1 & A_2
\end{pmatrix},
\begin{pmatrix}
B_L & B_R \\
B_0 & b_1 & B_2
\end{pmatrix}
\]

where $a_1$ and $b_1$ are columns. $C$

\[
\begin{pmatrix}
C_{TL} & * & * \\
C_{BL} & * & * \\
C_{BR} & * & *
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} & * & * \\
\gamma_{10} & \gamma_{11} & * \\
\gamma_{12} & c_{21} & C_{22}
\end{pmatrix}
\]

where $c_{10}$ is a row and $\gamma_{11}$ is a scalar. Notice the double lines have meaning:

\[
\begin{pmatrix}
D_{TL} = D_{00} \\
D_{BL} = D_{BL} \\
D_{BR} = D_{BR}
\end{pmatrix},
\begin{pmatrix}
A_L = A_0 \\
B_L = B_0 \\
B_R = (b_1 \mid B_2)
\end{pmatrix},
\begin{pmatrix}
C_{TL} = C_{00} \\
C_{BL} = C_{BL} \\
C_{BR} = C_{BR}
\end{pmatrix}
\]

Considering (10.2) and these repartionings, $D$ currently contains

\[
\begin{pmatrix}
B_L + B^T_L A_L + C_{TL} & * & * \\
C_{BL} & * & * \\
C_{BR} & * & *
\end{pmatrix}
= \begin{pmatrix}
C_{00} + A_0 B_0 + B_0 A_0 & * & * \\
C_{01} & \gamma_{11} & * \\
C_{20} & c_{21} & C_{22}
\end{pmatrix}
\]

After moving the double lines ahead, in preparation of the next iteration,

\[
\begin{pmatrix}
D_{TL} & D_{BL}^T & D_{BR}^T \\
D_{BL} & D_{00} & D_{01} \\
D_{BR} & D_{02} & D_{22}
\end{pmatrix},
\begin{pmatrix}
A_L & A_R \\
A_0 & a_1 & A_2
\end{pmatrix},
\begin{pmatrix}
B_L & B_R \\
B_0 & b_1 & B_2
\end{pmatrix},
\begin{pmatrix}
C_{TL} & * & * \\
C_{BL} & * & * \\
C_{BR} & * & *
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} & * & * \\
\gamma_{10} & \gamma_{11} & * \\
\gamma_{12} & c_{21} & C_{22}
\end{pmatrix}
\]

Thus the contents of $D$ must be updated like

\[
\begin{pmatrix}
C_{00} & * & * \\
\gamma_{10} & \gamma_{11} & * \\
\gamma_{12} & c_{21} & C_{22}
\end{pmatrix}
+ \begin{pmatrix}
A_0 & a_1 \end{pmatrix}^T \begin{pmatrix}
B_0 & b_1
\end{pmatrix} + \begin{pmatrix}
B_0 & b_1 \end{pmatrix}^T \begin{pmatrix}
A_0 & a_1
\end{pmatrix}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} + A_0^T B_0 + B_0^T A_0 & * & * \\
\gamma_{10} + a_1 b_1 & \gamma_{11} + a_1 b_1 & * \\
\gamma_{12} + a_1 b_1 & c_{21} & C_{22}
\end{pmatrix}
\]

We conclude that an unblocked algorithm that maintains the condition in (10.2) is given in Fig. 10.2 (left). Notice that the algorithm overwrites matrix $C$ with the result $A^T B + B^T A + C$. 

110
Algorithm 24 $C \leftarrow A^T B + B^T A + C$
(unblocked down-right moving
lazy w.r.t. $C$)

partition
$C \rightarrow \begin{pmatrix} C_{TL} & * \\ C_{BL} & C_{BR} \end{pmatrix}$
where $C_{TL}$ is $0 \times 0$.

partition
$A \rightarrow (A_L \parallel A_R)$
where $A_L$ has 0 columns.

partition
$B \rightarrow (B_L \parallel B_R)$
where $B_L$ has 0 columns.
do until $C_{BR}$ is $0 \times 0$

repartition
$\begin{pmatrix} C_{TL} & C_{TL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} C_0 & * & * \\ C_{10} & \gamma_{11} & * \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$
where $\gamma_{11}$ is a scalar

repartition
$(A_L \parallel A_R) \rightarrow (A_0 \parallel a_1 \mid A_2)$
where $a_1$ is a column

repartition
$(B_L \parallel B_R) \rightarrow (B_0 \parallel b_1 \mid B_2)$
where $b_1$ is a column

