Chapter 5

Triangular Matrix Multiplication

\( B \leftarrow LB \)

In this chapter, we revisit in more depth the implementation of the triangular matrix-matrix multiplication

\( B \leftarrow LB \)

where \( L \) is an \( m \times m \) lower triangular matrix and \( B \) is \( m \times n \). We start by deriving a number of different sequential algorithms. Subsequently, we show how to code the algorithms using FLAME.

Let us consider

(5.1) \[ D = LB \]

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

5.1 Algorithms that start by splitting \( L \)

Partition

\( L \rightarrow \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \)

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

\( D \rightarrow \begin{pmatrix} D_T \\ D_B \end{pmatrix} \), and \( B \rightarrow \begin{pmatrix} B_T \\ B_B \end{pmatrix} \)

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

Now, (5.1) becomes

\[ \begin{pmatrix} D_T \\ D_B \end{pmatrix} = \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \begin{pmatrix} B_T \\ B_B \end{pmatrix} \]

Multiplying out the right-hand-side yields

\[ \begin{pmatrix} D_T \\ D_B \end{pmatrix} = \begin{pmatrix} L_{TL}B_T \\ L_{BL}B_T + L_{BR}B_B \end{pmatrix} \]

This in turn exposes the equalities

\[ D_T = L_{TL}B_T \]
\[ D_B = L_{BL}B_T + L_{BR}B_B \]

which must hold.

Possible contents of \( D \) at this intermediate step are given in Fig. 5.1. Considering the comments in that table, only two viable conditions are left, the ones for which there are no comments.
\[
\begin{array}{|c|c|c|}
\hline
D \text{ contains} & \text{Comments} & \text{Viable?} \\
\hline
\left( \frac{\ast}{\ast} \right) & \text{This condition indicates no progress has been made.} & \text{NO} \\
\hline
\left( \frac{L_{TL}B_T}{\ast} \right) & \text{Since } B_T \text{ is to be overwritten by } D_T, \text{ this condition is not feasible since } B_T \text{ is still needed for the computation } L_{BL}B_T. & \text{NO} \\
\hline
\left( \frac{\ast}{L_{BL}B_T} \right) & \text{Since } B_T \text{ is to be overwritten by } D_T, \text{ this condition is not feasible since } B_T \text{ is still needed for the computation } L_{BL}B_T. & \text{NO} \\
\hline
\left( \frac{L_{TL}B_T}{L_{BL}B_T} \right) & \text{Since } B_T \text{ is to be overwritten by } D_T, \text{ this condition is not feasible since } B_T \text{ is still needed for the computation } L_{BR}B_T. & \text{NO} \\
\hline
\left( \frac{\ast}{L_{BR}B_T} \right) & \text{This condition indicates that the computation has completed.} & \text{YES} \\
\hline
\end{array}
\]

Figure 5.1: Possible contents of \( D \) at an intermediate step.

5.1.1 Row-lazy algorithm (relative to \( L \))

Consider the condition that currently

\[(5.2) \quad D \text{ contains } \left( \frac{L_{BL}B_T + L_{BR}B_T}{\ast} \right) \]

Notice that this indicates that \( L_{BL} \) and \( L_{BR} \) have been used to update \( D \), which will call a row-lazy algorithm relative to operand \( L \).

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

Unblocked algorithm

Repartition

\[(5.3) \quad \left( \frac{D_T}{D_B} \right) \rightarrow \left( \frac{D_0}{d_1^T} \frac{d_2^T}{D_2} \right) \]

where \( d_1^T \) is a row. Similarly, we repartition

\[(5.4) \quad \left( \frac{B_T}{B_B} \right) \rightarrow \left( \frac{B_0}{b_1^T} \frac{b_2^T}{B_2} \right) \quad \text{and} \quad \left( \begin{array}{c|c|c}
L_{TL} & 0 & 0 \\
L_{BL} & L_{BR} & \end{array} \right) \rightarrow \left( \begin{array}{c|c|c|c}
L_{00} & 0 & 0 \\
L_{10} & \lambda_{11} & 0 \\
L_{20} & l_{21} & L_{22} \\
\end{array} \right) \]

where \( b_1^T \) is a row and \( \lambda_{11} \) is a scalar.

