Developing Linear Algebra Algorithms: A Collection of Class Projects

STATUS AT END OF SEMESTER

FLAME Working Note #3

John A. Gunnels

Robert A. van de Geijn

Department of Computer Sciences
The University of Texas
Austin, TX 78712
{gunnels,rvdg}@cs.utexas.edu

May 31, 2001

Abstract

In this document we present a new approach to developing sequential and parallel dense linear algebra libraries. Given a linear algebra operation, we demonstrate how formal techniques can be used to derive a family of algorithms. Due to the systematic approach used, correctness of the algorithms can be asserted. Next, we introduce a library of routines that hides the manipulation of indices and allows the code to mirror the algorithms as they are naturally presented. The idea is that by having the code mirror the algorithm, the opportunity for introducing indexing errors is minimized. Thus, the correctness assertions regarding the algorithms carry over to the implementations.

The philosophy behind the approach is that one should start by systematically deriving the algorithms. The recipe for derivation is given in Chapter 2. Moreover, this derivation should be carefully documented. To facilitate this, we provide a set of LaTeX macros, given in an appendix. Once one or more algorithms have been developed, they are translated to implementations using a library of C routines, as part of the Formal Linear Algebra Methods Environment (FLAME). This library allows the code to look much like the algorithms as written using LaTeX. For all examples in the report we demonstrate that high performance can be attained on an Intel Pentium (R) III processor.

We illustrate the techniques with a large number of case studies, most of which were carried out by teams of computer science undergraduate students as part of a class taught in Spring 2001 at UT-Austin titled *High-Performance Parallel Algorithms.* The names of the members of the teams are given as the authors of the chapter on the operation assigned to that team. Thus we show that the approach makes the development and implementation of high-performance sequential and parallel algorithms for dense linear algebra operations accessible to novices.

It is important to realize that this document is meant to capture the progress of the project during a single semester. Thus, the document is incomplete in many ways. For example, the review of the literature is sparse at best. Many sections and chapters are missing or incomplete. Typographical errors are scattered throughout. For each operation, only a few algorithms are derived and implemented. The performance results are limited to a single architecture. While high-performance parallel implementations were also created using our Parallel Linear Algebra Package (PLAPACK), the discussion of these implementations did not make it into the document. It is our hope that there is value in this document despite these shortcomings.

Contents

Chapter 1

Introduction

When considering the unmanageable complexity of computer systems, Dijkstra recently made the following observations [10]:

(i) When exhaustive testing is impossible –i.e., almost always– our trust can only be based on proof (be it mechanized or not).
(ii) A program for which it is not clear why we should trust it, is of dubious value.
(iii) A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.
(iv) Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

Our Formal Linear Algebra Methods Environment (FLAME) is an attempt to address these concerns when coding linear algebra libraries [18].

The core undergraduate curriculum in computer science department often includes at least one course that focuses on the formal derivation and verification of algorithms [13]. Many of us in scientific computing may have, at some point in time, hastily dismissed this approach, arguing that this is all very nice for small, simple algorithms, but an academic exercise hardly applicable in "our world." Since it is often the case that our work involves libraries comprised of hundreds of thousands or even millions of lines of code, the knee-jerk reaction that this approach is much too cumbersome to take seriously is understandable. Furthermore, the momentum of established practices and "traditional wisdom" do little if anything to dissuade one from this line of reasoning. Yet, as the result of our search for superior methods for designing and constructing high-performance parallel linear algebra libraries, we have come to the conclusion that it is *only* through the systematic approach offered by formal methods that we will be able to deliver reliable, maintainable, flexible, yet highly efficient matrix libraries even in the relatively well-understood area of (sequential and parallel) dense linear algebra.

While some would immediately draw the conclusion that a change to a more modern programming language like C++ is at least highly desirable, if not a necessary precursor to writing elegant code, the fact is that most applications that call packages like the Linear Algebra PACKage (LAPACK) [4] and the Scalable Linear Algebra PACKage ScaLAPACK [8, 7] are still written in Fortran and/or C. Interfacing such an application with a library written in C++ presents certain complications. However, during the mid-nineties, the Message-Passing Interface (MPI) introduced to the scientific computing community a programming model, object-based programming, that possesses many of the advantages typically associated with the intelligent use of an object-oriented language [33]. Using objects (e.g. communicators in MPI) to encapsulate data structures and hide complexity, a much cleaner approach to coding can be achieved. Our own work on the Parallel Linear Algebra PACKage (PLAPACK) borrowed from this approach in order to hide details of data distribution and data mapping in the realm of parallel linear algebra libraries [3, 5, 16, 29, 30, 35, 37]. The primary concept also germane to this paper is that PLAPACK raises the level of abstraction at which one programs so that indexing is essentially removed from the code, allowing the routine to reflect the algorithm as it is naturally presented in a classroom setting. Since our initial work on PLAPACK, we have experimented with similar interfaces in such seemingly disparate contexts as (parallel) out-of-core linear algebra packages [19, 31, 32] and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [11, 12, 17, 28].

Our Formal Linear Algebra Methods Environment (FLAME) is the latest step in the evolution of these systems. It facilitates the use of a programming style that is equally applicable to everything from out-of-core, parallel systems to single-processor systems where cache-management is of paramount concern.

Over the last seven or eight years it has become apparent that what makes our task of library development more manageable is this systematic approach to deriving algorithms coupled with the abstractions we use to make our code reflect the algorithms thus produced. Further, from these experiences we can confidently state that this approach to programming greatly reduces the complexity of the resultant code and does not sacrifice high performance in order to do so.

Indeed, the formal techniques that we may have dismissed as merely academic or impractical make this possible, as we attempt to illustrate in this document.

1.1 The case for formal derivation

Ideally, an implementation should clearly reflect the algorithm as it is presented in a classroom setting. Additionally, some of the derivation of the algorithm should be apparent in the code and different variants of an algorithm should be recognizable as slight perturbations to an algorithmic "skeleton" or base code. indexbase code If the code is just a mechanically-realizable, straightforward translation of this algorithmic expression, there should be no opportunity for the introduction of logical errors or coding bugs. (Note: while we will frequently refer to translations from algorithms to code as being mechanical or automatic, this process is currently performed by hand.) Presumably, it should be possible to prove the algorithms correct, thus ensuring that the code is correct.

Typically, it is difficult to prove code correct precisely because one is not certain that the code truly mirrors the algorithm. With our approach, the chasm is largely bridged by the simple yet crucial fact that some very simple syntactic rewrite rules can produce the code from an algorithm expressed as one might in a classroom, using mathematical formulas and stylized matrix depictions. Since we can prove the correctness of the algorithm we wish to employ (the proof is generally constructive in nature, but this is of little consequence) and because the correctness of the translation from algorithm to code is at least as reliable as compiler technology, the complexity of the task at hand is greatly ameliorated. By assuming that components adhere to explicit "contractual obligations" [2], the algorithmic proof requires little alteration in order to be applicable to the code. In the case of a library constructed entirely through the methodology presented here, these components would be composed in like manner so as to make this task manageable. This is largely due to the fact that the approach presented here leads to a software architecture layered in such a way so as to require one to rely on the correctness of a very small number of base-level modules. Since those units are small, their correctness can be established through the application of standard formal methods. It is true that, in practice, one must accept that an application will need to interface with other libraries (for example, the vendor-supplied BLAS) that are not built in a "proof-friendly" format. However, it may still be possible to establish the correctness of a program if one is careful to impose minimal obligations on these, presumably time-tested and well-documented, pieces of code.

It should be noted that the "correctness" discussed so far does not address issues of numerical stability. We make no claim regarding the stability of the resulting algorithm.

Having said this, we will clarify through a simple example in Sectionsec:example. But first, we review commonly used matrix and vector notation. For those for whom linear algebra is not second nature, the most basic of operations are reviewed in A

1.2 Notation

A *(column) vector*, x, is the n-tuple of real or complex numbers

$$x = \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix}$$

Here χ_i are called the *components* of vector x. We will denote the set of all vectors with real components $\mathbf{R}^n$ and with complex components $\mathbf{C}^n$.

A *row* vector, x^T, is the n-tuple of real or complex numbers

$$x^T = \left(\begin{array}{c|c|c|c} \chi_1 & \chi_2 & \cdots & \chi_n \end{array} \right)$$

Here x^T indicates a transposed (column) vector. (We will always assume vectors are column vectors, unless transposed like this, or explicitly noted.) More about transposition next.

An $m \times n$ matrix, A, is the array

$$A = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{pmatrix}$$

with m rows and n columns. The (i, j) component, or *element*, of A refers to α_{ij}, which may be real or complex. The numbers m and n are the dimensions of A. If $m = n$ then the matrix is said to be **square**. Otherwise, it is said to be **rectangular**.

Notice that we use the convention introduced in [34] of using Greek letters for real or complex numbers, lowercase italicized letters for vectors, and uppercase italicized letters for matrices.

Frequently, we will wish to partition a matrix into blocks. For example, if A is an $m \times n$ matrix, it can be partitioned into a $M \times N$ matrix of submatrices like

$$A = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \cdots & A_{1N} \\ \hline A_{21} & A_{22} & \cdots & A_{2N} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M1} & A_{M2} & \cdots & A_{MN} \end{array} \right)$$

where A_{ij} is an $m_i \times n_j$ matrix, with $\sum_{i=1}^{N} n_i = n$ and $\sum_{i=1}^{M} m_i = m$.

Similarly, a vector can be partitioned into subvectors. For example, if x is a vector of length n, we may wish to partition like

$$x = \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_N \end{array} \right)$$

where x_i is an vector of length n_i, with $\sum_{i=1}^{N} n_i = n$.

Additional notation and basic linear algebra operations are reviewed in Appendix A.

1.3 A motivating example: LU factorization

We illustrate our approach by considering LU factorization without pivoting. Given an $n \times n$ matrix A we wish to compute an $n \times n$ lower triangular matrix L with unit main diagonal and an $n \times n$ upper triangular matrix U so that $A = LU$. The original matrix A is overwritten by L and U in the process.

The usual derivation of an algorithm for the LU factorization proceeds as follows: Partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right), \quad L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array} \right), \quad \text{and} \quad U = \left(\begin{array}{c|c} \upsilon & u_{12}^T \\ \hline 0 & U_{22} \end{array} \right)$$

Now $A = LU$ translates to

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right) = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array} \right), \left(\begin{array}{c|c} \upsilon & u_{12}^T \\ \hline 0 & U_{22} \end{array} \right) = \left(\begin{array}{c|c} \upsilon & u_{12}^T \\ \hline l_{21}\upsilon_{11} & l_{21}u_{12}^T + L_{22}U_{22} \end{array} \right)$$

so the following equalities hold:

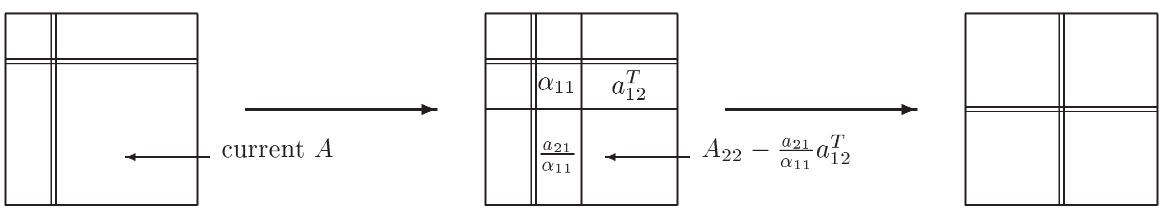
$$\left(\begin{array}{c|c} \alpha_{11} = \upsilon & a_{12}^T = u_{12}^T \\ \hline a_{21} = \upsilon_{11} l_{21} & a_{22} = l_{21}u_{12}^T + L_{22}U_{22} \end{array} \right)$$

Finally, we arrive at the following algorithm

- Overwrite α_{11} and a_{12}^T with υ_{11} and u_{12}^T, respectively (no-op).
- Update $a_{21} \leftarrow l_{21} = a_{21}/\upsilon_{11}$.
- Update $A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$.
- Recursively factor $A_{22} \rightarrow L_{22}U_{22}$.

While the algorithm is formulated as tail-recursive, it is usually implemented as a loop.

When presented in a classroom setting, the explanation is typically accompanied by the following progression of pictures:



with an indication that at a given stage the current active part of the matrix resides in the lower-right quadrant of the left picture. Next, the different parts to be updated are identified and the updates given (middle picture). Finally, the boundary indicating how far the computation has progressed is moved forward (right picture).

It is this progression depicted in the pictures that we try to capture both in the derivation and the implementation of the algorithm. We claim that the discussed algorithm for LU factorization is naturally given by the algorithm in Fig. 1.1. A code for implementing the algorithm using FLAME is given in Fig. 1.2.

The code can be obtained from the algorithm essentially via textual substitution. Notice that the calls to `FLA_Inv_scal` and `FLA_Ger` implement division of a vector by a scalar and rank-1 update of a matrix, respectively. The formatting of the code is a deliberate attempt to capture the partitioning and repartitioning in the algorithm.

1.4 Performance experiments

For each matrix operation discussed in this document, we report performance on an Intel Pentium III (650 MHz) processor with 16 Kbyte L1 data cache and a 256 Kbyte L2 cache running RedHat 7.1 Linux. All floating point calculations were performed in double precision (64-bit) arithmetic.

The FLAME routines that perform level 1 BLAS (vector-vector operations) and level 2 BLAS (matrix-vector operations) interface to a standard BLAS library. For performance experiments, the ATLAS library implementation was used [36]. In particular, the prebuilt version in

partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right)$ **where** A_{TL} **is** 0×0

do until A_{BR} **is** 0×0

repartition $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{02}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array} \right)$ **where** α_{11} **is a scalar**

$\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11}$
$a_{12}^T \leftarrow u_{12}^T = a_{12}^T$
$a_{21} \leftarrow l_{21} = a_{21} / \alpha_{11}$
$A_{22} \leftarrow A_{22} - l_{21} u_{12}^T$

---

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

enddo

Figure 1.1: An unblocked (rank-1 update based) eager LU factorization algorithm (without pivoting)

```
#include "FLAME.h"

void FLA_LU_nopivot_eager_level2( FLA_Obj A )
{
  FLA_Obj     ATL, ATR,   A00,  a01,     A02,
              ABL, ABR,   a10t, alpha11, a12t,
                          A20,  a21,     A22;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ) != 0 ){

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00,  /**/ &a01,     &A02,
                         /* ************* */   /* ************************* */
                                /**/            &a10t, /**/ &alpha11, &a12t,
                           ABL, /**/ ABR,       &A20,  /**/ &a21,     &A22,
            /* with */ 1, /* by */ 1, /* alpha11 split from */ FLA_BR );

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

    FLA_Inv_scal( alpha11, a21 );            /* a21 <- a21 / alpha11        */

    FLA_Ger( MINUS_ONE, a21, a12t, A22 );     /* A22 <- A22 - a21 * a12t     */

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

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00,  a01,     /**/ A02,
                                    /**/            a10t, alpha11, /**/ a12t,
                             /* ************** */ /* ************************ */
                               &ABL, /**/ &ABR,      A20,  a21,     /**/ A22,
            /* with alpha11 added to submatrix */ FLA_TL );
  }
}
```

Figure 1.2: An unblocked (rank-1 update based) eager LU factorization implementation (without pivoting) using FLAME.

atlas3.2.0_Linux_SSE1256.tgz

available from

http://www.netlib.org/atlas/archives/linux/

was used. In many of the performance graphs for level 3 BLAS (matrix-matrix) operations, we report performance by a reference implementation. This reference implementation is the one provided by ATLAS.

Notice that our implementations heavily rely on a high-performance matrix-matrix multiplication kernel. For our performance experiments, the `DGEMM` level 3 BLAS kernel for matrix-matrix multiplication was used. We use two different implementations:

- the implementation provided by ATLAS, and
- our own ITXGEMM implementation [17].

The latter implementation, which yields better performance for most matrix sizes, is further explained in Chapter 4.

We report performance as the rate at which the computation was performed in MFLOPS/sec. (millions of floating point operations per second). More precisely, if C equals the number of floating point operations required to complete the computation, and t equals the time required, the rate in MFLOPS/sec. is given by

$$\frac{C}{t} \times 10^{-6}$$

1.5 Overview

Our methodology for developing high-performance linear algebra algorithms is introduced in Chapter 2. The application programming interface (API) is introduced in Chapter 3 as a set of C routines. In Chapter 4, we discuss how high performance can be attained by a matrix-matrix multiplication kernel. In Chapters 5–16 we report a large number of case studies that demonstrate the methodology for developing algorithms. They also show how blocked algorithms can be used to formulate the algorithms in terms of matrix-matrix multiplications, which allows the high performance attained by this kernel to be exploited. It should be noted that the idea of implementing this particular set of operations in terms of matrix-matrix multiplication has been studied extensively elsewhere [8, 9, 20, 21, 24, 25, 26, 36]. In particular, one may recognize them as special cases of the level 3 BLAS.

1.6 Availability

Information related to FLAME is available at

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

Codes discussed in this document can be found at

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

The LaTeXcommands and environments used to typeset many of the formulas and algorithms in this document, which are described in Appendix B, can be found at

http://www.cs.utexas.edu/users/flame/LaTeX/

Chapter 2

Formal Derivation

In this chapter, we briefly review general techniques for the derivation of algorithms. We relate these techniques to the iterative algorithms encountered in subsequent chapters. Finally, we give a systematic recipe for deriving linear algebra algorithms. The recipe is illustrated for the triangular matrix-matrix multiplication $B \leftarrow LB$.

2.1 The correctness of loops

In a standard text [14] used to teach discrete mathematics to undergraduates in computer science we find the following material:

> We prefer to write a while loop using the syntax
>
> **do** $B \rightarrow S$ **od**
>
> where Boolean expression B is called the *guard* and statement S is called the *repetend*.
>
> [The l]oop is executed as follows: If B is *false*, then execution of the loop terminates; otherwise S is executed and the process is repeated.
>
> Each execution of repetend S is called an *iteration*. Thus, if B is initially *false*, then 0 iterations occur.
>
> [$\cdots$]
>
> We now state and prove the fundamental invariance theorem for loops. This theorem refers to an assertion P that holds before and after each iteration (provided it holds before the first). Such a predicate is called a *loop invariant*.
>
> (12.43) **Fundamental invariance theorem.** Suppose
>
> - $\{P \wedge B\}S\{P\}$ holds – i.e. execution of S begun in a state in which P and B are *true* terminates with P *true* – and
> - $\{P\}$ **do** $B \rightarrow S$ **od** *true* – i.e. execution of the loop begun in a state in which P is *true* terminates.
>
> Then $\{P\}$ **do** $B \rightarrow S$ **od** $\{P \wedge \neg B\}$ holds. [In other words, if the loop is entered in a state where P is *true*, it will complete in a state where P is *true* and guard B is *false*.]

The text proceeds to prove this theorem using mathematical induction.

Let us translate the above into our setting, which will accommodate linear algebra algorithms. Consider the loop

do until $\neg B$
  S
enddo

where B is some condition and S is the body of the loop. The above theorem says that *if*

- The loop is entered in a state where some condition P holds, and
- for each iteration, P holds at the top of the loop, and
- the body of the loop S has the property that if it is executed starting in a state where P holds it completes in a state where P holds.

then the loop will complete in a state where conditions P and $\neg B$ both hold.

A method that formally derives a loop (i.e., iterative implementation) approaches the problem of determining the body of the loop as follows:

- First, one must determine conditions B and P.
- Next, the body S should be developed so that it maintains condition P while making progress towards completing the iterative process (eventually B should become *false*).

As a consequence of the Fundamental Invariance Theorem, this approach implies correctness of the loop.

What we show in the remainder of this chapter, and the subsequent case studies in the remainder of this book, is that for a large class of dense linear algebra algorithms

- There is a systematic way of determining different conditions P that allow us develop loops to compute the result of a given linear algebra operation.
- This in turn yields different algorithms for computing the operation.

2.2 A recipe for deriving linear algebra algorithms

2.2.1 A typical operation

A typical linear algebra matrix-matrix operation involves up to three operands: matrices A, B, and C:

$$C \leftarrow \mathrm{op}(A, B, C)$$

> **Example (LTRMM)** A lower-triangular matrix-matrix multiplication (LTRMM) can be expressed as
>
> $$B \leftarrow LB$$
>
> where B is an $m \times n$ matrix and L is a $m \times m$ lower triangular matrix. Notice that the purpose of the game is to overwrite B with the results without requiring a work array in which to compute LB.

2.2.2 determining possible loop invariants

1. **Temporarily replace C by D.** Since C appears both on the left and the right of the operation to be performed, we temporarily replace one of the instances by a new operand, D.

   > **Example (LTRMM)** $B \leftarrow LB$ is replaced by $D = LB$. Notice that the arrow indicates that the operand is overwritten with the result. We now replace this by an equality, since the purpose of the game will be to determine equalities that must hold as the computation unfolds.

2. **Pick an Operand and Partition** The first step is to pick an operand and partition[1] it in a meaningful way:

[1]Note: we provide some useful LaTeXmacros for partitioning matrices in Appendix C.

- If the operand that is picked has a triangular storage structure, the partitioning should be into four quadrants, so that the quadrant that contains the block that is not used for storing the matrix can be identified.

  > **Example (LTRMM)** Pick L for partitioning:
  >
  > $$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$$
  >
  > where L_{TL} is a square block, say of size $k \times k$.

- If the operand has no special structure, it is typically partitioned into two submatrices, either horizontally or vertically.

  > **Example (LTRMM)** Pick B for partitioning:
  >
  > $$B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array}\right)$$
  >
  > where B_L has k columns. Note: this second partitioning does not necessarily lead to the same set of loop invariants as the partition of L above. Indeed, it doesn't.

3. **Partition the other operands conformal to the first one:** Given that the first operand has been partitioned, the other operands should be partitioned *conformally* to ensure that blocked multiplication of the submatrices makes sense.

   > **Example (LTRMM)** Let us concentrate of the case where L has been partitioned like
   >
   > $$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$$
   >
   > When considering $D = LB$, we notice that D and B must be partitioned by rows:
   >
   > $$D \rightarrow \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$
   >
   > where D_T and B_T have k rows.

4. **Rewrite the operation using the partitionings:** Next, plug the partitioned matrices into the operation $D = \text{op}(A, B, C)$.

   > **Example (LTRMM)** In the example $D = LB$, where D, L, and B have been partitioned as described, this yields
   >
   > $$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$

5. **Perform blocked matrix-matrix multiplications and additions:** Now that the operation has been expressed with blocked matrices, we perform the given operations.

   > **Example (LTRMM)** In our lower triangular matrix-matrix multiplication example we get
   >
   > $$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) = \left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$$

6. **Determine equalities that must hold:** Once the operations have been performed with the submatrices, equalities that must hold can be determined.

> **Example (LTRMM)** In our lower triangular matrix-matrix multiplication example we determined that
>
> $$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) = \left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$$
>
> Thus, we conclude that the following equalities must hold:
>
> $$\begin{array}{rcl} D_T & = & L_{TL}B_T \\ D_B & = & L_{BL}B_T + L_{BR}B_B \end{array}$$

7. **Determine possible contents of D:** Next, we determine possible contents of D under the assumption that some of the computations that appear in the equalities have occurred. One approach to this is to enumerate all the major operations that must be formed. At a given stage of the computation, each of these computations either has or has not occurred. By considering all possible combinations, one can enumerate essentially all possible conditions.

> **Example (LTRMM)** See Fig. 2.1.

8. **Eliminate unreasonable conditions:** Once possible contents of D have been enumerated, some of the conditions can typically be eliminated since they do not lead to reasonable loops.

   There are two reasons for rejecting a condition:

   (a) We want to derive a loop that maintains the condition while making progress towards the desired result. We need to be able to achieve that condition before we enter the loop. Furthermore, upon leaving the loop, the desired result should have been computed. If the condition doesn't allow this, then the condition is rejected.

   (b) The result overwrites part of one of the input operands before that part of the input operand is no longer needed.

   We will illustrate how the first approach can be used to reject possible conditions for the symmetric matrix multiply in Chapter 6. In Fig. 2.2, we use the second approach to reject a number of conditions for the triangular matrix-matrix multiplication example.

9. **Determine the direction of the computation:** The condition that determines the contents of D indicates a direction in which the computation naturally proceeds. Frequently encountered directions are

left	$\left(\begin{array}{c\|c\|\|c} & \leftarrow & \end{array}\right)$	right	$\left(\begin{array}{c\|\|c\|c} & \rightarrow & \end{array}\right)$	up	$\left(\begin{array}{c} \\ \hline \uparrow \\ \hline\hline \\ \end{array}\right)$
down	$\left(\begin{array}{c} \\ \hline\hline \downarrow \\ \hline \\ \end{array}\right)$	left-up	$\left(\begin{array}{c\|c\|\|c} & & \\ \hline & \nwarrow & \\ \hline\hline & & \end{array}\right)$	right-down	$\left(\begin{array}{c\|\|c\|c} & & \\ \hline\hline & \searrow & \\ \hline & & \end{array}\right)$

> **Example (LTRMM)** Consider the condition that currently D contains $\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right)$. 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.

Example (LTRMM) In our lower triangular matrix-matrix multiplication example we have determined that

$$\begin{array}{rcl} D_T & = & L_{TL}B_T \\ D_B & = & L_{BL}B_T + L_{BR}B_B \end{array}$$

The major operations to be performed are $L_{TL}B_T$, $L_{BL}B_T$, and $L_{BR}B_B$. Each of these either has or has not already been computed, leading to the $2^3 = 8$ possible conditions tabulated below. In the table, a $\star$ indicates the indicated part of D has yet to be computed.

Computed?			D contains
$L_{TL}B_T$	$L_{BL}B_T$	$L_{BR}B_B$	
NO	NO	NO	$\left(\frac{\star}{\star}\right)$
YES	NO	NO	$\left(\frac{L_{TL}B_T}{\star}\right)$
NO	YES	NO	$\left(\frac{\star}{L_{BL}B_T}\right)$
YES	YES	NO	$\left(\frac{L_{TL}B_T}{L_{BL}B_T}\right)$
NO	NO	YES	$\left(\frac{\star}{L_{BR}B_B}\right)$
YES	NO	YES	$\left(\frac{L_{TL}B_T}{L_{BR}B_B}\right)$
NO	YES	YES	$\left(\frac{\star}{L_{BL}B_T + L_{BR}B_B}\right)$
YES	YES	YES	$\left(\frac{L_{TL}B_T}{L_{BL}B_T + L_{BR}B_B}\right)$

Figure 2.1: Step 7 for the LTRMM example.

Example (LTRMM) In the following table, we again list possible contents of D. This time, we comment on each possibility.

D contains	Comments	Viable?
$\left(\begin{array}{c}\star \\ \hline\hline \star\end{array}\right)$	This condition indicates no progress has been made.	NO
$\left(\begin{array}{c}L_{TL}B_T \\ \hline\hline \star\end{array}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\begin{array}{c}\star \\ \hline\hline L_{BL}B_T\end{array}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\begin{array}{c}L_{TL}B_T \\ \hline\hline L_{BL}B_T\end{array}\right)$	Since B_B is to be overwritten by D_B, this condition is not feasible since B_B is still needed for the computation $L_{BR}B_B$.	NO
$\left(\begin{array}{c}\star \\ \hline\hline L_{BR}B_B\end{array}\right)$		YES
$\left(\begin{array}{c}L_{TL}B_T \\ \hline\hline L_{BR}B_B\end{array}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\begin{array}{c}\star \\ \hline\hline L_{BL}B_T + L_{BR}B_B\end{array}\right)$		YES
$\left(\begin{array}{c}L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B\end{array}\right)$	This condition indicates that the computation has completed.	NO

Considering the comments, only two viable conditions are left, the ones for which there are no comments.

Figure 2.2: Step 8 for the LTRMM example.

10. **Repartition the matrices:** In order to expose what elements must be updated to make progress, D is repartitioned. Similarly, the other matrices must be repartitioned conformally to expose submatrices needed to update D.

> **Example (LTRMM)** Since the computation moves through D in the "up" direction, we repartition D like
>
> $$\textbf{repartition} \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right) \textbf{ where } d_1^T \textbf{ is a row}$$
>
> Similarly, we must repartition the other matrices:
>
> $$\textbf{repartition} \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right) \textbf{ where } b_1^T \textbf{ is a row}$$
>
> $$\textbf{repartition} \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right) \textbf{ where } \lambda_{11} \textbf{ is a scalar}$$
>
> Notice that the double lines have meaning:
>
> $$\begin{array}{l} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T \end{array}\right) \\ \hline D_B = \quad D_2 \end{array} \qquad \begin{array}{l} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline B_B = \quad B_2 \end{array} \qquad \begin{array}{l||l} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & \\ \hline L_{BL} = \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) & L_{BR} = L_{22} \end{array}$$

11. **Determine what is currently in the matrix:** Given the repartitionings, we determine what is currently in matrix D.

> **Example (LTRMM)** D currently contains
>
> $$\begin{aligned} \left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) + L_{22}B_2 \end{array}\right) \\ &= \left(\begin{array}{c} \star \\ \hline\hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{aligned}$$

12. **Determine what needs to be in the matrix after the boundary shifts:** Notice that once the boundary shifts, the partitionings of the matrices indicate different submatrices of those matrices. To maintain the condition, an update to the contents of D is required.

> **Example (LTRMM)** After the boundaries shift
>
> $$\begin{array}{l} D_T = \quad D_0 \\ \hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \end{array} \qquad \begin{array}{l} B_T = \quad B_0 \\ \hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array} \qquad \begin{array}{l||l} L_{TL} = \quad L_{00} & \\ \hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \end{array}$$
>
> and thus D must hold
>
> $$\begin{aligned} \left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) B_0 + \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \\ &= \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T B_0 + \lambda_{11} b_1^T \\ \hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{array}\right) \end{aligned}$$

13. **Determine what update must occur:** The known contents of D before the shift of the boundaries and the desired contents of D after the shift of the boundaries determines the update that must occur.

> **Example (LTRMM)** The contents of D must change as follows
>
> $$\left(\begin{array}{c} \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \\ \hline\hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T B_0 + \lambda_{11} b_1^T \\ \hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{array}\right)$$

14. **State the algorithm:** At this point, the partitioning and repartition of the matrices have been derived, as have the steps required to maintain the desired condition. Thus, the algorithm can be given.

> **Example (LTRMM)** See Fig. 2.3.

> **Example (LTRMM)** One algorithm for the lower triangular matrix-matrix multiplication. Notice that we overwrite B with the result.
>
> **partition** $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ **where** B_B **has** 0 **rows**
>
> **partition** $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$ **where** L_{BR} **is** 0×0
>
> **do until** L_{TL} **is** 0×0
>
> **repartition** $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$ **where** b_1^T **is a row**
>
> **repartition** $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$ **where** λ_{11} **is a scalar**
>
> $b_1^T \leftarrow \lambda_{11} b_1^T$
> $b_1^T \leftarrow l_{10}^T B_0 + b_1^T$
>
> **continue with** $\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$
>
> **continue with** $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$
>
> **enddo**

Figure 2.3: Step 14 for the LTRMM example.

15. **Classify the algorithm:** We classify algorithms by

- the direction in which they move, and
- how aggressively they use and/or update data in the matrices.

While intuitively the classification is consistent, the different categories have slightly different meaning depending on whether we categorize with respect to an input or an output parameter. In particular:

Let X be the operand with respect to which we will categorize the algorithm. The following table explains the different categories:

right/left-moving algorithms: Consider the current partitioning on operand in a right- or left-moving algorithm w.r.t. X: $\left(X_L \,\|\, X_R \right)$. Then we will use the following categorization of algorithms w.r.t. X:

right-moving algorithm	
Lazy	The entries in X_L have been completely used and/or updated. The entries in X_R has not been touched (used or updated).
Eager	The entries in X_L have been completely used and/or updated. The entries in X_R have been updated as much as is possible without completing another column of X.

left-moving algorithm	
Lazy	The entries in X_R have been completely used and/or updated. The entries in X_L has not been touched (used or updated).
Eager	The entries in X_R have been completely used and/or updated. The entries in X_L have been updated as much as is possible without completing another column of X.

down/up-moving algorithms: Consider the current partitioning on operand in a down- or up-moving algorithm w.r.t. X: $\left(\frac{X_T}{X_B} \right)$. Then we will use the following categorization of algorithms w.r.t. X:

down-moving algorithm	
Lazy	The entries in X_T have been completely used and/or updated. The entries in X_B has not been touched (used or updated).
Eager	The entries in X_T have been completely used and/or updated. The entries in X_B have been updated as much as is possible without completing another row of X.

up-moving algorithm	
Lazy	The entries in X_B have been completely used and/or updated. The entries in X_T has not been touched (used or updated).
Eager	The entries in X_B have been completely used and/or updated. The entries in X_T have been updated as much as is possible without completing another row of X.

down-right/up-left-moving algorithms: Consider the current partitioning of an operand in a right-down- or left-up-moving algorithm w.r.t. X: $\left(\begin{array}{c||c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array} \right)$. Then we will use the following categorization of algorithms w.r.t. X:

right-down-moving algorithm	
Lazy	The entries in X_{TL} have been completely used and/or updated. The entries in X_{TR}, X_{BL}, and X_{BR} have not been touched (used or updated).
Row-lazy	The entries in X_{TL} and X_{TR} have been completely used and/or updated. The entries in X_{BL} and X_{BR} have not been touched (used or updated).
Column-lazy	The entries in X_{TL} and X_{BL} have been completely used and/or updated. The entries in X_{TR} and X_{BR} have not been touched (used or updated).
Both-lazy	The entries in X_{TL}, X_{TR}, and X_{BL} have been completely used and/or updated. The entries in X_{BR} have not been touched (used or updated).
Eager	The entries in X_{TL}, X_{TR}, and X_{BL} have been completely used and/or updated. The entries in X_{BR} have been updated as much as is possible without completing another row and column of X.

left-up-moving algorithm	
Lazy	The entries in X_{BR} have been completely used and/or updated. The entries in X_{TR}, X_{BL}, and X_{TL} have not been touched (used or updated).
Row-lazy	The entries in X_{BL} and X_{BR} have been completely used and/or updated. The entries in X_{TL} and X_{TR} have not been touched (used or updated).
Column-lazy	The entries in X_{TR} and X_{BR} have been completely used and/or updated. The entries in X_{TL} and X_{BL} have not been touched (used or updated).
Both-lazy	The entries in X_{BR}, X_{TR}, and X_{BL} have been completely used and/or updated. The entries in X_{TL} have not been touched (used or updated).
Eager	The entries in X_{BR}, X_{TR}, and X_{BL} have been completely used and/or updated. The entries in X_{TL} have been updated as much as is possible without completing another row and column of X.

Example (LTRMM) Consider the example that we have been using throughout this chapter. Using our categorization, we see that the algorithm that corresponds to the condition that currently D contains $\left(\frac{\star}{\overline{L_{BL}B_T + L_{BR}B_B}}\right)$ is up-moving and row-lazy with respect to matrix L since data in L_{BL} and L_{BR} will not be required for further computation. It is also lazy with respect to matrix D, since D_L has been completely computed.

The algorithm that corresponds to the the condition that currently D contains $\left(\frac{\star}{\overline{L_{BR}B_B}}\right)$ is also up-moving. However, it is lazy with respect to matrix L since only the data in L_{BR} will not be needed for further computation. It is also lazy with respect to matrix B, since B_B will not be needed for further computation.

Chapter 3

Coding Linear Algebra Algorithms

In this chapter we introduce a set of library routines that will allow us to capture in code linear algebra algorithms as they are naturally presented, for example in a classroom setting. The idea is that by making the code look as much like the algorithm in Fig. 1.1 the opportunity for the introduction of bugs is minimized. Readers familiar with MPI [15, 33] and/or our own PLAPACK [35] will recognize the programming style as being very similar to that used by those 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.

Since an application may wish to query whether the environment has already been initialized, we provide the inquiry routine

```
int FLA_Initialized ( )
```
Purpose: Check if FLAME is initialized.

return value `TRUE` if FLAME is already initialized
`FALSE` otherwise

3.2 Creating an object

Notice that there the following attributed 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 described how the two dimensional array is mapped to memory.

For now, we will assume that a matrix is stored using column-major ordering. Thus, the mapping to memory is described by a leading dimension that indicates the number of units through which one must stride in memory to get from one element in a row of the matrix to the next element in that row. The following call creates an object 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 as well as associated storage.

`datatype`	datatype of matrix
`m, n`	row dimensions of matrix
matrix	object that describes the matrix

Notice that the leading dimension of the array that is used to store the actual matrix is itself determined inside of this call.

Valid datatype value include

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

for the obvious datatypes that are commonly encountered. Additional datatype may be added at a future stage.

Sometimes it will be handy to create an object without storage attached. 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 an object:

```
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 associated storage.

`datatype`	datatype of matrix
`m, n`	dimensions of matrix
`matrix`	address of object that will describe the matrix

If an object has been created without storage attached, 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`.

`buff`	address of where buffer exists
`ldim`	leading dimension of array
`matrix`	object that describes the matrix

FLAME treats vectors as special cases of matrices, either as a $n \times 1$ matrix or an $1 \times n$ 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 )`, or
`FLA_Obj_create( FLA_DOUBLE, 1, n, &x )`,

where `x` has been declared as a `FLA_Obj` and `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 )`

where `alpha` has been declared as a `FLA_Obj`. 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.

`trans`	indicates whether to transpose
`old`	original object
`matrix`	new object

Valid values for `trans` include `FLA_NO_TRANSPOSE` , `FLA_TRANSPOSE`, and `FLA_CONJUGATE_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`.

3.2.1 Object destruction

If an object was created with `FLA_Obj_create` or `FLA_Obj_create_conf_to` a call to b `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`.

`obj`	object that describes the object

3.2.2 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). To extract the datatype and row and column dimensions of the matrix FLAME provides:

```
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.

`matrix`	object that describes the matrix
return value	datatype, row, or column dimension of matrix, respectively

To extract the address of the array that stores the matrix and the leading dimension of that array FLAME provides:

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

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

`matrix`	object that describes the matrix
return value	address and leading dimension of matrix, respectively

An example of how to use this information to implement a simple matrix-vector multiplication is given in Fig. 3.1. To understand the code one must understand that element α_{ij} of matrix A is stored in `buff_A[j*ldim_A+i]`, which conforms to column-major order. Similarly, χ_j, the jth element of x, and η_i, the ith element of y, are stored in `buff_x[j*inc_x]` and `buff_y[i*inc_y]`, respectively. (Here indexing starts at zero.)

```
#include "FLA.h"

void FLA_simple_mv_mult( FLA_Obj A, FLA_Obj x, FLA_Obj y )
{
  int
    datatype_A,    m_A, n_A, ldim_A,    m_x, n_y, inc_x,    m_y, n_y, inc_y;

  datatype_A = FLA_Obj_datatype( A );
  m_A        = FLA_Obj_length( A );
  n_A        = FLA_Obj_width ( A );
  ldim_A     = FLA_Obj_ldim  ( A );

  m_x        = FLA_Obj_length( x );
  n_x        = FLA_Obj_width ( x );
  m_y        = FLA_Obj_length( y );
  n_y        = FLA_Obj_width ( y );

  if ( m_x == 1 ) {
    m_x  = n_x;
    inc_x = FLA_Obj_ldim( x );
  }
  else inc_x = 1;

  if ( m_y == 1 ) {
    m_y  = n_y;
    inc_y = FLA_Obj_ldim( y );
  }
  else inc_y = 1;

  if ( datatype_A == FLA_DOUBLE ){
    double
      *buff_A, *buff_x, *buff_y;

    buff_A = ( double * ) FLA_Obj_datatype( A );
    buff_x = ( double * ) FLA_Obj_datatype( x );
    buff_y = ( double * ) FLA_Obj_datatype( y );

    for ( i=0; i<m_A; i++ ) buff_y[ i*inc_y ] = 0;

    for ( j=0; j<n_A; j++ )
      for ( i=0; i<m_A; i++ )
        buff_y[ i*inc_y ] += buff_A[ j*ldim_A+i ] * buff_x[ j*inc_x ];
  }
  else FLA_Abort( "datatype not yet supported", __LINE__, __FILE__ );
}
```

Figure 3.1: A simple matrix-vector multiplication routine.