$c_{10} \leftarrow c_{10}^T + a_1^T B_0 + b_1^T A_0$
$\gamma_{11} \leftarrow \gamma_{11} + a_1^T b_1 + b_1^T a_1$

continue with
$\begin{pmatrix} C_{TL} & C_{TL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} C_0 & * & * \\ C_{10} & \gamma_{11} & * \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$
continue with
$(A_L \parallel A_R) \leftarrow (A_0 \parallel a_1 \parallel A_2)$
continue with
$(B_L \parallel B_R) \leftarrow (B_0 \parallel b_1 \parallel B_2)$
endo

Algorithm 25 $C \leftarrow A^T B + B^T A + C$
(blocked down-right moving
lazy w.r.t. $C$)

partition
$C \rightarrow \begin{pmatrix} C_{TL} & * \\ C_{BL} & C_{BR} \end{pmatrix}$
where $C_{TL}$ is $0 \times 0$.

partition
$A \rightarrow (A_L \parallel A_R)$
where $A_L$ has 0 columns.

partition
$B \rightarrow (B_L \parallel B_R)$
where $B_L$ has 0 columns.
do until $C_{BR}$ is $0 \times 0$

repartition
$\begin{pmatrix} C_{TL} & C_{TL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} C_0 & * & * \\ C_{10} & C_{11} & * \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$
where $C_{11}$ is a matrix $n \times n$.

repartition
$(A_L \parallel A_R) \rightarrow (A_0 \parallel A_1 \parallel A_2)$
where $A_1$ is a $m \times n$ matrix.

repartition
$(B_L \parallel B_R) \rightarrow (B_0 \parallel B_1 \parallel B_2)$
where $B_1$ is a $m \times n$ matrix.

$c_{10} \leftarrow c_{10} + A_1^T B_0 + B_1^T A_0$
$c_{11} \leftarrow c_{11} + A_1^T B_1 + B_1^T A_1$

continue with
$\begin{pmatrix} C_{TL} & C_{TL}^T \\ C_{BL} & C_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} C_0 & * & * \\ C_{10} & C_{11} & * \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$
continue with
$(A_L \parallel A_R) \leftarrow (A_0 \parallel A_1 \parallel A_2)$
continue with
$(B_L \parallel B_R) \leftarrow (B_0 \parallel B_1 \parallel B_2)$
endo

Figure 10.2: Unblocked (left) and blocked (right) down right lazy syrk update algorithms.
Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

\[
\begin{pmatrix}
D_{TL} & D_{TL}^T \\
D_{BL} & D_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
D_{00} & D_{10} & * \\
D_{10}^T & D_{11} & * \\
D_{20} & D_{21} & D_{22}
\end{pmatrix}
\]

where \( D_{11} \) is a matrix of \( b \times b \) rows. Similarly, we repartition

\[
\begin{pmatrix}
C_{TL} & C_{TL}^T \\
C_{BL} & C_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{00} & C_{00} \\
C_{10} & C_{11} \\
C_{20} & C_{21} \\
C_{21} & C_{22}
\end{pmatrix}
\]

where \( C_{11} \) is a matrix of size \( b \times b \). continuing to partition:

\[
( A_L \parallel A_R ) \rightarrow ( A_0 \parallel A_1 \parallel A_2 ) \quad \text{and} \quad ( B_L \parallel B_R ) \rightarrow ( B_0 \parallel B_1 \parallel B_2 )
\]

where \( A_1 \) and \( B_1 \) are \( b \times n \) matrices. Again, the double lines have meaning:

\[
( A_L = A_0 \parallel A_R = ( A_1 \parallel A_2 ) )
\]

\[
( B_L = B_0 \parallel B_R = ( B_1 \parallel B_2 ) ) \quad \text{and} \quad \begin{pmatrix}
C_{TL} = C_{00} \\
C_{BL} = \begin{pmatrix} C_{10} \\ C_{20} \end{pmatrix} \\
C_{BR} = \begin{pmatrix} C_{11} \\ C_{21} \\ C_{22} \end{pmatrix}
\end{pmatrix}
\]