Notice that the double lines have meaning:

\[(5.5) \quad \left( \frac{D_T = \left( \frac{D_0}{d_1^T} \right)}{D_B = D_2} \right), \left( \frac{B_T = \left( \frac{B_0}{b_1^T} \right)}{B_B = B_2} \right) \quad \text{and} \quad \left( \frac{L_{TL} = \left( \frac{L_{00}}{L_{10}} \frac{\lambda_{11}}{\lambda_{11}} \right)}{L_{BL} = \left( \frac{L_{20}}{l_{21}} \right)} \right) \frac{L_{BR} = L_{22}}{\}

51
Considering (5.2) and these repartitionings, \( D \) currently contains

\[
\left( \frac{\star}{L_{BL}B_T + L_{BR}B_B} \right) = \left( \frac{\star}{L_{20} | l_{21}} \right) \left( \frac{b_0}{b_1^T} + L_{22}B_2 \right) = \left( \frac{\star}{l_{21}b_1^T + L_{22}B_2} \right)
\]

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

\[
\begin{align*}
(5.6) \quad & \frac{D_T = D_0}{D_B = \left( \frac{d_1^T}{D_2} \right)} \quad , \quad \frac{B_T = B_0}{B_B = \left( \frac{b_1^T}{B_2} \right)} \quad \text{and} \quad \left( \frac{L_{TL} = L_{00}}{L_{BL} = \left( \frac{t_1^T}{L_{20}} \right)} \right) \left( \frac{0}{L_{BR} = \begin{pmatrix} \lambda_1 & 0 \\ l_{21} & L_{22} \end{pmatrix}} \right) \\
\end{align*}
\]

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

\[
\begin{align*}
& \left( \frac{\star}{L_{20}B_0 + l_{21}b_1^T + L_{22}B_2} \right) \rightarrow \left( \frac{\star}{L_{20}B_0 + l_{21}b_1^T + \lambda_1 b_1^T + L_{22}B_2} \right) \\
\end{align*}
\]

Thus we conclude that an algorithm that maintains the condition in (5.2) is given in Fig. 5.2 (left). Notice that the algorithm overwrites matrix \( B \) with the result \( LB \).

**Theorem 1 (Cost of Algorithm 4)** Given that matrices \( B \) and \( L \) are \( m \times n \) and \( m \times m \), respectively, the unblocked row-lazy triangular matrix-matrix multiplication algorithm given in Fig. 5.2 (left) is \( m^2n \) floating point operations.

**Proof:** Consider the algorithm in Fig. 5.2 (right). The number of rows in \( B_B \) increases from 0 to \( m - 1 \) while \( L_{BR} \) increases from 0 to \((m - 1) \times (m - 1) \). Assuming \( B_B \) currently has \( k \) rows, and \( L_{BR} \) is thus currently \( k \times k \), the different parts of the matrices have the following dimensions:

\[
\begin{align*}
\bar{m} \{ & \begin{array}{c} B_0 \\ b_1^T \\ B_2 \end{array} \end{align*} \quad \text{and} \quad \begin{align*}
\bar{n} \{ & \begin{array}{c} L_{00} \\ t_1^T \\ L_{20} \end{array} \end{align*} \quad \text{and} \quad \begin{align*}
\bar{k} \{ & \begin{array}{c} \lambda_1 \\ 0 \\ l_{21} \end{array} \end{align*} \quad \text{with} \quad \bar{m} = m - k - 1 \quad \text{and} \quad \bar{n} = n - m + 1 \quad \text{and} \quad \bar{k} = k - 1
\]

The number of floating point operations for the updates in the loop are now given by

\[
\begin{align*}
b_1^T & \leftarrow \lambda_1 b_1^T \\
b_1^T & \leftarrow l_{t1}^T B_0 + b_1^T \quad 2(m - k - 1)n
\end{align*}
\]

since the first operation scales a row vector of length \( n \) and the vector-matrix multiply \( l_{t1}^T B_0 + b_1^T \) involves a matrix of size \((m - k - 1) \times n \). Thus, the total cost of the triangular matrix multiplication is given by

\[
\begin{align*}
\sum_{k=0}^{m-1} [n + 2(m - k - 1)n] &= \sum_{k=0}^{m-1} n + 2 \sum_{k=0}^{m-1} (m - k - 1)n \\
&= mn + 2 \sum_{k=0}^{m-1} kn = mn + 2 \frac{(m - 1)m}{2} n = mn + (m - 1)mn = m^2n
\end{align*}
\]

This proves the theorem. \( \square \)

**Blocked algorithm**

In order to derive a blocked algorithm, instead repartition like

\[
\begin{align*}
\left( \frac{D_T}{D_B} \right) & \rightarrow \left( \frac{D_0}{D_1} \right) \end{align*}
\]

52
Algorithm 4 $B \leftarrow LB$
\(\text{unblocked row-lazy w.r.t. } L\)
\[
\text{partition} \\
B \rightarrow \left( \frac{B_T}{B_B} \right) \\
\text{where } B_B \text{ has 0 rows}
\]
\[
\text{partition} \\
L \rightarrow \left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \\
\text{where } L_{BR} \text{ is } 0 \times 0
\]
do until $L_{TL}$ is $0 \times 0$
\[
\text{repartition} \\
\left( \frac{B_T}{B_B} \right) \rightarrow \left( \frac{B_0}{b^T_1} \right) \\
\text{where } b^T_1 \text{ is a row}
\]
\[
\text{repartition} \\
\left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \rightarrow \left( \frac{L_{00}}{L_{10}} \mid \frac{\lambda_{11}}{L_{20}} \mid \frac{0}{L_{22}} \right) \\
\text{where } \lambda_{11} \text{ is a scalar}
\]
\[
\begin{align*}
b^T_1 &\leftarrow \lambda_{11} b^T_1 \\
b^T_2 &\leftarrow l^T_{10} B_0 + b^T_1
\end{align*}
\]
\[
\text{continue with} \\
\left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \left( \frac{L_{00}}{L_{10}} \mid \frac{\lambda_{11}}{L_{20}} \mid \frac{0}{L_{22}} \right)
\]
\[
\text{continue with} \\
\left( \frac{B_T}{B_B} \right) \left( \frac{B_0}{b^T_1} \right) \left( \frac{B_1}{b^T_2} \right)
\]
enddo