3.2.3 Setting and extracting the contents

```
void FLA_Obj_set_contents     ( int trans, int m, int n, void *A, int ldA,+ &\\
                                FLA_Obj matrix )
void FLA_Obj_axpy_to_contents ( int trans, void *alpha, int m, int n, void *A,
                                int ldA, FLA_Obj matrix )
```

Purpose: Set the contents of the matrix object to those in $\mathtt{m} \times \mathtt{n}$ matrix `A` with leading dimension `ldA`. For the second call, α`A` is added to the current contents of `matrix`.

`trans`	indicates whether to transpose data
`alpha`	scaling factor
`m, n`	dimensions of `A`
`A`	array with data to be entered in matrix
`ldim`	leading dimension of `A`
`matrix`	object that describes the matrix

Here consistent means that the datatype of `ALPHA` and `A` must match that of the object `MATRIX`. Valid values for `trans` include `FLA_NO_TRANSPOSE`, `FLA_TRANSPOSE`, and `FLA_CONJUGATE_TRANSPOSE`. If `trans` equals `FLA_NO_TRANSPOSE`, m and n must equal the dimensions of `matrix`, respectively. Otherwise, they must equal the dimensions of the transpose of `matrix` and the data is transposed as it is entered in or added to `matrix`.

Similarly FLAME provides the following calls to extract the contents of an object:

```
void FLA_Obj_get_contents       ( int trans, FLA_Obj matrix,
                                  int m, int n, void A, int ldA )
void FLA_Obj_axpy_from_contents ( int trans, void *alpha, FLA_Obj matrix,
                                  int m, int n, void A, int ldA )
```

Purpose: Get the contents from the matrix object and store in $\mathtt{m} \times \mathtt{n}$ matrix `A` with leading dimension `ldA`. For the second call, α times the contents of `matrix` are added to the current contents of `A`.

`trans`	indicates whether data is to be transposed
`alpha`	scaling factor
`matrix`	object that describes the matrix
`m, n`	dimensions of `A`
`A`	array with data to be entered in matrix
`ldim`	leading dimension of `A`

3.3 A simple driver: matrix-vector multiplication

In Figure 3.2 we show a sample main program that uses most of the calls discussed so far.

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

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 A routine to be described next is used to fill A and x with values.

line 22 Compute $y = Ax$ using the FLAME matrix-vector multiply routine `FLA_Gemv` to be described later.

line 24–26 Print out the contents of A, x, and y. For each element the C print format `"%lf "` is used to print the contents as long floating point numbers. The calling sequence for `FLA_Obj_show` is given later in this chapter.

```
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   FLA_Gemv( FLA_NO_TRANSPOSE, ONE, A, x, ZERO, 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 }
```

Figure 3.2: A simple C driver for matrix-vector multiplication.

line 28 After the FLAME environment has finished it is finalized by a call to `FLA_Finalize`.

A sample routine for filling `A` and `x` with data is given in Fig. 3.3.

3.4 Views

Notice that in Fig. 2.3 became obvious that in stating a linear algorithm one frequently must partition a matrix, A, like

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

The primary mechanism used by our coding approach to hide complicated indexing is the notion of a *view*, which is simply 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:

```
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`

`A`	matrix to be partitioned
`mb, nb`	row and column dimensions of quadrant indicated by `quadrant`
`quadrant`	quadrant for which dimensions are given in `mb` and `nb`
ATL-ABR	views of TL, TR, BL, and BR quadrants

```
#include "FLAME.h"

#define BUFFER( i, j ) buff[ (j)*lda + (i) ]

void fill_matrix( FLA_Obj A )
{
  int datatype, m, n, lda;

  datatype = FLA_Obj_datatype( A );
  m        = FLA_Obj_length( A );
  n        = FLA_Obj_width ( A );
  lda      = FLA_Obj_ldim  ( A );

  if ( datatype == FLA_DOUBLE ){
    double *buff;
    int    i, j;

    buff = ( double * ) FLA_Obj_buffer( A );

    for ( j=0; j<n; j++ )
      for ( i=0; i<m; i++ )
        BUFFER( i,j ) = i+j*0.01;
  }
  else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
}
```

Figure 3.3: A simple routine for filling a matrix

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.

Also from Fig. 2.3, we notice that it is useful to be able to take a 2×2 partitioning of a given matrix A and repartition this so that submatrices can be identified that need to be updated and/or used for computation:

$$\textbf{repartition} \quad \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right) \quad \textbf{where} \quad A_{11} \textbf{ is } m_b \times n_b$$

Given that `ATL`, `ATR`, `ABL`, and `ABR` were the result of a call to `FLA_Part_2x2`, we would like create new views for this 3×3 partitioning from this 2×2 partitioning. 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`.

`ATL-ABR`	views of TL, TR, BL, and BR quadrants
`mb, nb`	row and column dimensions of A_{11}
`quadrant`	quadrant from which A_{11} is partitioned
`A00-A22`	views of A_{00}–A_{22}

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.

In order to update the partitioning of A into the four quadrants, we need to be able to update the descriptions of A_{TL}, A_{TR}, A_{BL}, and A_{BR}:

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

This update of the views 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`.

`ATL-ABR`	views of TL, TR, BL, and BR quadrants
`A00-A22`	views of A_{00}–A_{22}
`quadrant`	quadrant to which A_{11} is to be added

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.

We will see in subsequent chapters that we frequently will want to create a 2×1 partitioning of a given matrix A:

$$\textbf{partition } A \rightarrow \left(\frac{A_T}{\overline{A_B}} \right) \textbf{ where } A_T \textbf{ has } m_b \textbf{ rows}$$

For this we introduce 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.

`A`	matrix to be partitioned
`mb`	row dimension of side indicated by `side`
`side`	side for which row dimension is given
`AT, AB`	view of Top and Bottom part

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 like

$$\left(\frac{A_T}{\overline{A_B}} \right)$$

a repartitioning like

$$\textbf{repartition } \left(\frac{A_T}{\overline{A_B}} \right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array} \right) \textbf{ where } A_1 \textbf{ has } m_b \textbf{ rows}$$

is accomplished by 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`.

`AT, AB`	views of Top and Bottom sides
`mb`	row dimension of A_1
`side`	side from which A_1 is partitioned
`A0-A2`	views of A_0–A_2

Here `side` can take on the values `FLA_TOP` or `FLA_Bottom` 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, we may wish to update a 2×1 partitioning by adding A_1 to either A_T or A_B:

$$\textbf{continue with} \quad \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline\hline A_2 \end{array}\right)$$

For this FLAME provides 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`.

`AT, AB`	views of Top and Bottom sides
`A0-A2`	views of A_0–A_2
`side`	side from which A_1 is partitioned

Now `side` indicates whether A_1 is to be added to A_T or A_B.

Similarly, we may wish to create a 1×2 partitioning of a given matrix A:

$$\textbf{partition } A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \textbf{ where } \ A_T \textbf{ has } n \textbf{ rows}$$

For this we introduce the call

```
void FLA_Part_1x2 ( FLA_Obj A, FLA_Obj *AT, FLA_Obj *AB,
                    int nb, int side )
```

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

`A`	matrix to be partitioned
`nb`	column dimension of side indicated by `side`
`side`	side for which column dimension is given
`AL, AR`	view of Left and Right part

Here `side` can take on the values `FLA_LEFT` or `FLA_RIGHT` to indicate that `nb` equals the column dimension of A_L or A_R, respectively.

Given that matrix A is already partitioned like

$$\left(\begin{array}{c||c} A_L & A_R \end{array} \right)$$

a repartitioning like

$$\textbf{repartition} \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array}\right) \textbf{ where } \ A_1 \textbf{ has } n \textbf{ rows}$$

is accomplished by the call

```
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`.

`AL, AR`	views of Left and Right sides
`A0-A2`	views of A_0–A_2
`nb`	column dimension of A_1
`side`	side from which A_1 is partitioned

Now `side` indicates whether A_1 is partitioned from A_L or A_R.

Given a 1×3 partitioning of a given matrix A, updating a 1×2 partitioning by adding A_1 to either A_L or A_R,

$$\left(\begin{array}{c||c} A_L & A_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_0 & A_1 & A_2 \end{array} \right)$$

is 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`.

`AL, AR`	views of Left and Right sides
`side`	side to which A_1 is added
`A0-A2`	views of A_0–A_2

Parameter `side` indicates whether A_1 is added to A_L or A_R.

3.5 Other useful routines

To examine the contents of an object, we recommend the following routine:

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

Purpose: Print the contents of A.

`string1`	string to be printed before contents
`A`	descriptor for A
`format`	format to be used to print each individual element
`string2`	string to be printed after contents

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.

Chapter 4

Matrix-Matrix Multiplication: The Key to High Performance

by

John A. Gunnels

Greg M. Henry

Robert A. van de Geijn

As will will see in subsequent chapters, most important dense linear algebra operations can be organized so that most of the computation is in matrix-matrix multiplication. Thus, it is important to understand why and how matrix-matrix multiplication can be implemented to achieve high performance on modern microprocessors with hierarchical memories. In this chapter, we describe the basic approach used by our ITXGEMM matrix-matrix multiplication implementation [17].

4.1 The object of the game

The basics behind the design of a highly efficient matrix multiplication implementation are rather simple. To implement $C = AB + C$ where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively, one starts by partitioning these matrices like

$$C = \left(\begin{array}{c|c|c} C_{11} & \dots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \dots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \dots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \dots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \dots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \dots & B_{KN} \end{array}\right)$$

where C_{ij} is $m_b \times n_b$, A_{ip} is $m_b \times k_b$, and B_{pj} is $k_b \times n_b$. (Naturally, some blocks may not be exactly this block size, a minor detail.) Now, $C_{ij} = A_{i1}B_{1j} + \cdots + A_{iK}B_{Kj} + C_{ij}$. Given that C, A, and B all reside in main memory, the blockings of these matrices and the ordering of the updates $C_{ij} = A_{ip}B_{pj} + C_{ij}$ needs to be orchestrated so that movement into the caches of the processor is best amortized over computation.

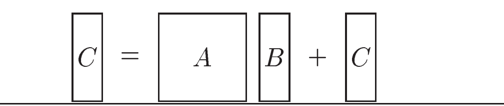
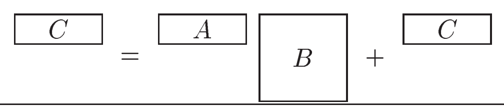
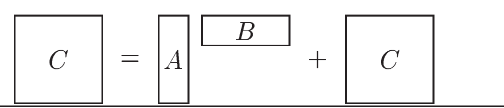
For the moment considering an architecture with two layers of cache memory, this initial partitioning creates blocks to be moved in and out of the L2 cache. Now C_{ij}, A_{ip}, and B_{pj} are themselves blocked and the computation with these even smaller blocks is orchestrated to optimally utilize the L1 cache. Finally, one further blocking is necessary to optimally utilize the registers. The purpose of the game now is to determine the optimal block size and the optimal ordering of the loops so that data movement between levels of the memory hierarchy is amortized over as much computation as possible.

Notice that for each level of the memory hierarchy we may need as many as three nested loops, not counting the registers. Thus, for a typical architecture with two caches and a main memory, one must consider as many as nine nested loops. Since the ordering of these loops will affect memory access patterns, one must consider up to $9! = 362880$ different loop orderings. By realizing that it is only the three loops for a given level of the hierarchy that need to be ordered, there are $3 \times 3! = 18$ possible loop orderings. For each of these loop orderings, one needs to consider different blockings at each level of the memory hierarchy. In other words, without some reasonable way of pruning the space of possible algorithms, one faces a formidable task.

The theory that we develop in the first part of this paper will allow us to propose a sensible heuristic for pruning the space of possible algorithms. By combining this heuristic with practical considerations we can reduce the number of different algorithms to only eight. Simultaneously, theoretical and practical considerations allow us to severely restrict the range of reasonable block sizes. The net result is a highly efficient implementation of matrix multiplication.

4.2 Special cases of matrix-matrix multiplication

The general form of a matrix-matrix multiply is $C \leftarrow \alpha AB + \beta C$ where C is $m \times n$, A is $m \times k$, and B is $k \times n$. We will use the following terminology when referring to a matrix-matrix multiply when two dimensions are large and one is small:

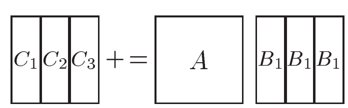
	Condition	Shape
Matrix-panel multiply	n is small	
Panel-matrix multiply	m is small	
Panel-panel multiply	k is small	

The following observation will become key to understanding concepts encountered in the rest of the paper: Partition

$$X = \left(X_1 \mid \cdots \mid X_{N_X} \right) = \begin{pmatrix} \hat{X}_1 \\ \hline \vdots \\ \hline \hat{X}_{M_X} \end{pmatrix}$$

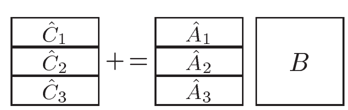
for $X \in \{A, B, C\}$, where C_j is $m \times n_j$, $\hat{C}_i$ is $m_i \times n$, A_p is $m \times k_p$, $\hat{A}_i$ is $m_i \times k$, B_j is $k \times n_j$, and $\hat{B}_p$ is $k_p \times n$. Then $C \leftarrow AB + C$ can be achieved as
multiple matrix-panel multiplies:

$$C_j \leftarrow AB_j + C_j \text{ for } j = 1, \ldots, N_C$$



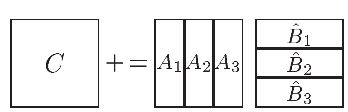
multiple panel-matrix multiplies:

$$\hat{C}_i \leftarrow \hat{A}_i B + \hat{C}_i \text{ for } i = 1, \ldots, M_C$$



or multiple panel-panel multiplies

$$C \leftarrow A_1 \hat{B}_1 + \cdots + A_{N_A} \hat{B}_{N_A}$$



4.3 A cost model for hierarchical memories

The memory hierarchy of a modern microprocessor is often viewed as the pyramid given in Fig. 4.1: At the

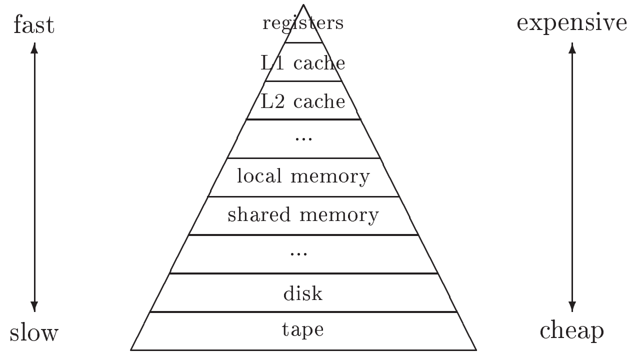


Figure 4.1: The hierarchical memories viewed as a pyramid.

top of the pyramid, there are the processor registers, with extremely fast access. At the bottom, there are disks and even slower media. As one goes down the pyramid, the amount of memory increases as well as the time required to access that that memory, while the cost of memory decreases.

We will model the above mentioned hierarchy naively as follows:

1. The memory hierarchy consists of L levels, indexed $0, \ldots, L-1$. Level 0 corresponds to the registers. We will often denote the ith level by L_i. Notice that on a typical current architecture L_1 and L_2 correspond the level 1 and level 2 data caches and L_3 corresponds to RAM.

2. Level h of the memory hierarchy can store S_h floating point numbers. Generally $S_0 \leq S_1 \leq \cdots \leq S_{L-1}$.

3. Loading a floating point number stored in level $h+1$ to level h costs time ρ_h. We will assume that $\rho_0 < \rho_1 < \cdots < \rho_{L-1}$.

4. Storing a floating point number from level h to level $h+1$ costs time σ_h. We will assume that $\sigma_0 < \sigma_1 < \cdots < \sigma_{L-1}$.

5. If $m_h \times n_h$ matrix C, $m_h \times k_h$ matrix A, and $k_h \times n_h$ matrix B are all stored in level h of the memory hierarchy then forming $C \leftarrow AB + C$ costs time $2 m_h n_h k_h \gamma_h$. (Notice that γ_h will depend on m_h, n_h, and k_h).

4.4 Building-blocks for matrix multiplication

Consider the matrix multiplication $C \leftarrow AB + C$ where $m_{h+1} \times n_{h+1}$ matrix C, $m_{h+1} \times k_{h+1}$ matrix A, and $k_{h+1} \times n_{h+1}$ matrix B are all stored in L_{h+1}. Let us assume that somehow an efficient matrix multiplication kernel exists for matrices stored in L_h. In this section, we develop three distinct approaches for matrix multiplication kernels for matrices stored in L_{h+1}.

Partition

$$
(4.1) \quad C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array} \right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array} \right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array} \right)
$$

where C_{ij} is $m_h \times n_h$, A_{ip} is $m_h \times k_h$, and B_{pj} is $k_h \times n_h$. We must now determine the optimal m_h, n_h, and k_h.

Algorithm 1

for $j = 1, \ldots, N$
    for $i = 1, \ldots, M$
        Load C_{ij} from L_{h+1} to L_h. $\quad m_h n_h \rho_h$
        for $p = 1, \ldots, K$
            Load A_{ip} from L_{h+1} to L_h. $\quad m_h k_h \rho_h$
            Load B_{pj} from L_{h+1} to L_h. $\quad k_h n_h \rho_h$
            Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$ $\quad 2 m_h n_h k_h \gamma_h$
        endfor
        Store C_{ij} from L_h to L_{h+1} $\quad m_h n_h \sigma_h$
    endfor
endfor

Figure 4.2: Multiple panel-panel multiply based blocked matrix-matrix multiplication.

4.4.1 Multiple panel-panel multiplies in L_h

Noting that $C_{ij} \leftarrow \sum_{p=1}^{K} A_{ip} B_{pj} + C_{ij}$, let us consider the algorithm in Fig. 4.2 for computing the matrix-matrix multiplication. In that figure the costs of the various operations are shown to the right. The order of the outer-most loops is irrelevant to the analysis.

The cost for updating C is given by

$$\sum_{i=1}^{M} \sum_{j=1}^{N} \left[m_h n_h \rho_h + m_h n_h \sigma_h + \sum_{p=1}^{K} \left[k_h m_h \rho_h + k_h n_h \rho_h + 2 m_h n_h k_h \gamma_h \right] \right]$$

$$= m_{h+1} n_{h+1} (\rho_h + \sigma_h) + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{n_h} + m_{h+1} n_{h+1} k_{h+1} \frac{\rho_h}{m_h} + 2 m_{h+1} n_{h+1} k_{h+1} \gamma_h \tag{4.2}$$

Since γ_{h+1} is defined to be the cost of a floating point operation when all three matrices are stored in L_{h+1}, we find that by we also have that the cost is given by

$$2 m_{h+1} n_{h+1} k_{h+1} \gamma_{h+1} \tag{4.3}$$

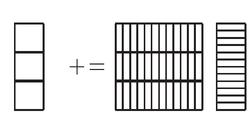
Thus, by dividing 4.2 by $2 m_{h+1} n_{h+1} k_{h+1}$ the effective cost per floating point operation at this level is given by

$$\gamma_{h+1} = \frac{\rho_h + \sigma_h}{2 k_{h+1}} + \frac{\rho_h}{2 n_h} + \frac{\rho_h}{2 m_h} + \gamma_h$$

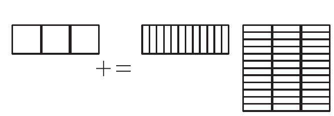
The question now is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \le S_h$. The smaller k_h, the more space in L_h can be dedicated to C_{ij} and thus the smaller the fractions ρ_h / m_h and ρ_h / n_h can be made. A good strategy is thus to let essentially all of L_h be dedicated to C_{ij}, i.e., $m_h n_h \approx S_h$. The minimum is then attained when essentially $m_h \approx n_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $n_{h+1} = n_h$ for the above cost of γ_{h+1} to be achieved. Thus, the above already for the special cases

$$\left(\begin{array}{c} C_{11} \\ \hline\hline \vdots \\ \hline\hline C_{M1} \end{array} \right) \mathrel{+}= \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline\hline \vdots & & \vdots \\ \hline\hline A_{M1} & \cdots & A_{MK} \end{array} \right) \left(\begin{array}{c} B_{11} \\ \hline \vdots \\ \hline B_{K1} \end{array} \right) \tag{4.4}$$



$$\left(\begin{array}{c||c||c} C_{11} & \cdots & C_{1N} \end{array} \right) \mathrel{+}= \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \end{array} \right) \left(\begin{array}{c||c||c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array} \right) \tag{4.5}$$



Here the distance between single/thin lines is k_h and between double/thick lines $m_h = n_h$, where k_h is much smaller than m_h and n_h. The significance of this will become apparent later.

Algorithm 2

for $p = 1, \ldots, K$	
for $i = 1, \ldots, M$	
Load A_{ip} from level $h+1$ to level h.	$m_h k_h \rho_h$
for $j = 1, \ldots, N$	
Load C_{ij} from level $h+1$ to level h.	$m_h n_h \rho_h$
Load B_{pj} from level $h+1$ to level h.	$k_h n_h \rho_h$
Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$	$2 m_h n_h k_h \gamma_h$
Store C_{ij} from level h to level $h+1$	$m_h n_h \sigma_h$
endfor	
endfor	
endfor	

Figure 4.3: Multiple matrix-panel multiply based blocked matrix-matrix multiplication.

Note 1 *The above analysis shows that for the ordering of the loops given in Alg. 1, the strategy should be to load L_h with blocks of C that fill most of L_h. The intuitive reason is that the cost of moving blocks C_{ij} between L_h and L_{h+1} is amortized over computation with many smaller blocks A_{ip} and B_{pj}, which are "streamed" from L_{h+1}. Simultaneously, the cost of bringing each of these smaller blocks into L_h is itself amortized over many computations, since C_{ij} is essentially as large as possible and almost square.*

The inner-most loop in Alg. 1 implements multiple panel-panel multiplies since k_h is small relative to m_h and n_h. Thus the name of this section.

4.4.2 Multiple matrix-panel multiplies in L_h

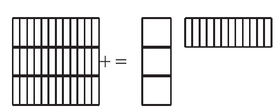
Moving the loops over l and i to the outside we get the algorithm in Fig. 4.3. Performing an analysis similar to that given in Section 4.4.1 the effective cost of a floating point operation is now given by

$$\gamma_{h+1} = \frac{\rho_h}{2n_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2m_h} + \gamma_h \tag{4.6}$$

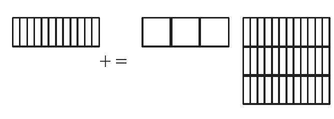
Again, the question is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \le S_h$. Note that the smaller n_h, the more space in L_h can be dedicated to A_{il} and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2m_h$ can be made. A good strategy is thus to let essentially all of L_h be dedicated to A_{il}, i.e., $m_h k_h \approx S_h$. The minimum is then attained when essentially $m_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $m_{h+1} = m_h$ or $k_{h+1} = k_h$ for the above cost of γ_{h+1} to be achieved. In other words, the above holds for the special cases

$$\left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline\hline \vdots & & \vdots \\ \hline\hline C_{M1} & \cdots & C_{MN} \end{array}\right) += \left(\begin{array}{c} A_{11} \\ \hline\hline \vdots \\ \hline\hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right) \tag{4.7}$$



$$\left(\begin{array}{c||c||c} C_{11} & \cdots & C_{1N} \end{array}\right) += \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \end{array}\right) \left(\begin{array}{c||c||c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right) \tag{4.8}$$



here the distance between single/thin lines is n_h and between double/thick lines is $m_h = k_h$, where n_h is much smaller than m_h and k_h. This will become important later when we notice that these occur naturally as we move up and down the memory hierarchy.

Note 2 *The above analysis shows that for the ordering of the loops given in Alg. 2, the strategy should be to load L_h with blocks of A that fill most of L_h. The intuitive reason is that the cost of moving blocks*

Algorithm 3

for $j = 1, \ldots, N$

  for $p = 1, \ldots, K$

    Load B_{pj} from level $h+1$ to level h. $\qquad k_h n_h \rho_h$

    for $i = 1, \ldots, M$

      Load C_{ij} from level $h+1$ to level h. $\qquad m_h n_h \rho_h$

      Load A_{ip} from level $h+1$ to level h. $\qquad m_h k_h \rho_h$

      Update $C_{ij} \leftarrow A_{ip} B_{pj} + C_{ij}$ $\qquad 2 m_h n_h k_h \gamma_h$

      Store C_{ij} from level h to level $h+1$ $\qquad m_h n_h \sigma_h$

    endfor

  endfor

endfor

Figure 4.4: Multiple panel-matrix multiply based blocked matrix-matrix multiplication.

A_{ip} between L_h and L_{h+1} is amortized over computation with many smaller blocks C_{ij} and B_{pj}, which are "streamed" from L_{h+1}. Simultaneously, the cost of bringing each of these smaller blocks into L_h is itself amortized over many computations, since A_{ip} is essentially as large as possible and almost square.

The inner-most loop in Alg. 2 implements multiple matrix-panel multiplies since n_h is small relative to m_h and k_h. Thus the name of this section.

4.4.3 Multiple panel-matrix multiplies in L_h

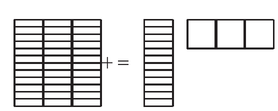
Finally, moving the loops over p and j to the outside we get the algorithm given in Fig. 4.4. This time, the effective cost of a floating point operation is given by

$$\gamma_{h+1} = \frac{\rho_h}{2m_{h+1}} + \frac{\rho_h + \sigma_h}{2k_h} + \frac{\rho_h}{2n_h} + \gamma_h \tag{4.9}$$

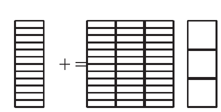
Again, the question is how to find the m_h, n_h, and k_h that minimize γ_{h+1} under the constraint that C_{ij}, A_{ik} and B_{kj} all fit in L_h, i.e., $m_h n_h + m_h k_h + n_h k_h \leq S_h$. Note that the smaller m_h, the more space in L_h can be dedicated to B_{pj} and thus the smaller the fractions $(\rho_h + \sigma_h)/2k_h$ and $\rho_h/2n_h$ can be made. A good strategy is thus to let essentially all of L_h be dedicated to B_{pj}, i.e., $n_h k_h \approx S_h$. The minimum is then attained when essentially $n_h \approx k_h \approx \sqrt{S_h}$.

Notice that it suffices to have $n_{h+1} = n_h$ and/or $k_{h+1} = k_h$ for the above cost of γ_{h+1} to be achieved. In other words, the above holds for the special cases

$$\left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right) + = \left(\begin{array}{c} A_{11} \\ \hline \vdots \\ \hline A_{M1} \end{array}\right) \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \end{array}\right) \tag{4.10}$$



$$\left(\begin{array}{c} C_{11} \\ \hline \vdots \\ \hline C_{M1} \end{array}\right) + = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right) \left(\begin{array}{c} B_{11} \\ \hline \vdots \\ \hline B_{K1} \end{array}\right) \tag{4.11}$$



an observation that will become important later.

Note 3 *The above analysis shows that for the ordering of the loops given in Alg. 3, the strategy should be to load L_h with blocks of B that fill most of L_h. The intuitive reason is that the cost of moving block B_{pj} between L_h and L_{h+1} is amortized over computation with many smaller blocks C_{ij} and A_{ip}, which are "streamed" from L_{h+1}. Simultaneously, the cost of bringing each of these smaller blocks into L_h is itself amortized over many computations, since B_{pj} is essentially as large as possible and almost square.*

4.5 A heuristic for a multi-level algorithm

Key observations so far are

- From Section 4.4: If one were to perform a matrix-matrix multiplication with all operands stored in L_{L-1} (as would naturally occur as part of an application) then this operation should be staged to perform multiple panel-panel, matrix-panel, or panel-matrix multiplies, moving data to and/from L_{L-2}.
- Each of the individual panel-panel, matrix-panel, or panel-matrix multiplication has the property that all operands reside in L_{L-2} and should be staged itself be implemented by utilizing L_{L-3} efficiently.
- Whenever all operands of the matrix-matrix multiply fill most of L_{h+1}
  - a panel-panel multiply can be efficiently implemented by performing multiple matrix-panel or panel-matrix multiplies in L_h. This follows from (4.7) and (4.10).
  - a matrix-panel multiply can be efficiently implemented by performing multiple panel-panel or panel-matrix multiplies in L_h. This follows from (4.4) and (4.11).
  - a panel-matrix multiply can be efficiently implemented by performing multiple panel-panel or matrix-panel multiplies in L_h. This follows from (4.5) and (4.8).
- From Section 4.2 we conclude that even the matrix-matrix multiply in L_{L-1} can be staged as multiple panel-panel, matrix-panel, or panel-matrix multiplies.

Thus, we conclude that at each layer of the memory hierarchy we should stage the matrix-matrix multiply as multiple panel-panel, matrix-panel, or panel-matrix multiplies.

These observations leads to the following heuristic for implementing the matrix-matrix multiply:

- If in level L_{h+1} one encounters a panel-panel multiply, an optimal implementation will utilize a matrix-panel or panel-matrix multiply in L_h. Moreover, the optimal matrix-panel or panel-matrix multiply in L_h will pick $k_h \approx \sqrt{S_h}$ and thus $k_{h+1} = k_h \approx \sqrt{S_h}$. (Recall that we already determined that $m_{h+1} \approx n_{h+1} \approx \sqrt{S_{h+1}}$ was a desirable blocking.)
- If in level L_{h+1} one encounters a matrix-panel multiply, an optimal implementation will utilize a panel-panel or panel-matrix multiply in L_h. Moreover, the optimal panel-panel or panel-matrix multiply in L_h will pick $n_h \approx \sqrt{S_h}$ and thus $n_{h+1} = n_h \approx \sqrt{S_h}$.
- If in level L_{h+1} one encounters a panel-matrix multiply, an optimal implementation will utilize a panel-panel or matrix-panel multiply in L_h. Moreover, the optimal panel-panel or matrix-panel multiply in L_h will pick $m_h \approx \sqrt{S_h}$ and thus $m_{h+1} = m_h \approx \sqrt{S_h}$.

The decision made at a give level L_{h+1} is summarized in Fig. 4.5. In other words, at each level of the hierarchy, L_h, one of the three operands is chosen to be approximately $\sqrt{S_h} \times \sqrt{S_h}$ and fills most of that memory layer while the other two operands are either approximately $\sqrt{S_{h-1}} \times \sqrt{S_h}$ of $\sqrt{S_h} \times \sqrt{S_{h-1}}$. Another way of viewing this is that one of the operands is moved into level L_h while the other two operands are streamed (moved in in smaller submatrices) from level L_{h+1}. Notice that if m_0, n_0, and k_0 are now given, all block sizes are approximately determined by the above analysis.

The above heuristic leaves a number of questions:

1. What choices to make in memory layer L_{L-1} since there the shape of the matrices may not cleanly fall into any of these categories. In particular, What if at some level L_{L-1} the smallest dimension is much larger than $\sqrt{S_{L-2}}$? Notice that our theory actually does answer this question since...
2. What if at some level L_{h+1} the "small" dimension is much smaller than $\sqrt{S_h}$. Indeed, what if more than one dimension is "small" relative to $\sqrt{S_h}$.

We will visit these questions later.

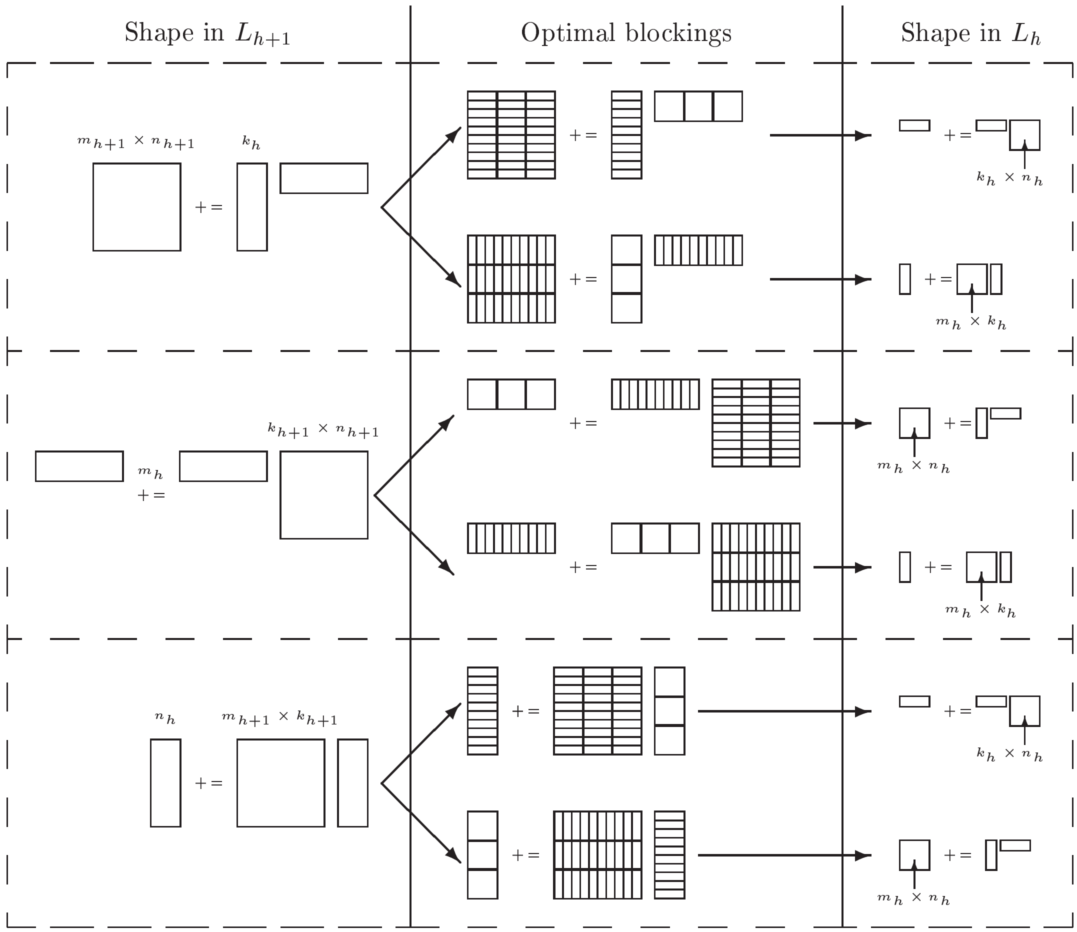


Figure 4.5: Optimal partitioning at memory level L_h and resulting shapes in level L_{h+1}.

4.6 Practical considerations

In the previous section we developed a heuristic for implementation of matrix-matrix multiplication that best amortizes movement of data between memory hierarchies from a local point of view. However, there are many issues associated with actual implementation that are ignored by the analysis and the heuristic. In this section we discuss implementation details that do take some of those issues into account. We do so by noting certain machine characteristics that to our knowledge hold for a wide variety of architectures.

Rather than making the register level our lowest level, we start with L_1, the L1 cache. The reason for this is that at that level loop indexing is a major concern and thus a lot of machine details must be considered. We will not discuss how registers come into play since this goes beyond the scope of this chapter.

4.6.1 L_1-kernels (lowest level)

While previously we have discussed the shape of the computation to be performed in the L_1 cache to be a panel-panel, matrix-panel or panel-matrix multiply, in order to keep loop indexing down to a minimum, at that level our kernels actually perform multiple such operations. Specifically, this allows the loops to be unrolled to eliminate most of the loop overhead.

Matrix-panel L_1-kernel

Our theory indicates that one of the operations that may be encountered at the L_1 level is a matrix-panel multiply. Instead, we consider the operation $C \leftarrow AB + C$ where C is $m_1 \times n$, A is $m_1 \times k_1$, and B is $k_1 \times n$, with $n >> n_0$. The idea is that the overhead of performing the multiple matrix-panel multiplies encountered in the matrix-panel multiply based approaches discussed in Sec. 4.4.2 is amortized over many such matrix-panel multiplies.

The question is how to perform the computation so that elements of A are used with a frequency so that the cache-replacement policy keeps A in the L_1 cache. To achieve this, C is computed a few columns at a time. For example, if C is computed a single column at a time, for every m_1 elements of C and k_1 elements of B all $m_1 \times k_1$ elements of A are accessed and which tends to keep A in the L1 cache.

Panel-matrix L_1-kernel

We similar treat the case where the shape of matrices in L_1 is a panel-matrix multiply. This time, we consider the operation $C \leftarrow AB + C$ where C is $m \times n_1$, A is $m \times k_1$, and B is $k_1 \times n_1$, with $m >> m_0$. The idea is that the overhead of performing the multiple panel-matrix multiplies encountered in the panel-matrix multiply based approaches discussed in Sec. 4.4.3 is amortized over many such panel-matrix multiplies.

This time, the computation must be orchestrated in such a way that the elements of B are used with a frequency so that the cache-replacement policy keeps B in the L_1 cache. To achieve this, C is computed a few rows at a time.

Panel-panel L_1-kernel

We could similarly treat the case where the shape of matrices in L_1 is a panel-panel multiply. If so, we would consider the operation $C \leftarrow AB + C$ where C is $m_1 \times n_1$, A is ${}_1m \times k$, and B is $k \times n_1$, with $k >> k_0$. However, for this approach leads to a utilization of the registers that requires elements of C to be loaded to and stored from registers with great frequency. This inherently leads to an L_1-kernel that is slower than either the matrix-panel or panel-matrix multiply kernel. Thus, we don't consider this approach to be a viable L_1-kernel. We will analyze approaches for L_2-kernels under the assumption that this particular L_1-kernel is not available. **Thus, in Fig. 4.6 we delete from Fig. 4.5 the panel-panel multiply as a possible shape in the L_1 level.**

4.6.2 L_2-kernel

We now ask ourselves the question of what possible algorithms can be implemented when we perform a multiplication with matrices that are stored in L_2. To answer this question, we turn to Fig. 4.6 in which we delete the shapes and algorithms in Fig. 4.5 that now cannot be supported.