Considering (10.2) and these repartitions, \( D \) currently contains

\[
\begin{pmatrix}
A_L^T B_L + B_L^T A_L + C_{TL} \\
C_{BL}
\end{pmatrix}
\]

\[
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix}
\]

After moving the double lines ahead, in preparation of the next iteration,

\[
\begin{pmatrix}
D_{TL} & D_{TL}^T \\
D_{BL} & D_{BL}^T
\end{pmatrix}
\rightarrow
\begin{pmatrix}
D_{00} & D_{10} & D_{11} & * \\
D_{10}^T & D_{11}^T & * & * \\
D_{20} & D_{21} & D_{22}
\end{pmatrix}
\]

( \( A_L \parallel A_R \rightarrow ( A_0 \parallel A_1 \parallel A_2 ) \),

( \( B_L \parallel B_R \rightarrow ( B_0 \parallel B_1 \parallel B_2 ) \)

\[
\begin{pmatrix}
C_{TL} \\
C_{BL}
\end{pmatrix}
\]

\[
\begin{pmatrix}
* \\
C_{BR}
\end{pmatrix}
\]

Thus the contents of \( D \) must be updated to contain

\[
\begin{pmatrix}
( C_{00} + A_0^T B_0 + B_0^T A_0 ) \\
( C_{10} + A_1^T B_0 + B_0^T A_1 ) \\
( C_{20} + A_2^T B_0 + B_0^T A_2 )
\end{pmatrix}
\]

\[
\begin{pmatrix}
* \\
( C_{11} + A_1^T B_1 + B_1^T A_1 ) \\
( C_{21} + A_2^T B_1 + B_1^T A_2 ) \\
( C_{22} + A_2^T B_2 + B_2^T A_2 )
\end{pmatrix}
\]

Hence the contents of \( D \) must be updated like

\[
\begin{pmatrix}
C_{00} + A_0^T B_0 + B_0^T A_0 \\
C_{10} + A_1^T B_0 + B_0^T A_1 \\
C_{20} + A_2^T B_0 + B_0^T A_2
\end{pmatrix}
\]

\[
\begin{pmatrix}
* \\
C_{11} + A_1^T B_1 + B_1^T A_1 \\
C_{21} + A_2^T B_1 + B_1^T A_2 \\
C_{22} + A_2^T B_2 + B_2^T A_2
\end{pmatrix}
\]

We conclude that a blocked algorithm that maintains the condition in (10.2) is given in Fig. 10.2 (right). Notice that the algorithm overwrites matrix \( C \) with the result \( A^T B + B^T A + C \).
10.2 Implementation

These codes, and related test routines, can be found at

http://www.cs.utexas.edu/users/flame/materials/syr2k_lt/

10.3 Performance
```c
#include "FLAME.h"

int Syr2k_l_t_downright_wrt_C_unb ( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
    FLA_Obj CTX, CTR, C00, c01, c02, AL, A0, BL, BO,
    CEL, CER, c10, c11, c12, AR, a1, BR, b1,
    C20, C21, C22, A2, E2;

    /* Partition C into a 2X2 */
    FLA_Part_2x2( C, &CTX, /** &CTR,
                  /* ************ */
                      &CEL, /** &CER,
                      0, 0, /* submatrix */ FLA_TL );

    /* Partition A & B into 1X2 */
    FLA_Part_1x2( A, &AL, /** &AR,
                  0, /* width submatrix */ FLA_LEFT );

    FLA_Part_1x2( B, &BL, /** &BR,
                  0, /* width submatrix */ FLA_LEFT );

    while (FLA_Obj_length(CER)!=0) {
        FLA_Repart_2x2_to_3x3( CTX, /** CTR, C00, /** c01, c02,
                               /* ************ */
                               /** c10, /** c11, c12,
                               CEL, /** CER, C20, /** C21, C22,
                               1, 1, /* C11 from */ FLA_BR );

        FLA_Repart_1x2_to_1x3( AL, /** AR, A0, /** a1, A2,
                               1, /* width al from */ FLA_RIGHT );

        FLA_Repart_1x2_to_1x3( BL, /** BR, BO, /** b1, B2,
                               1, /* width bi from */ FLA_RIGHT );

        /* ************ */
        FLA_Gemv( FLA_TRANSPOSE, ONE, BO, a1, ONE, c10 );
        FLA_Gemv( FLA_TRANSPOSE, ONE, A0, b1, ONE, c10 );
        FLA_Dot_x( ONE,a1,b1,ONE,c11);
        FLA_Dot_x( ONE,b1,a1,ONE,c11);

        /* ************ */
        FLA_Cont_with_3x3_to_2x2( &CTX, /** &CTR, C00, c01, /** C02,
                                  /** c10, c11, /** C12,
                                  /** &CEL, /** &CER, C20, C21, /** C22,
                                  /* C11 added to */ FLA_TL );

        FLA_Cont_with_1x3_to_1x2( &AL, /** &AR, A0, a1, /** A2,
                                  /* al added to */ FLA_LEFT );

        FLA_Cont_with_1x3_to_1x2( &BL, /** &BR, BO, b1, /** B2,
                                  /* bi added to */ FLA_LEFT );
    }
}
```