Algorithm 5 $B \leftarrow LB$
\(\text{blocked row-lazy w.r.t. } L\)
\[
\text{partition} \\
B \rightarrow \left( \frac{B_T}{B_B} \right) \\
\text{where } B_B \text{ has 0 rows}
\]
\[
\text{partition} \\
L \rightarrow \left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \\
\text{where } L_{BR} \text{ is } 0 \times 0
\]
do until $L_{TL}$ is $0 \times 0$
\[
\text{determine block size } b \\
\text{repartition} \\
\left( \frac{B_T}{B_B} \right) \rightarrow \left( \frac{B_0}{B_1} \right) \\
\text{where } B_1 \text{ has } b \text{ rows}
\]
\[
\text{repartition} \\
\left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \rightarrow \left( \frac{L_{00}}{L_{10}} \mid \frac{L_{11}}{L_{20}} \mid \frac{0}{L_{22}} \right) \\
\text{where } L_{11} \text{ is } b \times b
\]
\[
\begin{align*}
B_1 &\leftarrow L_{11} B_1 \\
B_1 &\leftarrow L_{10} B_0 + B_1
\end{align*}
\]
\[
\text{continue with} \\
\left( \frac{L_{TL}}{L_{BL}} \mid \frac{0}{L_{BR}} \right) \rightarrow \left( \frac{L_{00}}{L_{10}} \mid \frac{L_{11}}{L_{20}} \mid \frac{0}{L_{22}} \right)
\]
\[
\text{continue with} \\
\left( \frac{B_T}{B_B} \right) \rightarrow \left( \frac{B_0}{B_1} \right) \left( \frac{B_1}{B_2} \right)
\]
enddo

Figure 5.2: Unblocked (left) and blocked (right) row-lazy triangular matrix-matrix multiplication algorithms.
where $D_1$ is a block of $b$ rows. Similarly, we repartition

$$
\begin{pmatrix}
B_T \\
B_B
\end{pmatrix}
\rightarrow
\begin{pmatrix}
B_0 \\
B_1 \\
B_2
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
L_{TL} \\
L_{BL} \\
L_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
L_{00} & 0 & 0 \\
L_{10} & L_{11} & 0 \\
L_{20} & L_{21} & L_{22}
\end{pmatrix}
$$

where $B_1$ is a block of $b$ rows and $L_{11}$ is $b \times b$.

Again, the double lines have meaning:

$$
(5.7) \quad \left( \frac{D_T = \begin{pmatrix} D_0 \\ D_1 \end{pmatrix}}{D_B = D_2} \right), \quad \left( \frac{B_T = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}}{B_B = B_2} \right) \quad \text{and} \quad \left( \frac{L_{TL} = \begin{pmatrix} L_{00} \\ L_{10} \\ L_{20} \end{pmatrix}}{L_{BL} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}} \right) = \left( \frac{0}{L_{BR} = L_{22}} \right)
$$

Considering (5.2) and these repartitionings, $D$ currently contains

$$
\left( \frac{L_{BL}B_T + L_{BR}B_B}{*} \right) = \left( \frac{*}{(L_{20} \mid L_{21}) \left( \frac{B_0}{B_1} \right) + L_{22}B_2} \right) = \left( \frac{*}{L_{20}B_0 + L_{21}B_1 + L_{22}B_2} \right)
$$

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

$$
\left( \frac{D_T = D_0}{D_B = \begin{pmatrix} D_1 \\ D_2 \end{pmatrix}} \right), \quad \left( \frac{B_T = B_0}{B_B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}} \right) \quad \text{and} \quad \left( \frac{L_{TL} = L_{00}}{L_{BL} = \begin{pmatrix} L_{10} \\ L_{20} \end{pmatrix}} \right) = \left( \frac{0}{L_{BR} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} \mid L_{22}} \right)
$$

Thus the contents of $D$ must be updated like

$$
\left( \frac{L_{BL}B_0 + L_{21}B_1 + L_{22}B_2}{*} \right) \rightarrow \left( \frac{\frac{\star}{L_{20}B_0 + L_{21}B_1 + L_{22}B_2}}{L_{10}B_0 + L_{11}B_1 + L_{22}B_2} \right)
$$

Thus we conclude that an algorithm that maintains the condition in (5.2) is given in Fig. 5.2 (right). Notice that the algorithm overwrites matrix $B$ with the result $LB$.