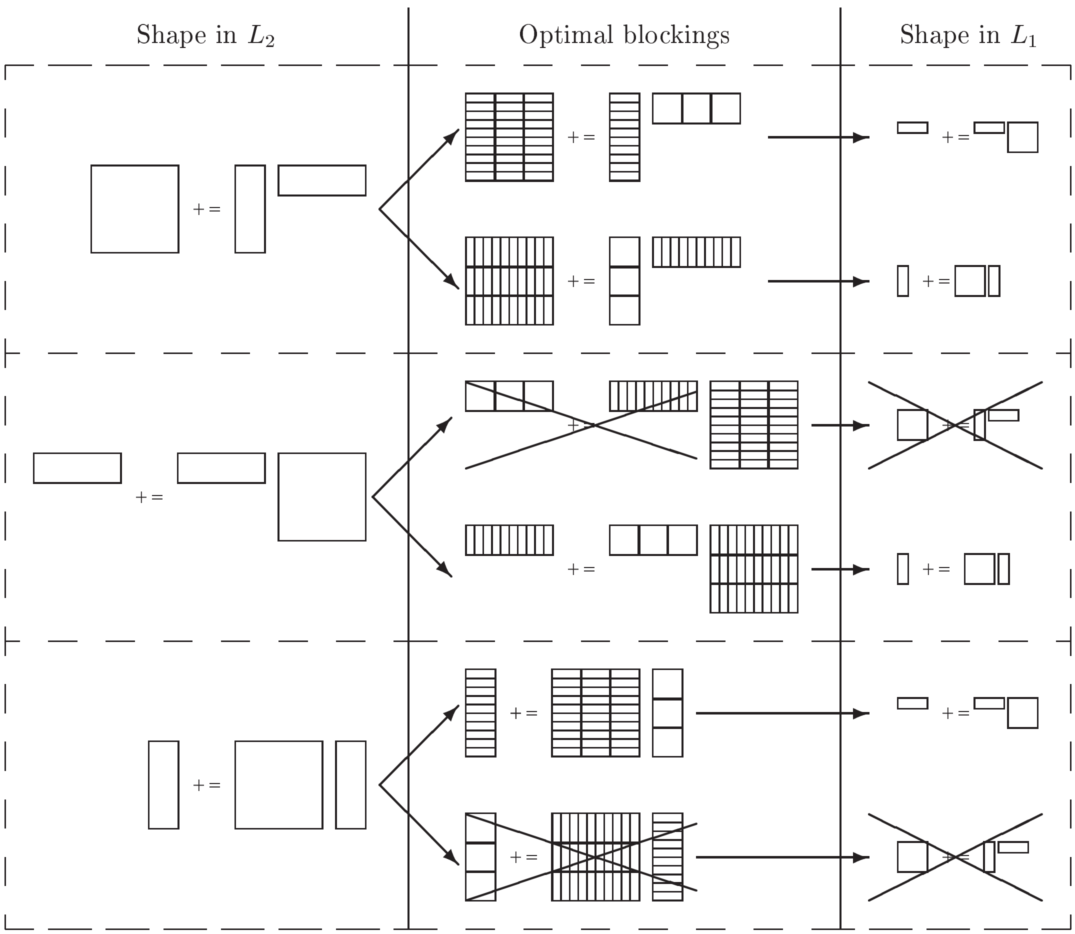


Figure 4.6: Possible algorithms for matrices in memory level L_2 given that our kernel at L_1 can only accommodate matrix-panel and panel-matrix multiplication.

Just like for the L_1-kernel, our L_2-kernel assumes that one matrix of size approximately $\sqrt{S_2} \times \sqrt{S_2}$ is moved into L_2 after which multiple panel-panel, matrix-panel, or panel-matrix multiplies commence. Depending on which matrix occupies most of L_2, one of the four blockings remaining in Fig. 4.6 is used to smaller subproblems that can be passed to the L_1-kernel.

For the sake of consistency, our L_2-kernel performs multiple panel-panel, matrix-panel, or panel-matrix multiplies, like the L_1-kernel. This is illustrated in Fig. 4.7.

Notice that if the L_2-kernel implements multiple panel-panel multiplies, there is a choice of two possible blockings. One leads to a panel-matrix multiply as basic operation in the L_1 level, the other to a matrix-panel multiply. The question naturally becomes which of the two to use. Notice from (4.9) that the panel-matrix multiply based algorithm has a cost of

$$\gamma_2^{\mathrm{PM}} = \frac{\rho_1}{2m_2} + \frac{\rho_1 + \sigma_1}{2k_1} + \frac{\rho_1}{2n_1} + \gamma_1$$

while (4.6) shows the matrix-panel multiply based algorithm has a cost of

$$\gamma_2^{\mathrm{MP}} = \frac{\rho_1}{2n_2} + \frac{\rho_1 + \sigma_1}{2k_1} + \frac{\rho_1}{2m_1} + \gamma_1$$

If parameters ρ_1 and σ_1 are equal in both these equations, the panel-matrix multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_2^{\mathrm{PM}} < \gamma_2^{\mathrm{MP}}$ or

$$\frac{1}{m_2} + \frac{1}{n_1} < \frac{1}{n_2} + \frac{1}{m_1}$$

Note that one can expect γ_1 to be equal for both equations since in our situation the L_1-kernel is one and the same for both approaches.

4.6.3 L_3-kernel

For current generation microprocessors, the L_3 level is typically the primary RAM of the processor. For this reason, our discussion will target that situation.

Notice that while in this level on the surface it may appear that one should analyze the general matrix-matrix multiplication $C \leftarrow \alpha AB + \beta C$ for general $m \times n$ matrix C, $m \times k$ matrix A, and $k \times n$ matrix B. However, a common use of matrix-matrix multiplication is as part of the implementation of other dense linear algebra algorithms, e.g. for factorization operations like LU, Cholesky, and QR factorization. In those algorithms, as implemented in LAPACK, the matrix-matrix multiply invariably appears as a panel-panel, matrix-panel or panel-matrix multiply. Indeed, the width of the panels involved are determined by the width that makes matrix-matrix multiplication operate at peak performance. Thus, the most important cases of to analyze are exactly those where one of m, n, or k equals approximately $\sqrt{S_2}$. Thus, we again analyze the panel-panel, matrix-panel, and panel-matrix multiply before proceeding with the general case. To do so, again consider Fig. 4.5.

L_3-kernel for panel-panel multiply

In this case $k = k_2 \approx \sqrt{S_2}$. As for the L_2-kernel there are now two choices for implementation: a panel-matrix multiply based algorithm and a matrix-panel multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\mathrm{PM}} = \frac{\rho_2}{2m} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\mathrm{MP}} = \frac{\rho_2}{2n} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2m_2} + \gamma_2$$

If parameters ρ_2, σ_2, and γ_2 are equal in both these equations, the panel-matrix multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_3^{\mathrm{PM}} < \gamma_3^{\mathrm{MP}}$ or

$$\frac{1}{m} + \frac{1}{n_2} < \frac{1}{n} + \frac{1}{m_2}$$

Note that one cannot expect γ_2 to be equal for both equations since in our situation the L_2-kernel for each of the L_2-kernels is not the same for both approaches.

L_3-kernel for panel-matrix multiply

In this case $m = m_2 \approx \sqrt{S_2}$. Unlike for the L_2-kernel there are now two choices for implementation: a panel-panel or matrix-panel multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\rm PP} = \frac{\rho_2}{2m_2} + \frac{\rho_2 + \sigma_2}{2k} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\rm MP} = \frac{\rho_2}{2n} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2m_2} + \gamma_2$$

Again, if parameters ρ_2, σ_2, and γ_2 are equal in both these equations, the panel-panel multiply based algorithm outperforms the matrix-panel multiply based algorithm when $\gamma_3^{\rm PP} < \gamma_3^{\rm MP}$ or

$$??? < \frac{1}{n} + \frac{1}{m_2}$$

Again, one cannot expect γ_2 to be equal for both equations since in the L_2-kernel for the two approaches is not the same.

L_3-kernel for matrix-panel multiply

In this case $n = n_2 \approx \sqrt{S_2}$. There are two choices for implementation: a panel-panel and a panel-matrix multiply based algorithm. The first yields an effective cost per floating point operation of

$$\gamma_3^{\rm PP} = \frac{\rho_2}{2m_2} + \frac{\rho_2 + \sigma_2}{2k} + \frac{\rho_2}{2n_2} + \gamma_2$$

while the second yields

$$\gamma_3^{\rm PM} = \frac{\rho_2}{2m} + \frac{\rho_2 + \sigma_2}{2k_2} + \frac{\rho_2}{2n_2} + \gamma_2$$

If parameters ρ_2, σ_2, and γ_2 are equal in both these equations, the panel-panel multiply based algorithm outperforms the panel-matrix multiply based algorithm when $\gamma_3^{\rm PP} < \gamma_3^{\rm PM}$ or

$$\frac{1}{m} + \frac{1}{n_2} < ???$$

Again, one cannot expect γ_2 to be equal for both equations since in the L_2-kernel for the two approaches is not the same.

4.7 A family of algorithms

We now turn the observations made above into a practical implementation.

High-performance implementations of matrix multiplication typically start with an "inner-kernel". This kernel carefully orchestrates the movement of data in and out of the registers and the computation under the assumption that one or more of the operands are in the L1 cache. For our implementation on the Intel Pentium (R) III processor, the inner-kernel performs the operation $C = A^T B + \beta C$ where 64×8 matrix A is kept in the L1 cache. Matrices B and C have a large number of columns, which we view as multiple-panels, with each panel of width one. Thus, our inner-kernel performs a multiple matrix-panel multiply (MMP) with a transposed resident matrix A. The technical reasons why this particular shape was selected go beyond the scope of this paper.

While it may appear that we thus only have one of the three kernels for operation in the L1 cache, notice that for the submatrices with which we compute at that level one can instead compute $C^T = B^T A + C^T$,

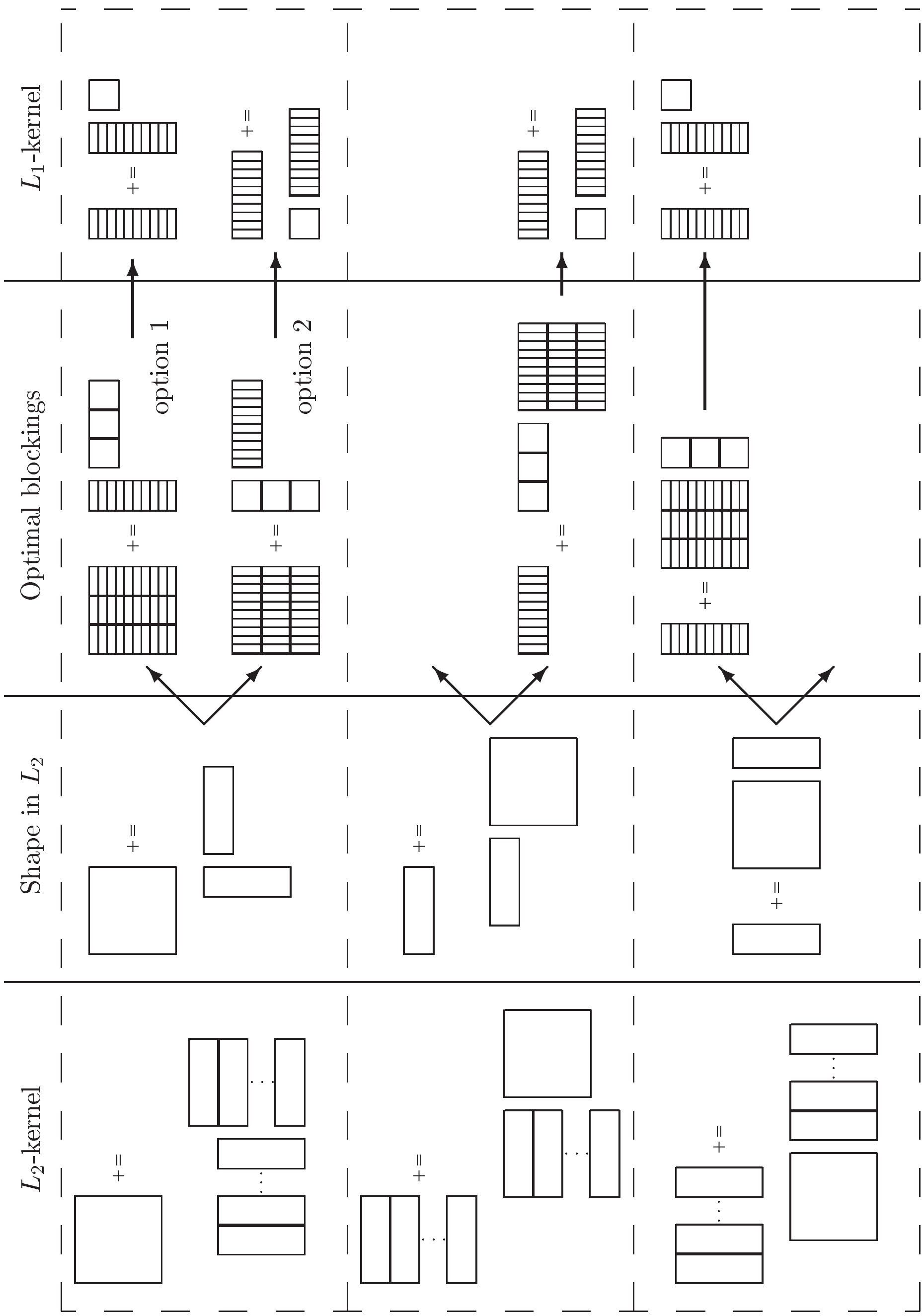


Figure 4.7: A tree of possible algorithms for matrices in memory level L_3 given that our kernel at L_1 can only accommodate matrix-panel and panel-matrix multiplication.

reversing the role of A and B. This simple observation allows us to claim that we also have an inner-kernel that performs a multiple panel-matrix multiply (MPM).

Let us introduce a naming convention for a family of algorithms that perform the discussed algorithms at different levels of the memory hierarchy:

$$\langle\text{kernel at } L_3\rangle\text{-}\langle\text{kernel at } L_2\rangle\text{-}\langle\text{kernel at } L_1\rangle.$$

For example MPP-MPM-MMP will indicate that the L_3-kernel uses multiple panel-panel multiplies, calls the L_2-kernel that uses multiple matrix-panel multiplies, which in turn calls the L_1-kernel that uses multiple panel-matrix multiplies. Given the constraint that only two of the possible three kernel algorithms are implemented at L_1, the tree of algorithms in Fig. 4.7 can be constructed.

4.8 Performance

In this section, we report performance attained by the different algorithms. Details regarding the performance test bed can be found in 1.4. For the usual matrix dimensions m, n, and k, we use the operation count $2mnk$ for a matrix-matrix multiplication. We tested performance of the operation $C = C - AB$ ($\alpha = -1$ and $\beta = 1$) since this is the case most frequently encountered when matrix multiplication is used in libraries like LAPACK.

4.8.1 Implementations tested

It turns out that whenever all operands start by being stored in main memory, there is no noticeable difference between the different loop orderings at that level. In other words, MPM-MMP-MPM achieves performance that is essentially identical to MPP-MMP-MPM. Thus, we only report performance for the following variants: MPM-MMP-MPM, MMP-MPM-MMP, MPM-MPP-MPM, and MPM-MPP-MMP.

4.8.2 Determining optimal block sizes

Our first experiment is intended to demonstrate that the block size chosen for the matrix that remains resident in the L2 cache has a clear effect on the overall performance of the matrix multiplication routine. In Fig. 4.8(a) we report performance attained as a function of the fraction of the L2 cache filled with the resident matrix when a matrix multiplication with $k = m = n = 1000$ is executed. This experiments tests our theory that reuse of data in the L2 cache impacts overall performance as well as our theory that the resident matrix should occupy "most" of the L2 cache. Note that performance improves as a larger fraction of the L2 cache is filled with the resident matrix. Once the resident matrix fills more than half of the L2 cache, performance starts to diminish. This is consistent with the theory which tells us that some of the cache must be used for the matrices that are being streamed from main memory. Once more than 3/4 of the L2 cache is filled with the resident matrix, performs drops significantly. This is consistent with the scenario where parts of the other matrices start vacating parts of the resident matrix from the L2 cache.

The exact reason why the MPM-MMP-MPM variant performs better when the block size is chosen appropriately is not entirely clear. Most likely, it has to do with the details of the packing and unpacking routines that are part of the implementation.

Based on the above experiment, we fix the block size for the resident matrix in the L2 cache to 128×128, which fills exactly half of this cache, for the remaining experiments.

4.8.3 Resident matrices

The next set of experiments show that the cost of moving a submatrix into the L2 cache and then amortizing the cost of this memory operation over as much computation as possible is indeed observable in practice.

Matrix A resident in L2: In Fig. 4.8(b), dimensions m and k are fixed to 128. This implies that matrix A fills half of the L2 cache. Notice that variant MPM-MMP-MPM will keep 128×128 submatrices of A resident in the L2 cache. Thus, one would expect performance to increase smoothly as n is increased. For the other variants, one would expect a drop in performance whenever n becomes slightly larger than a multiple of 128, since they attempt to keep 128×128 submatrices of C or B in the L2 cache: whenever n is slightly larger than 128, one of the submatrices of A or B is relatively small.

Matrix B resident in L2: In Fig. 4.8(c), dimensions n and k are fixed to 128. This implies that matrix B fills half of the L2 cache. Notice that variant MMP-MPM-MMP will keep 128×128 submatrices of B resident in the L2 cache. Thus, one would expect performance to increase smoothly as m is increased. For the other variants, one would expect a drop in performance whenever k becomes slightly larger than a multiple of 128, since they attempt to keep 128×128 submatrices of C or A in the L2 cache: whenever m is slightly larger than 128, one of the submatrices of C or A is relatively small.

Matrix C resident in L2: In Fig. 4.8(d), dimensions m and n are fixed to 128. This implies that matrix C fills half of the L2 cache. Notice that variants MPM-MPP-MPM and MPM-MPP-MMP will both keep 128×128 submatrices of C resident in the L2 cache. Thus, one would expect performance to increase smoothly as k is increased. For the other variants, one would expect a drop in performance whenever k becomes slightly larger than a multiple of 128, since they attempt to keep 128×128 submatrices of A or B in the L2 cache: whenever k is slightly larger than 128, one of the submatrices of A or B is relatively small. Unfortunately, we cannot observe this phenomena, since the L1 kernel takes a performance hit every time k is slightly larger than 64.

4.8.4 Commonly encountered shapes

The most commonly encountered special cases of matrix-matrix multiplication are the matrix-panel, panel-matrix, and panel-panel multiplications. Not only did they show up in this paper as the shape that is encountered at each level of the memory hierarchy, but it is also the shape that shows up when implementing other matrix operations like LU, Cholesky, and QR factorization, for example as part of LAPACK.

Matrix-panel multiply In Fig. 4.9(a) we report performance as a function of n (the number of columns in the panel) when m and k are fixed to be large. Notice that our theory indicates that when a matrix-panel multiply is performed in main memory, the L2 kernel should perform multiple panel-matrix or panel-panel multiplies. The theory indicates that MPP-MMP-MPM and MPM-MMP-MPM, which perform a matrix-panel multiply in the L2 level, should not be good choices. The data in Fig. 4.9(a) supports this.

Panel-matrix multiply In Fig. 4.9(b) we report performance as a function of m (the number of columns in the panel) when n and k are fixed to be large. Notice that our theory indicates that when a panel-matrix multiply is performed in main memory, the L2 kernel should perform multiple matrix-panel or panel-panel multiplies. Thus, MPM-MMP-MPM, MPM-MPP-MPM, or MPM-MPP-MMP should perform well. The theory indicates that MMP-MPM-MMP, which perform a panel-matrix multiply in the L2 level, would not be good choice. The data in Fig. 4.9(b) supports this in the sense that MMP-MPM-MMP, which for other matrix shapes frequently did well, does not perform quite as well for small m.

Panel-panel multiply In Fig. 4.9(c) we report performance as a function of k (the number of columns in the panel) when m and n are fixed to be large. Notice that our theory indicates that when a panel-panel multiply is performed in main memory, the L2 kernel should perform multiple matrix-panel or panel-matrix multiplies. The theory indicates that variants *-MPP-*, which perform multiple panel-panel multiplies in the L2 level, should not be good choices. The data in Fig. 4.9(c) supports this in the range $64 \leq k \leq 128$. Interestingly enough, it does not appear to be true in the range $k \leq 64$ for MPM-MPP-MMP. We believe that can be attributed to the fact that in that range there isn't sufficient opportunity for reuse of data in the L2 cache. As a result, it is best to ignore it, which is essentially what MPM-MPP-MMP does.

4.8.5 Square matrices

Matrix multiplication with square matrices is relatively uncommon in practice. However, it is commonly presented in papers. Thus, for good measure, we include Fig. 4.9(a).

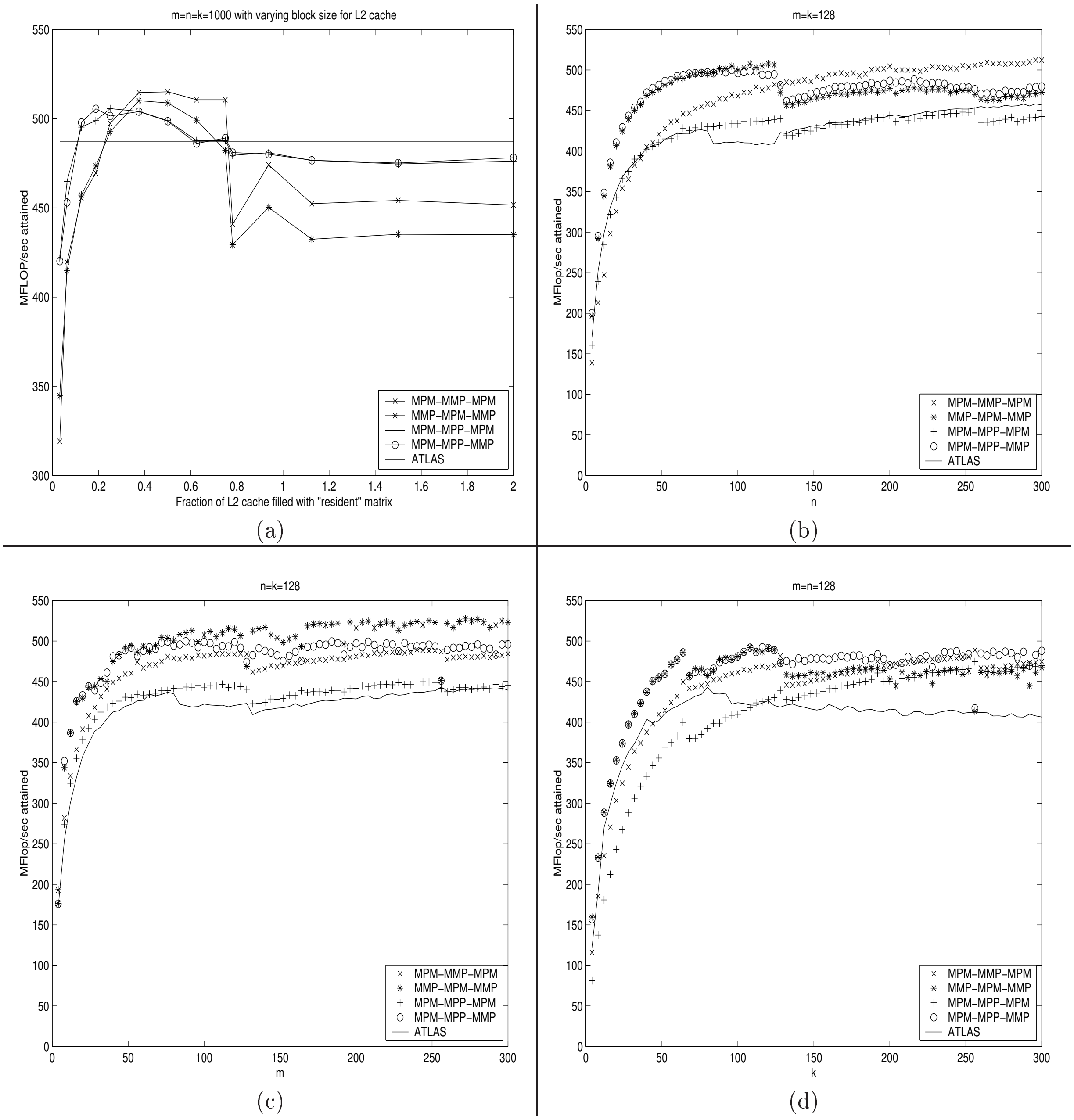


Figure 4.8: Performance of the different members of the family on an Intel Pentium (R) III processor, using ATLAS' `DGEMM` kernel.

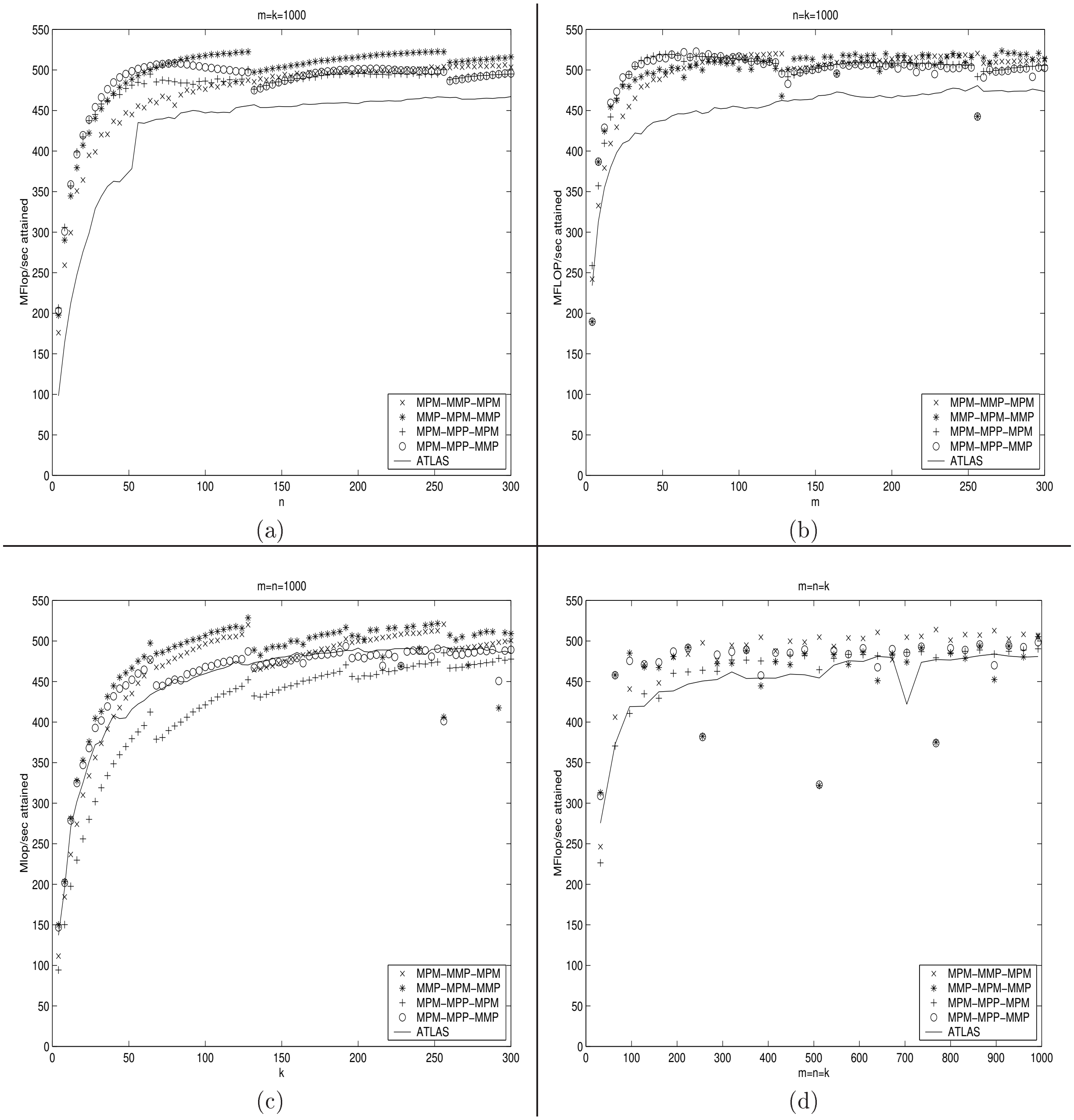


Figure 4.9: Performance of the different members of the family on an Intel Pentium (R) III processor, using the ITXGEMM `DGEMM` kernel.

4.9 Impact on algorithm design

While most of the above discussion is relatively machine independent, many of the decisions regarding block sizes that were used in the actual implementations for the Intel Pentium (R) III processor are specific for that processor. Furthermore, the block sizes are also specific to the ITXGEMM implementation of the matrix-matrix multiplication.

The general lesson to be learned, independent of the details of the implementation or processor, is that for every level layer of the memory hierarchy, there is a block size that allows optimal reuse of data in that memory layer. For example, for ITXGEMM on the Pentium (R) III processor, high performance is achieved when one of the dimensions (m, n, or k) equals 128. This allows 128×128 submatrices to reside in the L2 cache. The next magic number is $k = 64$, which allows 8×64 submatrices of A or 64×8 submatrices of B to reside in the L1 cache. In implementing other linear algebra algorithms we will observe the importance of these block sizes.

4.10 Related work

The current pursuit of highly optimized matrix kernels constructed by coding in a high-level programming language started with the implementation of the FORTRAN implementation of Basic Linear Algebra Subprograms (BLAS) [11] for the IBM POWER2 (TM) [1]. Subsequently, the PHiPAC project [6] demonstrated that high-performance matrix multiplication kernels can be written in C and that code generators could be used to automatically generate many different blockings, allowing automatic tuning. Next, the ATLAS project [36] extended these ideas by reducing the kernel that is called once matrices are massaged to be in the L1 cache into one specific case: $C = A^T B + \beta C$ for small matrices A, B, and C and by reducing the space searched for optimal blockings. Furthermore it marketed the methodology allowing it to gain wide-spread acceptance and igniting the current trend in the linear algebra community towards automatically tuned libraries. Finally, there has been a considerable recent interest in recursive algorithms and recursive data structures. The idea here is that by recursively partitioning the operands, blocks that fit in the different levels of the caches will automatically be encountered [22]. By storing matrices recursively, blocks that are encountered during the execution of the recursive algorithms will be in contiguous memory [20, 23].

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

$$D = LB \tag{5.1}$$

keeping in mind that D will overwrite B.

5.1 Algorithms that start by splitting L

Partition

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

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

$$D \rightarrow \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

where D_T and B_T have k rows.

Now, (5.1) becomes

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

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c} L_{TL}B_T \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_T & = & L_{TL}B_T \\ D_B & = & L_{BL}B_T + L_{BR}B_B \end{array}$$

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.

D contains	Comments	Viable?
$\left(\frac{\star}{\star}\right)$	This condition indicates no progress has been made.	NO
$\left(\frac{L_{TL}B_T}{\star}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\frac{\star}{L_{BL}B_T}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\frac{L_{TL}B_T}{L_{BL}B_T}\right)$	Since B_B is to be overwritten by D_B, this condition is not feasible since B_B is still needed for the computation $L_{BR}B_B$.	NO
$\left(\frac{\star}{L_{BR}B_B}\right)$		YES
$\left(\frac{L_{TL}B_T}{L_{BR}B_B}\right)$	Since B_T is to be overwritten by D_T, this condition is not feasible since B_T is still needed for the computation $L_{BL}B_T$.	NO
$\left(\frac{\star}{L_{BL}B_T + L_{BR}B_B}\right)$		YES
$\left(\frac{L_{TL}B_T}{L_{BL}B_T + L_{BR}B_B}\right)$	This condition indicates that the computation has completed.	NO

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

$$D \quad \text{contains} \quad \left(\frac{\star}{L_{BL}B_T + L_{BR}B_B}\right) \tag{5.2}$$

Notice that this indicates that L_{BL} and L_{BR} have been used to update D, which will 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

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right) \tag{5.3}$$

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

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

where b_1^T is a row and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) & L_{BR} = L_{22} \end{array}\right) \tag{5.5}$$

Considering (5.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) = \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c|c} L_{20} & l_{21} \end{array} \right) \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) + L_{22}B_2 \end{array}\right) = \left(\begin{array}{c} \star \\ \hline\hline l_{21}b_1^T + L_{22}B_2 \end{array}\right)$$

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

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \end{array}\right) \tag{5.6}$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \\ \hline\hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} l_{10}^T B_0 + \lambda_{11} b_1^T \\ \hline L_{20}B_0 + l_{21}b_1^T + L_{22}B_2 \end{array}\right) \end{array}\right)$$

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 $m^2 n$ 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 0×0 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:

$$\begin{array}{r} \bar{m}\{ \\ 1\{ \\ k\{ \end{array} \overbrace{\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline B_2 \end{array}}^{n} \qquad \begin{array}{c||c|c} \overbrace{L_{00}}^{\bar{m}} & \overbrace{0}^{1} & \overbrace{0}^{k} \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \begin{array}{l} \}\bar{m} = m-k-1 \\ \}1 \\ \}k \end{array}$$

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

$$\begin{array}{ll} b_1^T \leftarrow \lambda_{11} b_1^T & \qquad\qquad n \\ b_1^T \leftarrow l_{10}^T B_0 + b_1^T & \qquad\qquad 2(m-k-1)n \end{array}$$

since the first operation scales a row vector of length n and the vector-matrix multiply $l_{10}^T B_0 + b_1^T$ involves a matrix of size $(m-k-1) \times n$. Thus, the total cost of the triangular matrix multiplication is given by

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

This proves the theorem. $\square$

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline\hline D_2 \end{array}\right)$$

Algorithm 4 $B \leftarrow LB$
(unblocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where B_B **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0

repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$
where b_1^T **is a row**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$
where λ_{11} **is a scalar**

$b_1^T \leftarrow \lambda_{11} b_1^T$
$b_1^T \leftarrow l_{10}^T B_0 + b_1^T$

---

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

Algorithm 5 $B \leftarrow LB$
(blocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where B_B **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0
determine block size b
repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$
where B_1 **has** b **rows**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$
where L_{11} **is** $b \times b$

$B_1 \leftarrow L_{11} B_1$
$B_1 \leftarrow L_{10} B_0 + B_1$

---

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

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

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

Again, the double lines have meaning:

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

Considering (5.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} \star \\ \hline\hline L_{BL}B_T + L_{BR}B_B \end{array}\right) = \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right) \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) + L_{22}B_2 \end{array}\right) = \left(\begin{array}{c} \star \\ \hline\hline L_{20}B_0 + L_{21}B_1 + L_{22}B_2 \end{array}\right)$$

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

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} D_1 \\ \hline D_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \end{array}\right)$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \\ \hline\hline L_{20}B_0 + L_{21}B_1 + L_{22}B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} \star \\ \hline\hline \left(\begin{array}{c} L_{10}B_0 + L_{11}B_1 \\ \hline L_{20}B_0 + L_{21}B_1 + L_{22}B_2 \end{array}\right) \end{array}\right)$$

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 $m^2 n$ 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 0×0 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:

$$\begin{array}{r} \bar{m}\{ \\ b\{ \\ k\{ \end{array} \overbrace{\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline B_2 \end{array}}^{n} \qquad \begin{array}{c||c|c} \overbrace{L_{00}}^{\bar{m}} & \overbrace{0}^{b} & \overbrace{0}^{k} \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \begin{array}{l} \}\bar{m} = m - k - b \\ \}b \\ \}k \end{array}$$

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

$$\begin{array}{ll} B_1 \leftarrow L_{11}B_1 & b^2 n \\ B_1 \leftarrow L_{10}B_0 + B_1 & 2(m-k-b)bn \end{array}$$

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 $b \times (m-k-b)$ matrix times an $b \times n$ matrix. Notice that

$k = ib$ 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

$$\begin{aligned}
\sum_{i=0}^{\frac{m}{b}-1} \left[b^2 n + 2(m - ib - b) bn \right] &= \sum_{i=0}^{\frac{m}{b}-1} b^2 n + 2 \sum_{i=0}^{\frac{m}{b}-1} (m - ib - b)\, bn \\
&= \frac{m}{b} b^2 n + 2 \sum_{i=0}^{\frac{m}{b}-1} (ib)\, bn = mbn + 2 \frac{(\frac{m}{b} - 1)\frac{m}{b}}{2} b^2 n = mbn + (m - b) mn = m^2 n
\end{aligned}$$

This proves the theorem. $\square$

5.1.2 Lazy algorithm (relative to L)

Next, consider the condition that currently

$$D \quad \text{contains} \quad \left(\frac{\star}{\overline{L_{BR} B_B}} \right) \tag{5.8}$$

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

$$\left(\frac{\star}{\overline{L_{BR} B_B}} \right) = \left(\frac{\star}{\overline{L_{22} B_2}} \right)$$

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

$$\left(\frac{\star}{\overline{L_{BR} B_B}} \right) = \left(\frac{\star}{\overline{\left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right) \left(\frac{b_1^T}{B_2} \right)}} \right) = \left(\frac{\star}{\overline{\left(\frac{\lambda_{11} b_1^T}{l_{21} b_1^T + L_{22} B_2} \right)}} \right)$$

Thus the contents of D must be updated like

$$\left(\frac{\left(\frac{\star}{\star} \right)}{\overline{L_{22} B_2}} \right) \rightarrow \left(\frac{\star}{\overline{\left(\frac{\lambda_{11} b_1^T}{l_{21} b_1^T + L_{22} B_2} \right)}} \right)$$

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
$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$
where B_B **has** 0 **rows**
partition
$$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0

repartition
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$$
where b_1^T **is a row**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$
where λ_{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
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$$
enddo

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

partition
$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$
where B_B **has** 0 **rows**
partition
$$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0
determine block size b
repartition
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$$
where B_1 **has** b **rows**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$$
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
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$$
enddo

Figure 5.3: Unblocked (left) and blocked (right) lazy triangular matrix-matrix multiplication algorithms.