Figure 10.3: Unblocked down-right moving (w.r.t. C) symmetric rank-2k update algorithm using FLAME.
#include "FLAME.h"

int Syr2k_lt_downright_wrt_C_blk ( int rec, FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
{
    FLA_Obj CTL, CTR, C00, C01, C02, A0, B0, C10, C11, C12, A1, B1, C20, C21, C22, A2, B2, E2;

    int b;

    FLA_Part_2x2( C, &CTL, /**< &CTR,
                  /* ************** */
                  &C00, /**< &C01, &C02,
                  /* ************** */
                  &C10, /**< &C11, &C12,
                  /* */ &CER,
                    0, 0, /**< submatrix */ FLA_TL );

    FLA_Part_1x2( A, &AL, /**< &AR,
                  0, /**< width submatrix */ FLA_LEFT );

    FLA_Part_1x2( B, &BL, /**< &BR,
                  0, /**< width submatrix */ FLA_LEFT );

    while ( FLA_Obj_length(CER) != 0 )
    {
        b = min( FLA_Obj_length(CER), nb_alg );

        FLA_Report_2x2_to_3x3( CTL, /**< CTR,
                               /* ************** */
                               &C00, /**< &C01, &C02,
                               /* ************** */
                               &C10, /**< &C11, &C12,
                               /* */ &CER,
                                 b, b, /**< C11 from */ FLA_BR );

        FLA_Report_1x2_to_1x3( A, /**< AR,
                               &A0, /**< &A1, &A2,
                               b, /**< width ai from */ FLA_RIGHT );

        FLA_Report_1x2_to_1x3( B, /**< BR,
                               &B0, /**< &B1, &B2,
                               b, /**< width bi from */ FLA_RIGHT );

        /* ************************************************** */

        FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A1, B0, ONE, C10 );

        FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, B1, A0, ONE, C10 );

        if ( rec == FLA_RECRSIVE && b > 8 )
            Syr2k_lt_downright_wrt_C_blk( rec, A1, B1, C11, nb_alg/2 );
        else
            Syr2k_lt_downright_wrt_C_unb( A1, B1, C11 );

        /* ************************************************** */

        FLA_Cont_with_3x3_to_2x2( &CTL, /**< &CTR,
                                   /* */ &C00, C01, /**< C02,
                                   /* */ &C10, C11, /**< C12,
                                   /* */ &CER,
                                   /* */ &C20, C21, /**< C22,
                                   /* */ C11 added to */ FLA_TL );

        FLA_Cont_with_1x3_to_1x2( &AL, /**< &AR,
                                    &A0, A1, /**< A2,
                                    /* A1 added to */ FLA_LEFT );

        FLA_Cont_with_1x3_to_1x2( &BL, /**< &BR,
                                    &B0, B1, /**< B2,
                                    /* B1 added to */ FLA_LEFT );
    }
}

Figure 10.4: Blocked down-right moving (w.r.t. C) symmetric rank-2k update algorithm using FLAME.
Figure 10.5: Performance of a down-right moving (w.r.t. C) symmetric rank-k update algorithm for a block size of 32 (left) and 128 (right). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.