**Theorem 2 (Cost of Algorithm 5)** Given that matrices $B$ and $L$ are $m \times n$ and $m \times m$, respectively, the blocked row-lazy triangular matrix-matrix multiplication algorithm given in Fig. 5.2 (right) is $m^2n$ floating point operations.

**Proof:** We will only prove the theorem for the case where the block size $b$ is always the same and $m$ is an integer multiple of $b$.

Consider the algorithm in Fig. 5.2 (right). The number of rows in $B_B$ increases from 0 to $m - b$ while $L_{BR}$ increases from $0 \times 0$ to $(m - b) \times (m - b)$. Assuming $B_B$ currently has $k$ rows, and $L_{BR}$ is thus currently $k \times k$, the different parts of the matrices have the following dimensions:

$n$ \quad \begin{pmatrix} B_0 \\ L_{00} \end{pmatrix} \quad \begin{pmatrix} \bar{m} \\ \lambda_0 \end{pmatrix} \quad \begin{pmatrix} b \\ \lambda_1 \end{pmatrix} \quad k$\quad $\bar{m} = m - k - b$

$b$ \quad \begin{pmatrix} b_1' \\ L_{10} \end{pmatrix} \quad \begin{pmatrix} \bar{b} \\ \lambda_1 \end{pmatrix} \quad \begin{pmatrix} \bar{b} \\ \lambda_2 \end{pmatrix} \quad k$

$k$ \quad \begin{pmatrix} B_2 \\ L_{20} \end{pmatrix} \quad \begin{pmatrix} \bar{k} \\ \lambda_2 \end{pmatrix} \quad \begin{pmatrix} \bar{k} \\ \lambda_2 \end{pmatrix} \quad k$

The number of floating point operations for the updates in the loop are now given by

\[ B_1 \leftarrow L_{11}B_1 \]
\[ B_1 \leftarrow L_{10}B_0 + B_1 \]
\[ 2(m - k - b)bn \]

since the first operation is a triangular matrix multiplication, which can be accomplished by executing Alg. 4, and the second is a matrix-matrix multiplication of an $b \times (m - b)$ matrix times an $b \times n$ matrix. Notice that

54
\( k = ib \) for some \( i \), and \( i \) ranges from 0 to \( \frac{m}{b} - 1 \). Thus, the total cost of the triangular matrix multiplication is given by

\[
\sum_{i=0}^{\frac{m}{b}-1} [b^2n + 2(m - ib - b)bn] = \sum_{i=0}^{\frac{m}{b}-1} b^2n + 2 \sum_{i=0}^{\frac{m}{b}-1} (m - ib - b)bn
\]

\[
= \frac{m}{b}b^2n + 2 \sum_{i=0}^{\frac{m}{b}-1} (ib)n = mn + 2 \left( \frac{m}{b} - 1 \right) \frac{m}{b}b^2n = mn + (m - b)mn = m^2n
\]

This proves the theorem. \( \square \)

### 5.1.2 Lazy algorithm (relative to \( L \))

Next, consider the condition that currently

\[
D \text{ contains } \left( \frac{\ast}{L_{BR}B_B} \right)
\]

Notice that this indicates that only \( L_{BR} \) has been used to update \( D \), which will will call a lazy algorithm relative to operand \( L \).

Again, this algorithm naturally moves through matrices \( D \) and \( B \) in the “up” direction.

#### Unblocked algorithm

Repartition again as in Eqns. (5.3) and (5.4) Again, the double lines have the same meaning as in (5.5). Considering (5.8) and these repartitionings, we find that \( D \) currently contains

\[
\left( \frac{\ast}{L_{BR}B_B} \right) = \left( \frac{\ast}{L_{22}B_2} \right)
\]

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the meaning in (5.6) and \( D \) must be updated to contain

\[
\left( \frac{\ast}{L_{BR}B_B} \right) \rightarrow \left( \frac{\ast}{\lambda_{11} b_1' + L_{22} B_2} \right)
\]

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

\[
\left( \frac{\ast}{L_{22} B_2} \right) \rightarrow \left( \frac{\ast}{\lambda_{11} b_1' + L_{22} B_2} \right)
\]

We conclude that an algorithm that maintains the condition in (5.8) is given in Fig. 5.3 (left). Again, the algorithm overwrites matrix \( B \) with the result \( LB \).

#### Blocked algorithm

We leave the derivation of the lazy unblocked algorithm, given in Fig. 5.3 (right), as an exercise for the reader.