First, consider partitioning B by rows:

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

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

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

where L_{TL} is $k \times k$. Notice that we use a 2×2 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:

$$B = \left(\; B_L \;\|\; B_R \; \right)$$

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

$$D = \left(\; D_L \;\|\; D_R \; \right)$$

Now, (5.1) becomes

$$\left(\; D_L \;\|\; D_R \; \right) = L \left(\; B_L \;\|\; B_R \; \right)$$

Multiplying out the right-hand-side yields

$$\left(\; D_L \;\|\; D_R \; \right) = \left(\; LB_L \;\|\; LB_R \; \right)$$

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

D contains	Comments
$\left(\; \star \;\|\; \star \; \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\; LB_L \;\|\; \star \; \right)$	
$\left(\; LB_L \;\|\; LB_R \; \right)$	This indicates that the computation has finished. Clearly not an interesting intermediate condition.
$\left(\; \star \;\|\; LB_R \; \right)$	

5.2.1 Right-moving (lazy) algorithm

Let us consider the condition that currently

$$D \quad \text{contains} \quad \left(\; LB_L \;\|\; \star \; \right) \tag{5.9}$$

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

$$\left(\; D_L \;\|\; D_R \; \right) \rightarrow \left(\; D_0 \;\|\; d_1 \;|\; D_2 \; \right)$$

where d_1 is a column. Similarly, we repartition

$$\left(\; B_L \;\|\; B_R \; \right) \rightarrow \left(\; B_0 \;\|\; b_1 \;|\; B_2 \; \right)$$

where b_1 is a column. Notice that

$$\left(D_L = D_0 \,\|\, D_R = \left(d_1 \mid D_2 \right) \right) \quad \text{and} \quad \left(B_L = B_0 \,\|\, B_R = \left(b_1 \mid B_2 \right) \right) \tag{5.10}$$

Thus D currently contains

$$\left(LB_0 \,\|\, \left(\star \mid \star \right) \right)$$

After moving the double lines ahead,

$$\left(D_L = \left(D_0 \mid d_1 \right) \,\|\, D_R = D_2 \right) \quad \text{and} \quad \left(B_L = \left(B_0 \mid b_1 \right) \,\|\, B_R = B_2 \right)$$

and thus D must hold

$$\left(L\left(B_0 \mid b_1 \right) \,\|\, \star \right) = \left(\left(LB_0 \mid Lb_1 \right) \,\|\, \star \right)$$

Thus, the contents of D must be update like

$$\left(LB_0 \,\|\, \left(\star \mid \star \right) \right) \rightarrow \left(\left(LB_0 \mid Lb_1 \right) \,\|\, \star \right)$$

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

Algorithm 8 $B \leftarrow LB$ *(unblocked right-moving (lazy) w.r.t. B)*

partition
  $B \rightarrow \left(B_L \,\|\, B_R \right)$
    where B_L **has** 0 **columns**
do until B_R **has** 0 **columns**
  repartition
    $\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \mid B_2 \right)$
      where b_1 **is a column**

  $b_1 \leftarrow Lb_1$

  ---
  continue with
    $\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \mid b_1 \,\|\, B_2 \right)$
enddo

Algorithm 9 $B \leftarrow LB$ *(blocked right-moving (lazy) w.r.t. B)*

partition
  $B \rightarrow \left(B_L \,\|\, B_R \right)$
    where B_L **has** 0 **columns**
do until B_R **has** 0 **columns**
  determine block size b
  repartition
    $\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \mid B_2 \right)$
      where B_1 **has** b **columns**

  $B_1 \leftarrow LB_1$

  ---
  continue with
    $\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \mid B_1 \,\|\, B_2 \right)$
enddo

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

`http://www.cs.utexas.edu/users/flame/materials/trmm_lln/`

```
#include "FLAME.h"

void Trmm_left_lower_notrans_unb( int variant, FLA_Obj A, FLA_Obj B )
{
  FLA_Obj     ATL, ATR,     A00,  a01,     A02,     BT,      B0,
              ABL, ABR,     a10t, alpha11, a12t,    BB,      b1t,
                            A20,  a21,     A22,              B2;
  int         b;

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

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

  while ( 0 != FLA_Obj_length( ATR ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00,  &a01,     /**/ &A02,
                                /**/              &a10t, &alpha11, /**/ &a12t,
                        /* ************* */    /* ************************** */
                           ABL, /**/ ABR,         &A20,  &a21,     /**/ &A22,
                           1, 1, /* alpha11 from */ FLA_TL );

    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                                                 &b1t,
                          /**/                   /**/
                           BB,                   &B2,
                           1, /* length b1t from */ FLA_TOP );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Ger( ONE, a21, b1t, B2 );

    FLA_Scal( alpha11, b1t );

    if ( variant == FLA_VARIANT_ROWLAZY )
      FLA_Gemv( FLA_TRANSPOSE, ONE, B0, a10t, ONE, b1t );

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

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00,  /**/ a01,     A02,
                            /* ************** */   /* ************************ */
                                    /**/               a10t, /**/ alpha11, a12t,
                              &ABL, /**/ &ABR,         A20,  /**/ a21,     A22,
                              /* alpha11 added to */ FLA_BR );

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                             /***/                    /**/
                                                       b1t,
                              &BB,                     B2,
                              /* b1t  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, ABR,     A10, A11, A12,    BB,           B1,
                            A20, A21, A22,                  B2;
  int         b;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,   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_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &A01, /**/ &A02,
                                /**/              &A10, &A11, /**/ &A12,
                         /* ************* */    /* ********************* */
                           ABL, /**/ ABR,         &A20, &A21, /**/ &A22,
                           b, b, /* A11 from */ FLA_TL );

    FLA_Repart_2x1_to_3x1( BT,                  &B0,
                                                &B1,
                          /**/                  /**/
                           BB,                  &B2,
                           b, /* length B1 from */ FLA_TOP );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A21, B1, ONE, B2 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      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_ROWLAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B0, ONE, B1 );

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

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

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                             /***/                    /**/
                                                       B1,
                              &BB,                     B2,
                              /* 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(  BL, /**/      BR,
                           &B0, /**/  &b1, &B2,
                           1, /* width b1 from */ FLA_RIGHT );

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

    FLA_Trmv( 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(  BL, /**/      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_Trmm` implemented 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.

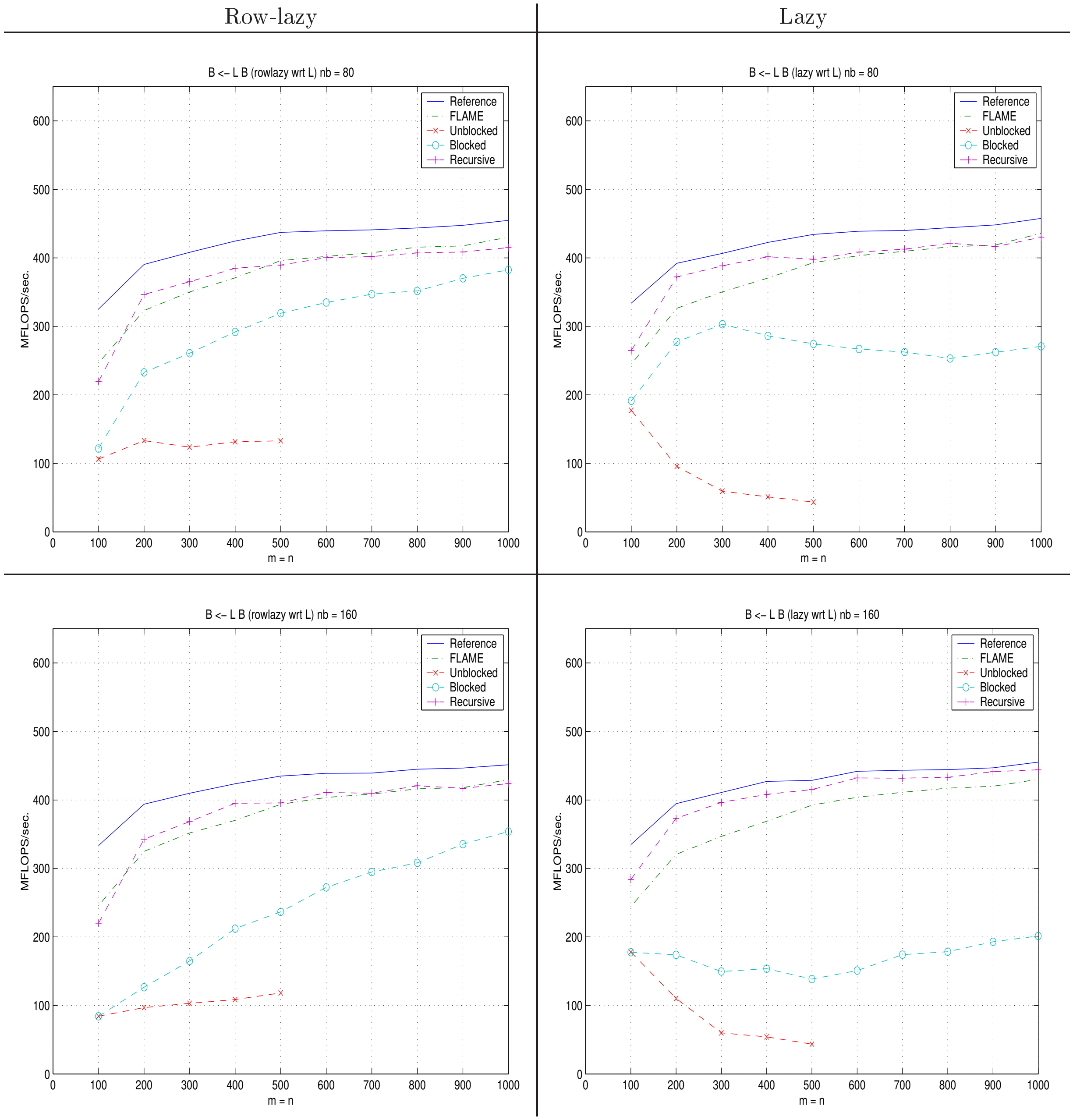


Figure 5.9: Performance of the row lazy (left) and lazy (right) (w.r.t. L) lower triangular matrix-matrix multiplication algorithms for a block size of 80 (top) and 160 (bottom). For these experiments, the ATLAS matrix-matrix multiplication kernel was used.

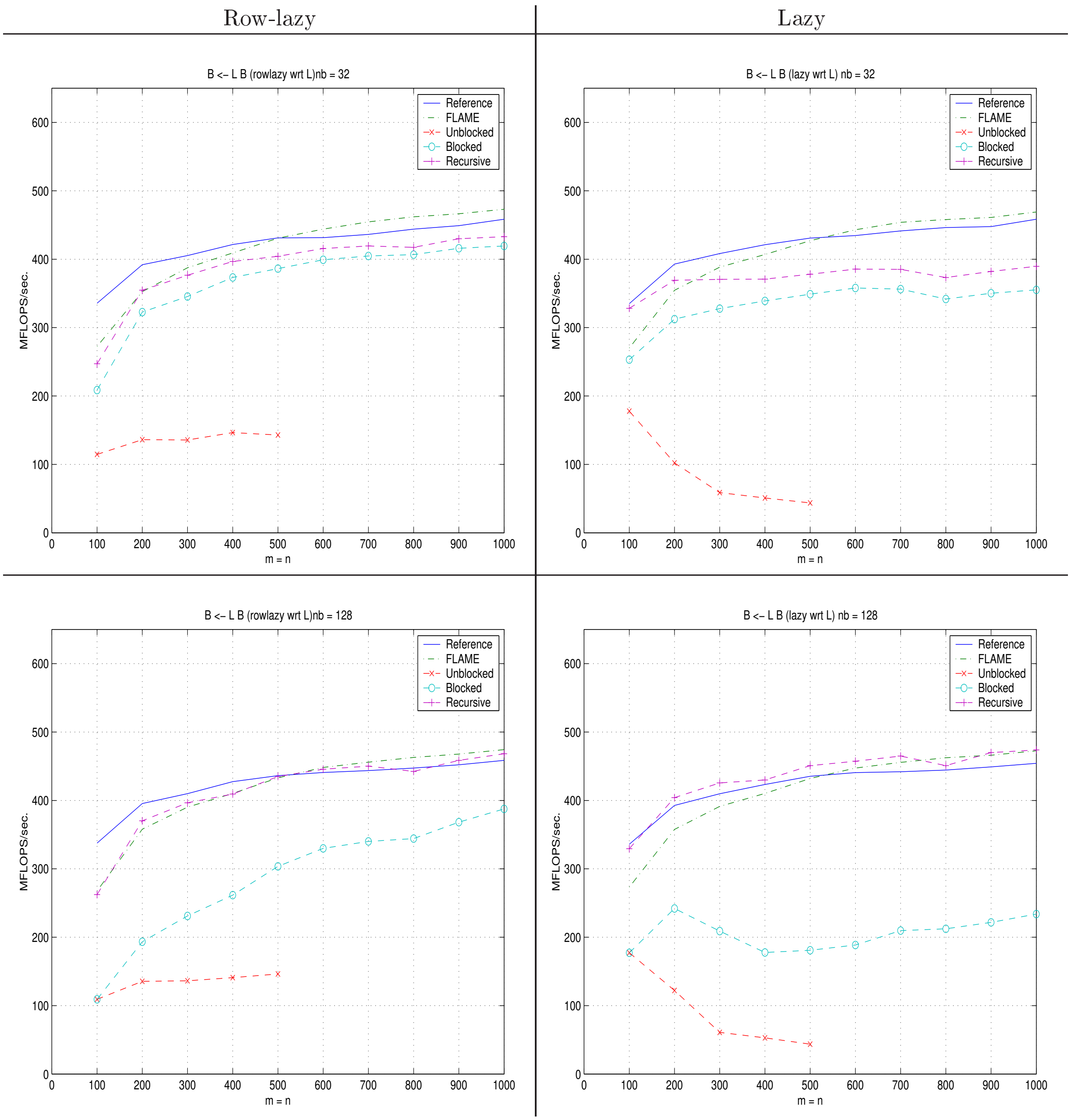


Figure 5.10: Performance of the row lazy (left) and lazy (right) (w.r.t. L) lower triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

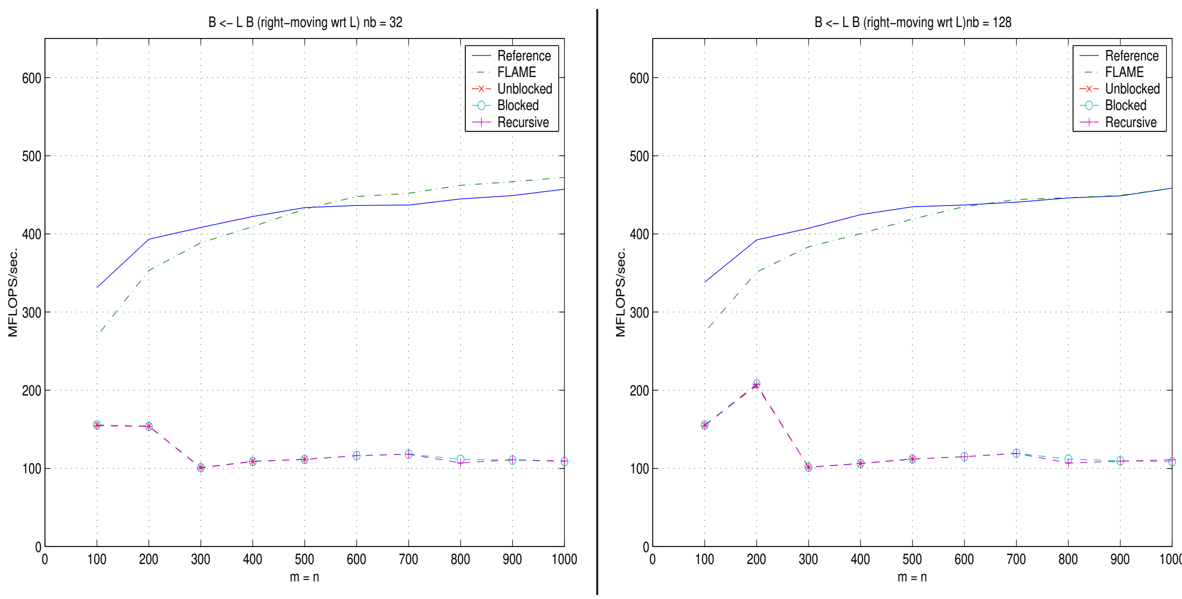


Figure 5.11: Performance of the right-moving (w.r.t. B) triangular matrix-matrix multiplication algorithms for a block size of 32 (left) and 128 (right), using ITXGEMM.

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

$$D = BA + C \tag{6.1}$$

keeping in mind that D will overwrite C. Partition

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

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

$$C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array} \right), \quad \text{and} \quad D \rightarrow \left(\begin{array}{c||c} D_L & D_R \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

where C_L and D_L and B_L have k columns.

Now, (6.1) becomes

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) = \left(\begin{array}{c||c} B_L & B_R \end{array} \right) \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) + \left(\begin{array}{c||c} C_L & C_R \end{array} \right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c|c} D_L & D_R \end{array} \right) = \left(\begin{array}{c|c} B_L A_{TL} + B_R A_{TR}^T + C_L & B_L A_{TR} + B_R A_{BR} + C_R \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_L & = & B_L A_{TL} + B_R A_{TR}^T + C_L \\ D_R & = & B_L A_{TR} + B_R A_{BR} + C_R \end{array}$$

which must hold. Possible contents of D at this intermediate step are given in Fig. 6.1.

D contains	Comments
$\left(C_L \parallel C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_L A_{TL} + C_L \parallel C_R \right)$	
$\left(B_R A_{TR}^T + C_L \parallel C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_L A_{TL} + B_R A_{TR}^T + C_L \parallel C_R \right)$	
$\left(C_L \parallel B_L A_{TR} + C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_L A_{TL} + C_L \parallel B_L A_{TR} + C_R \right)$	
$\left(B_R A_{TR}^T + C_L \parallel B_L A_{TR} + C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_L A_{TL} + B_R A_{TR}^T + C_L \parallel B_L A_{TR} + C_R \right)$	
$\left(C_L \parallel B_R A_{BR} + C_R \right)$	
$\left(B_L A_{TL} + C_L \parallel B_R A_{BR} + C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_R A_{TR}^T + C_L \parallel B_R A_{BR} + C_R \right)$	
$\left(B_L A_{TL} + B_R A_{TR}^T + C_L \parallel B_R A_{BR} + C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(C_L \parallel B_L A_{TR} + B_R A_{BR} + C_R \right)$	
$\left(B_L A_{TL} + C_L \parallel B_L A_{TR} + B_R A_{BR} + C_R \right)$	Not viable based on 8a in Section 2.2.
$\left(B_R A_{TR}^T + C_L \parallel B_L A_{TR} + B_R A_{BR} + C_R \right)$	
$\left(B_L A_{TL} + B_R A_{TR}^T + C_L \parallel B_L A_{TR} + B_R A_{BR} + C_R \right)$	Not viable based on 8a in Section 2.2.

Figure 6.1: Some possible contents of D at an intermediate step.

Let us take this opportunity to illustrate how 8a in Section 2.2 can be used to reject conditions. Consider the condition that D contains $\left(B_R A_{TR}^T + C_L \parallel C_R \right)$. In the loop for computing the symmetric matrix-matrix multiplication $C \leftarrow BA + C$, A and C are partitioned like

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{TR}^T & A_{BR} \end{array} \right) \quad \text{and} \quad C = \left(\begin{array}{c|c} C_L & C_R \end{array} \right)$$

where either

1. The algorithm is a down-right moving algorithm (relative to A) and
   - initially A_{TL} is 0×0 and C_L has no columns, and
   - upon completion A_{BR} is 0×0 and C_R has no columns,

   or

2. The algorithm is an up-left moving algorithm (relative to A) and
   - initially A_{BR} is 0×0 and C_R has no columns, and
   - upon completion A_{TL} is 0×0 and C_L has no columns.

We now show that neither of these cases work. In particular, the condition must hold before the loop commences and the condition must hold upon completion. Thus, corresponding to the two cases above, we require that either

1. For the down-right moving algorithm

   - D initially contains $\left(B_R A_{TR}^T + C_L \,\|\, C_R \right) = \left(\;\|\, C_R \right) = C$ since all matrices on the left are void (contain no rows and/or columns). This is good, since initially D should contain C if it is to overwrite matrix C.
   - D upon completion contains $\left(B_R A_{TR}^T + C_L \,\|\, C_R \right) = \left(C_L \,\|\; \right) = C$ since B_R, A_{TR}, and C_R are all void (contain no rows and/or columns). This is bad, since this indicates that upon completion of the loop D will contain C rather than the desired result $BA + C$.

2. For the up-left moving algorithm

   - D initially contains $\left(B_R A_{TR}^T + C_L \,\|\, C_R \right) = \left(C_L \,\|\; \right) = C$ since all matrices on the left are void (contain no rows and/or columns). This is good, since initially D should contain C if it is to overwrite matrix C.
   - D upon completion contains $\left(B_R A_{TR}^T + C_L \,\|\, C_R \right) = \left(C_R \,\|\; \right) = C$ since B_R, A_{TR}, and C_R are all void (contain no rows and/or columns). This is bad, since this indicates that upon completion of the loop D will contain C rather than the desired result $BA + C$.

Thus, we conclude that neither the down-right moving nor the up-left moving algorithm will compute the correct result if the loop maintains the indicated invariant.

6.1.1 Column-lazy algorithm (relative to A)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(B_L A_{TL} + B_R A_{TR}^T + C_L \,\|\, C_R \right) \tag{6.2}$$

Notice that this indicates that A_{TL} and A_{TR}^T (or A_{BL}) have been used to update D, which we will call a column-lazy algorithm relative to operand A.

In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved to the right. Thus, this algorithm naturally moves through matrices D and B from the left to the right.

Unblocked algorithm

Repartition

$$\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,\|\, d_1 \mid D_2 \right)$$

and

$$\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, c_1 \mid C_2 \right)$$

where d_1 and c_1 are columns or vectors. Similarly, we repartition

$$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \mid B_2 \right) \quad \text{and} \quad \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{01}^T & \alpha_{11} & a_{12}^T \\ \hline A_{02}^T & a_{12} & A_{22} \end{array} \right)$$

where b_1 is a column and α_{11} is a scalar. Notice that the double lines have meaning:

$$\left(D_L = D_0 \,\|\, D_R = \left(d_1 \mid D_2 \right) \right), \left(B_L = B_0 \,\|\, B_R = \left(b_1 \mid B_2 \right) \right), \left(C_L = C_0 \,\|\, C_R = \left(c_1 \mid C_2 \right) \right), \tag{6.3}$$

and

$$\left(\begin{array}{c||c} A_{TL} = A_{00} & A_{TR} = \left(a_{01} \mid A_{02} \right) \\ \hline\hline A_{TR}^T = \left(\begin{array}{c} a_{01}^T \\ \hline A_{02}^T \end{array} \right) & A_{BR} = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{12} & A_{22} \end{array} \right) \end{array} \right) \tag{6.4}$$

Considering (6.1) and these repartitionings, D currently contains

$$\begin{aligned}\left(B_L A_{TL} + B_R A_{TR}^T + C_L \;\middle\|\; C_R \right) &= \left(B_0 A_{00} + \left(b_1 \mid B_2 \right) \left(\frac{a_{01}^T}{A_{02}^T} \right) + C_0 \;\middle\|\; \left(c_1 \mid C_2 \right) \right) \\ &= \left(B_0 A_{00} + b_1 a_{01}^T + B_2 A_{02}^T + C_0 \;\|\; \left(c_1 \mid C_2 \right) \right)\end{aligned}$$

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

$$\left(D_L = \left(D_0 \mid d_1 \right) \| D_R = D_2 \right), \left(B_L = \left(B_0 \mid b_1 \right) \| B_R = B_2 \right), \left(C_L = \left(C_0 \mid c_1 \right) \| C_R = C_2 \right),$$

and

$$\left(\begin{array}{c|c} A_{TL} = \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{01}^T & \alpha_{11} \end{array} \right) & A_{TR} = \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \end{array} \right) \\ \hline\hline A_{TR}^T = \left(A_{02}^T \mid a_{12} \right) & A_{BR} = A_{22} \end{array} \right)$$

Thus the contents of D must be updated to contain

$$\begin{aligned}&\left(\left(B_0 \mid b_1 \right) \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{01}^T & \alpha_{11} \end{array} \right) + B_2 \left(\frac{A_{02}}{a_{12}^T} \right)^T + \left(C_0 \mid c_1 \right) \;\middle\|\; C_2 \right) \\ &= \left(\left(B_0 A_{00} + b_1 a_{01}^T \mid B_0 a_{01} + b_1 \alpha_{11} \right) + \left(B_2 A_{02}^T \mid B_2 a_{12} \right) + \left(C_0 \mid c_1 \right) \| C_2 \right) \\ &= \left(\left(B_0 A_{00} + b_1 a_{01}^T + B_2 A_{02}^T + C_0 \mid B_0 a_{01} + b_1 \alpha_{11} + B_2 a_{12} + c_1 \right) \| C_2 \right)\end{aligned}$$

We conclude that the contents of D must change like

$$\begin{aligned}&\left(B_0 A_{00} + b_1 a_{01}^T + B_2 A_{02}^T + C_0 \;\|\; \left(c_1 \mid C_2 \right) \right) \rightarrow \\ &\quad \left(\left(B_0 A_{00} + b_1 a_{01}^T + B_2 A_{02}^T + C_0 \mid B_0 a_{01} + b_1 \alpha_{11} + B_2 a_{12} + c_1 \right) \| C_2 \right)\end{aligned}$$

An algorithm that maintains the condition in (6.1) is given in Fig. 6.2 (left). In that figure, we also give the corresponding blocked algorithm. Notice that the algorithm overwrites matrix C with the result $BA + C$.

6.1.2 Lazy algorithm (relative to A)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(B_L A_{TL} + C_L \;\|\; \star \right) \tag{6.5}$$

Notice that this indicates that A_{TL} has been used to update D, which we'll call a lazy algorithm relative to operand A since it operates upon only one of the quadrants of A.

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

Blocked algorithm

Repartition

$$\left(D_L \| D_R \right) \rightarrow \left(D_0 \| D_1 \mid D_2 \right) \quad \text{and} \quad \left(C_L \| C_R \right) \rightarrow \left(C_0 \| C_1 \mid C_2 \right)$$

where D_1 and C_1 are b columns or vectors. Similarly, we repartition

$$\left(B_L \| B_R \right) \rightarrow \left(B_0 \| B_1 \mid B_2 \right) \quad \text{and} \quad \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{01}^T & A_{11} & A_{12} \\ \hline A_{02}^T & A_{12}^T & A_{22} \end{array} \right)$$

where B_1 is a block of b columns and A_{11} is a $b \times b$ submatrix. Again the double lines have a similar meaning to those in (6.3) and (6.4)

Algorithm 10 $C \leftarrow BA + C$
(unblocked col.-lazy w.r.t. A)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right), C \rightarrow \left(C_L \,\|\, C_R \right)$
where B_L **and** C_L **have** 0 **columns**
partition
$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right)$
where A_{TL} **is** 0×0
do until A_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
$\left(C_T \,\|\, C_B \right) \rightarrow \left(C_0 \,\|\, c_1 \,|\, C_2 \right)$
where b_1 **and** c_1 **are columns**
repartition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{01}^T & \alpha_{11} & a_{12}^T \\ \hline A_{02}^T & a_{12} & A_{22} \end{array} \right)$$
where α_{11} **is a scalar**

$c_1 \leftarrow \alpha_{11} b_1 + c_1$
$c_1 \leftarrow B_0 a_{01} + c_1$
$c_1 \leftarrow B_2 a_{12} + c_1$

---

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, c_1 \,\|\, C_2 \right)$
continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{01}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{02}^T & a_{12} & A_{22} \end{array} \right)$$
enddo

Algorithm 11 $C \leftarrow BA + C$
(blocked col.-lazy w.r.t. A)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right), C \rightarrow \left(C_L \,\|\, C_R \right)$
where B_L **and** C_L **have** 0 **columns**
partition
$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right)$
where A_{TL} **is** 0×0
do until A_{BR} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,|\, C_2 \right)$
where B_1 **and** C_1 **have** b **columns**
repartition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{01}^T & A_{11} & A_{12} \\ \hline A_{02}^T & A_{12}^T & A_{22} \end{array} \right)$$
where A_{11} **is** $b \times b$

$C_1 \leftarrow C_1 + B_0 A_{01}$
$C_1 \leftarrow C_1 + B_1 A_{11}$
$C_1 \leftarrow C_1 + B_2 A_{12}^T$

---

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,\|\, C_2 \right)$
continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{01}^T & A_{11} & A_{12} \\ \hline\hline A_{02}^T & A_{12}^T & A_{22} \end{array} \right)$$
enddo

Figure 6.2: Unblocked (left) and blocked (right) column-lazy upper triangular stored matrix-matrix multiplication algorithms.

```
#include "FLAME.h"

void Symm_right_upper_unb( int variant, FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00,  a01,     A02,  BL,  B0,  CL,  C0,
              ABL, ABR,     a10t, alpha11, a12t, BR,  b1,  CR,  c1,
                            A20,  a21,     A22,       B2,       C2;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                      &ABL, /**/ &ABR,
                  0, 0,     /* submatrix */ FLA_TL );

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

  FLA_Part_1x2( C,  &CL, /**/ &CR,     0, /*  width submatrix */ FLA_LEFT );

  while ( FLA_Obj_length( ABR )>0 ){
     FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &a01,     &A02,
                         /* ************* */    /* ************************ */
                                 /**/              &a10t, /**/ &alpha11, &a12t,
                            ABL, /**/ ABR,         &A20,  /**/ &a21,     &A22,
                            1, 1, /* alpha11 from */ FLA_BR );

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  /**/ &b1,  &B2,
                            1, /* width c1 from */ FLA_RIGHT );

    FLA_Repart_1x2_to_1x3( CL,  /**/ CR,          &C0,  /**/ &c1,  &C2,
                            1, /* width c1 from */ FLA_RIGHT );

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

    FLA_Gemv( FLA_NO_TRANSPOSE, ONE, B0, a01, ONE, c1 );

    FLA_Axpy( alpha11, b1, c1);

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Ger( ONE, b1, a01, C0 );
    else
      FLA_Gemv( FLA_NO_TRANSPOSE, ONE, B2, a12t, ONE, c1 );

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

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00,  a01,     /**/ A02,
                                         /**/          a10t, alpha11, /**/ a12t,
                             /* ************** */    /* ***************** */
                               &ABL, /**/ &ABR,         A20,  a21,     /**/ A22,
                               /* alpha11 added to */ FLA_TL );

    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,           B0,  b1,  /**/ B2,
                               /* b1  added to */ FLA_LEFT );

    FLA_Cont_with_1x3_to_1x2( &CL, /**/ &CR,           C0,  b1,  /**/ C2,
                               /* b1  added to */ FLA_LEFT );
  }
}
```

Figure 6.3: Unblocked lazy and column-lazy symmetric matrix-matrix multiplication.

```
#include "FLAME.h"

int Symm_right_upper_blk( int variant, int rec, FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,  BL,  B0,  CL,  C0,
              ABL, ABR,     A10, A11, A12,  BR,  B1,  CR,  C1,
                            A20, A21, A22,       B2,       C2;
  int         b;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
                 0, 0,     /* submatrix */ FLA_TL );

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

  FLA_Part_1x2( C,  &CL, /**/ &CR,     0, /*  width submatrix */ FLA_LEFT );

  while ( b = min( FLA_Obj_length( ABR ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                         /* ************* */    /* ******************** */
                                 /**/             &A10, /**/ &A11, &A12,
                            ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
                            b, b, /* A11 from */ FLA_BR );

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  /**/ &B1,  &B2,
                            b, /* width B1 from */ FLA_RIGHT );

    FLA_Repart_1x2_to_1x3( CL,  /**/ CR,          &C0,  /**/ &C1,  &C2,
                            b, /* width C1 from */ FLA_RIGHT );

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

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,    ONE, B0, A01, ONE, C1 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      Symm_right_upper_blk( variant, rec, A11, B1, C1, nb_alg/2 );
    else
      Symm_right_upper_unb( variant, A11, B1, C1 );

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,    ONE, B1, A01, ONE, C0 );
    else
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,    ONE, B2, A12, ONE, C1 );