### 5.2 Algorithms that start by splitting \( B \)

Alternatively, we can start by partitioning matrix \( B \).
Algorithm 6 $B \leftarrow LB$
(unblocked lazy w.r.t. $L$)

**partition**

$B \rightarrow \left( \frac{B_T}{B_B} \right)$

where $B_B$ has 0 rows

**partition**

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

where $L_{BR}$ is $0 \times 0$

**do until** $L_{TL}$ is $0 \times 0$

**repartition**

$\left( \begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c} L_{00} & 0 \\ \hline l_{10} & \lambda_{11} \\ \hline l_{20} & l_{21} \\ \hline \end{array} \right)$

where $\lambda_{11}$ is a scalar

$B_2 \leftarrow l_{21}b_1^T + B_2$

$b_1^T \leftarrow \lambda_{11}b_1^T$

continue with

$\left( \begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c} L_{00} & 0 \\ \hline l_{10} & \lambda_{11} \\ \hline l_{20} & l_{21} \\ \hline \end{array} \right)$

continue with

$\left( \begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c} L_0 \hline B_T \\ \hline B_B \end{array} \right)$

enddo

Algorithm 7 $B \leftarrow LB$
(blocked lazy w.r.t. $L$)

**partition**

$B \rightarrow \left( \frac{B_T}{B_B} \right)$

where $B_B$ has 0 rows

**partition**

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

where $L_{BR}$ is $0 \times 0$

**do until** $L_{TL}$ is $0 \times 0$

**determine block size $b$**

**repartition**

$\left( \begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \\ \hline L_{20} & L_{21} \\ \hline \end{array} \right)$

where $L_{11}$ is $b \times b$

$B_2 \leftarrow L_{21}B_1 + B_2$

$B_1 \leftarrow L_{11}B_1$

continue with

$\left( \begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \\ \hline L_{20} & L_{21} \\ \hline \end{array} \right)$

continue with

$\left( \begin{array}{c|c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left( \begin{array}{c|c} B_0 \\ \hline B_1 \\ \hline \end{array} \right)$

enddo
First, consider partitioning $B$ by rows:

$$B = \begin{pmatrix} B_T \\ B_B \end{pmatrix}$$

where $B_T$ has $k$ rows. In order to partition $L$ conformally, we find that

$$L = \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix}$$

where $L_{TL}$ is $k \times k$. Notice that we use a $2 \times 2$ partitioning in order to expose the block of zeroes. Clearly, this partitioning of $B$ leads to the same algorithms as were derived in Section 5.1.

First, consider partitioning $B$ by columns:

$$B = \begin{pmatrix} B_L & B_R \end{pmatrix}$$

where $B_L$ has $k$ columns. Notice that this time $L$ is not partitioned at all and

$$D = \begin{pmatrix} D_L & D_R \end{pmatrix}$$

Now, (5.1) becomes

$$\begin{pmatrix} D_L & D_R \end{pmatrix} = L \begin{pmatrix} B_L & B_R \end{pmatrix}$$

Multiplying out the right-hand-side yields

$$\begin{pmatrix} D_L & D_R \end{pmatrix} = \begin{pmatrix} LB_L & LB_R \end{pmatrix}$$

This in turn exposes the equalities

$$D_L = LB_L \quad \text{and} \quad D_R = LB_R$$

which must hold.

Possible contents of $D$ at this intermediate step are given by

<table>
<thead>
<tr>
<th>$D$ contains</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>($\ast \parallel \ast$)</td>
<td>This indicates no progress has been made. Clearly not a desirable state.</td>
</tr>
<tr>
<td>($LB_L \parallel \ast$)</td>
<td>This indicates that the computation has finished. Clearly not an interesting intermediate condition.</td>
</tr>
<tr>
<td>($LB_L \parallel LB_R$)</td>
<td></td>
</tr>
<tr>
<td>($\ast \parallel LB_R$)</td>
<td></td>
</tr>
</tbody>
</table>

5.2.1 Right-moving (lazy) algorithm

Let us consider the condition that currently

$$D \quad \text{contains} \quad \begin{pmatrix} LB_L \parallel \ast \end{pmatrix}$$

Notice that since $B_L$ will not be used again, and no other part of $B$ has been used, we will call this algorithm lazy (w.r.t. $B$). Clearly, to expand the part of $D$ that has been computed, we will have to move right within $D$ and $B$. Thus, we call it a right-moving algorithm (w.r.t. $B$).

Unblocked algorithm

Repartition

$$\begin{pmatrix} D_L \parallel D_R \end{pmatrix} \rightarrow \begin{pmatrix} D_0 \parallel d_1 \parallel D_2 \end{pmatrix}$$

where $d_1$ is a column. Similarly, we repartition

$$\begin{pmatrix} B_L \parallel B_R \end{pmatrix} \rightarrow \begin{pmatrix} B_0 \parallel b_1 \parallel B_2 \end{pmatrix}$$
where \( b_1 \) is a column. Notice that

\[
(5.10) \quad \left( D_L = D_0 \parallel D_R = ( d_1 \mid D_2 ) \right) \quad \text{and} \quad \left( B_L = B_0 \parallel B_R = ( b_1 \mid B_2 ) \right)
\]