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

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

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

    FLA_Cont_with_1x3_to_1x2( &CL, /**/ &CR,           C0,  C1,  /**/ C2,
                                /* C1  added to */ FLA_LEFT );
  }
}
```

Figure 6.4: Blocked lazy and column-lazy symmetric matrix-matrix multiplication.

Considering (6.1) and these repartitionings, D currently contains

$$\left(\begin{array}{c\|c} B_L A_{TL} + C_L & C_R \end{array} \right) = \left(\begin{array}{c\|c} B_0 A_{00} + C_0 & \left(\begin{array}{c|c} C_1 & C_2 \end{array} \right) \end{array} \right)$$

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

$$\left(\begin{array}{c\|c} D_L = \left(\begin{array}{c|c} D_0 & D_1 \end{array} \right) & D_R = D_2 \end{array} \right), \left(\begin{array}{c\|c} B_L = \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) & B_R = B_2 \end{array} \right), \left(\begin{array}{c\|c} C_L = \left(\begin{array}{c|c} C_0 & C_1 \end{array} \right) & C_R = C_2 \end{array} \right),$$

and

$$\left(\begin{array}{c\|c} A_{TL} = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{01}^T & A_{11} \end{array} \right) & A_{TR} = \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array} \right) \\ \hline\hline A_{TR}^T = \left(\begin{array}{c|c} A_{02}^T & A_{12}^T \end{array} \right) & A_{BR} = A_{22} \end{array} \right)$$

Thus the contents of D be updated to contain

$$\begin{aligned} &\left(\begin{array}{c\|c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{01}^T & A_{11} \end{array} \right) + \left(\begin{array}{c|c} C_0 & C_1 \end{array} \right) & C_2 \end{array} \right) \\ &\quad = \left(\begin{array}{c\|c} \left(\begin{array}{c|c} B_0 A_{00} + B_1 A_{01}^T & B_0 A_{01} + B_1 A_{11} \end{array} \right) + \left(\begin{array}{c|c} C_0 & C_1 \end{array} \right) & C_2 \end{array} \right) \\ &\quad = \left(\begin{array}{c\|c} \left(\begin{array}{c|c} B_0 A_{00} + B_1 A_{01}^T + C_0 & B_0 A_{01} + B_1 A_{11} + C_1 \end{array} \right) & C_2 \end{array} \right) \end{aligned}$$

Thus we conclude that an algorithm that maintains the condition in (6.5 is given in Fig. 6.2 (right). Notice that the algorithm overwrites matrix C with the result $BA + C$.

6.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/symm_ru/`

6.3 Performance

Algorithm 12 $C \leftarrow BA + C$
(unblocked lazy w.r.t. A)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right), C \rightarrow \left(C_L \,\|\, C_R \right)$
where B_L **and** C_L **have** 0 **columns**
partition
$$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right)$$
where A_{TL} **is** 0×0
do until A_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
$\left(C_T \,\|\, C_B \right) \rightarrow \left(C_0 \,\|\, c_1 \,|\, C_2 \right)$
where b_1 **and** c_1 **are columns**
repartition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{01}^T & \alpha_{11} & a_{12}^T \\ \hline A_{02}^T & a_{12} & A_{22} \end{array} \right)$$
where α_{11} **is a scalar**

$C_0 \leftarrow b_1 a_{01}^T + C_0$
$c_1 \leftarrow B_0 a_{01} + \alpha_{11} b_1 + c_1$

---

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1^T \,\|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, c_1^T \,\|\, C_2 \right)$
continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{01}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{02}^T & a_{12} & A_{22} \end{array} \right)$$
enddo

Algorithm 13 $C \leftarrow BA + C$
(blocked lazy w.r.t. A)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right), C \rightarrow \left(C_L \,\|\, C_R \right)$
where B_L **and** C_L **have** 0 **columns**
partition
$$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right)$$
where A_{TL} **is** 0×0
do until A_{BR} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,|\, C_2 \right)$
where B_1 **and** C_1 **have** b **columns**
repartition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{01}^T & A_{11} & A_{12} \\ \hline A_{02}^T & A_{12}^T & A_{22} \end{array} \right)$$
where A_{11} **is** $b \times b$

$C_0 \leftarrow B_1 A_{01}^T + C_0$
$C_1 \leftarrow B_0 A_{01} + B_1 A_{11} + C_1$

---

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
$\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,\|\, C_2 \right)$
continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{TR}^T & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{01}^T & A_{11} & A_{12} \\ \hline\hline A_{02}^T & A_{12}^T & A_{22} \end{array} \right)$$
enddo

Figure 6.5: Unblocked (left) and blocked (right) lazy upper triangular stored matrix-matrix multiplication algorithms.

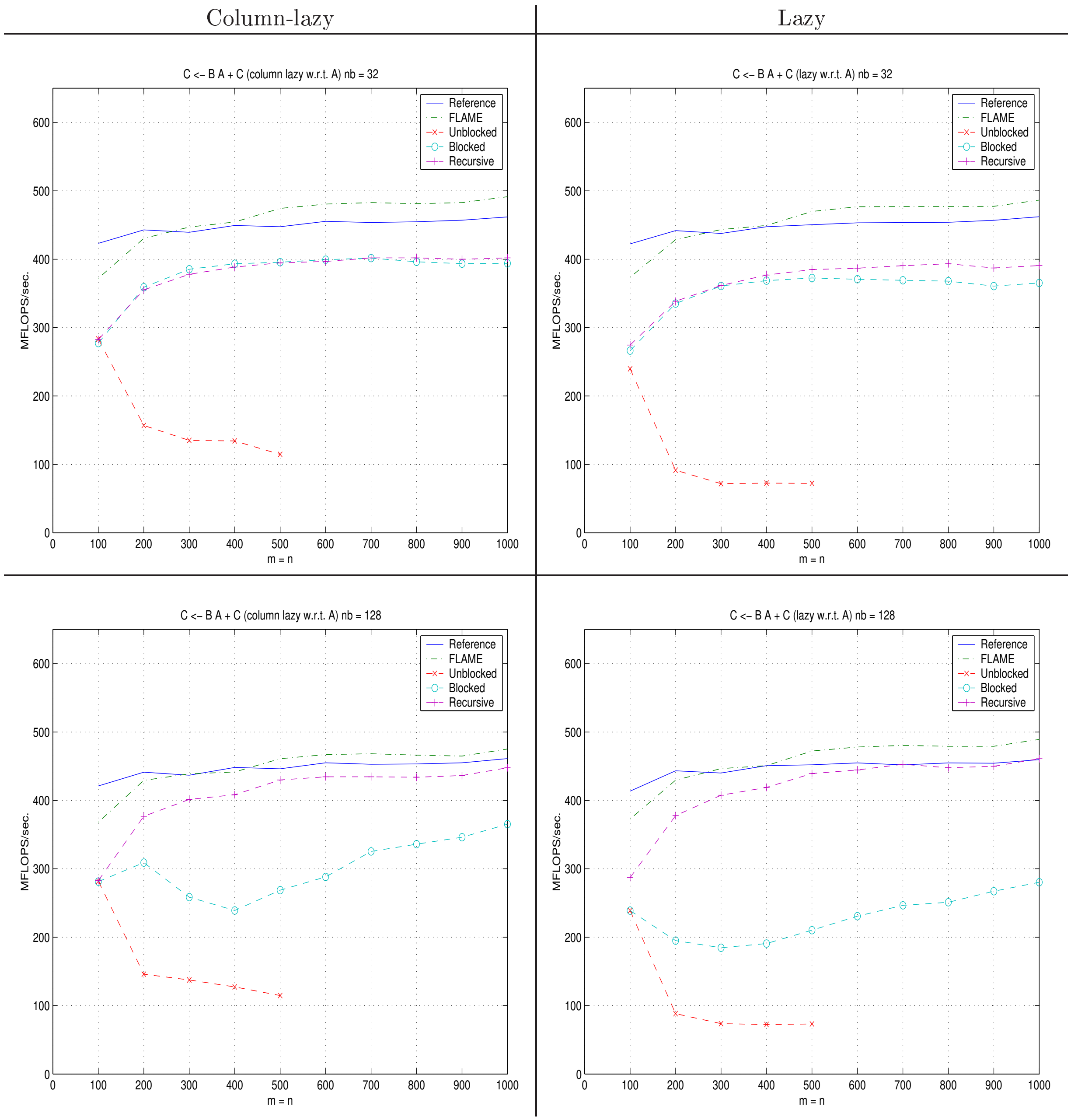


Figure 6.6: Performance of the column lazy (left) and lazy (right) (w.r.t. A) symmetric matrix-multiplication algorithms for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Chapter 7

Symmetric Matrix Multiplication $C \leftarrow AB + C$ A symmetric, stored in lower triangle

by

Eric G. Geordi

Stephen Y. Hui

In this chapter, we discuss the implementation of a symmetric matrix-matrix multiplication:

$$C \leftarrow AB + C$$

where A is a symmetric $m \times m$ matrix and B, C and D are $m \times n$ matrices. Since A is symmetric, only the lower triangle is stored.

Let us instead consider

$$D = AB + C \tag{7.1}$$

keeping in mind that D will overwrite C.

7.1 Algorithms that start by splitting A

Partition

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

where A_{TL} is $k \times k$. Also, partition

$$D \rightarrow \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right), \; B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right), \text{ and } C \rightarrow \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array} \right)$$

where D_T, B_T, and C_T have k rows.

Now, (7.1) becomes

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right) + \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array} \right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array} \right)$$

This generates the equalities

$$\begin{array}{rcl} D_T & = & A_{TL}B_T + A_{BL}^T B_B + C_T \\ D_B & = & A_{BL}B_T + A_{BR}B_B + C_B \end{array}$$

which relate the contents of the different submatrices.

Possible contents of D at this intermediate step are given in Fig. 7.1. In this chapter, we will only consider

D contains	Comments	Viable?
$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$	No progress has been made.	NO
$\left(\begin{array}{c} A_{TL}B_T + C_T \\ \hline\hline C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{BL}^T B_B + C_T \\ \hline\hline C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + C_T \\ \hline\hline C_B \end{array}\right)$		YES
$\left(\begin{array}{c} C_T \\ \hline\hline A_{BL}B_T + C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} A_{TL}B_T + C_T \\ \hline\hline A_{BL}B_T + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{BL}^T B_B + C_T \\ \hline\hline A_{BL}B_T + C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + C_T \\ \hline\hline A_{BL}B_T + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} C_T \\ \hline\hline A_{BR}B_B + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{TL}B_T + C_T \\ \hline\hline A_{BR}B_B + C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} A_{BL}^T B_B + C_T \\ \hline\hline A_{BR}B_B + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + C_T \\ \hline\hline A_{BR}B_B + C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{TL}B_T + C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right)$	Not viable based on 8a in Section 2.2.	NO
$\left(\begin{array}{c} A_{BL}^T B_B + C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right)$		YES
$\left(\begin{array}{c} A_{TL}B_T + A_{BR}^T B_B + C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right)$	The computation has completed.	NO

Figure 7.1: Possible contents of D at an intermediate step.

two viable options.

7.1.1 Up-moving lazy algorithm (relative to C)

Consider the condition that

$$D \quad \text{contains} \quad \left(\begin{array}{c} \star \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right) \tag{7.2}$$

Notice that this indicates that C_B has been completely updated, while C_T is yet untouched. Thus, we will call the resulting algorithm a lazy algorithm relative to operand C. In order to make progress towards completion of the computation of D, the boundary indicating the progress of the computation must be moved up. Thus, this algorithm naturally moves through matrices B, C and D in the "up" direction.

Unblocked algorithm

Repartition

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right)$$

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

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right), \quad \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline c_1^T \\ \hline\hline C_2 \end{array}\right), \text{ and } \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_{00} & a_{10} & A_{20}^T \\ \hline a_{10}^T & \alpha_{11} & a_{21}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

where b_1^T and c_1^T are rows and α_{11} is a scalar.

Notice that the double lines have meaning

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \quad \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right), \quad \left(\begin{array}{c} C_T = \left(\begin{array}{c} C_0 \\ \hline c_1^T \end{array}\right) \\ \hline\hline C_B = C_2 \end{array}\right),$$

and

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

Considering 7.2 and these repartitionings, D currently contains

$$\begin{aligned}\left(\begin{array}{c} C_T \\ \hline\hline A_{BL}B_T + A_{BR}B_B + C_B \end{array}\right) &= \left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline c_1^T \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} A_{20} & A_{21} \end{array}\right)\left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) + A_{22}B_2 + C_2 \end{array}\right) \\ &= \left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline c_1^T \end{array}\right) \\ \hline\hline A_{20}B_0 + A_{21}b_1^T + A_{22}B_2 + C_2 \end{array}\right)\end{aligned}$$

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

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \end{array}\right), \quad \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right), \quad \left(\begin{array}{c} C_T = C_0 \\ \hline\hline C_B = \left(\begin{array}{c} c_1^T \\ \hline C_2 \end{array}\right) \end{array}\right),$$

and

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

Thus, the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline c_1^T \end{array}\right) \\ \hline\hline A_{20}B_0 + A_{21}b_1^T + A_{22}B_2 + C_2 \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline \left(\begin{array}{c} a_{10}^T B_0 + \alpha_{11}b_1^T + (a_{21})^T B_2 + c_1^T \\ \hline A_{20}B_0 + A_{21}b_1^T + A_{22}B_2 + C_2 \end{array}\right) \end{array}\right)$$

We conclude that the algorithm maintains the condition in (7.2) is given in Fig. 7.2 (left). Notice that the algorithm overwrites matrix C with the result of $AB + C$.

Algorithm 14 $C \leftarrow AB + C$
(unblocked up-moving lazy w.r.t. C)

partition

$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$$

where B_B **and** C_B **have** 0 **rows**

partition

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

where A_{BR} **is** 0×0

do until A_{TL} **is** 0×0

repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$$

where c_1^T **is a row**

repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$$

where b_1^T **is a row**

repartition

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

where α_{11} **is a scalar**

$c_1^T \leftarrow a_{10}^T B_0 + c_1^T$
$c_1^T \leftarrow \alpha_{11} b_1^T + c_1^T$
$c_1^T \leftarrow a_{21}^T B_2 + c_1^T$

---

continue with

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

continue with

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline\hline c_1^T \\ \hline C_2 \end{array}\right)$$

continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$$

enddo

Algorithm 15 $C \leftarrow AB + C$
(blocked up-moving lazy w.r.t. C)

partition

$$C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline C_B \end{array}\right)$$

where B_B **and** C_B **have** 0 **rows**

partition

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

where A_{BR} **is** 0×0

do until A_{TL} **is** 0×0

determine block size b

repartition

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right)$$

where C_1 **has** b **rows**

repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$$

where B_1 **has** b **rows**

repartition

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

where A_{11} **is** $b \times b$

$C_1 \leftarrow A_{10} B_0 + C_1$
$C_1 \leftarrow A_{11} B_1 + C_1$
$C_1 \leftarrow A_{21}^T B_2 + C_1$

---

continue with

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

continue with

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right)$$

continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$$

enddo

Figure 7.2: Unblocked (left) and blocked (right) symmetric matrix-matrix multiply algorithms (up-moving lazy w.r.t. C).

Blocked Algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline\hline D_2 \end{array}\right)$$

where D_1 is a block of b rows. Similarly, we repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right), \quad \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right), \text{ and } \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_{00} & A_{10}^T & A_{20}^T \\ \hline A_{10} & A_{11} & A_{21}^T \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where B_1 and C_1 are blocks with b rows and A_{11} is $b \times b$.

Again, the double lines have meaning:

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \quad \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right), \quad \left(\begin{array}{c} C_T = \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline C_B = C_2 \end{array}\right),$$

and

$$\left(\begin{array}{c||c} A_{TL} = \left(\begin{array}{c|c} A_{00} & A_{10}^T \\ \hline A_{10} & A_{11} \end{array}\right) & A_{TR} = \left(\begin{array}{c} A_{20}^T \\ \hline A_{21}^T \end{array}\right) \\ \hline\hline A_{BL} = \left(\begin{array}{c|c} A_{20} & A_{21} \end{array}\right) & A_{BR} = A_{22} \end{array}\right)$$

Considering (7.2) and these repartitionings, D currently contains

$$\begin{aligned} \left(\begin{array}{c} C_T \\ \hline\hline A_{BR}B_T + A_{BR}B_B + C_B \end{array}\right) &= \left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} A_{20} & A_{21} \end{array}\right) \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) + A_{22}B_2 + C_2 \end{array}\right) \\ &= \left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline A_{20}B_0 + A_{21}B_1 + A_{22}B_2 + C_2 \end{array}\right) \end{aligned}$$

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

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} D_1 \\ \hline D_2 \end{array}\right) \end{array}\right), \quad \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \end{array}\right), \quad \left(\begin{array}{c} C_T = C_0 \\ \hline\hline C_B = \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right),$$

and

$$\left(\begin{array}{c||c} A_{TL} = A_{00} & A_{TR} = \left(\begin{array}{c|c} A_{10}^T & A_{20}^T \end{array}\right) \\ \hline\hline A_{BL} = \left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right) & A_{BR} = \left(\begin{array}{c|c} A_{11} & A_{21}^T \\ \hline A_{21} & A_{22} \end{array}\right) \end{array}\right)$$

Thus the contents of D must update as

$$\left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline C_0 \end{array}\right) \\ \hline\hline A_{20}B_0 + A_{21}B_1 + A_{22}B_2 + C_2 \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline \left(\begin{array}{c} A_{10}B_0 + A_{11}B_1 + A_{21}^T B_2 + C_1 \\ \hline A_{20}B_0 + A_{21}B_1 + A_{22}B_2 + C_2 \end{array}\right) \end{array}\right)$$

We can conclude that an algorithm which maintains the condition in 7.1 is given by Fig. 7.2 (right). Notice that the algorithm overwrites matrix C with the result $AB + C$.

7.1.2 Down-moving algorithm (relative to C)

Consider the condition that

$$D \quad \text{contains} \quad \left(\frac{A_{TL}B_T + A_{BR}^T B_B + C_T}{C_B} \right) \tag{7.3}$$

Notice that this indicates that C_T has been completely updated while C_B has been left untouched. Thus, we will call the resulting algorithm a down-moving lazy algorithm relative to operand C.

Unblocked Algorithm

Repartition

$$\left(\frac{D_T}{D_B} \right) \to \left(\begin{array}{c} D_0 \\ \hline\hline d_1^T \\ \hline D_2 \end{array} \right)$$

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

$$\left(\frac{B_T}{B_B} \right) \to \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array} \right), \quad \left(\frac{C_T}{C_B} \right) \to \left(\begin{array}{c} C_0 \\ \hline\hline c_1^T \\ \hline C_2 \end{array} \right), \text{ and } \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \to \left(\begin{array}{c||c|c} A_{00} & a_{10} & A_{20}^T \\ \hline\hline a_{10}^T & \alpha_{11} & a_{21}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array} \right)$$

where b_1^T and c_1^T are rows and α_{11} is a scalar.

Again, notice that the double lines have meaning

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\frac{d_1^T}{D_2} \right) \end{array} \right), \quad \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\frac{b_1^T}{B_2} \right) \end{array} \right), \quad \left(\begin{array}{c} C_T = C_0 \\ \hline\hline C_B = \left(\frac{c_1^T}{C_2} \right) \end{array} \right),$$

and

$$\left(\begin{array}{c||c} A_{TL} = A_{00} & A_{BL}^T = \left(\begin{array}{c|c} (a_{10}^T)^T & (A_{20})^T \end{array} \right) \\ \hline\hline A_{BL} = \left(\frac{a_{10}^T}{A_{20}} \right) & A_{BR} = \left(\begin{array}{c|c} \alpha_{11} & a_{21}^T \\ \hline a_{21} & A_{22} \end{array} \right) \end{array} \right)$$

Considering 7.3 and these repartitionings, D currently contains

$$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + C_T}{C_B} \right) = \left(\begin{array}{c} A_{00}B_0 + \left(\begin{array}{c|c} a_{10} & A_{20}^T \end{array} \right) \left(\frac{b_1^T}{B_2} \right) + C_0 \\ \hline\hline \left(\frac{c_1^T}{C_2} \right) \end{array} \right) =$$

$$\left(\begin{array}{c} A_{00}B_0 + a_{10}b_1^T + A_{20}^T B_2 + C_0 \\ \hline\hline \left(\frac{c_1^T}{C_2} \right) \end{array} \right)$$

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

$$\left(\begin{array}{c} D_T = \left(\frac{D_0}{d_1^T} \right) \\ \hline\hline D_B = D_2 \end{array} \right), \quad \left(\begin{array}{c} B_T = \left(\frac{B_0}{b_1^T} \right) \\ \hline\hline B_B = B_2 \end{array} \right), \quad \left(\begin{array}{c} C_T = \left(\frac{C_0}{c_1^T} \right) \\ \hline\hline C_B = C_2 \end{array} \right),$$

and

$$\left(\begin{array}{c||c} A_{TL} = \left(\begin{array}{c|c} A_{00} & (a_{10}^T)^T \\ \hline a_{10}^T & \alpha_{11} \end{array} \right) & A_{BL}^T = \left(\frac{(A_{20})^T}{(a_{21})^T} \right) \\ \hline\hline A_{BL} = \left(\begin{array}{c|c} A_{20} & A_{21} \end{array} \right) & A_{BR} = A_{22} \end{array} \right)$$

Thus, the contents of D must be updated as

$$\left(\begin{array}{c} A_{00}B_0 + a_{10}b_1^T + A_{20}^T B_2 + C_0 \\ \hline\hline \left(\begin{array}{c} c_1^T \\ \hline C_2 \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} A_{00}B_0 + a_{10}b_1^T + A_{20}^T B_2 + C_0 \\ \hline a_{10}^T B_0 + \alpha_{11} b_1^T + a_{21}^T B_2 + C_1^T \end{array}\right) \\ \hline\hline C_2 \end{array}\right)$$

We conclude that the algorithm maintains the condition in (7.3) is given in Fig. 7.3 (left). Notice that the algorithm overwrites matrix C with the result of $AB + C$.

Blocked Algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline\hline D_1 \\ \hline D_2 \end{array}\right)$$

where D_1 is a block of b rows. Similarly, we repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right), \text{ and } \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{10}^T & A_{20}^T \\ \hline\hline A_{10} & A_{11} & A_{21}^T \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where B_1 and C_1 are blocks with b rows and A_{11} is $b \times b$.

Again, the double lines have meaning:

$$\text{(7.4)} \qquad \left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} D_1 \\ \hline D_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} C_T = C_0 \\ \hline\hline C_B = \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c||c} A_{TL} = A_{00} & A_{BL}^T = \left(\begin{array}{c|c} (A_{10}^T)^T & (A_{20})^T \end{array}\right) \\ \hline\hline A_{BL} = \left(\begin{array}{c} A_{10}^T \\ \hline A_{20} \end{array}\right) & A_{BR} = \left(\begin{array}{c|c} A_{11} & A_{21}^T \\ \hline A_{21} & A_{22} \end{array}\right) \end{array}\right)$$

Considering (7.3) and these repartitionings, D currently contains

$$\left(\begin{array}{c} A_{TL}B_T + A_{BL}^T B_B + C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} A_{00}B_0 + \left(\begin{array}{c|c} A_{10} & A_{20}^T \end{array}\right) \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) + C_0 \\ \hline\hline \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right) =$$

$$\left(\begin{array}{c} A_{00}B_0 + A_{10}B_1 + A_{20}^T B_2 + C_0 \\ \hline\hline \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right)$$

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

$$\text{(7.5)} \qquad \left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right), \left(\begin{array}{c} C_T = \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline C_B = C_2 \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c||c} A_{TL} = \left(\begin{array}{c|c} A_{00} & A_{10}^T \\ \hline A_{10} & A_{11} \end{array}\right) & A_{TR} = \left(\begin{array}{c} A_{20}^T \\ \hline A_{21}^T \end{array}\right) \\ \hline\hline A_{BL} = \left(\begin{array}{c|c} A_{20} & A_{21} \end{array}\right) & A_{BR} = A_{22} \end{array}\right)$$

Algorithm 16 $C \leftarrow AB + C$
(unblocked down-moving lazy w.r.t. C)

partition

$$C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline C_B \end{array}\right)$$

where B_T **and** C_T **have** 0 **rows**

partition

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

where A_{BR} **is** 0×0

do until A_{TL} **is** 0×0

repartition

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline c_1^T \\ \hline C_2 \end{array}\right)$$

where c_1^T **is a row**

repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$$

where b_1^T **is a row**

repartition

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

where α_{11} **is a scalar**

$c_1^T \leftarrow a_{10}^T B_0 + c_1^T$
$c_1^T \leftarrow \alpha_{11} b_1^T + c_1^T$
$c_1^T \leftarrow a_{21}^T B_2 + c_1^T$

---

continue with

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

continue with

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline c_1^T \\ \hline\hline C_2 \end{array}\right)$$

continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$$

enddo

Algorithm 17 $C \leftarrow AB + C$
(blocked down-moving lazy w.r.t. C)

partition

$$C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \quad \text{and} \quad C \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline C_B \end{array}\right)$$

where B_T **and** C_T **have** 0 **rows**

partition

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

where A_{TL} **is** 0×0

do until A_{BR} **is** 0×0

determine block size b

repartition

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right)$$

where C_1 **has** b **rows**

repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$$

where B_1 **has** b **rows**

repartition

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

where A_{11} **is** $b \times b$

$C_1 \leftarrow A_{10} B_0 + C_1$
$C_1 \leftarrow A_{11} B_1 + C_1$
$C_1 \leftarrow A_{21}^T B_2 + C_1$

---

continue with

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

continue with

$$\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right)$$

continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$$

enddo

Figure 7.3: Unblocked (left) and blocked (right) symmetric matrix-matrix multiply algorithms (down-moving lazy w.r.t. C).

Thus the contents of D must update as

$$\left(\begin{array}{c} A_{00}B_0 + A_{10}B_1^T + A_{20}^T B_2 + C_0 \\ \hline\hline \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} A_{00}B_0 + A_{10}B_1^T + A_{20}^T B_2 + C_0 \\ \hline A_{10}^T B_0 + A_{11}B_1^T + A_{21}^T B_2 + C_1^T \end{array}\right) \\ \hline\hline C_2 \end{array}\right)$$

We conclude that an algorithm which maintains the condition in 7.3 is given by Fig. 7.3 (right). Notice that the algorithm overwrites matrix C with the result $AB + C$.

7.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/symm_ll/`

7.3 Performance

```
#include "FLAME.h"

int Symm_ll_up_lazy_wrt_C_unb ( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
  FLA_Obj     ATL, ATR,  A00, a01,      A02,  BT,  B0,   CT,  C0,
              ABL, ABR,  a10, alpha11,  a12,  BB,  b1t,  CB,  c1t,
                         A20, a21,      A22,       B2,        C2;
  int         b;

  FLA_Part_2x2 (A, &ATL, &ATR,
                   &ABL, &ABR,
                0, 0, FLA_BR);
  FLA_Part_2x1 (B, &BT,
                   &BB,
                0, FLA_BOTTOM);
  FLA_Part_2x1 (C, &CT,
                   &CB,
                0, FLA_BOTTOM);

  while (FLA_Obj_length (ATL) != 0){
    FLA_Repart_2x2_to_3x3 (ATL, ATR,  &A00,  &a01,     &A02,
                                      &a10,  &alpha11, &a12,
                           ABL, ABR,  &A20,  &a21,     &A22,
                           1, 1, FLA_TL);
    FLA_Repart_2x1_to_3x1 (BT,         &B0,
                                       &b1t,
                           BB,         &B2,
                           1, FLA_TOP);
    FLA_Repart_2x1_to_3x1 (CT,         &C0,
                                       &c1t,
                           CB,         &C2,
                           1, FLA_TOP);

    /* ------------ Compute ---------------------------------------------- */
    FLA_Gemv ( FLA_TRANSPOSE, ONE, B0, a10, ONE, c1t);
    FLA_Axpy (alpha11, b1t, c1t);
    FLA_Gemv ( FLA_TRANSPOSE, ONE, B2, a21, ONE, c1t);
    /* ------------------------------------------------------------------- */

    FLA_Cont_with_3x3_to_2x2 (&ATL, &ATR,  A00,  a01,     A02,
                                           a10,  alpha11, a12,
                              &ABL, &ABR,  A20,  a21,     A22,
                              FLA_BR);
    FLA_Cont_with_3x1_to_2x1 (&BT,         B0,
                                           b1t,
                              &BB,         B2,
                              FLA_BOTTOM);
    FLA_Cont_with_3x1_to_2x1 (&CT,         C0,
                                           c1t,
                              &CB,         C2,
                              FLA_BOTTOM);
  }
}
```

Figure 7.4: Unblocked up-moving lazy (w.r.t. C) symmetric matrix-matrix multiply.

```
#include "FLAME.h"

int Symm_ll_up_lazy_wrt_C_blk ( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,  BT,  B0,  CT,  C0,
              ABL, ABR,     A10, A11, A12,  BB,  B1,  CB,  C1,
                            A20, A21, A22,       B2,       C2;
  int         b;

  FLA_Part_2x2 (A, &ATL, &ATR,
                   &ABL, &ABR,
                0, 0, FLA_BR);
  FLA_Part_2x1 (B, &BT,
                   &BB,
                0, FLA_BOTTOM);
  FLA_Part_2x1 (C, &CT,
                   &CB,
                0, FLA_BOTTOM);

  while (b = min (FLA_Obj_length (ATL), nb_alg)) {
    FLA_Repart_2x2_to_3x3 (ATL, ATR,  &A00, &A01, &A02,
                                      &A10, &A11, &A12,
                           ABL, ABR,  &A20, &A21, &A22,
                           b, b, FLA_TL);
    FLA_Repart_2x1_to_3x1 (BT,         &B0,
                                       &B1,
                           BB,         &B2,
                           b, FLA_TOP);
    FLA_Repart_2x1_to_3x1 (CT,         &C0,
                                       &C1,
                           CB,         &C2,
                           b, FLA_TOP);

    /* ------------ Compute ----------------------------------------------- */
    FLA_Gemm (FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B0, ONE, C1);
    Symm_ll_up_lazy_wrt_C_unb ( A11, B1, C1 );
    FLA_Gemm (FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A21, B2, ONE, C1);
    /* -------------------------------------------------------------------- */

    FLA_Cont_with_3x3_to_2x2 (&ATL, &ATR,  A00, A01, A02,
                                           A10, A11, A12,
                              &ABL, &ABR,  A20, A21, A22,
                              FLA_BR);
    FLA_Cont_with_3x1_to_2x1 (&BT,         B0,
                                           B1,
                              &BB,         B2,
                              FLA_BOTTOM);
    FLA_Cont_with_3x1_to_2x1 (&CT,         C0,
                                           C1,
                              &CB,         C2,
                              FLA_BOTTOM);
  }
}
```

Figure 7.5: Blocked up-moving lazy (w.r.t. C) symmetric matrix-matrix multiply.

```
#include "FLAME.h"

int Symm_ll_up_lazy_wrt_C_rec ( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,  BT,  B0,  CT,  C0,
              ABL, ABR,     A10, A11, A12,  BB,  B1,  CB,  C1,
                            A20, A21, A22,       B2,       C2;
  int         b;

  FLA_Part_2x2 (A, &ATL, &ATR,
                   &ABL, &ABR,
                0, 0, FLA_BR);
  FLA_Part_2x1 (B, &BT,
                   &BB,
                0, FLA_BOTTOM);
  FLA_Part_2x1 (C, &CT,
                   &CB,
                0, FLA_BOTTOM);
  while (b = min (FLA_Obj_length (ATL), nb_alg)) {
    FLA_Repart_2x2_to_3x3 (ATL, ATR,  &A00, &A01, &A02,
                                      &A10, &A11, &A12,
                           ABL, ABR,  &A20, &A21, &A22,
                           b, b, FLA_TL);
    FLA_Repart_2x1_to_3x1 (BT,         &B0,
                                       &B1,
                           BB,         &B2,
                           b, FLA_TOP);
    FLA_Repart_2x1_to_3x1 (CT,         &C0,
                                       &C1,
                           CB,         &C2,
                           b, FLA_TOP);

    /* ------------ Compute ----------------------------------------------- */
    FLA_Gemm (FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B0, ONE, C1);

    if (b <= 8)
      Symm_ll_up_lazy_wrt_C_unb ( A11, B1, C1 );
    else
      Symm_ll_up_lazy_wrt_C_rec ( A11, B1, C1, nb_alg/2 );

    FLA_Gemm (FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A21, B2, ONE, C1);
    /* -------------------------------------------------------------------- */

    FLA_Cont_with_3x3_to_2x2 (&ATL, &ATR,  A00, A01, A02,
                                           A10, A11, A12,
                              &ABL, &ABR,  A20, A21, A22,
                              FLA_BR);
    FLA_Cont_with_3x1_to_2x1 (&BT,         B0,
                                           B1,
                              &BB,         B2,
                              FLA_BOTTOM);
    FLA_Cont_with_3x1_to_2x1 (&CT,         C0,
                                           C1,
                              &CB,         C2,
                              FLA_BOTTOM);
  }
}
```

Figure 7.6: Recursive up-moving lazy (w.r.t. C) symmetric matrix-matrix multiply.

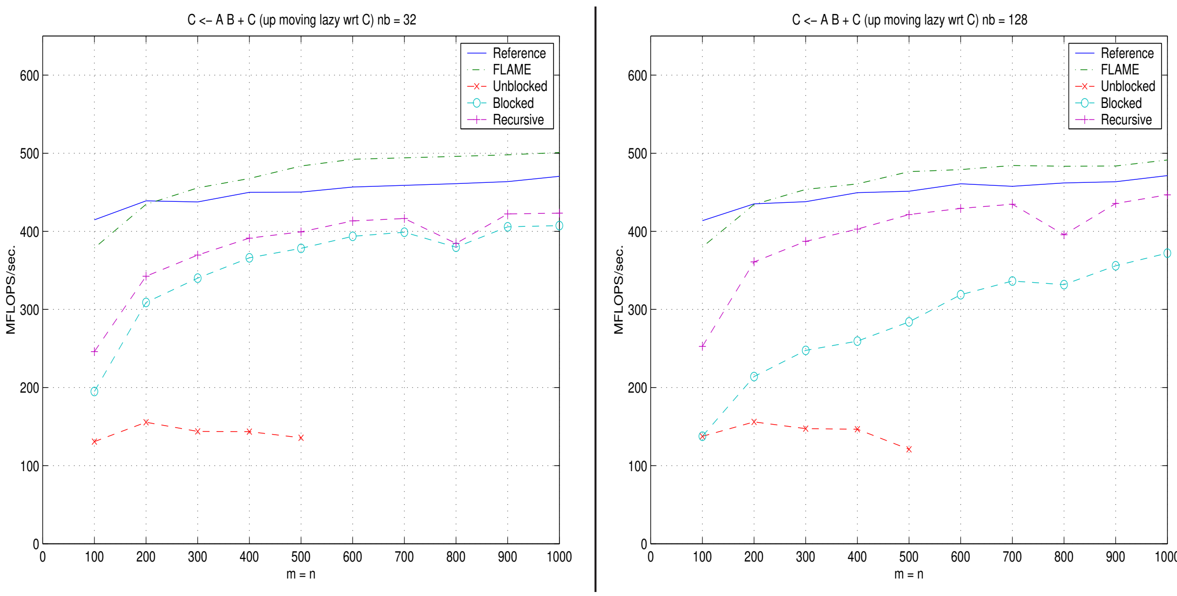


Figure 7.7: Performance of up-moving lazy w.r.t. C symmetric matrix-matrix multiplication algorithm for a block size of 32 (left) and block size 128 (right).

Chapter 8

Symmetric Rank-k Update $C \leftarrow AA^T + C$ C symmetric, stored in lower triangle

by

Daniel J. Atkinson

Brian R. Walker

In this chapter, we discuss the implementation of symmetric rank-k update:

$$C \leftarrow AA^T + C$$

where C is a symmetric $n \times n$ matrix stored in the lower triangle and A is $n \times k$.

8.1 Algorithms that start by splitting C

Let us consider

$$D = AA^T + C \tag{8.1}$$

keeping in mind that D will overwrite C.

Partition

$$C \rightarrow \left(\begin{array}{c\|c} C_{TL} & C_{BL}^T \\ \hline C_{BL} & C_{BR} \end{array}\right), \quad A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right), \quad \text{and} \quad D \rightarrow \left(\begin{array}{c\|c} D_{TL} & D_{BL}^T \\ \hline D_{BL} & D_{BR} \end{array}\right)$$

where C_{TL} and D_{TL} are $k \times k$ matrices and A_T has k rows.

Now, (8.1) becomes

$$\begin{aligned} \left(\begin{array}{c\|c} D_{TL} & D_{BL}^T \\ \hline D_{BL} & D_{BR} \end{array}\right) &= \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \left(\begin{array}{c\|c} A_T^T & A_B^T \end{array}\right) + \left(\begin{array}{c\|c} C_{TL} & C_{BL}^T \\ \hline C_{BL} & C_{BR} \end{array}\right) \\ &= \left(\begin{array}{c\|c} A_T A_T^T + C_{TL} & A_T A_B^T + C_{BL}^T \\ \hline A_B A_T^T + C_{BL} & A_B A_B^T + C_{BR} \end{array}\right) \end{aligned}$$

This exposes the equalities

$$\begin{array}{rcl} D_{TL} & = & A_T A_T^T + C_{TL} \\ D_{BL} & = & A_B A_T^T + C_{BL} \\ D_{BR} & = & A_B A_B^T + C_{BR} \end{array}$$

which must hold.

Possible contents of D at this intermediate step are given in Fig. 8.1.

D contains	Comments
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c\|c} A_T A_T^T + C_{TL} & \star \\ \hline A_B A_T^T + C_{BL} & A_B A_B^T + C_{BR} \end{array}\right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(\begin{array}{c\|c} A_T A_T^T + C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & A_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B A_T^T + C_{BL} & A_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} A_T A_T^T + C_{TL} & \star \\ \hline C_{BL} & A_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} A_T A_T^T + C_{TL} & \star \\ \hline A_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	

Figure 8.1: Possible contents of D at an intermediate step.

8.1.1 Up-left moving algorithm (lazy w.r.t. C)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & A_B A_B^T + C_{BR} \end{array}\right) \tag{8.2}$$

Notice that this indicates that $A_B A_B^T$ has been used to update D.

In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved left and up. Thus, this algorithm naturally moves through matrices D and C in the "up-left" direction.

Unblocked algorithm

,

Repartition

$$\left(\begin{array}{c|c} D_{TL} & \star \\ \hline D_{BL} & D_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_{00} & \star & \star \\ \hline d_{10}^T & \delta_{11} & \star \\ \hline D_{20} & d_{21} & D_{22} \end{array}\right)$$

where d_{10}^T is a row and δ_{11} is a scalar. Similarly, we repartition

$$\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline A_2 \end{array}\right)$$

where a_1^T is a row and γ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(\begin{array}{c||c} D_{TL} = \left(\begin{array}{c|c} D_{00} & \star \\ \hline d_{10}^T & \delta_{11} \end{array}\right) & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c|c} D_{20} & d_{21} \end{array}\right) & D_{BR} = D_{22} \end{array}\right), \quad \left(\begin{array}{c||c} C_{TL} = \left(\begin{array}{c|c} C_{00} & \star \\ \hline c_{10}^T & \gamma_{11} \end{array}\right) & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c|c} C_{20} & c_{21} \end{array}\right) & C_{BR} = C_{22} \end{array}\right), \tag{8.3}$$

and

$$\left(\begin{array}{c} A_T = \left(\begin{array}{c} A_0 \\ \hline a_1^T \end{array}\right) \\ \hline\hline A_B = A_2 \end{array}\right) \tag{8.4}$$

Considering (8.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & A_B A_B^T + C_{BR} \end{array}\right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline c_{10}^T & \gamma_{11} \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21} \end{array}\right) & A_2 A_2^T + C_{22} \end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL} = D_{00} & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c} d_{10}^T \\ \hline D_{20} \end{array}\right) & D_{BR} = \left(\begin{array}{c|c} \delta_{11} & \star \\ \hline d_{21} & D_{22} \end{array}\right) \end{array}\right), \quad \left(\begin{array}{c||c} C_{TL} = C_{00} & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & C_{BR} = \left(\begin{array}{c|c} \gamma_{11} & \star \\ \hline c_{21} & C_{22} \end{array}\right) \end{array}\right),$$

and

$$\left(\begin{array}{c} A_T = A_0 \\ \hline\hline A_B = \left(\begin{array}{c} a_1^T \\ \hline A_2 \end{array}\right) \end{array}\right)$$

Thus the contents of D be updated to hold

$$\begin{aligned} \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & A_B A_B^T + C_{BR} \end{array}\right) &= \left(\begin{array}{c||c} C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c} a_1^T \\ \hline A_2 \end{array}\right)\left(\begin{array}{c} a_1^T \\ \hline A_2 \end{array}\right)^T + \left(\begin{array}{c||c} \gamma_{11} & \star \\ \hline\hline c_{21} & C_{22} \end{array}\right) \end{array}\right) \\ &= \left(\begin{array}{c||c} C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c|c} \gamma_{11} + a_1^T a_1 & \star \\ \hline c_{21} + A_2 a_1 & A_2 A_2^T + C_{22} \end{array}\right) \end{array}\right) \end{aligned}$$

Thus, maintaining the condition requires the following update:

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline c_{10}^T & \gamma_{11} \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21} \end{array}\right) & A_2 A_2^T + C_{22} \end{array}\right) \rightarrow \left(\begin{array}{c||c} C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c|c} \gamma_{11} + a_1^T a_1 & \star \\ \hline c_{21} + A_2 a_1 & A_2 A_2^T + C_{22} \end{array}\right) \end{array}\right)$$

An algorithm that maintains the condition in (8.2) is given in Fig. 8.2 (left). Notice that the algorithm overwrites matrix C with the result $AA^T + C$.

Blocked algorithm

For the blocked algorithm, we repartition as follows: Repartition

$$\left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} D_{00} & \star & \star \\ \hline D_{10} & D_{11} & \star \\ \hline\hline D_{20} & D_{21} & D_{22} \end{array}\right)$$

where D_{11} is $b \times b$ Similarly, we repartition

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline\hline C_{20} & C_{21} & C_{22}\end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} A_T \\ \hline\hline A_B\end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline\hline A_2 \end{array}\right)$$

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

Again, the double lines have meaning:

$$\left(\begin{array}{c||c} D_{TL} = \left(\begin{array}{c|c} D_{00} & \star \\ \hline D_{10} & D_{11}\end{array}\right) & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c|c} D_{20} & D_{21}\end{array}\right) & D_{BR} = D_{22}\end{array}\right), \quad \left(\begin{array}{c||c} C_{TL} = \left(\begin{array}{c|c} C_{00} & \star \\ \hline C_{10} & C_{11}\end{array}\right) & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c|c} C_{20} & C_{21}\end{array}\right) & C_{BR} = C_{22}\end{array}\right),$$

and

$$\left(\begin{array}{c} A_T = \left(\begin{array}{c} A_0 \\ \hline A_1\end{array}\right) \\ \hline\hline A_B = A_2\end{array}\right)$$

Usingthese repartitionings, D currently contains

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & A_B A_B^T + C_{BR}\end{array}\right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline C_{10} & C_{11}\end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21}\end{array}\right) & A_2 A_2^T + C_{22}\end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL} = D_{00} & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c} D_{10} \\ \hline D_{20}\end{array}\right) & D_{BR} = \left(\begin{array}{c|c} D_{11} & \star \\ \hline D_{21} & D_{22}\end{array}\right)\end{array}\right), \left(\begin{array}{c||c} C_{TL} = C_{00} & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c} C_{10} \\ \hline C_{20}\end{array}\right) & C_{BR} = \left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22}\end{array}\right)\end{array}\right),$$

and

$$\left(\begin{array}{c} A_T = A_0 \\ \hline\hline A_B = \left(\begin{array}{c} A_1 \\ \hline A_2\end{array}\right)\end{array}\right)$$

Thus the contents of D must change like

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline C_{10} & C_{11}\end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21}\end{array}\right) & A_2 A_2^T + C_{22}\end{array}\right) \rightarrow \left(\begin{array}{c||c} C_{00} & \star \\ \hline\hline \left(\begin{array}{c} C_{10} \\ \hline C_{20}\end{array}\right) & \left(\begin{array}{c|c} C_{11} + A_1^T A_1 & \star \\ \hline C_{21} + A_2 A_1 & A_2 A_2^T + C_{22}\end{array}\right)\end{array}\right)$$

A blocked algorithm that maintains the condition in (8.2) is given in Fig. 8.2 (right).

8.1.2 Down-right moving algorithm (lazy w.r.t. C)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c||c} A_T A_T^T + C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) \tag{8.5}$$

Notice that this indicates that $A_T A_T^T$ has been used to update D.

In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved right and down. Thus, this algorithm naturally moves through matrices D and C in the "down-right" direction.

Algorithm 18 $C \leftarrow AA^T + C$
(unblocked up-left moving lazy w.r.t. C)

partition
$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$
where C_{BR} **is** 0×0
partition
$$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)$$
where A_B **has** 0 **rows**
do until C_{TL} **is** 0×0

repartition
$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline c_{10}^T & \gamma_{11} & \star \\ \hline\hline C_{20} & c_{21} & C_{22} \end{array}\right)$$
where γ_{11} **is a scalar**
repartition
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline\hline A_2 \end{array}\right)$$
where a_1^T **is a row**

$\gamma_{11} \leftarrow \gamma_{11} + a_1 a_1^T$
$c_{21} \leftarrow c_{21} + A_2 a_1^T$

---

continue with
$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline\hline a_1^T \\ \hline A_2 \end{array}\right)$$
enddo

Algorithm 19 $C \leftarrow AA^T + C$
(blocked up-left moving lazy w.r.t. C)

partition
$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$
where C_{BR} **is** 0×0
partition
$$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)$$
where A_B **has** 0 **rows**
do until C_{TL} **is** 0×0

repartition
$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right)$$
where C_{11} **is** $b \times b$
repartition
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline\hline A_2 \end{array}\right)$$
where A_1 **is b rows**

$C_{11} \leftarrow C_{11} + A_1 A_1^T$
$C_{21} \leftarrow C_{21} + A_2 A_1^T$

---

continue with
$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array}\right)$$
enddo

Figure 8.2: Unblocked (left) and blocked (right) up-left moving symmetric rank-k update algorithms (lazy w.r.t. C).

Unblocked algorithm

If one follows the same steps are in Section 8.1.1, one finds that in order to maintain the Condition (8.5), one must update the contents of D like:

$$\left(\begin{array}{c||c} A_0 A_0^T + C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c|c} \gamma_{11} & \star \\ \hline c_{21} & C_{22} \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} + A_0 A_0^T & \star \\ \hline c_{10}^T + a_1 A_0^T & \gamma_{11} + a_1 a_1^T \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21} \end{array}\right) & C_{22} \end{array}\right)$$

This leads to the algorithm given in Fig. 8.3 (left). Notice that the algorithm overwrites matrix C with the result $AA^T + C$.

Blocked algorithm

For the blocked algorithm, the update is given by

$$\left(\begin{array}{c||c} C_{00} & \star \\ \hline\hline \left(\begin{array}{c} C_{10} \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22} \end{array}\right) \end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} + A_0 A_0^T & \star \\ \hline C_{10} + A_1 A_0^T & C_{11} + A_1 A_1^T \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21} \end{array}\right) & C_{22} \end{array}\right)$$

leading to the algorithm in Fig. 8.3 (right).

8.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/syrk_ln/`

8.3 Performance

Algorithm 20 $C \leftarrow AA^T + C$
(unblocked down-right moving lazy w.r.t. C)

partition

$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

where C_{BR} **is** 0×0

partition

$$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)$$

where A_B **has** 0 **rows**

do until C_{TL} **is** 0×0

repartition

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right)$$

where γ_{11} **is a scalar**

repartition

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline a_1^T \\ \hline A_2 \end{array}\right)$$

where a_1^T **is a row**

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

---

continue with

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline c_{10}^T & \gamma_{11} & \star \\ \hline\hline C_{20} & c_{21} & C_{22} \end{array}\right)$$

continue with

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline\hline A_2 \end{array}\right)$$

enddo

Algorithm 21 $C \leftarrow AA^T + C$
(blocked down-right moving lazy w.r.t. C)

partition

$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

where C_{BR} **is** 0×0

partition

$$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)$$

where A_B **has** 0 **rows**

do until C_{TL} **is** 0×0

repartition

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

where C_{11} **is** $b \times b$

repartition

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2 \end{array}\right)$$

where A_1 **is b rows**

$C_{11} \leftarrow C_{11} + A_1 A_1^T$
$C_{10} \leftarrow C_{10} + A_1 A_0^T$

---

continue with

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

continue with

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline\hline A_2 \end{array}\right)$$

enddo

Figure 8.3: Unblocked (left) and blocked (right) down-right moving symmetric rank-k update algorithms (lazy w.r.t. C).

```
#include "FLAME.h"

void Syrk_ln_upleft_wrt_C_unb( FLA_Obj A, FLA_Obj C )
{
  FLA_Obj     CTL, CTR,     C00,  C01,     C02,         AT,      A0,
              CBL, CBR,     c10t, gamma11, C12,         AB,      a1t,
                            C20,  c21,     C22,         A2;
  int         b;

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                 /* ************** */
                   &CBL, /**/ &CBR,   0, 0,     /* submatrix */ FLA_TL );

  FLA_Part_2x1( A,  &AT,
                  /***/
                   &AB,               0, /* length submatrix */ FLA_TOP );

  while ( 0 != FLA_Obj_length( CBR ) ){
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00,  &C01,     /**/   &C02,
                                                  &c10t, &gamma11, /**/   &C12,
                        /* ************* */    /* *************************** */
                           CBL, /**/ CBR,         &C20,  &c21,     /**/   &C22,
                           1, 1, /* gamma11 from */ FLA_BR );

    FLA_Repart_2x1_to_3x1( AT,                   &A0,
                                                 &a1t,
                           /**/                  /**/
                           AB,                   &A2,
                           1, /* length b1t from */ FLA_BOTTOM );

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

    FLA_Dot_x( ONE, a1t, a1t, ONE, gamma11 );

    FLA_Gemv(FLA_NO_TRANSPOSE, ONE, A2, a1t, ONE, c21 );

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

    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,         C00,  /**/ C01,     C02,
                             /* ************** */   /* ************************ */
                                    /**/               c10t, /**/ gamma11, C12,
                              &CBL, /**/ &CBR,         C20,  /**/ c21,     C22,
                              /* gamma11 added to */ FLA_TL );

    FLA_Cont_with_3x1_to_2x1( &AT,                    A0,
                             /***/                   /**/
                                                      a1t,
                              &AB,                    A2,
                              /* a1t added to */ FLA_TOP );
  }
}
```

Figure 8.4: Unblocked up-left moving (w.r.t. C) symmetric rank-k update algorithm using FLAME.

```
#include "FLAME.h"

void Syrk_ln_upleft_wrt_C_unb( FLA_Obj A, FLA_Obj C )
{
  FLA_Obj     CTL, CTR,     C00,  C01,     C02,         AT,      A0,
              CBL, CBR,     c10t, gamma11, C12,         AB,      a1t,
                            C20,  c21,     C22,         A2;
  int         b;

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                 /* ************** */
                   &CBL, /**/ &CBR,   0, 0,     /* submatrix */ FLA_TL );

  FLA_Part_2x1( A,  &AT,
                  /***/
                   &AB,               0, /* length submatrix */ FLA_TOP );

  while ( 0 != FLA_Obj_length( CBR ) ){
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00,  &C01,     /**/   &C02,
                                                  &c10t, &gamma11, /**/   &C12,
                       /* ************* */    /* *************************** */
                           CBL, /**/ CBR,         &C20,  &c21,     /**/   &C22,
                           1, 1, /* gamma11 from */ FLA_BR );

    FLA_Repart_2x1_to_3x1( AT,                   &A0,
                                                 &a1t,
                           /**/                  /**/
                           AB,                   &A2,
                           1, /* length b1t from */ FLA_BOTTOM );

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

    FLA_Dot_x( ONE, a1t, a1t, ONE, gamma11 );

    FLA_Gemv(FLA_NO_TRANSPOSE, ONE, A2, a1t, ONE, c21 );

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

    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,         C00,  /**/ C01,     C02,
                            /* ************** */   /* ************************ */
                                     /**/              c10t, /**/ gamma11, C12,
                             &CBL, /**/ &CBR,         C20,  /**/ c21,     C22,
                             /* gamma11 added to */ FLA_TL );

    FLA_Cont_with_3x1_to_2x1( &AT,                    A0,
                            /***/                  /**/
                                                      a1t,
                             &AB,                     A2,
                             /* a1t added to */ FLA_TOP );
  }
}
```

Figure 8.5: Blocked up-left moving (w.r.t. C) symmetric rank-k update algorithm using FLAME.

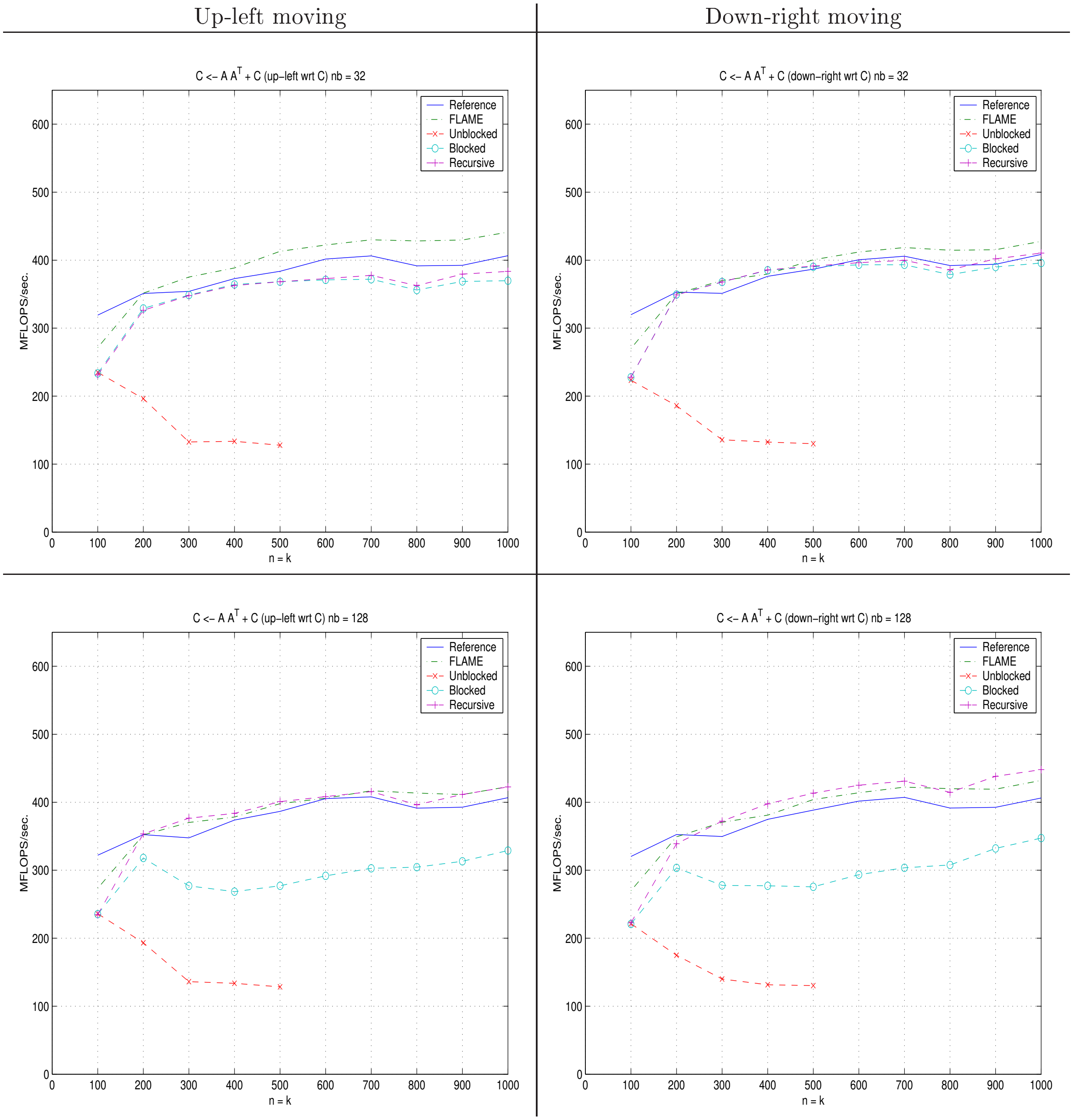


Figure 8.6: Performance of the up-right moving (left) and down-right moving (right) (w.r.t. C) symmetric rank-k update for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Chapter 9

Symmetric Rank-2k Update $C \leftarrow AB^T + BA^T + \beta C$ C symmetric, stored in lower triangle

by

Jin Pan

Hoa V. Phan

Yu-Hung (Eddie) Wang

In this chapter, we discuss the implementation of triangular matrix-matrix multiplication:

$$C \leftarrow (AB^T + BA^T) + C$$

where C is a symmetric lower triangular matrix.

Let us consider

$$D = (AB^T + BA^T) + C \tag{9.1}$$

keeping in mind that D will overwrite C.

9.1 Algorithms that start by splitting C

Partition

$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

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

$$D \rightarrow \left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR} \end{array}\right), \quad A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$

where D_T, A_T and B_T have k rows.

Now, (9.1) becomes

$$\left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) = \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)^T + \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right)^T + \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) = \left(\begin{array}{c||c} A_T B_T^T & \star \\ \hline\hline A_B B_T^T & A_B B_B^T \end{array}\right) + \left(\begin{array}{c||c} B_T A_T^T & \star \\ \hline\hline B_B A_T^T & B_B A_B^T \end{array}\right) + \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

$$= \left(\begin{array}{c||c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline\hline A_B B_T^T + B_B A_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_{TL} & = & \left[A_T B_T^T + B_T A_T^T\right] + C_{TL} \\ D_{BL} & = & A_B B_T^T + B_B A_T^T + C_{BL} \\ D_{BR} & = & \left[A_B B_B^T + B_B A_B^T\right] + C_{BR} \end{array}$$

which must hold. Here we use the brackets, [ ] to indicate that to preserve symmetry, the computations $A_T B_T^T + B_T A_T^T$ and $A_B B_B^T + B_B A_B^T$ should be considered a single update rather than two separate computations.

Possible contents of D at this intermediate step are given in Fig. 9.1. Considering the comments in that table, only two viable conditions are left, the ones for which there are no comments.

9.1.1 Lazy algorithm (down-right moving relative to C)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c||c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \tag{9.2}$$

Notice that this indicates that A_T and B_T have been used to update D, which will call a Right-bottm algorithm relative to operand A_T and B_T.

Unblocked algorithm

Repartition

$$\left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} D_{00} & \star & \star \\ \hline\hline d_{10}^T & \delta_{11} & \star \\ \hline D_{20} & d_{21} & D_{22} \end{array}\right)$$

where d_{10}^T is a row. Similarly, repartition

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline a_1^T \\ \hline A_2 \end{array}\right), \quad \text{and} \quad \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right), \quad \text{and} \quad \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right)$$

where a_1^T , b_1^T are rows and γ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(\begin{array}{c||c} D_{TL} = D_{00} & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c} d_{10}^T \\ \hline D_{20} \end{array}\right) & D_{BR} = \left(\begin{array}{c|c} \delta_{11} & \star \\ \hline d_{21} & D_{22} \end{array}\right) \end{array}\right), \left(\begin{array}{c||c} C_{TL} = C_{00} & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c} c_{10}^T \\ \hline C_{20} \end{array}\right) & C_{BR} = \left(\begin{array}{c|c} \gamma_{11} & \star \\ \hline c_{21} & C_{22} \end{array}\right) \end{array}\right) \tag{9.3}$$

$$\left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} A_T = A_0 \\ \hline\hline A_B = \left(\begin{array}{c} a_1^T \\ \hline A_2 \end{array}\right) \end{array}\right) \tag{9.4}$$

D contains	Comments
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B B_T^T + C_{BL} & C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline A_B B_T^T + C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline B_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline B_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B B_T^T + B_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline A_B B_T^T + B_B A_T^T + C_{BL} & C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B B_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline A_B B_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline B_B A_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline B_B A_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline A_B B_T^T + B_B A_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	
$\left(\begin{array}{c\|c} A_T B_T^T + B_T A_T^T + C_{TL} & \star \\ \hline A_B B_T^T + B_B A_T^T + C_{BL} & A_B B_B^T + B_B A_B^T + C_{BR} \end{array}\right)$	This condition can be rejected using 8a of the recipe.

Figure 9.1: Possible contents of D at an intermediate step.

Considering (9.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c||c} A_TB_T^T+B_TA_T^T+C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) = \left(\begin{array}{c||c} A_0B_0^T+B_0A_0^T+C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline C_{20}\end{array}\right) & \left(\begin{array}{c|c}\gamma_{11} & \star \\ \hline c_{21} & C_{22}\end{array}\right)\end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL}=\left(\begin{array}{c|c} D_{00} & \star \\ \hline d_{10}^T & \delta_{11}\end{array}\right) & \star \\ \hline\hline D_{BL}=\left(\begin{array}{c|c} D_{20} & d_{21}\end{array}\right) & D_{BR}=D_{22}\end{array}\right), \left(\begin{array}{c||c} C_{TL}=\left(\begin{array}{c|c} C_{00} & \star \\ \hline c_{10}^T & \gamma_{11}\end{array}\right) & \star \\ \hline\hline C_{BL}=\left(\begin{array}{c|c} C_{20} & c_{21}\end{array}\right) & C_{BR}=C_{22}\end{array}\right) \tag{9.5}$$

$$\left(\begin{array}{c} B_T=\left(\begin{array}{c} B_0 \\ \hline b_1^T\end{array}\right) \\ \hline\hline B_B=B_2\end{array}\right), \left(\begin{array}{c} A_T=\left(\begin{array}{c} A_0 \\ \hline a_1^T\end{array}\right) \\ \hline\hline A_B=A_2\end{array}\right) \tag{9.6}$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c||c} A_0B_0^T+B_0A_0^T+C_{00} & \star \\ \hline\hline \left(\begin{array}{c} c_{10}^T \\ \hline D_{20}\end{array}\right) & \left(\begin{array}{c|c}\gamma_{11} & \star \\ \hline c_{21} & C_{22}\end{array}\right)\end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} A_0B_0^T+B_0A_0^T+C_{00} & \star \\ \hline a_1^TB_0^T+b_1^TA_0^T+c_{10}^T & a_1^Tb_1+b_1^Ta_1+\gamma_{11}\end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21}\end{array}\right) & C_{22}\end{array}\right)$$

Thus we conclude that an algorithm that maintains the condition in (9.2) is given in Fig. 9.2 (left). Notice that the algorithm overwrites matrix C with the result AB^T+BA^T+C.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c||c|c} D_{00} & \star & \star \\ \hline\hline D_{10} & D_{11} & \star \\ \hline D_{20} & D_{21} & D_{22}\end{array}\right)$$

where D_{11} is $b\times b$. Similarly, we repartition

$$\left(\begin{array}{c} A_T \\ \hline\hline A_B\end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1 \\ \hline A_2\end{array}\right), \quad\text{and}\quad \left(\begin{array}{c} B_T \\ \hline\hline B_B\end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2\end{array}\right), \quad\text{and}\quad \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22}\end{array}\right)$$

where A_1 and B_1 have b rows and C_{11} is a $b\times b$ matrix. Again, the double lines have meaning:

$$\left(\begin{array}{c||c} D_{TL}=D_{00} & \star \\ \hline\hline D_{BL}=\left(\begin{array}{c} D_{10}^T \\ \hline D_{20}\end{array}\right) & D_{BR}=\left(\begin{array}{c|c} D_{11} & \star \\ \hline D_{21} & D_{22}\end{array}\right)\end{array}\right), \left(\begin{array}{c||c} C_{TL}=C_{00} & \star \\ \hline\hline C_{BL}=\left(\begin{array}{c} C_{10} \\ \hline C_{20}\end{array}\right) & C_{BR}=\left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22}\end{array}\right)\end{array}\right)$$

$$\left(\begin{array}{c} B_T=B_0 \\ \hline\hline B_B=\left(\begin{array}{c} B_1 \\ \hline B_2\end{array}\right)\end{array}\right) \quad\text{and}\quad \left(\begin{array}{c} A_T=A_0 \\ \hline\hline A_B=\left(\begin{array}{c} A_1 \\ \hline A_2\end{array}\right)\end{array}\right)$$

We conclude that D currently contains

$$\left(\begin{array}{c||c} A_TB_T^T+B_TA_T^T+C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) = \left(\begin{array}{c||c} A_0B_0^T+B_0A_0^T+C_{00} & \star \\ \hline\hline \left(\begin{array}{c} C_{10} \\ \hline C_{20}\end{array}\right) & \left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22}\end{array}\right)\end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL}=\left(\begin{array}{c|c} D_{00} & \star \\ \hline D_{10} & D_{11}\end{array}\right) & \star \\ \hline\hline D_{BL}=\left(\begin{array}{c|c} D_{20} & D_{21}\end{array}\right) & D_{BR}=D_{22}\end{array}\right), \left(\begin{array}{c||c} C_{TL}=\left(\begin{array}{c|c} C_{00} & \star \\ \hline C_{10} & C_{11}\end{array}\right) & \star \\ \hline\hline C_{BL}=\left(\begin{array}{c|c} C_{20} & C_{21}\end{array}\right) & C_{BR}=C_{22}\end{array}\right)$$

Algorithm 22 $C \leftarrow (AB^T + BA^T) + C$
(unblocked row-lazy w.r.t. C)

partition
$C \rightarrow \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$
where C_{BR} **is** 0×0
partition
$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where A_B, B_B **are** 0×0
do until A_T, B_T **have** 0 **rows**

repartition
$$\left(\begin{array}{c||c} C_{TL} & 0 \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & 0 & 0 \\ \hline\hline c_{10}^T & \gamma_{11} & 0 \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right)$$
where γ_{11}^T **is a scalar**
repartition
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline a_1^T \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1 \\ \hline B_2 \end{array}\right)$$
where a_1, b_1 **have 1 row**

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

---

continue with
$$\left(\begin{array}{c||c} C_{TL} & 0 \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & 0 & 0 \\ \hline c_{10}^T & \gamma_{11} & 0 \\ \hline\hline C_{20} & c_{21} & C_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline\hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$$
enddo

Algorithm 23 $C \leftarrow (AB^T + BA^T) + C$
(blocked row-lazy w.r.t. C)

partition
$C \rightarrow \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$
where C_{BR} **is** 0×0
partition
$A \rightarrow \left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where A_B, B_B **are** 0×0
do until A_T, B_T **have** 0 **rows**
determine block size c
repartition
$$\left(\begin{array}{c||c} C_{TL} & 0 \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & 0 & 0 \\ \hline\hline C_{10}^T & C_{11} & 0 \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$
where C_{11}^T **is a matrix**
repartition
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline\hline A_1^T \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1^T \\ \hline B_2 \end{array}\right)$$
where A_1^T, B_1^T **have** c **rows**

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

---

continue with
$$\left(\begin{array}{c||c} C_{TL} & 0 \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & 0 & 0 \\ \hline C_{10}^T & C_{11} & 0 \\ \hline\hline C_{20} & c_{21} & C_{22} \end{array}\right)$$
continue with
$$\left(\begin{array}{c} A_T \\ \hline\hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1^T \\ \hline\hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1^T \\ \hline\hline B_2 \end{array}\right)$$
enddo

Figure 9.2: Unblocked (left) and blocked (right) row-lazy triangular matrix-matrix algorithms.

$$\left(\frac{B_T = \left(\frac{B_0}{B_1}\right)}{B_B = B_2}\right), \left(\frac{A_T = \left(\frac{A_0}{A_1}\right)}{A_B = A_2}\right)$$

Thus the contents of D must change be updated like

$$\left(\begin{array}{c||c} A_0 B_0^T + B_0 A_0^T + C_{00} & \star \\ \hline\hline \left(\begin{array}{c} C_{10} \\ \hline C_{20} \end{array}\right) & \left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22} \end{array}\right) \end{array}\right)$$

$$\rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} A_0 B_0^T + B_0 A_0^T + C_{00} & \star \\ \hline A_1 B_0^T + B_1 A_0^T + C_{10} & B_1 A_1^T + A_1 B_1^T + C_{11} \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21} \end{array}\right) & C_{22} \end{array}\right)$$

We conclude that an algorithm that maintains the condition in (9.2) is given in Fig. 9.2 (right).

9.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/syr2k_ln/`

9.3 Performance

```
#include "FLAME.h"

void Syr2k_ln_downright_wrt_C_unb( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
  FLA_Obj     CTL, CTR,     C00, C01, C02,    AT,           A0,    BT,           B0,
              CBL, CBR,     C10, C11, C12,    AB,           A1,    BB,           B1,
                            C20, C21, C22,                  A2,                  B2;

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                    /* ************** */
                    &CBL, /**/ &CBR,
                    0, 0,     /* submatrix */ FLA_TL );
  FLA_Part_2x1( A,  &AT,
                   /***/
                    &AB,
                0, /* length submatrix */ FLA_TOP );
  FLA_Part_2x1( B,  &BT,
                   /***/
                    &BB,
                0, /* length submatrix */ FLA_TOP );

  while ( FLA_Obj_length( CBR ) ){
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00, /**/ &C01, &C02,
                           /* ************* */    /* ******************** */
                                /**/              &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,         &C20, /**/ &C21, &C22,
                           1, 1, /* C11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,                    &A0,
                          /**/                    /**/
                                                  &A1,
                           AB,                    &A2,
                           1, /* length A1 from */ FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( BT,                    &B0,
                          /**/                    /**/
                                                  &B1,
                           BB,                    &B2,
                           1, /* length B1 from */ FLA_BOTTOM );

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

    FLA_Gemv( FLA_NO_TRANSPOSE, ONE, B0, A1, ONE, C10 );
    FLA_Gemv( FLA_NO_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( &CTL, /**/ &CTR,         C00, C01, /**/ C02,
                                    /**/               C10, C11, /**/ C12,
                            /* ************** */    /* ****************** */
                              &CBL, /**/ &CBR,         C20, C21, /**/ C22,
                              /* C11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                     A0,
                                                       A1,
                             /***/                    /**/
                              &AB,                     A2,
                              /* A1  added to */ FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                              &BB,                     B2,
                              /* B1  added to */ FLA_TOP );
  }
}
```

Figure 9.3: Unblocked down-right moving (w.r.t. C) symmetric rank-2k update algorithm using FLAME.

```
#include "FLAME.h"

void Syr2k_ln_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,    AT,           A0,    BT,           B0,
              CBL, CBR,     C10, C11, C12,    AB,           A1,    BB,           B1,
                            C20, C21, C22,                  A2,                  B2;
  int         b;

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                  /* ************** */
                    &CBL, /**/ &CBR,
                    0, 0,     /* submatrix */ FLA_TL );
  FLA_Part_2x1( A,  &AT,
                   /***/
                    &AB,
                0, /* length submatrix */ FLA_TOP );
  FLA_Part_2x1( B,  &BT,
                   /***/
                    &BB,
                0, /* length submatrix */ FLA_TOP );

  while ( b = min( FLA_Obj_length( CBR ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00, /**/ &C01, &C02,
                        /* ************* */    /* ********************* */
                                /**/              &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,         &C20, /**/ &C21, &C22,
                           b, b, /* C11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,                    &A0,
                          /**/                    /**/
                                                  &A1,
                           AB,                    &A2,
                           b, /* length A1 from */ FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( BT,                    &B0,
                          /**/                    /**/
                                                  &B1,
                           BB,                    &B2,
                           b, /* length B1 from */ FLA_BOTTOM );
    /* ********************************************************************************************************* */
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A2, B1, ONE, C21 );
    if ( rec == FLA_RECURSIVE && b > 8 )
      Syr2k_ln_downright_wrt_C_blk( rec, A1, B1, C11, nb_alg/2 );
    else
      Syr2k_ln_downright_wrt_C_unb( A1, B1, C11 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, B2, A1, ONE,  C21 );
    /* ********************************************************************************************************* */
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,         C00, C01, /**/ C02,
                                    /**/               C10, C11, /**/ C12,
                            /* ************** */    /* ****************** */
                              &CBL, /**/ &CBR,         C20, C21, /**/ C22,
                              /* C11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                     A0,
                                                       A1,
                             /***/                    /**/
                              &AB,                     A2,
                              /* A1  added to */ FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                              &BB,                     B2,
                              /* B1  added to */ FLA_TOP );
  }
}
```

Figure 9.4: Blocked down-right moving (w.r.t. C) symmetric rank-2k update algorithm using FLAME.

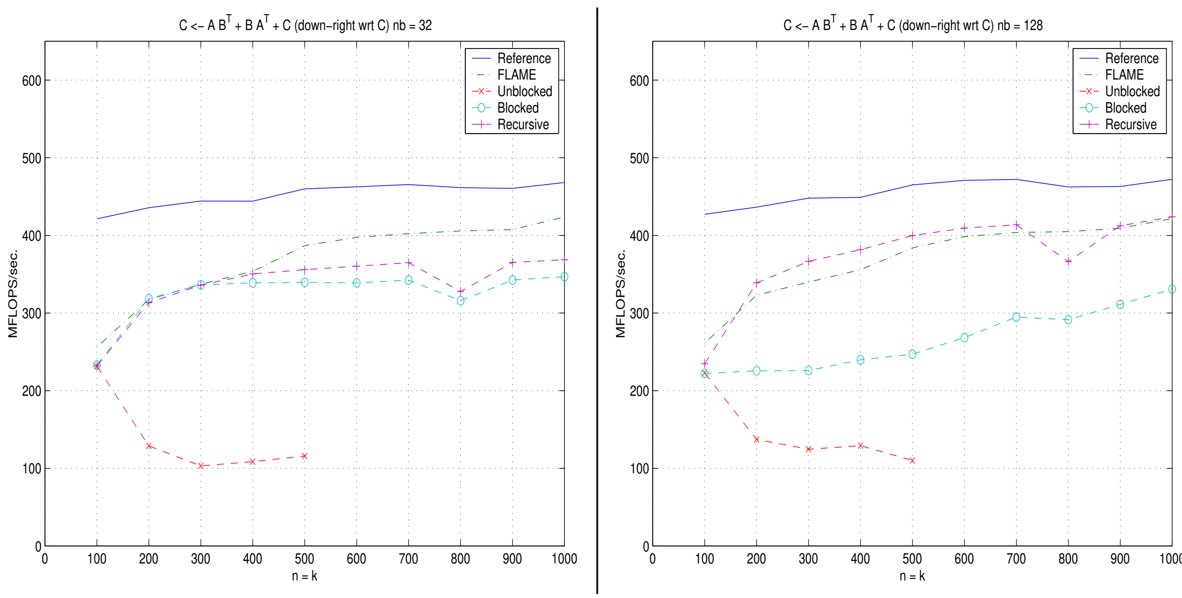


Figure 9.5: Performance of a down-right moving (w.r.t. C) symmetric rank-k update algorithm for a block size of 32 (left) and 128 (right). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

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 matrixmatrix 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

$$D = A^T B + B^T A + C \tag{10.1}$$

keeping in mind that D will overwrite C.

10.1 Algorithms that start by splitting C

Partition

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{BL}^T \\ \hline C_{BL} & C_{BR} \end{array} \right)$$

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

$$A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array} \right), \quad B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right) \quad \text{and} \quad D \rightarrow \left(\begin{array}{c|c} D_{TL} & D_{BL}^T \\ \hline D_{BL} & D_{BR} \end{array} \right)$$

where A_T and B_T have k rows.

Now. (10.1) becomes

$$\left(\begin{array}{c|c} D_{TL} & D_{BL}^T \\ \hline D_{BL} & D_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_L & A_R \end{array} \right)^T \left(\begin{array}{c|c} B_L & B_R \end{array} \right) + \left(\begin{array}{c|c} B_L & B_R \end{array} \right)^T \left(\begin{array}{c|c} A_L & A_R \end{array} \right) + \left(\begin{array}{c|c} C_{TL} & C_{BL}^T \\ \hline C_{BL} & C_{BR} \end{array} \right)$$

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.)

$$\left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) = \left(\begin{array}{c||c} A_L^T B_L + B_L^T A_L + C_{TL} & D_{BL}^T \\ \hline\hline A_R^T B_L + B_R^T B_L + C_{BL} & A_R^T B_R + B_R^T A_R + C_{BR} \end{array}\right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_{TL} & = & A_L^T B_L + B_L^T A_L + C_{TL} \\ D_{BL} & = & A_R^T B_L + B_R^T B_L + C_{BL} \\ D_{BR} & = & A_R^T B_R + B_R^T A_R + C_{BR} \end{array}$$

which must hold.

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

D contains	Comments
$\left(\begin{array}{c\|\|c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c\|\|c} D_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$	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.
$\left(\begin{array}{c\|\|c} D_{TL} & \star \\ \hline\hline D_{BL} & C_{BR} \end{array}\right)$	This one is a viable option. This time as progress proceeds from the left to right you again use then write over elements in C that you will not need later in computations.
$\left(\begin{array}{c\|\|c} D_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(\begin{array}{c\|\|c} C_{TL} & \star \\ \hline\hline C_{BL} & D_{BR} \end{array}\right)$	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.
$\left(\begin{array}{c\|\|c} C_{TL} & \star \\ \hline\hline D_{BL} & D_{BR} \end{array}\right)$	This one is a viable option.
$\left(\begin{array}{c\|\|c} D_{TL} & \star \\ \hline\hline C_{BL} & D_{BR} \end{array}\right)$	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

$$D \quad \text{contains} \quad \left(\begin{array}{c||c} D_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \tag{10.2}$$

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

$$\left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c||c|c} D_{00} & \star & \star \\ \hline\hline d_{10}^T & \delta_{11} & \star \\ \hline D_{20} & d_{21} & D_{22}\end{array}\right)$$

where d_{10}^T is a row and δ_{11} is a scalar. Similarly, we repartition A, B,

$$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_0 & a_1 & A_2 \end{array}\right), \quad \left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} B_0 & b_1 & B_2 \end{array}\right)$$

where a_1 and b_1 are columns. C

$$\left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22}\end{array}\right)$$

where c_{10}^T is a row and γ_{11} is a scalar. Notice the double lines have meaning:

$$\left(\begin{array}{c||c} D_{TL} = D_{00} & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c} d_{10}^T \\ \hline D_{20}\end{array}\right) & D_{BR} = \left(\begin{array}{c|c} \delta_{11} & \star \\ \hline d_{21} & D_{22}\end{array}\right)\end{array}\right), \left(\begin{array}{c||c} A_L = A_0 & A_R = \left(\begin{array}{c|c} a_1 & A_2\end{array}\right)\end{array}\right)$$

$$\left(\begin{array}{c||c} B_L = B_0 & B_R = \left(\begin{array}{c|c} b_1 & B_2\end{array}\right)\end{array}\right) \text{ and } \left(\begin{array}{c||c} C_{TL} = C_{00} & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c} c_{10}^T \\ \hline C_{20}\end{array}\right) & C_{BR} = \left(\begin{array}{c|c} \gamma_{11} & \star \\ \hline c_{21} & C_{22}\end{array}\right)\end{array}\right)$$

Considering (10.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c||c} A_L^T B_L + B_L^T A_L + C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) = \left(\begin{array}{c||c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star & \star \\ \hline\hline C_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & C_{21} & C_{22}\end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c|c||c} D_{00} & \star & \star \\ \hline d_{10}^T & \delta_{11} & \star \\ \hline\hline D_{20} & d_{21} & D_{22}\end{array}\right), \left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_0 & a_1 & A_2 \end{array}\right),$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} B_0 & b_1 & B_2 \end{array}\right) \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR}\end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline c_{10}^T & \gamma_{11} & \star \\ \hline\hline C_{20} & c_{21} & C_{22}\end{array}\right)$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline c_{10}^T & \gamma_{11}\end{array}\right) + \left(\begin{array}{c|c} A_0 & a_1\end{array}\right)^T \left(\begin{array}{c|c} B_0 & b_1\end{array}\right) + \left(\begin{array}{c|c} B_0 & b_1\end{array}\right)^T \left(\begin{array}{c|c} A_0 & a_1\end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21}\end{array}\right) & C_{22}\end{array}\right) \rightarrow$$

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star \\ \hline c_{10}^T + a_1^T B_0 + b_1^T A_0 & \gamma_{11} + a_1^T b_1 + b_1^T a_1\end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & c_{21}\end{array}\right) & C_{22}\end{array}\right)$$

We conclude that a 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 $A^T B + B^T A + C$.

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

partition

$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

where C_{TL} **is** 0×0**.**

partition

$A \rightarrow \left(\begin{array}{c||c} A_L & A_R \end{array}\right)$

where A_L **has 0 colums.**

partition

$B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array}\right)$

where B_L **has 0 colums.**

do until C_{BR} **is** 0×0

repartition

$$\left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline c_{10}^T & \gamma_{11} & \star \\ \hline C_{20} & c_{21} & C_{22} \end{array}\right)$$

where γ_{11} **is a scalar**

repartition

$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_0 & a_1 & A_2 \end{array}\right)$

where a_1 **is a column**

repartition

$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} B_0 & b_1 & B_2 \end{array}\right)$

where b_1 **is a column**

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

---

continue with

$$\left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline c_{10}^T & \gamma_{11} & \star \\ \hline\hline C_{20} & c_{21} & C_{22} \end{array}\right)$$

continue with

$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_0 & a_1 & A_2 \end{array}\right)$

continue with

$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} B_0 & b_1 & B_2 \end{array}\right)$

enddo

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

partition

$$C \rightarrow \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right)$$

where C_{TL} **is** 0×0**.**

partition

$A \rightarrow \left(\begin{array}{c||c} A_L & A_R \end{array}\right)$

where A_L **has 0 colums.**

partition

$B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array}\right)$

where B_L **has 0 colums.**

do until C_{BR} **is** 0×0

repartition

$$\left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

where C_{11} **is a matrix** $n \times n$**.**

repartition

$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_0 & A_1 & A_2 \end{array}\right)$

where A_1 **is a** $m \times n$ **matrix.**

repartition

$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array}\right)$

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

$$\left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

continue with

$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_0 & A_1 & A_2 \end{array}\right)$

continue with

$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} B_0 & B_1 & B_2 \end{array}\right)$

enddo

Figure 10.2: Unblocked (left) and blocked (right) down right lazy syr2k update algorithms.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} D_{00} & \star & \star \\ \hline\hline D_{10} & D_{11} & \star \\ \hline D_{20} & D_{21} & D_{22} \end{array}\right)$$

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

$$\left(\begin{array}{c||c} C_{TL} & C_{BL}^T \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_{00} & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

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

(10.3) $$\left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_0 & A_1 & A_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where A_1 and B_1 are $b \times n$ matrixcies. Again, the double lines have meaning:

(10.4) $$\left(\begin{array}{c||c} D_{TL} = D_{00} & \star \\ \hline\hline D_{BL} = \left(\begin{array}{c} D_{10} \\ \hline D_{20} \end{array}\right) & D_{BR} = \left(\begin{array}{c|c} D_{11} & \star \\ \hline D_{21} & D_{22} \end{array}\right) \end{array}\right), \left(\begin{array}{c||c} A_L = A_0 & A_R = \left(\begin{array}{c|c} A_1 & A_2 \end{array}\right) \end{array}\right)$$

$$\left(\begin{array}{c||c} B_L = B_0 & B_R = \left(\begin{array}{c|c} B_1 & B_2 \end{array}\right) \end{array}\right) \text{ and } \left(\begin{array}{c||c} C_{TL} = C_{00} & \star \\ \hline\hline C_{BL} = \left(\begin{array}{c} C_{10} \\ \hline C_{20} \end{array}\right) & C_{BR} = \left(\begin{array}{c|c} C_{11} & \star \\ \hline C_{21} & C_{22} \end{array}\right) \end{array}\right)$$

Considering (10.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c||c} A_L^T B_L + B_L^T A_L + C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

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

$$\left(\begin{array}{c||c} D_{TL} & D_{BL}^T \\ \hline\hline D_{BL} & D_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} D_{00} & \star & \star \\ \hline D_{10} & D_{11} & \star \\ \hline\hline D_{20} & D_{21} & D_{22} \end{array}\right), \left(\begin{array}{c||c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_0 & A_1 & A_2 \end{array}\right),$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} B_0 & B_1 & B_2 \end{array}\right) \left(\begin{array}{c||c} C_{TL} & \star \\ \hline\hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline\hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

Thus the contents of D must be updated to contain

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} & \star \\ \hline C_{10} & C_{11} \end{array}\right) + \left(\begin{array}{c|c} A_0 & A_1 \end{array}\right)^T \left(\begin{array}{c|c} B_0 & B_1 \end{array}\right) + \left(\begin{array}{c|c} B_0 & B_1 \end{array}\right)^T \left(\begin{array}{c|c} A_0 & A_1 \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21} \end{array}\right) & C_{22} \end{array}\right)$$

$$= \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star \\ \hline C_{10} + A_1^T B_0 + B_1^T A_0 & C_{11} + A_1^T B_1 + B_1^T A_1 \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21} \end{array}\right) & C_{22} \end{array}\right)$$

Hence the contents of D must be updated like

$$\left(\begin{array}{c||c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star & \star \\ \hline\hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} C_{00} + A_0^T B_0 + B_0^T A_0 & \star \\ \hline C_{10} + A_1^T B_0 + B_1^T A_0 & C_{11} + A_1^T B_1 + B_1^T A_1 \end{array}\right) & \star \\ \hline\hline \left(\begin{array}{c|c} C_{20} & C_{21} \end{array}\right) & C_{22} \end{array}\right)$$

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

```
#include "FLAME.h"

int Syr2k_lt_downright_wrt_C_unb ( FLA_Obj A, FLA_Obj B, FLA_Obj C )
{
  FLA_Obj     CTL, CTR,     C00, c01, C02,    AL,           A0,    BL,           B0,
              CBL, CBR,     c10, c11, C12,    AR,           a1,    BR,           b1,
                            C20, C21, C22,                  A2,                  B2;

  /* Partion C into a 2X2 */
  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                   /* ************** */
                     &CBL, /**/ &CBR,
                     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(CBR)!=0) {
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00, /**/ &c01, &C02,
                           /* ************* */    /* ********************* */
                                /**/              &c10, /**/ &c11, &C12,
                           CBL, /**/ CBR,         &C20, /**/ &C21, &C22,
                           1, 1, /* C11 from */ FLA_BR );

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

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  /**/ &b1,  &B2,
                           1, /* width  b1 from */ FLA_RIGHT );

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

    FLA_Gemv( FLA_TRANSPOSE, ONE, B0, 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( &CTL, /**/ &CTR,         C00, c01, /**/ C02,
                                    /**/               c10, c11, /**/ C12,
                              /* ************** */   /* ****************** */
                              &CBL, /**/ &CBR,         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 10.3: 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,    AL,           A0,    BL,           B0,
              CBL, CBR,     C10, C11, C12,    AR,           A1,    BR,           B1,
                            C20, C21, C22,                  A2,                  B2;

  int         b;

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                   /* ************** */
                     &CBL, /**/ &CBR,
                     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(CBR) != 0 ) {
    b = min( FLA_Obj_length(CBR), nb_alg );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,         &C00, /**/ &C01, &C02,
                        /* ************* */    /* ******************** */
                                 /**/             &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,         &C20, /**/ &C21, &C22,
                           b, b, /* C11 from */ FLA_BR );

    FLA_Repart_1x2_to_1x3( AL,  /**/ AR,          &A0,  /**/ &A1,  &A2,
                           b, /* width  a1 from */ FLA_RIGHT );

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  /**/ &B1,  &B2,
                           b, /* width  b1 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_RECURSIVE && 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,
                            /* ************** */    /* ****************** */
                              &CBL, /**/ &CBR,         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.