Thus \( D \) currently contains

\[
( LB_0 \parallel ( \star \mid \star ) )
\]

After moving the double lines ahead,

\[
( D_L = ( D_0 \mid d_1 ) \parallel D_R = D_2 ) \quad \text{and} \quad ( B_L = ( B_0 \mid b_1 ) \parallel B_R = B_2 )
\]

and thus \( D \) must hold

\[
( L ( B_0 \mid b_1 ) \parallel \star ) = ( LB_0 \parallel Lb_1 ) \parallel \star)
\]

Thus, the contents of \( D \) must be updated like

\[
( LB_0 \parallel ( \star \mid \star ) ) \rightarrow ( LB_0 \parallel Lb_1 ) \parallel \star)
\]

This motivates the algorithm given in Fig. 5.4 (left).

**Algorithm 8** \( B \leftarrow LB \)

**(unblocked right-moving)**

**(lazy) w.r.t. \( B \)**

**partition**

\( B \rightarrow ( B_L \parallel B_R ) \)

where \( B_L \) has 0 columns

**do until** \( B_R \) has 0 columns

**repartition**

\( ( B_L \parallel B_R ) \rightarrow ( B_0 \parallel b_1 \mid B_2 ) \)

where \( b_1 \) is a column

\( b_1 \leftarrow Lb_1 \)

**continue with**

\( ( B_L \parallel B_R ) \leftarrow ( B_0 \mid b_1 \parallel B_2 ) \)

**endo**

**Algorithm 9** \( B \leftarrow LB \)

**(blocked right-moving)**

**(lazy) w.r.t. \( B \)**

**partition**

\( B \rightarrow ( B_L \parallel B_R ) \)

where \( B_L \) has 0 columns

**do until** \( B_R \) has 0 columns

**determine block size \( b \)**

**repartition**

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

where \( B_1 \) has \( b \) columns

\( B_1 \leftarrow LB_1 \)

**continue with**

\( ( B_L \parallel B_R ) \leftarrow ( B_0 \mid B_1 \parallel B_2 ) \)

**endo**

Figure 5.4: Unblocked (left) and blocked (right) right-moving (lazy) triangular matrix-matrix multiplication algorithms.

**Blocked algorithm**

We leave the derivation of the blocked algorithm given in Fig. 5.4 as an exercise for the reader.

**5.2.2 Left-moving (lazy) algorithm**

We leave the derivation of left-moving unblocked and blocked algorithms to the reader.

**5.3 Implementation**

The sequential implementations for the lazy, row-lazy, and right-moving algorithms using FLAME are given in Figs. 5.5–5.8. These codes, and corresponding test routines, can be found at

\[ \text{http://www.cs.utexas.edu/users/flame/materials/trmm_lln/} \]
```c
#include "FLAME.h"

void Trmm_left_lower_no_trans_unb( int variant, FLA_Obj A, FLA_Obj B )
{
    FLA_Obj ATL, ATR, A00, a01, A02, B0, AEL, ABR, a10t, alpha11, a12t, BB, bit,
    A20, a21, A22, B2;

    int b;

    FLA_Part_2x2( A, &ATL, //** ATR, 
    /* ****************** */
        &AEL, /*&ABR, 0, 0, */ submatrix /* FLA_BR */;
    FLA_Part_2x1( B, &BT,
    /***/
        &BB, 0, /* length submatrix */ FLA_BOTTOM);

    while ( 0 != FLA_Obj_length( ATR ) ){
        FLA_Part_2x2_to_3x3( ATL, //** ATR,
            &A00, //** a01, //** //A02,
            /***/
                &a10t, &alpha11, /*&a12t,
                /* ****************** */
                    AEL, /*&ABR, &a20, //** //A22,
                    i, /* alpha11 from */ FLA_TL );

        FLA_Part_2x1_to_3x1( BT, &B0,
            &bit,
                /***/
                    BB, &Ｂ2,
                        i, /* length bit from */ FLA_TUP );

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

        if ( variant == FLA_VARIANT.LAZY )
            FLA_Ger( ONE, a21, bit, B2 );
        if ( variant == FLA_VARIANT_ROW_LAZY )
            FLA_Gemv( FLA_TRANSPOSE, ONE, B0, a10t, ONE, bit );

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

        FLA_Cut_with_3x3_to_2x2( &ATL, //** &ATR, A00, /*&a01, A02,
            /* ****************** */
                a10t, //&alpha11, a12t,
                    &AEL, /*&ABR, A20, //** //A22,
                        /* alpha11 added to */ FLA_BR );

        FLA_Cut_with_3x1_to_2x1( &BT, B0,
            /***/
                bit,
                    &BB, B2,
                        /* bit added to */ FLA_BOTTOM );
    }
}
```