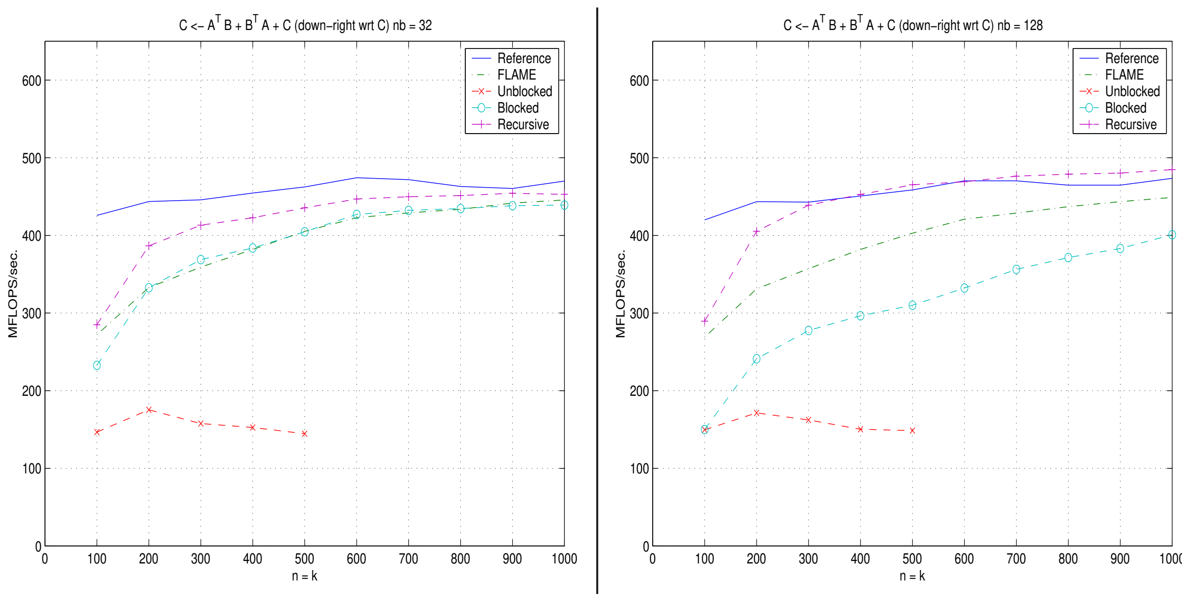


Figure 10.5: Performance of a down-right moving (w.r.t. C) symmetric rank-k update algorithm for a block size of 32 (left) and 128 (right). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Chapter 11

Triangular Matrix Multiplication $B \leftarrow L^T B$

by

Nickou Oskouipour

Jason S. Stirman

In this chapter, we discuss the implementation of triangular transposed matrix-matrix multiplication:

$$B \leftarrow L^T B$$

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. Finally, we discuss a parallel implementation with PLAPACK.

Let us consider

$$D = L^T B \tag{11.1}$$

keeping in mind that D will overwrite B.

11.1 Algorithms that start by splitting L

Partition

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

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

$$D \rightarrow \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

where D_T and B_T have k rows.

Now, (11.1) becomes

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array} \right) \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c} L_{TL}^T B_T + L_{BL}^T B_B \\ \hline\hline L_{BR}^T B_B \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_T &=& L_{TL}^T B_T + L_{BL}^T B_B \\ D_B &=& L_{BR}^T B_B \end{array}$$

which must hold.

Possible contents of D at this intermediate step are given in Fig. 11.1. Considering the comments in that

D contains	Comments
$\left(\frac{\star}{\star}\right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\frac{L_{BL}^T B_B}{\star}\right)$	This one is not a viable option since if D_T overwrites B_T, unless we provide a temporary in which to store this intermediate result, the original B_T is lost.
$\left(\frac{L_{TL}^T B_T}{L_{BR}^T B_B}\right)$	This one is not a viable option since if D_B overwrites B_B, unless we provide a temporary in which to store this intermediate result, the original B_B is lost.
$\left(\frac{L_{TL}^T B_T + L_{BL}^T B_B}{L_{BR}^T B_B}\right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(\frac{L_{BL}^T B_B}{L_{BR}^T B_B}\right)$	This one is not a viable option since if D_T overwrites B_T, unless we provide a temporary in which to store this intermediate result, the original B_T is lost.
$\left(\frac{L_{TL}^T B_T + L_{BL}^T B_B}{\star}\right)$	
$\left(\frac{L_{TL}^T B_T}{\star}\right)$	

Figure 11.1: Possible contents of D at an intermediate step.

table, only two viable conditions are left, the ones for which there are no comments.

11.1.1 Column-lazy algorithm (relative to L)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\frac{L_{TL}^T B_T + L_{BL}^T B_B}{\star}\right) \tag{11.2}$$

Notice that this indicates that L_{TL}^T and L_{BL}^T have been used to update D, which will 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 down. Thus, this algorithm naturally moves through matrices D and B in the "down" direction.

Unblocked algorithm

Repartition

$$\left(\frac{D_T}{D_B}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline\hline d_1^T \\ \hline D_2 \end{array}\right)$$

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

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

where b_1^T is a row and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2\end{array}\right)\end{array}\right) \left(\begin{array}{c} B_T=B_0 \\ \hline\hline B_B = \left(\begin{array}{c} b_1^T\\ \hline B_2\end{array}\right)\end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL}=L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20}\end{array}\right) & L_{BR} = \left(\begin{array}{c|c}\lambda_{11} & 0 \\ \hline l_{21} & L_{22}\end{array}\right)\end{array}\right) \tag{11.3}$$

Considering (11.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} L_{TL}^T B_T + L_{BL}^T B_B \\ \hline\hline \star \end{array}\right) = \left(\begin{array}{c} L_{00}^T B_0 + \left(\begin{array}{c} l_{10}^T \\ \hline L_{20}\end{array}\right)^T \left(\begin{array}{c} b_1^T \\ \hline B_2\end{array}\right) \\ \hline\hline \star \end{array}\right) = \left(\begin{array}{c} L_{00}^T B_0 + l_{10} b_1^T + L_{20}^T B_2 \\ \hline\hline \star \end{array}\right) \tag{11.4}$$

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

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T\end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T\end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right), \text{ and } \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00}^T & 0 \\ \hline l_{10}^T & \lambda_{11}\end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & l_{21}\end{array}\right) & L_{BR} = L_{22}\end{array}\right) \tag{11.5}$$

Thus the contents of D must change be updated like

$$\left(\begin{array}{c} L_{TL}^T B_T + L_{BL}^T B_B \\ \hline\hline \left(\begin{array}{c} \star \\ \hline \star \end{array}\right)\end{array}\right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11}\end{array}\right) B_0 + \left(\begin{array}{c} L_{20} \\ \hline l_{21}\end{array}\right)^T \left(\begin{array}{c} b_1^T \\ \hline B_2\end{array}\right) \\ \hline\hline \star \end{array}\right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} L_{00} B0 + l_{10} b_1^T + L_{20}^T B_2 \\ \hline \lambda_{11} b_1^T + l_{21}^T B_2 \end{array}\right) \\ \hline\hline \star \end{array}\right) \tag{11.6}$$

Thus we conclude that an algorithm that maintains the condition in (11.2) is given in Fig. 11.2 (left). Notice that the algorithm overwrites matrix B with the result LB.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline\hline D_1 \\ \hline D_2 \end{array}\right)$$

where D_1 is a block of b rows. Similarly, we repartition

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

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

Again, the double lines have meaning:

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} D_1 \\ \hline D_2\end{array}\right)\end{array}\right), \left(\begin{array}{c} B_T=B_0 \\ \hline\hline B_B = \left(\begin{array}{c} B_1\\ \hline B_2\end{array}\right)\end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL}=L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} L_{10} \\ \hline L_{20}\end{array}\right) & L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22}\end{array}\right)\end{array}\right)$$

Considering (11.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} L_{TL}^T B_T + L_{BL}^T B_B \\ \hline\hline \star \end{array}\right) = \left(\begin{array}{c} L_{00}^T B_0 + \left(\begin{array}{c} L_{10} \\ \hline L_{20}\end{array}\right)^T \left(\begin{array}{c} B_1 \\ \hline B_2\end{array}\right) \\ \hline\hline \star \end{array}\right) = \left(\begin{array}{c} L_{00}^T B_0 + L_{10}^T B_1 + L_{20}^T B_2 \\ \hline\hline \star \end{array}\right)$$

Algorithm 26 $B \leftarrow LB$
(unblocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where B_T **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0

repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$
where b_1^T **is a row**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$
where λ_{11} **is a scalar**

$b_1^T \leftarrow \lambda_{11} b_1^T$
$b_1^T \leftarrow l_{21}^T B_2 + b_1^T$

---

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

Algorithm 27 $B \leftarrow LB$
(blocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
where B_T **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0
determine block size b
repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$
where B_1 **has** b **rows**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
where L_{11} **is** $b \times b$

$B_1 \leftarrow L_{11}^T B_1$
$B_1 \leftarrow L_{21}^T B_2 + B_1$

---

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

Figure 11.2: Unblocked (left) and blocked (right) row-lazy triangular matrix-matrix multiplication algorithms.

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

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right) \\ \hline\hline D_B = D_0 \end{array} \right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) \\ \hline\hline B_B = B_2 \end{array} \right), \text{ and } \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right) & L_{BR} = L_{22} \end{array} \right)$$

Thus the contents of D must change be updated like

$$\left(\begin{array}{c} L_{00}^T B_0 + L_{20}^T B_1 + L_{21}^T B_2 \\ \hline\hline \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \end{array} \right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} L_{00}^T B_0 + L_{10}^T B_1 + L_{20}^T B_2 \\ \hline L_{21}^T B_2 + L_{11}^T B_1 \end{array}\right) \\ \hline\hline \star \end{array} \right)$$

We conclude that an algorithm that maintains the condition in (11.2) is given in Fig. 11.2 (right). Notice that the algorithm overwrites matrix B with the result LB.

11.1.2 Lazy algorithm (relative to L)

Next, consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c} L_{TL}^T B_T \\ \hline\hline \star \end{array} \right) \tag{11.7}$$

Notice that this indicates that only L_{TL}^T 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 "down" direction.

Unblocked algorithm

Repartition

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline\hline d_1^T \\ \hline D_2 \end{array} \right)$$

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

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

where b_1^T is a row and λ_{11} is a scalar.

Again, the double lines have the same meaning as illustrated in (11.3). Considering (11.7) and these repartitionings, D currently contains

$$\left(\begin{array}{c} L_{TL}^T B_T \\ \hline\hline \star \end{array} \right) = \left(\begin{array}{c} L_{00}^T B_0 \\ \hline\hline \star \end{array} \right)$$

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the meaning in (11.1.1) and D must be updated to contain

$$\left(\begin{array}{c} L_{TL}^T B_T \\ \hline\hline \star \end{array} \right) = \left(\begin{array}{c} \left(\begin{array}{c|c} L_{00}^T & l_{10}^T \\ \hline 0 & \lambda_{11} \end{array}\right) \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline\hline \star \end{array} \right) = \left(\begin{array}{c} \left(\begin{array}{c} L_{00}^T B_0 + l_{10}^T b_1^T \\ \hline \lambda_{11} \end{array}\right) \\ \hline\hline \star \end{array} \right)$$

Thus the contents of D must change be updated like

$$\left(\begin{array}{c} L_{00}^T B_0 \\ \hline\hline \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) \end{array} \right) \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} L_{00}^T B_0 + l_{10}^T b_1^T \\ \hline \lambda_{11} b_1^T \end{array}\right) \\ \hline\hline \star \end{array} \right)$$

Thus we conclude that an algorithm that maintains the condition in (11.7) is given in Fig. 11.3 (left). Again, the algorithm overwrites matrix B with the result LB.

Algorithm 28 $B \leftarrow LB$
(unblocked lazy w.r.t. L)

partition

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

where B_T **has** 0 **rows**

partition

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

where L_{TL} **is** 0×0

do until L_{BR} **is** 0×0

repartition

$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right)$

where b_1^T **is a row**

repartition

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

where λ_{11} **is a scalar**

$B_0 \leftarrow l_{10}^T b_1^T + B_0$
$b_1^T \leftarrow \lambda_{11} b_1^T$

continue with

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

continue with

$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$

enddo

Algorithm 29 $B \leftarrow LB$
(blocked lazy w.r.t. L)

partition

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

where B_T **has** 0 **rows**

partition

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

where L_{TL} **is** 0×0

do until L_{BR} **is** 0×0

determine block size b

repartition

$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$

where B_1 **has** b **rows**

repartition

$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$

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

$B_0 \leftarrow L_{10}^T B_1 + B_0$
$B_1 \leftarrow L_{11}^T B_1$

continue with

$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$

continue with

$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$

enddo

Figure 11.3: Unblocked (left) and blocked (right) lazy triangular matrix-matrix multiplication algorithms.

Blocked algorithm

We leave the derivation of the lazy unblocked algorithm, given in Fig. 11.3 (right), as an exercise for the reader.

11.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trmm_llt/`

11.3 Performance

```
#include "FLAME.h"

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

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
                 0, 0,     /* submatrix */ FLA_TL );

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

  while ( FLA_Obj_length( ABR ) > 0 ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                         /* ************* */    /* ********************* */
                                 /**/             &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
                           1, 1, /* A11 from */ FLA_BR );

    FLA_Repart_2x1_to_3x1( BT,                    &B0,
                          /**/                    /**/
                                                  &B1,
                           BB,                    &B2,
                           1, /* length B1 from */ FLA_BOTTOM );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Ger( ONE, A10, B1, B0 );

    FLA_Scal( A11, B1);

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Gemv( FLA_TRANSPOSE, ONE, B2 , A21, ONE, B1);

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

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

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                              /***/                   /**/
                               &BB,                    B2,
                               /* B1  added to */ FLA_TOP );
  }
}
```

Figure 11.4: Unblocked column-lazy and lazy (w.r.t. L) triangular matrix-matrix multiplication algorithm using FLAME.

```
#include "FLAME.h"

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

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
                 0, 0,     /* submatrix */ FLA_TL );

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

  while ( b = min( FLA_Obj_length( ABR ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                        /* ************* */    /* ********************* */
                                /**/              &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
                           b, b, /* A11 from */ FLA_BR );

    FLA_Repart_2x1_to_3x1( BT,                    &B0,
                          /**/                    /**/
                                                  &B1,
                           BB,                    &B2,
                           b, /* length B1 from */ FLA_BOTTOM );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B1, ONE, B0 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      Trmm_left_lower_trans_blk( variant, rec, A11, B1, nb_alg/2 );
    else
      Trmm_left_lower_trans_unb( variant, A11, B1 );

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A21, B2, ONE, B1 );

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

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

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                   /**/
                              &BB,                     B2,
                              /* B1  added to */ FLA_TOP );
  }

  return FLA_SUCCESS;
}
```

Figure 11.5: Blocked column-lazy and lazy (w.r.t. L) triangular matrix-matrix multiplication algorithm using FLAME.

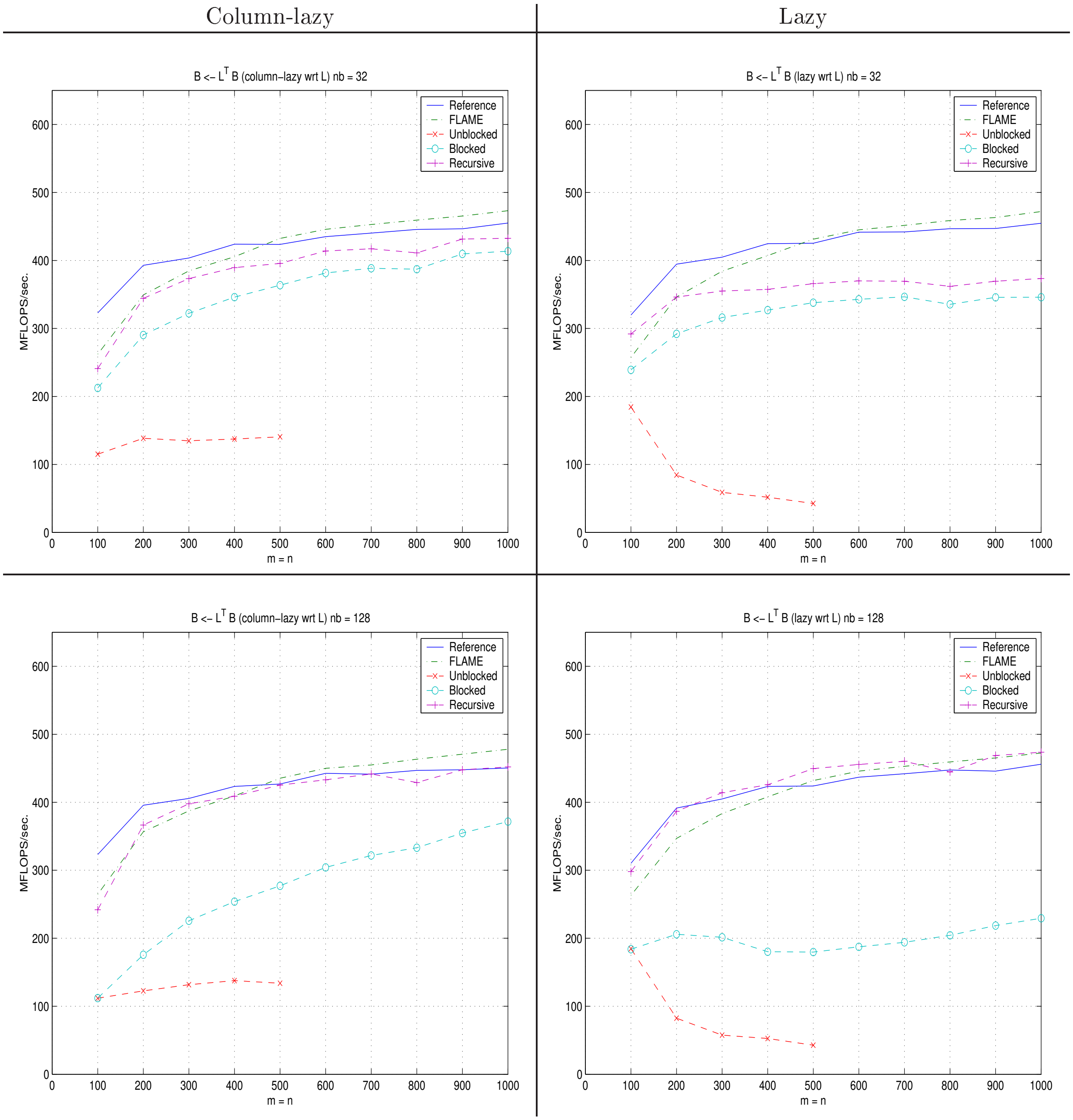


Figure 11.6: Performance of the Column-lazy (left) and lazy (right) triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom).

Chapter 12

Triangular Matrix Multiplication $B \leftarrow BL^T$

by

Deepak Giridharagopal

Sammuel A. Jarmon

Shivraj Ramanan

In this chapter, we discuss the implementation of triangular matrix multiplication:

$$B \leftarrow BL^T$$

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. Finally, we discuss a parallel implementation with PLAPACK.

Let us consider

$$D = BL^T \tag{12.1}$$

keeping in mind that D will overwrite B.

12.1 Algorithms that start by splitting L

Partition

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

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

$$D \rightarrow \left(\begin{array}{c||c} D_L & D_L \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

where D_L and B_L have k columns. Now, (12.1) becomes

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) = \left(\begin{array}{c||c} B_L & B_R \end{array} \right) \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)^T$$

Applying the transpose operation to L gives

$$\left(\begin{array}{c\|c} D_L & D_R \end{array} \right) = \left(\begin{array}{c\|c} B_L & B_R \end{array} \right) \left(\begin{array}{c\|c} L_{TL}{}^T & L_{BL}{}^T \\ \hline\hline 0 & L_{BR}{}^T \end{array} \right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c\|c} D_L & D_R \end{array} \right) = \left(\begin{array}{c\|c} B_L L_{TL}{}^T & B_L L_{BL}{}^T + B_R L_{BR}{}^T \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_L & = & B_L L_{TL}{}^T \\ D_R & = & B_L L_{BL}{}^T + B_R L_{BR}{}^T \end{array}$$

which must hold.

Possible contents of D at this intermediate step are given in Fig. 12.1. Considering the comments in that

D contains	Comments
$\left(\begin{array}{c} \star \\ \hline\hline \star \end{array} \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c\|c} B_L L_{TL}{}^T & \star \end{array} \right)$	This one is not a viable option since if D_L overwrites B_L, unless we provide a temporary in which to store this intermediate result, the original B_L is lost. The original B_L is needed in the calculation of D_R.
$\left(\begin{array}{c\|c} B_L L_{TL}{}^T & B_L L_{BL}{}^T \end{array} \right)$	This one is not a viable option since if D_R overwrites B_R, unless we provide a temporary in which to store this intermediate result, the original B_R is lost. The original B_R is needed in the calculation of D_R.
$\left(\begin{array}{c\|c} B_L L_{TL}{}^T & B_R L_{BR}{}^T \end{array} \right)$	This one is not a viable option since if D_L overwrites B_L, unless we provide a temporary in which to store this intermediate result, the original B_L is lost. The original B_L is needed in the calculation of D_R.
$\left(\begin{array}{c\|c} B_L L_{TL}{}^T & B_L L_{BL}{}^T + B_R L_{BR}{}^T \end{array} \right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(\begin{array}{c\|c} \star & B_L L_{BL}{}^T \end{array} \right)$	This one is not a viable option since if D_R overwrites B_R, unless we provide a temporary in which to store this intermediate result, the original B_R is lost. The original B_R is needed in the calculation of D_R.
$\left(\begin{array}{c\|c} \star & B_L L_{BL}{}^T + B_R L_{BR}{}^T \end{array} \right)$	
$\left(\begin{array}{c\|c} \star & B_R L_{BR}{}^T \end{array} \right)$	

Figure 12.1: Possible contents of D at an intermediate step.

table, only two viable conditions are left, the ones for which there are no comments.

12.1.1 Lazy algorithm (relative to L)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c\|c} \star & B_R L_{BR}{}^T \end{array} \right) \tag{12.2}$$

Notice that this indicates that L_{BR} has been used to update D, which will call a 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 left. Thus, this algorithm naturally moves through matrices D and B in the "left" direction.

Unblocked algorithm

Repartition

$$\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,|\, d_1 \,\|\, D_2 \right)$$

where d_1 is a column. Similarly, we repartition

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

where b_1 is a column and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(D_L = \left(D_0 \,|\, d_1 \right) \,\|\, D_R = D_2 \right), \left(B_L = \left(B_0 \,|\, b_1 \right) \,\|\, B_R = B_2 \right),$$

(12.3) $$\text{and} \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(L_{20} \,|\, l_{21} \right) & L_{BR} = L_{22} \end{array}\right)^T$$

Considering (12.2) and these repartitionings, D currently contains

$$\left(\star \,\|\, B_R {L_{BR}}^T \right) = \left(\star \,\|\, B_2 {L_{22}}^T \right)$$

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

$$\left(D_L = D_0 \,\|\, D_R = \left(d_1 \,|\, D_2 \right) \right), \left(B_L = B_0 \,\|\, B_R = \left(b_1 \,|\, B_2 \right) \right),$$

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

Thus the contents of D must change be updated like

$$\left(\star \,\|\, B_2 {L_{22}}^T \right) \rightarrow \left(\star \,\|\, \left(b_1\lambda_{11} \,\|\, b_1 {l_{21}}^T + B_2 {L_{22}}^T \right) \right)$$

Thus we conclude that an algorithm that maintains the condition in (12.2) is given in Fig. 12.2 (left). Notice that the algorithm overwrites matrix B with the result BL^T.

Theorem 3 (Cost of Algorithm 30) *Given that matrices B and L are $m \times n$ and $m \times m$, respectively, the unblocked lazy triangular matrix-matrix multiplication algorithm given in Fig. 12.2 (left) is $m^2 n$ floating point operations.*

Proof: Consider the algorithm in Fig. 12.2 (left). The number of columns in B_R increases from 0 to $m-1$ while L_{BR} increases from 0×0 to $(m-1) \times (m-1)$. Assuming B_R currently has k columns, and L_{BR} is thus currently $k \times k$, the different parts of the matrices have the following dimensions:

$$n\{ \; \overbrace{B_0}^{\bar m} \,\Big\|\, \overbrace{b_1}^{1} \,\Big|\, \overbrace{B_2}^{k} \qquad \begin{array}{c||c|c} \overbrace{L_{00}}^{\bar m} & \overbrace{0}^{1} & \overbrace{0}^{k} \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \begin{array}{l} \}\bar m = m-k-1 \\ \}1 \\ \}k \end{array}$$

$$\bar m = m-k-1$$

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

$$\begin{array}{ll} b_1 \leftarrow b_1\lambda_{11} & \qquad n \\ B_2 \leftarrow b_1 l_{21}^T + B_2 & \qquad 2kn \end{array}$$

Algorithm 30 $B \leftarrow BL^T$
(unblocked lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L^T \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)^T$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
where b_1 **is a column**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$
where λ_{11} **is a scalar**

$b_1 \leftarrow b_1 \lambda_{11}$
$B_2 \leftarrow b_1 {l_{21}}^T + B_2$

---

continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
enddo

Algorithm 31 $B \leftarrow BL^T$
(blocked lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L^T \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)^T$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
where B_1 **has** b **columns**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$
where L_{11} **is** $b \times b$

$B_1 \leftarrow B_1 {L_{11}}^T$
$B_2 \leftarrow B_1 {L_{21}}^T + B_2$

---

continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
enddo

Figure 12.2: Unblocked (left) and blocked (right) lazy triangular matrix multiplication algorithms.

since the first operation scales a vector of length n and the vector-matrix multiply $b_1 l_{21}^T + B_2$ involves a matrix of size $k \times n$. Thus, the total cost of the triangular matrix multiplication is given by

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

This proves the theorem. $\square$

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$
\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} D_0 & D_1 & D_2 \end{array} \right)
$$

where D_1 is a block of b rows. Similarly, we repartition

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

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

Again, the double lines have meaning:

$$
\left(\begin{array}{c||c} D_L = \left(\begin{array}{c|c} D_0 & D_1 \end{array} \right) & D_R = D_2 \end{array} \right), \left(\begin{array}{c||c} B_L = \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) & B_R = B_2 \end{array} \right),
$$

$$
\text{and} \quad \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array} \right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & L_{21} \end{array} \right) & L_{BR} = L_{22} \end{array} \right)
$$

Considering (12.2) and these repartitionings, D currently contains

$$
\left(\begin{array}{c||c} \star & B_R {L_{BR}}^T \end{array} \right) = \left(\begin{array}{c||c} \star & B_2 {L_{22}}^T \end{array} \right)
$$

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

$$
\left(\begin{array}{c||c} D_L = D_0 & D_R = \left(\begin{array}{c|c} D_1 & D_2 \end{array} \right) \end{array} \right), \left(\begin{array}{c||c} B_L = B_0 & B_R = \left(\begin{array}{c|c} B_1 & B_2 \end{array} \right) \end{array} \right),
$$

$$
\text{and} \quad \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array} \right) & L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right) \end{array} \right)
$$

Thus the contents of D must change and be updated like

$$
\left(\begin{array}{c||c} \star & B_2 {L_{22}}^T \end{array} \right) \rightarrow \left(\begin{array}{c||c} \star & \left(\begin{array}{c|c} B_1 & B_2 \end{array} \right) \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right)^T \end{array} \right)
$$

Applying the transpose operator, we get

$$
\left(\begin{array}{c||c} \star & B_2 {L_{22}}^T \end{array} \right) \rightarrow \left(\begin{array}{c||c} \star & \left(\begin{array}{c|c} B_1 & B_2 \end{array} \right) \left(\begin{array}{c|c} {L_{11}}^T & {L_{21}}^T \\ \hline 0 & {L_{22}}^T \end{array} \right) \end{array} \right)
$$

After simplification, we can see that the contents of D must change and be updated like

$$
\left(\begin{array}{c||c} \star & B_2 {L_{22}}^T \end{array} \right) \rightarrow \left(\begin{array}{c||c} \star & \left(\begin{array}{c|c} B_1 {L_{11}}^T & B_1 {L_{21}}^T + B_2 {L_{22}}^T \end{array} \right) \end{array} \right)
$$

Thus we conclude that an algorithm that maintains the condition in (12.2) is given in Fig. 12.2 (right). Notice that the algorithm overwrites matrix B with the result BL^T.

12.1.2 Row-lazy algorithm (relative to L)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\star \;\Big\|\; B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) \tag{12.5}$$

This will be called 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 left. Again, this algorithm naturally moves through matrices D and B in the "left" direction.

Unblocked algorithm

Repartition

$$\left(D_L \;\|\; D_R \right) \rightarrow \left(D_0 \;|\; d_1 \;\|\; D_2 \right)$$

where d_1 is a column. Similarly, we repartition

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

where b_1 is a column and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(D_L = \left(D_0 \;|\; d_1 \right) \;\|\; D_R = D_2 \right), \left(B_L = \left(B_0 \;|\; b_1 \right) \;\|\; B_R = B_2 \right),$$
$$\text{and} \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array} \right) & 0 \\ \hline\hline L_{BL} = \left(L_{20} \;|\; l_{21} \right) & L_{BR} = L_{22} \end{array} \right)^T$$

Considering (12.5) and these repartitionings, D currently contains

$$\left(\star \;\|\; B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) = \left(\star \;\Big\|\; \left(B_0 \;\|\; b_1 \right) \left(L_{20} \;\|\; l_{21} \right)^T + B_2 {L_{22}}^T \right)$$

Simplifiying, we get

$$\left(\star \;\|\; B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) = \left(\star \;\|\; B_0 {L_{20}}^T + b_1 {l_{21}}^T + B_2 {L_{22}}^T \right)$$

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

$$\left(D_L = D_0 \;\|\; D_R = \left(d_1 \;|\; D_2 \right) \right), \left(B_L = B_0 \;\|\; B_R = \left(b_1 \;|\; B_2 \right) \right),$$
$$\text{and} \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array} \right) & L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right) \end{array} \right)^T$$

Thus the contents of D must change be updated like

$$\left(\star \;\|\; B_0 {L_{20}}^T + b_1 {l_{21}}^T + B_2 {L_{22}}^T \right) \rightarrow \left(\star \;\Bigg\|\; B_0 \left(l_{10} \;|\; {L_{20}}^T \right) + \left(b_1 \;|\; B_2 \right) \left(\begin{array}{c|c} \lambda_{11} & {l_{21}}^T \\ \hline 0 & {L_{22}}^T \end{array} \right) \right)$$

After simplification, the contents of D are updated as follows:

$$\left(\star \;\|\; B_0 {L_{20}}^T + b_1 {l_{21}}^T + B_2 {L_{22}}^T \right) \rightarrow \left(\star \;\|\; \left(B_0 l_{10} + b_1 \lambda_{11} \;\Big|\; B_0 {L_{20}}^T + b_1 {l_{21}}^T + B_2 {L_{22}}^T \right) \right)$$

Thus we conclude that an algorithm that maintains the condition in (12.5) is given in Fig. 12.3 (left). Notice that the algorithm overwrites matrix B with the result BL^T.

Algorithm 32 $B \leftarrow BL^T$
(unblocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L^T \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)^T$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
where b_1 **is a column**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array} \right)$$
where λ_{11} **is a scalar**

$b_1 \leftarrow b_1 \lambda_{11}$
$b_1 \leftarrow B_0 l_{10} + b_1$

---

continue with
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
enddo

Algorithm 33 $B \leftarrow BL^T$
(blocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L^T \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)^T$
where L_{BR} **is** 0×0
do until L_{TL} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
where B_1 **has** b **columns**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array} \right)$$
where L_{11} **is** $b \times b$

$B_1 \leftarrow B_1 {L_{11}}^T$
$B_1 \leftarrow B_0 {L_{10}}^T + B_1$

---

continue with
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
enddo

Figure 12.3: Unblocked (left) and blocked (right) row-lazy triangular matrix multiplication algorithms.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,|\, D_1 \,\|\, D_2 \right)$$

where D_1 is a block of b rows. Similarly, we repartition

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

where B_1 is a block of b columns and L_{11} is $b \times b$. Again, the double lines have meaning:

$$\left(D_L = \left(D_0 \,|\, D_1 \right) \,\|\, D_R = D_2 \right), \left(B_L = \left(B_0 \,|\, B_1 \right) \,\|\, B_R = B_2 \right),$$

$$\text{and} \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(L_{20} \,|\, L_{21} \right) & L_{BR} = L_{22} \end{array}\right) \tag{12.6}$$

Considering (12.5) and these repartitionings, D currently contains

$$\left(\star \,\|\, B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) = \left(\star \,\|\, \left(B_0 \,|\, B_1 \right) \left(L_{20} \,|\, L_{21} \right)^T + B_2 {L_{22}}^T \right)$$

Simplyfing,

$$\left(\star \,\|\, B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) = \left(\star \,\|\, B_0 {L_{20}}^T + B_1 {L_{21}}^T + B_2 {L_{22}}^T \right)$$

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

$$\left(D_L = D_0 \,\|\, D_R = \left(D_1 \,|\, D_2 \right) \right), \left(B_L = B_0 \,\|\, B_R = \left(B_1 \,|\, B_2 \right) \right),$$

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

Thus the contents of D must change and be updated like

$$\left(\star \,\|\, B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) \rightarrow \left(\star \,\Big\|\, B_0 \left(L_{10} \,|\, L_{20} \right)^T + \left(B_1 \,|\, B_2 \right) \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right)^T \right)$$

Applying the transpose operators and multiplying out, we get

$$\left(\star \,\|\, B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) \rightarrow \left(\star \,\|\, \left(B_0 {L_{10}}^T \,|\, B_0 {L_{20}}^T \right) + \left(B_1 {L_{11}}^T \,|\, B_1 {L_{21}}^T + B_2 {L_{22}}^T \right) \right)$$

Simplifying,

$$\left(\star \,\|\, B_L {L_{BL}}^T + B_R {L_{BR}}^T \right) \rightarrow \left(\star \,\|\, \left(B_0 {L_{10}}^T + B_1 {L_{11}}^T \,\Big|\, B_0 {L_{20}}^T + B_1 {L_{21}}^T + B_2 {L_{22}}^T \right) \right)$$

After simplification, we can see that the contents of D must change and be updated like

$$\left(\star \,\|\, B_0 {L_{20}}^T + B_1 {L_{21}}^T + B_2 {L_{22}}^T \right) \rightarrow \left(\star \,\|\, \left(B_0 {L_{10}}^T + B_1 {L_{11}}^T \,\Big|\, B_0 {L_{20}}^T + B_1 {L_{21}}^T + B_2 {L_{22}}^T \right) \right)$$

Thus we conclude that an algorithm that maintains the condition in (12.5) is given in Fig. 12.3 (right). Notice that the algorithm overwrites matrix B with the result BL^T.

12.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trmm_rlt/`

12.3 Performance

Performance for our lazy and row-lazy algorithms are reported in Fig. 12.6.

```
#include "FLAME.h"

void Trmm_right_lower_trans_unb ( int variant, FLA_Obj A, FLA_Obj B )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,         BL,  BR,      B0,  B1,  B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;
  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                     &ABL, /**/ &ABR,
                     0, 0,     /* submatrix */ FLA_BR );

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

  while ( FLA_Obj_length( ATL ) ) {
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &A01, /**/ &A02,
                                /**/              &A10, &A11, /**/ &A12,
                         /* ************* */    /* ********************* */
                            ABL, /**/ ABR,        &A20, &A21, /**/ &A22,
                            1,  1, /* A11 from */ FLA_TL );

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  &B1,  /**/ &B2,
                            1, /* width  B1 from */ FLA_LEFT );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Ger( ONE, B1, A21, B2 );

    FLA_Scal( A11, B1 );

    if ( variant == FLA_VARIANT_ROWLAZY )
      FLA_Gemv( FLA_NO_TRANSPOSE, ONE, B0, A10, ONE, B1 );

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

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

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

Figure 12.4: Unblocked lazy and column-lazy (w.r.t. L) triangular matrix-matrix multiplication algorithm using FLAME.

```
#include "FLAME.h"

void Trmm_right_lower_trans_blk ( int variant, int rec, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,         BL,  BR,      B0,  B1,  B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;

  int        b;

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

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

  while ( b = min( FLA_Obj_length( ATL ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &A01, /**/ &A02,
                                /**/              &A10, &A11, /**/ &A12,
                         /* ************* */    /* ********************* */
                            ABL, /**/ ABR,        &A20, &A21, /**/ &A22,
                            b,  b, /* A11 from */ FLA_TL );

    FLA_Repart_1x2_to_1x3( BL,  /**/ BR,          &B0,  &B1,  /**/ &B2,
                            b, /* width  B1 from */ FLA_LEFT );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, B1, A21, ONE, B2 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      Trmm_right_lower_trans_blk ( variant, rec, A11, B1, nb_alg/2 );
    else
      Trmm_right_lower_trans_unb ( variant, A11, B1 );

    if ( variant == FLA_VARIANT_ROWLAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, B0, A10, ONE, B1 );

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

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

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

Figure 12.5: Blocked lazy and column-lazy (w.r.t. L) triangular matrix-matrix multiplication algorithm using FLAME.

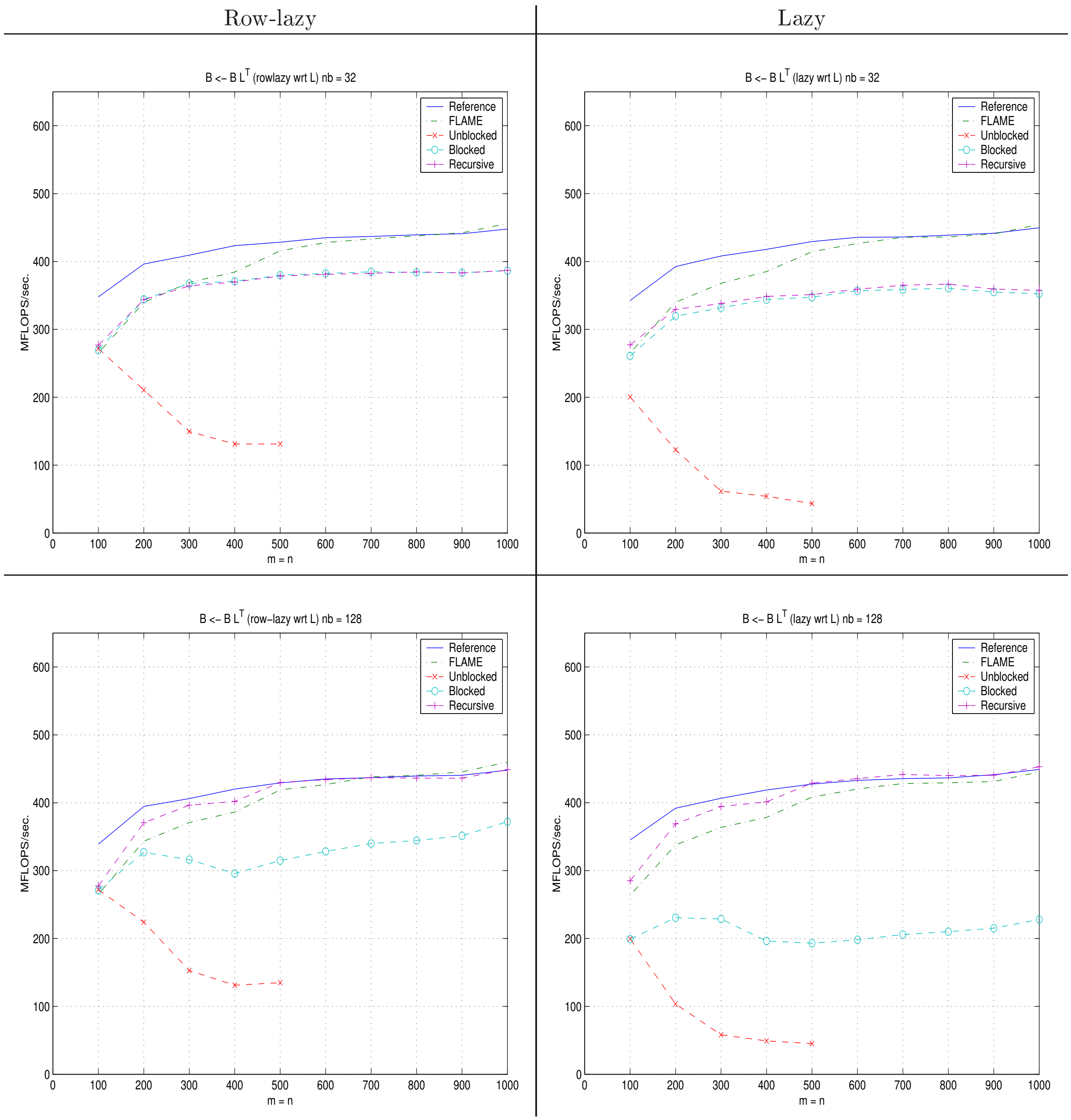


Figure 12.6: Performance of the lazy (left) and row-lazy (right) (w.r.t. L) lower triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Chapter 13

Triangular Matrix Multiplication $B \leftarrow \alpha BU$

by

Joshua L. Reid

Stephanie E. Rowland

Robert V. Wilder

In this chapter, we discuss the implementation of triangular matrix-matrix multiplication:

$$B \leftarrow \alpha BU$$

where U is an $m \times m$ upper triangular matrix and B is $n \times m$. We start by deriving a number of different sequential algorithms. Subsequently, we show how to code the algorithms using FLAME. Finally, we discuss a parallel implementation with PLAPACK.