Figure 5.5: Unblocked lazy and row-lazy (w.r.t. L) triangular matrix-matrix multiplication algorithm using FLAME.
`include "FLAME.h"

void Trmm_left_lower_notrans_blk( int variant, int rec, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj        ATL, ATR, A00, A01, A02, BT, B0,
                 ABL, AER, A10, A11, A12, BB, B1,
                 A20, A21, A22, B2;
  int b;

  FLA_Part_2x2( A, &ATL, /**/ &ATR,
               /** */ *** */ &ABL, /**/ &AER, 0, 0, /* submatrix */ FLA_BR );

  FLA_Part_2x1( B, &BT,
               /**/ &BB,
               0, /* length submatrix */ FLA_BOTTOM );

  while ( b = min( FLA_Obj_length( ATR ), nb_alg ) ){ 

    FLA_Rep_2x2_to_3x3( ATL, /**/ ATR,
                        &A00, &A01, /**/ &A02,
                        /**/ &A10, &A11, /**/ &A12,
                        /** */ *** */ &ABL, /**/ &AER,
                        &A20, &A21, /**/ &A22,
                        b, b, /* A11 from */ FLA_LB );

    FLA_Rep_2x1_to_3x1( BT,  &B0,
                        &B1,
                        /**/ &BB,
                        0, /*length B1 from */ FLA_TOP );

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

    if ( variant == FLA_VARIANT_LAZY )
        FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A21, B1, ONE, B2 );
    else if ( rec == FLA_RECURSIVE & b > 0 )
        Trmm_left_lower_notrans_blk( variant, rec, A11, B1, nb_alg/2 );
    else
        Trmm_left_lower_notrans_unb( variant, A11, B1 );

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

    if ( variant == FLA_VARIANT_ROW_LAZY )
        FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B0, ONE, B1 );
    else
        /* ************************************************************** */

        FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,
                                 /**/ &A00, /**/ &A01, &A02,
                                 /** */ *** */ &ABL, /**/ &AER,
                                 /**/ &A10, /**/ &A11, &A12,
                                 /**/ &A20, /**/ &A21, &A22,
                                 **/ A11 added to */ FLA_BR );

        FLA_Cont_with_3x1_to_2x1( &BT,  &B0,
                                 /**/ &BB,
                                 0, /* B1 added to */ FLA_BOTTOM );
  }
}

Figure 5.6: Blocked lazy and row-lazy (w.r.t. \( L \)) triangular matrix-matrix multiplication algorithm using FLAME. Recursion is optionally supported.
#include "FLAME.h"

void Trmm_lln_right_wrt_B_unb( FLA_Obj A, FLA_Obj B )
{
  FLA_Obj   BL, BR, B0, b1, B2;
  FLA_Part_1x2( B, &BL, /**&BR, 0, /* width submatrix */ FLA_LEFT );
  while ( 0 != FLA_Obj_width( BR ) ){
    FLA_Repart_1x2_to_1x3( EL, /** BR,
      &B0, /**&b1, &B2,
      1, /* width b1 from */ FLA_RIGHT );
    /* ******************************************************* */
    FLA_Tzrv( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
      A, b1 );
    /* ******************************************************* */
    FLA_Cont_with_1x3_to_1x2( &BL, /**&BR,
      &B0, /**&b1, &B2,
      /* with b1 added to */ FLA_LEFT );
  }
}

Figure 5.7: Unblocked right-moving (w.r.t. B) triangular matrix-matrix multiplication algorithm using FLAME.

#include "FLAME.h"

void Trmm_lln_right_wrt_B_blk( FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj   BL, BR, B0, B1, B2;
  int   b;
  FLA_Part_1x2( B, &BL, /**&BR, 0, /* width submatrix */ FLA_LEFT );
  while ( b = min( FLA_Obj_width( BR ), nb_alg ) ){  
    FLA_Repart_1x2_to_1x3( EL, /** BR,
      &B0, /**&b1, &B2,
      b, /* width B1 from */ FLA_RIGHT );
    /* ******************************************************* */
    Trmm_lln_right_wrt_B_unb( A, B1 );
    /* ******************************************************* */
    FLA_Cont_with_1x3_to_1x2( &BL, /**&BR,
      &B0, /**&b1, &B2,
      /* with b1 added to */ FLA_LEFT );
  }
}

Figure 5.8: Blocked right-moving (w.r.t. B) triangular matrix-matrix multiplication algorithm using FLAME.
5.4 Performance

In this section, we discuss the performance attained by the different variants for computing $B \leftarrow LB$. In each of the Figs. 5.9 and 5.10, we compare the performance attained by five different implementations:

- Reference DTRMM as implemented as part of ATLAS,
- FLAME FLA_Trm as part of FLAME as of this writing,
- Unblocked the unblocked implementation in Fig. 5.5,
- Blocked the blocked implementation in Fig. 5.6 called with rec == FLA_NON_RECURSIVE, and
- Recursive the blocked implementation in Fig. 5.6 called with rec == FLA_RECURSIVE.

In Fig. 5.9, all FLAME implementations are based on the matrix-matrix multiplication (DGEMM) provided by ATLAS. Notice that for the platform on which we performed the experiments, multiples of 40 are good block sizes when using the ATLAS matrix-matrix multiplication kernel. We note that the unblocked algorithms perform badly, since they are rich in matrix-vector operations that do not benefit much from cache memories. The blocked algorithms performed better for relatively small block sizes (nb_alg=80) than for larger block sizes (nb_alg=160). The reason for this is that too much of the computation is performed in the subprogram for which the unblocked algorithm is used. The recursive implementations benefit from larger block sizes, since they overcome this problem that plagues the blocked algorithm. In addition they benefit from the fact that the DGEMM kernel performs better for larger blocks.

In Fig. 5.10, all FLAME implementations are based on the matrix-matrix multiplication (DGEMM) provided by ITXGEMM. This time, as discussed in Section 4.9, 128 is a magic block size. Again, as expected, the unblocked implementations perform badly. Again, the blocked algorithm benefits from smaller block sizes and the recursive algorithm performs best. It is interesting to note that the lazy algorithm performs much better for small problem sizes when the outer-most block size is chosen to be 128. This is probably mostly due to the fact that the unblocked lazy algorithm performs better.

It is interesting to note that asymptotically, the recursive row-lazy implementation performs somewhat worse than the recursive lazy implementation, when ATLAS is used for the matrix-matrix multiply (Fig. 5.9). We attribute this to the fact that when $m$ is relatively small and $n$ and $k$ are large (e.g., in a panel-matrix multiply), the ATLAS matrix-matrix multiplication does not perform as well as when $k$ is relatively small and $m$ and $n$ are large (e.g., in a panel-panel multiply). Since the row-lazy and lazy algorithms are rich in panel-matrix and panel-panel multiplies, respectively, the lazy algorithm attains better performance. This is not observed as dramatically in in the experiments where ITXGEMM was used for matrix-matrix multiplication (Fig. 5.10). This is due to the fact that ITXGEMM attains performance that is similar regardless of the shape of the matrix.

Notice that the right-moving algorithms perform miserably (Fig. 5.11). We point out that the blocked and recursive algorithms attain essentially the same performance as the unblocked algorithm. This is not surprising since all computation is in a triangular matrix-vector multiply, which does not perform very well, regardless of how the blocking proceeds.

Since in the recursive algorithms at each level a different variant could be called to solve the subproblem, it may be possible to improve performance further by combining different variants. We have not yet studied this.
Figure 5.9: Performance of the row lazy (left) and lazy (right) (w.r.t. \( L \)) lower triangular matrix-matrix multiplication algorithms for a block size of 80 (top) and 160 (bottom). For these experiments, the ATLAS matrix-matrix multiplication kernel was used.
Figure 5.10: Performance of the row lazy (left) and lazy (right) (w.r.t. $L$) lower triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.
Figure 5.11: Performance of the right-moving (w.r.t. $B$) triangular matrix-matrix multiplication algorithms for a block size of 32 (left) and 128 (right), using ITXGEMM.
Chapter 6

Symmetric Matrix Multiplication
$C \leftarrow BA + C$

A symmetric, stored in upper triangle

by
Wynne A. Hexamer

In this chapter, we examine the implementation of the symmetric matrix-matrix multiplication

$C \leftarrow BA + C$

where $A$ is a symmetric $n \times n$ matrix stored in the upper triangle and $B$ and $C$ are $m \times n$ matrices. Symmetric matrices are necessarily square and have the property that $A = A^T$. All entries except those along the diagonal occur in pairs mirrored along the diagonal.

6.1 Derivation of a Family of Algorithms

Let us consider

(6.1) \[ D = BA + C \]

keeping in mind that $D$ will overwrite $C$. Partition

\[ A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{TR}^T & A_{BR} \end{pmatrix} \]

where $A_{TL}$ is $k \times k$. Keep in mind that square submatrices on the diagonal, $A_{TL}$ are $A_{BR}$, are themselves symmetric and stored only in the upper triangle. Also, partition

\[ C \rightarrow \begin{pmatrix} C_L & C_R \end{pmatrix}, \quad \text{and} \quad D \rightarrow \begin{pmatrix} D_L & D_R \end{pmatrix}, \quad \text{and} \quad B \rightarrow \begin{pmatrix} B_L & B_R \end{pmatrix} \]

where $C_L$ and $D_L$ and $B_L$ have $k$ columns.

Now, (6.1) becomes

\[ \begin{pmatrix} D_L & D_R \end{pmatrix} = \begin{pmatrix} B_L & B_R \end{pmatrix} \begin{pmatrix} A_{TL} & A_{TR} \\ A_{TR}^T & A_{BR} \end{pmatrix} \begin{pmatrix} A_{TL} & A_{TR} \\ A_{TR}^T & A_{BR} \end{pmatrix} + \begin{pmatrix} C_L & C_R \end{pmatrix} \]

66