Let us consider

(13.1) $$D = \alpha BU$$

keeping in mind that D will overwrite B. Throughout the derivations, α will be excluded for simplicity and will be added to the final equation of the derivation.

13.1 Algorithms that start by splitting U

Partition

$$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$$

where U_{TL} is $k \times k$. Also, partition

$$D \rightarrow \left(\begin{array}{c||c} D_L & D_R \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

where D_L and B_L have k columns.

Now, (13) becomes

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) = \left(\begin{array}{c||c} B_L & B_R \end{array} \right) \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c|c} D_L & D_R \end{array} \right) = \left(\begin{array}{c|c} B_L U_{TL} & B_L U_{TR} + B_R U_{BR} \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_L & = & B_L U_{TL} \\ D_R & = & B_L U_{TR} + B_R U_{BR} \end{array}$$

which must hold.

Possible contents of D at this intermediate step are given in Fig. 13.1. Considering the comments in that

D contains	Comments
$\left(\star \,\Vert\, \star \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(B_L U_{TL} \,\Vert\, \star \right)$	This one is not a viable option since if D_L overwrites B_L, unless we provide a temporary in which to store this intermediate result, the original B_L is lost.
$\left(B_L U_{TL} \,\Vert\, B_L U_{TR} \right)$	This one is not a viable option since if D_R overwrites B_R, unless we provide a temporary in which to store this intermediate result, the original B_R is lost.
$\left(B_L U_{TL} \,\Vert\, B_L U_{TR} + B_R U_{BR} \right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(B_L U_{TL} \,\Vert\, B_R U_{BR} \right)$	This one is not a viable option since if D_R overwrites B_R, unless we provide a temporary in which to store this intermediate result, the original B_R is lost.
$\left(\star \,\Vert\, B_L U_{TR} + B_R U_{BR} \right)$	
$\left(\star \,\Vert\, B_R U_{BR} \right)$	

Figure 13.1: Possible contents of D at an intermediate step.

table, only two viable conditions are left, the ones for which there are no comments.

13.1.1 Column-lazy algorithm (relative to U)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c|c} \star & B_L U_{TR} + B_R U_{BR} \end{array} \right) \tag{13.2}$$

Notice that this indicates that U_{TR} and U_{BR} have been used to update D, which will will call a row-lazy algorithm relative to operand U.

In order to move the boundary that indicates how far the computation has proceeded, that boundary must be moved left. Thus, this algorithm naturally moves through matrices D and B in the "left" direction.

Unblocked algorithm

Repartition

$$\left(\begin{array}{c|c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} D_0 & d_1 & D_2 \end{array} \right)$$

where d_1 is a column. Similarly, we repartition

$$\left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & b_1 & B_2 \end{array} \right) \quad \text{and} \quad \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \lambda_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$

where b_1 is a column and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$\left(\; D_L = \left(\; D_0 \; \middle| \; d_1 \; \right) \; \middle\| \; D_R = D_2 \; \right),$$
$$\left(\; B_L = \left(\; B_0 \; \middle| \; b_1 \; \right) \; \middle\| \; B_R = B_2 \; \right),$$
$$\text{and} \quad \left(\begin{array}{c||c} U_{TL} = \left(\begin{array}{c|c} U_{00} & u_{01} \\ \hline 0 & \lambda_{11} \end{array} \right) & U_{TR} = \left(\; U_{02} \; \middle| \; u_{12} \; \right) \\ \hline\hline 0 & U_{BR} = U_{22} \end{array} \right)$$

Considering (13.2) and these repartitionings, D currently contains

$$\left(\; \star \; \middle\| \; B_L U_{TR} + B_R U_{BR} \; \right) = \left(\; \star \; \middle\| \; \left(\; B_0 \; \middle| \; b_1 \; \right) \left(\frac{U_{02}}{u_{12}} \right) + B_2 U_{22} \; \right) = \left(\; \star \; \middle\| \; B_0 U_{02} + b_1 u_{12} + B_2 U_{22} \; \right)$$

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

$$\left(\; D_L = D_0 \; \middle\| \; D_R = \left(\; d_1 \; \middle| \; D_2 \; \right) \; \right),$$
$$\left(\; B_L = B_0 \; \middle\| \; B_R = \left(\; b_1 \; \middle| \; B_2 \; \right) \; \right),$$
$$\text{and} \quad \left(\begin{array}{c||c} U_{TL} = U_{00} & U_{TR} = \left(\; u_{01} \; \middle| \; U_{02} \; \right) \\ \hline\hline 0 & U_{BR} = \left(\begin{array}{c|c} \lambda_{11} & u_{12} \\ \hline 0 & U_{22} \end{array} \right) \end{array} \right)$$

Thus the contents of D must change be updated like

$$\left(\; \left(\; \star \; \middle| \; \star \; \right) \; \middle\| \; \alpha B_0 U_{02} + \alpha b_1 u_{12} + \alpha B_2 U_{22} \; \right) \rightarrow \left(\; \star \; \middle\| \; \left(\; \alpha B_0 u_{01} + \alpha b_1 \lambda_{11} \; \middle| \; \alpha B_0 U_{02} + \alpha b_1 u_{12} + \alpha B_2 U_{22} \; \right) \; \right)$$

Thus we conclude that an algorithm that maintains the condition in (13.2) is given in Fig. 13.2 (left). Notice that the algorithm overwrites matrix B with the result $\alpha B U$.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\; D_L \; \middle\| \; D_R \; \right) \rightarrow \left(\; D_0 \; \middle| \; D_1 \; \middle\| \; D_2 \; \right)$$

where D_1 is a block of b columns. Similarly, we repartition

$$\left(\; B_L \; \middle\| \; B_R \; \right) \rightarrow \left(\; B_0 \; \middle| \; B_1 \; \middle\| \; B_2 \; \right) \quad \text{and} \quad \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$

where B_1 is a block of b columns and U_{11} is $b \times b$.

Again, the double lines have meaning:

$$\left(\; D_L = \left(\; D_0 \; \middle| \; D_1 \; \right) \; \middle\| \; D_R = D_2 \; \right),$$
$$\left(\; B_L = \left(\; B_0 \; \middle| \; B_1 \; \right) \; \middle\| \; B_R = B_2 \; \right),$$
$$\text{and} \quad \left(\begin{array}{c||c} U_{TL} = \left(\begin{array}{c|c} U_{00} & U_{01} \\ \hline 0 & U_{11} \end{array} \right) & U_{TR} = \left(\frac{U_{02}}{U_{12}} \right) \\ \hline\hline 0 & U_{BR} = U_{22} \end{array} \right)$$

Considering (13.2) and these repartitionings, D currently contains

$$\left(\; \star \; \middle\| \; B_L U_{TR} + B_R U_{BR} \; \right) = \left(\; \star \; \middle\| \; \left(\; B_0 \; \middle| \; B_1 \; \right) \left(\frac{U_{02}}{U_{12}} \right) + B_2 U_{22} \; \right) = \left(\; \star \; \middle\| \; B_0 U_{02} + B_1 U_{12} + B_2 U_{22} \; \right)$$

Algorithm 34 $B \leftarrow \alpha BU$ *(unblocked column-lazy w.r.t. U)*

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$
where U_{TL} **is** 0×0
do until U_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
where b_1 **is a column**
repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \lambda_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$
where λ_{11} **is a scalar**

$b_1 \leftarrow \alpha \lambda_{11} b_1$
$b_1 \leftarrow \alpha B_0 u_{01} + \alpha b_1$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \lambda_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
enddo

Algorithm 35 $B \leftarrow \alpha BU$ *(blocked column-lazy w.r.t. U)*

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$
where U_{TL} **is** 0×0
do until U_{BR} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
where B_1 **has** b **columns**
repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$
where U_{11} **is** $b \times b$

$B_1 \leftarrow \alpha B_1 U_{11}$
$B_1 \leftarrow \alpha B_0 U_{01} + \alpha B_1$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
enddo

Figure 13.2: Unblocked (left) and blocked (right) column-lazy triangular matrix-matrix multiplication algorithms.

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

$$\left(D_L = D_0 \;\middle\|\; D_R = \left(D_1 \mid D_2 \right) \right),$$
$$\left(B_L = B_0 \;\middle\|\; B_R = \left(B_1 \mid B_2 \right) \right),$$
$$\text{and} \quad \left(\begin{array}{c||c} U_{TL} = U_{00} & U_{TR} = \left(U_{01} \mid U_{02} \right) \\ \hline\hline 0 & U_{BR} = \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array} \right) \end{array} \right)$$

Thus the contents of D must be updated like

$$\left(\left(\star \mid \star \right) \;\middle\|\; \alpha B_0 U_{02} + \alpha B_1 U_{12} + \alpha B_2 U_{22} \right) \rightarrow \left(\star \;\middle\|\; \left(\alpha B_0 U_{01} + \alpha B_1 U_{11} \mid \alpha B_0 U_{02} + \alpha B_1 U_{12} + \alpha B_2 U_{22} \right) \right)$$

Thus we conclude that an algorithm that maintains the condition in (13.2) is given in Fig. 13.2 (right). Notice that the algorithm overwrites matrix B with the result αBU.

13.1.2 Lazy algorithm (relative to U)

Next, consider the condition that currently

$$D \quad \text{contains} \quad \left(\star \;\middle\|\; B_R U_{BR} \right) \tag{13.3}$$

Notice that this indicates that only U_{BR} has been used to update D, which will will call a lazy algorithm relative to operand U.

Again, this algorithm naturally moves through matrices D and B in the "left" direction.

Unblocked algorithm

Repartition

$$\left(D_L \;\middle\|\; D_R \right) \rightarrow \left(D_0 \mid d_1 \;\middle\|\; D_2 \right)$$

where d_1 is a column. Similarly, we repartition

$$\left(B_L \;\middle\|\; B_R \right) \rightarrow \left(B_0 \mid b_1 \;\middle\|\; B_2 \right) \quad \text{and} \quad \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \lambda_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$

where b_1 is a column and λ_{11} is a scalar.

Again, the double lines have the same meaning as illustrated in (13.3). Considering (13.3) and these repartitionings, D currently contains

$$\left(\star \;\middle\|\; B_R U_{BR} \right) = \left(\star \;\middle\|\; B_2 U_{22} \right)$$

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the meaning in (13.3) and D must be updated to contain

$$\left(\star \;\middle\|\; B_R U_{BR} \right) = \left(\star \;\middle\|\; \left(b_1 \mid B_2 \right) \left(\begin{array}{c|c} \lambda_{11} & u_{12} \\ \hline 0 & U_{22} \end{array} \right) \right) = \left(\star \;\middle\|\; \left(b_1 \lambda_{11} \mid B_0 U_{02} + b_1 u_{12} + B_2 U_{22} \right) \right)$$

Thus the contents of D must change be updated like

$$\left(\left(\star \mid \star \right) \;\middle\|\; \alpha B_2 U_{22} \right) \rightarrow \left(\star \;\middle\|\; \left(\alpha b_1 \lambda_{11} \mid \alpha b_1 u_{12} + \alpha B_2 U_{22} \right) \right)$$

Thus we conclude that an algorithm that maintains the condition in (13.3) is given in Fig. 13.3 (left). Again, the algorithm overwrites matrix B with the result αBU.

Blocked algorithm

We leave the derivation of the lazy unblocked algorithm, given in Fig. 13.3 (right), as an exercise for the reader.

Algorithm 36 $B \leftarrow \alpha BU$
(unblocked lazy w.r.t. U)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$
where U_{BR} **is** 0×0
do until U_{TL} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
where b_1 **is a column**
repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \lambda_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$
where λ_{11} **is a scalar**

$B_2 \leftarrow \alpha b_1 l_{12} + \alpha B_2$
$b_1 \leftarrow \alpha b_1 \lambda_{11}$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \lambda_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
enddo

Algorithm 37 $B \leftarrow \alpha BU$
(blocked lazy w.r.t. U)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_R **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$
where U_{BR} **is** 0×0
do until U_{TL} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
where B_1 **has** b **columns**
repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$
where U_{11} **is** $b \times b$

$B_2 \leftarrow \alpha B_1 U_{12} + \alpha B_2$
$B_1 \leftarrow \alpha B_1 U_{11}$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
enddo

Figure 13.3: Unblocked (left) and blocked (right) lazy triangular matrix-matrix multiplication algorithms.

13.2 Algorithms that start by splitting B

Alternatively, we can start by partitioning matrix B.

First, consider partitioning B by columns:

$$B = \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

where B_L has k columns. In order to partition U conformally, we find that

$$U = \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)$$

where U_{TL} is $k \times k$. Notice that we use a 2×2 partitioning in order to expose the block of zeroes. Clearly, this partitioning of B leads to the same algorithms as were derived in Section 13.1.

Next, consider partitioning B by rows:

$$B = \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$

where B_T has k rows. Notice that this time U is not partitioned at all and

$$D = \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right)$$

Now, (13) becomes

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right) U$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array} \right) = \left(\begin{array}{c} B_T U \\ \hline\hline B_B U \end{array} \right)$$

This in turn exposes the equalities

$$D_T = B_T U \quad \text{and} \quad D_B = B_B U$$

which must hold.

Possible contents of D at this intermediate step are given by

D contains	Comments
$\left(\begin{array}{c} \star \\ \hline\hline \star \end{array} \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\begin{array}{c} B_T U \\ \hline\hline \star \end{array} \right)$	
$\left(\begin{array}{c} B_T U \\ \hline\hline B_B U \end{array} \right)$	This indicates that the computation has finished. Clearly not an interesting intermediate condition.
$\left(\begin{array}{c} \star \\ \hline\hline B_B U \end{array} \right)$	

13.2.1 Down-moving (lazy) algorithm

Let us consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c} B_T U \\ \hline\hline \star \end{array} \right) \tag{13.4}$$

Notice that since B_T 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 down within D and B. Thus, we call it a down-moving algorithm (w.r.t. B).

Unblocked algorithm

Repartition

$$\left(\frac{D_T}{D_B}\right) \rightarrow \left(\frac{D_0}{\frac{d_1^T}{D_2}}\right)$$

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

$$\left(\frac{B_T}{B_B}\right) \rightarrow \left(\frac{B_0}{\frac{b_1^T}{B_2}}\right)$$

where b_1^T is a row. Notice that

$$\left(\frac{D_T = D_0}{D_B = \left(\frac{d_1^T}{D_2}\right)}\right) \quad \text{and} \quad \left(\frac{B_T = B_0}{B_B = \left(\frac{b_1^T}{B_2}\right)}\right) \tag{13.5}$$

Thus D currently contains

$$\left(\frac{B_0 U}{\left(\frac{\star}{\star}\right)}\right)$$

After moving the double lines ahead,

$$\left(\frac{D_T = \left(\frac{D_0}{d_1^T}\right)}{D_B = D_2}\right) \quad \text{and} \quad \left(\frac{B_T = \left(\frac{B_0}{b_1^T}\right)}{B_B = B_2}\right)$$

and thus D must hold

$$\left(\frac{\left(\frac{B_0}{b_1^T}\right) U}{\star}\right) = \left(\frac{\left(\frac{B_0 U}{b_1^T U}\right)}{\star}\right)$$

Thus, the contents of D must be updated like

$$\left(\frac{\alpha B_0 U}{\left(\frac{\star}{\star}\right)}\right) \rightarrow \left(\frac{\left(\frac{\alpha B_0 U}{\alpha b_1^T U}\right)}{\star}\right)$$

This motivates the algorithm given in Fig. 13.4 (left).

Blocked algorithm

We leave the derivation of the blocked algorithm given in Fig. 13.4 as an exercise for the reader.

13.2.2 Up-moving (lazy) algorithm

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

13.3 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trmm_run/`

13.4 Performance

Algorithm 38 $B \leftarrow \alpha BU$
(unblocked down-moving (lazy) w.r.t. B)

partition
$B \rightarrow \left(\frac{B_T}{B_B}\right)$
where B_T **has** 0 **columns**
do until B_B **has** 0 **columns**

repartition
$\left(\frac{B_T}{B_B}\right) \rightarrow \begin{pmatrix} B_0 \\ \hline b_1^T \\ \hline B_2 \end{pmatrix}$
where b_1^T **is a row**

$b_1^T \leftarrow \alpha b_1^T U$

---

continue with
$\left(\frac{B_T}{B_B}\right) \leftarrow \begin{pmatrix} B_0 \\ \hline b_1^T \\ \hline B_2 \end{pmatrix}$
enddo

Algorithm 39 $B \leftarrow \alpha BU$
(blocked down-moving (lazy) w.r.t. B)

partition
$B \rightarrow \left(\frac{B_T}{B_B}\right)$
where B_T **has** 0 **columns**
do until B_B **has** 0 **columns**
determine block size b
repartition
$\left(\frac{B_T}{B_B}\right) \rightarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$
where B_1 **has** b **columns**

$B_1 \leftarrow \alpha B_1 U$

---

continue with
$\left(\frac{B_T}{B_B}\right) \leftarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$
enddo

Figure 13.4: Unblocked (left) and blocked (right) down-moving (lazy) triangular matrix-matrix multiplication algorithms.

```
#include "FLAME.h"

void Trmm_right_upper_notrans_unb( int variant, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00,  a01,    A02,    BL,      B0,
              ABL, ABR,     a10, alpha11, a12,    BR,      b1,
                            A20,  a21,    A22,             B2;
  int         b;


  FLA_Part_1x2( B,  &BL, /***/ &BR,   0, /* length submatrix */ FLA_RIGHT );


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

  while ( 0 != FLA_Obj_length( ATL ) ){
    FLA_Repart_1x2_to_1x3( BL, /**/ BR,    &B0, &b1, /**/ &B2,    1, /* length b1 from */ FLA_LEFT );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00,  &a01,    /**/ &A02,
                               /**/              &a10, &alpha11, /**/ &a12,
                        /* ************* */    /* *************************** */
                           ABL, /**/ ABR,         &A20,  &a21,    /**/ &A22,
                           1, 1, /* alpha11 from */ FLA_TL );


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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Ger( ONE, b1, a12, B2 );

    FLA_Scal( alpha11, b1 );

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Gemv( FLA_NO_TRANSPOSE, ONE, B0, a01, ONE, b1 );

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

    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,     B0, /**/ b1,  B2,     /* b1  added to */ FLA_RIGHT );


    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00,  /**/ a01,     A02,
                            /* ************** */   /* ************************ */
                                    /**/               a10,  /**/ alpha11, a12,
                              &ABL, /**/ &ABR,         A20,  /**/ a21,     A22,
                              /* alpha11 added to */ FLA_BR );

  }
}
```

Figure 13.5: Unblocked lazy and column-lazy (w.r.t. U) triangular matrix-matrix multiplication algorithm using FLAME.

```
#include "FLAME.h"

void Trmm_right_upper_notrans_blk( int variant, int rec, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,          BL,  BR,      B0,  B1,  B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;
  int         b;

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

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

  while ( b = min ( FLA_Obj_length( ATR ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &A01, /**/ &A02,
                                /**/              &A10, &A11, /**/ &A12,
                         /* ************* */    /* ********************* */
                            ABL, /**/ ABR,        &A20, &A21, /**/ &A22,
                            b, b, /* A11 from */ FLA_TL );

    FLA_Repart_1x2_to_1x3( BL, /**/ BR,           &B0,  &B1,  /**/ &B2,
                            b, /* width  B1 from */ FLA_LEFT );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, B1, A12, ONE, B2 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      Trmm_right_upper_notrans_blk( variant, rec, A11, B1, nb_alg/2 );
    else
      Trmm_right_upper_notrans_unb( variant, A11, B1, nb_alg/2 );

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, B0, A01, ONE, B1 );

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

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

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

Figure 13.6: Blocked lazy and column-lazy (w.r.t. U) triangular matrix-matrix multiplication algorithm using FLAME. Recursion is optionally supported.

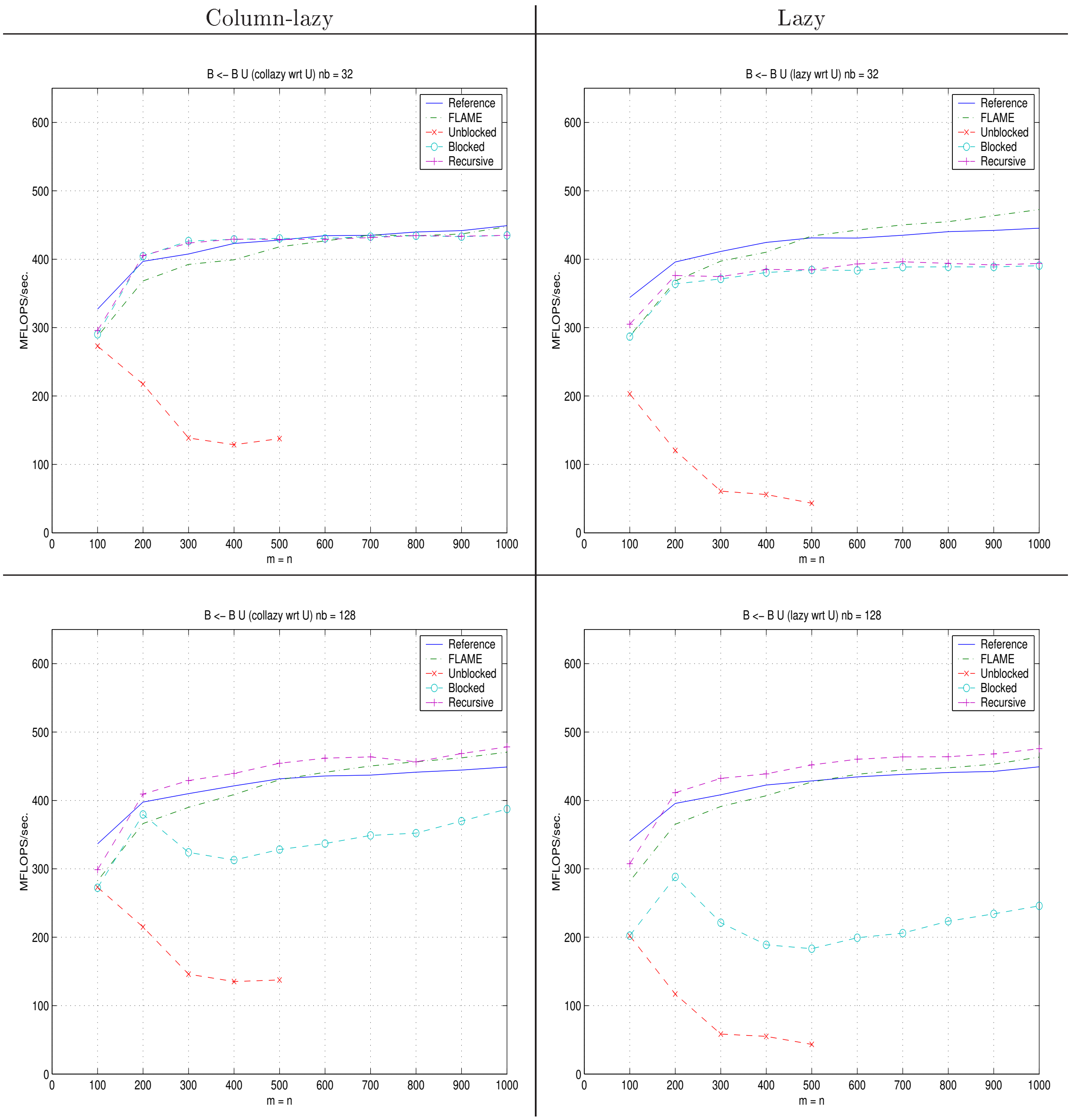


Figure 13.7: Performance of the lazy (left) and column lazy (right) (w.r.t. U) lower triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used for the FLAME experiments.

Chapter 14

Triangular Matrix Multiplication $B \leftarrow BU^T$

by

Eugene Ho

Lea Kulapaditharom

In this chapter, we discuss the implementation of triangular matrix-matrix multiplication

$$B \leftarrow BU^T$$

where U is an $n \times n$ upper triangular matrix and B is $m \times n$.

14.1 Algorithms that start by splitting U

To begin this operation, it would be easier to see B before and after the update as two separate matrices. We change the operation from

$$B \leftarrow BU^T$$

to

$$D = BU^T.$$

Then, we can proceed to partition the matrices. As a result of the transpose, U^T becomes a lower triangular matrix. Hence, U partitions as

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$$

As for B, if it is partitioned by rows

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$

the operation becomes

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) = \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)^T$$

B_T and B_B do not multiply to any distinct partitions of the Matrix U. Therefore it is inadvisable to partition Matrix B by rows. Alternatively, if B is partitioned by columns

$$\left(\begin{array}{c||c} B_L & B_R \end{array}\right)$$

the matrix multiplication can be represented as

$$\left(B_L \,\|\, B_R \right) \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)^T$$

Since B_L and B_R respectively multiply to the top and bottom sections of Matrix U, it makes more sense to partition this way since it effectively splits the work into two distinct sections. As D is supposed to be an update on B, it also makes sense to partition D by columns as well. So, our operation now changes from

$$D = BU^T$$

to

$$\left(D_L \,\|\, D_R \right) = \left(B_L \,\|\, B_R \right) \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right)^T$$

Now that the matrix is partitioned into blocks, the matrix operation can be performed on each separate block

$$\left(D_L \,\|\, D_R \right) = \left(B_L \,\|\, B_R \right) \left(\begin{array}{c||c} U_{TL}^T & 0 \\ \hline\hline U_{TR}^T & U_{BR}^T \end{array} \right) = \left(B_L U_{TL}^T + B_R U_{TR}^T \,\|\, B_R U_{BR}^T \right)$$

From this calculation, it logically follows that

$$\left(D_L \,\|\, D_R \right) = \left(B_L U_{TL}^T + B_R U_{TR}^T \,\|\, B_R U_{BR}^T \right)$$

deriving

$$\begin{array}{rcl} D_L & = & B_L U_{TL}^T + B_R U_{TR}^T \\ D_R & = & B_R U_{BR}^T \end{array}$$

Possible contents of D at this intermediate step are given in Fig. 14.1. Considering the comments in that

D contains	Comments
$\left(\star \,\|\, \star \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(\star \,\|\, B_R U_{BR}^T \right)$	This one is not a viable option since if D_R overwrites B_R, unless we provide a temporary in which to store this intermediate result, the original B_R is lost.
$\left(B_L U_{TL}^T \,\|\, B_R U_{BR}^T \right)$	This one is not a viable option since if D_L overwrites B_L, unless we provide a temporary in which to store this intermediate result, the original B_L is lost.
$\left(B_L U_{TL}^T + B_R U_{TR}^T \,\|\, B_R U_{BR}^T \right)$	Notice that this assumes that the computation has finished, which is not very interesting.
$\left(B_R U_{TR}^T \,\|\, B_R U_{BR}^T \right)$	This one is not a viable option since if D_L overwrites B_L, unless we provide a temporary in which to store this intermediate result, the original B_L is lost.
$\left(B_L U_{TL}^T + B_R U_{TR}^T \,\|\, \star \right)$	
$\left(B_L U_{TL}^T \,\|\, \star \right)$	

Figure 14.1: Possible contents of D at an intermediate step.

table, only two viable conditions are left, the ones for which there are no comments.

$$\left(B_L U_{TL}^T + B_R U_{TR}^T \,\|\, \star \right)$$

or

$$\left(\begin{array}{c||c} B_R U_{TR}^T & \star \end{array}\right)$$

From these equations, it can determined that B_R is essential for both equations and should not be altered before repartitioning. Therefore, the flow of the operation should go rightwards.

14.1.1 Row-Lazy algorithm (relative to U)

For row-lazy computation we consider the progression as

$$\left(\begin{array}{c||c} B_L U_{TL}^T + B_R U_{TR}^T & \star \end{array}\right)$$

Unblocked algorithm

With the flow of computation now determined, the uncalculated partition can now be **repartitioned**

$$\left(\begin{array}{c||c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} D_0 & d_1 & D_2 \end{array}\right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} B_0 & b_1 & B_2 \end{array}\right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & v_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

which also affect the contents of the D matrix

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T + B_R U_{TR}^T & \star \end{array}\right) &= \left(\begin{array}{c||c} B_0 U_{00}^T + \left(\begin{array}{c|c} b_1 & B_2 \end{array}\right) \left(\begin{array}{c} u_{01}^T \\ \hline U_{02}^T \end{array}\right) & \star \end{array}\right) \\ &= \left(\begin{array}{c||c} B_0 U_{00}^T + b_1 u_{01}^T + B_2 U_{02}^T & \star \end{array}\right) \end{aligned}$$

With this step complete, it is now possible to shift the boundaries of each main partition.

$$\left(\begin{array}{c||c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} D_0 & d_1 & D_2 \end{array}\right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} B_0 & b_1 & B_2 \end{array}\right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & v_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

Now, the D matrix can be recalculated for the new boundaries as such:

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T + B_R U_{TR}^T & \star \end{array}\right) &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & b_1 \end{array}\right) \left(\begin{array}{c|c} U_{00}^T & 0 \\ \hline u_{01}^T & v_{11} \end{array}\right) + B_2 \left(\begin{array}{c|c} U_{02}^T & u_{12} \end{array}\right) & \star \end{array}\right) \\ &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + b_1 u_{01}^T + B_2 U_{02}^T & b_1 v_{11} + B_2 u_{12} \end{array}\right) & \star \end{array}\right) \end{aligned}$$

Comparing the matrix D before and after the boundary shift, it is clear which section of the matrix has to be updated with each shift of the boundary.

$$\left(\begin{array}{c||c} B_0 U_{00}^T + b_1 u_{01}^T + B_2 U_{02}^T & \left(\begin{array}{c|c} \star & \star \end{array}\right) \end{array}\right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + b_1 u_{01}^T + B_2 U_{02}^T & b_1 v_{11} + B_2 u_{12} \end{array}\right) & \star \end{array}\right)$$

So, with each boundary shift, $b_1 v_{11} + B_2 u_{12}$ is calculated and updates to a row of D.

Blocked algorithm

Repartitioning using the blocked algorithm means that the next step is split into a vector instead of a single scalar value. Doing so means that parallel matrix-matrix operations can be used to perform the calculations.

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} D_0 & D_1 & D_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$

Similarly, the contents of D changes.

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T + B_R U_{TR}^T & \star \end{array} \right) &= \left(\begin{array}{c||c} B_0 U_{00}^T + \left(\begin{array}{c|c} B_1 & B_2 \end{array} \right) \left(\begin{array}{c} U_{01}^T \\ \hline U_{02}^T \end{array} \right) & \star \end{array} \right) \\ &= \left(\begin{array}{c||c} B_0 U_{00}^T + B_1 U_{01}^T + B_2 U_{02}^T & \star \end{array} \right) \end{aligned}$$

With this step complete, it is now possible to shift the boundaries of each main partition.

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} D_0 & D_1 & D_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} B_0 & B_1 & B_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array} \right)$$

Now, the D matrix can be recalculated for the new boundaries as such:

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T + B_R U_{TR}^T & \star \end{array} \right) &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} U_{00}^T & 0 \\ \hline U_{01}^T & U_{11}^T \end{array} \right) + B_2 \left(\begin{array}{c|c} U_{02}^T & U_{12}^T \end{array} \right) & \star \end{array} \right) \\ &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + B_1 U_{01}^T + B_2 U_{02}^T & B_1 U_{11}^T + B_2 U_{12}^T \end{array} \right) & \star \end{array} \right) \end{aligned}$$

Comparing the matrix D before and after the boundary shift, it is clear which section of the matrix has to be updated with each shift of the boundary.

$$\left(\begin{array}{c||c} B_0 U_{00}^T + B_1 U_{01}^T + B_2 U_{02}^T & \left(\begin{array}{c|c} \star & \star \end{array} \right) \end{array} \right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + B_1 U_{01}^T + B_2 U_{02}^T & B_1 U_{11}^T + B_2 U_{12}^T \end{array} \right) & \star \end{array} \right)$$

So, with each boundary shift, $B_1 U_{11}^T + B_2 U_{12}$ is calculated and updates to a row of D.

14.1.2 Lazy Algorithm (relative to U)

For Lazy computation we consider the progression as

$$\left(\begin{array}{c||c} B_L U_{TL}^T & \star \end{array} \right)$$

Again, this algorithm moves in the the same direction as the row-lazy algorithm, which is to the right.

Algorithm 40
$B \leftarrow BU^T$

(unblocked row-lazy w.r.t. U)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **rows**

partition

$$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$$

where U_{TL}^T **is a** 0×0 **matrix.**

do until U_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
where b_1 *is a column*

repartition

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \upsilon_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where υ_{11} *is a scalar*

$b_1 \leftarrow b_1 \upsilon_{11}$
$b_1 \leftarrow B_2 u_{12} + b_1$

---

continue with

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$

enddo

Algorithm 41
$B \leftarrow BU^T$

(blocked row-lazy w.r.t. U)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **rows**

partition

$$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$$

where U_{TL}^T **is a** 0×0 **matrix.**

do until U_{BR} **is** 0×0

determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
where B_1 *has* b *columns*

repartition

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where U_{11}^T *is* $b \times b$

$B_1 \leftarrow B_1 U_{11}^T$
$B_1 \leftarrow B_2 U_{12} + B_1$

---

continue with

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$

enddo

Figure 14.2: Unblocked (left) and blocked (right) row-lazy transposed upper-triangular matrix-matrix multiplication algorithms.

Unblocked Algorithm

Repartition

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} D_0 & d_1 & D_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} B_0 & b_1 & B_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \upsilon_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$

With that, D currently contains

$$\left(\begin{array}{c||c} B_L U_{TL}^T & \star \end{array} \right) = \left(\begin{array}{c||c} B_0 U_{00}^T & \star \end{array} \right)$$

In the next iteration,

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T & \star \end{array} \right) &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & b_1 \end{array} \right) \left(\begin{array}{c|c} U_{00}^T & 0 \\ \hline u_{01}^T & \upsilon_{11} \end{array} \right) & \star \end{array} \right) \\ &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + b_1 u_{01}^T & b_1 \upsilon_{11} \end{array} \right) & \star \end{array} \right) \end{aligned}$$

Thus the contents of D changes as such

$$\left(\begin{array}{c||c} B_0 U_{00}^T & \left(\begin{array}{c|c} \star & \star \end{array} \right) \end{array} \right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + b_1 u_{01}^T & b_1 \upsilon_{11} \end{array} \right) & \star \end{array} \right)$$

Blocked Algorithm

Repartition

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} D_0 & D_1 & D_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array} \right)$$

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array} \right)$$

With that, D currently contains

$$\left(\begin{array}{c||c} B_L U_{TL}^T & \star \end{array} \right) = \left(\begin{array}{c||c} B_0 U_{00}^T & \star \end{array} \right)$$

In the next iteration,

$$\begin{aligned} \left(\begin{array}{c||c} B_L U_{TL}^T & \star \end{array} \right) &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} U_{00}^T & 0 \\ \hline U_{01}^T & U_{11}^T \end{array} \right) & \star \end{array} \right) \\ &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + B_1 U_{01}^T & B_1 U_{11}^T \end{array} \right) & \star \end{array} \right) \end{aligned}$$

Thus the contents of D changes as such

$$\left(\begin{array}{c||c} B_0 U_{00}^T & \left(\begin{array}{c|c} \star & \star \end{array} \right) \end{array} \right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 U_{00}^T + B_1 U_{01}^T & B_1 U_{11}^T \end{array} \right) & \star \end{array} \right)$$

Algorithm 42
$B \leftarrow BU^T$

(unblocked lazy w.r.t. U)
partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **rows**

partition
$$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$$
where U_{TL}^T **is a** 0×0 **matrix.**

do until U_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
where b_1 *is a column*

repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)^T \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline 0 & \upsilon_{11} & u_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$
where υ_{11}^T *is a scalar*

$B_0 \leftarrow b_1 u_{01}^T + B_0$
$b_1 \leftarrow b_1 u_{11}$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)^T \rightarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$

enddo

Algorithm 43
$B \leftarrow BU^T$

(blocked lazy w.r.t. U)
partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **rows**

partition
$$U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$$
where U_{TL}^T **is a** 0×0 **matrix.**

do until U_{BR} **is** 0×0

determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
where B_1 *has b columns*

repartition
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$
where U_{11}^T *is $b \times b$*

$B_0 \leftarrow B_1 U_{01}^T + B_0$
$B_1 \leftarrow B_1 U_{11}$

---

continue with
$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$

enddo

Figure 14.3: Unblocked (left) and blocked (right) lazy transposed upper-triangular matrix-matrix multiplication algorithms.

```
#include "FLAME.h"

void Trmm_right_upper_trans_unb( int variant, FLA_Obj A, FLA_Obj B ){

  FLA_Obj     ATL, ATR,     A00, A01, A02,    BL,BR,      B0,B1,B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;

   FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,   0, 0,     /* submatrix */ FLA_TL );

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

   while ( 0 != FLA_Obj_length( ABR ) ){

     FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                            /* ************* */    &A10, /**/ &A11, &A12,
                                 /**/            /* ******************** */
                            ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
                            1, 1, /* A11 from */ FLA_BR );

     FLA_Repart_1x2_to_1x3( BL, /**/ BR,              &B0,/**/ &B1,&B2,
                           1, /* length B1 from */ FLA_RIGHT );

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

     if ( variant == FLA_VARIANT_LAZY )
       FLA_Ger( ONE, B1, A01, B0 );

     FLA_Scal( A11, B1 );

     if ( variant == FLA_VARIANT_ROWLAZY )
       FLA_Gemv (FLA_NO_TRANSPOSE, ONE , B2, A12, ONE, B1);

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

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

    FLA_Cont_with_1x3_to_1x2( &BL,  /**/  &BR,        B0, B1, /**/  B2,

                               /* B1  added to */ FLA_LEFT );
  }
}
```

Figure 14.4: Unblocked row-lazy and lazy (w.r.t. U) triangular matrix-matrix multiplication algorithm using FLAME.

```
#include "FLAME.h"

void Trmm_right_upper_trans_blk( int variant, int rec, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,    BL,BR,      B0,B1,B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;

  int         b;

   FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,   0, 0,     /* submatrix */ FLA_TL );

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

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

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

     FLA_Repart_1x2_to_1x3( BL, /**/ BR,              &B0,/**/ &B1,&B2,
                           b, /* length B1 from */ FLA_RIGHT );

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

     if ( variant == FLA_VARIANT_LAZY )
       FLA_Gemm (FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE , B1, A01, ONE, B0);

     if ( rec == FLA_RECURSIVE && b > 8 )
       Trmm_right_upper_trans_blk ( variant, rec, A11, B1, nb_alg/2 );
     else
       Trmm_right_upper_trans_unb ( variant, A11, B1 );

     if ( variant == FLA_VARIANT_ROWLAZY )
       FLA_Gemm (FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE , B2, A12, ONE, B1);

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

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

    FLA_Cont_with_1x3_to_1x2( &BL,  /**/  &BR,        B0, B1, /**/  B2,

                              /* B1  added to */ FLA_LEFT );
  }
}
```

Figure 14.5: Blocked row-lazy and lazy (w.r.t. U) triangular matrix-matrix multiplication algorithm using FLAME.

14.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trmm_rut/`

14.3 Performance

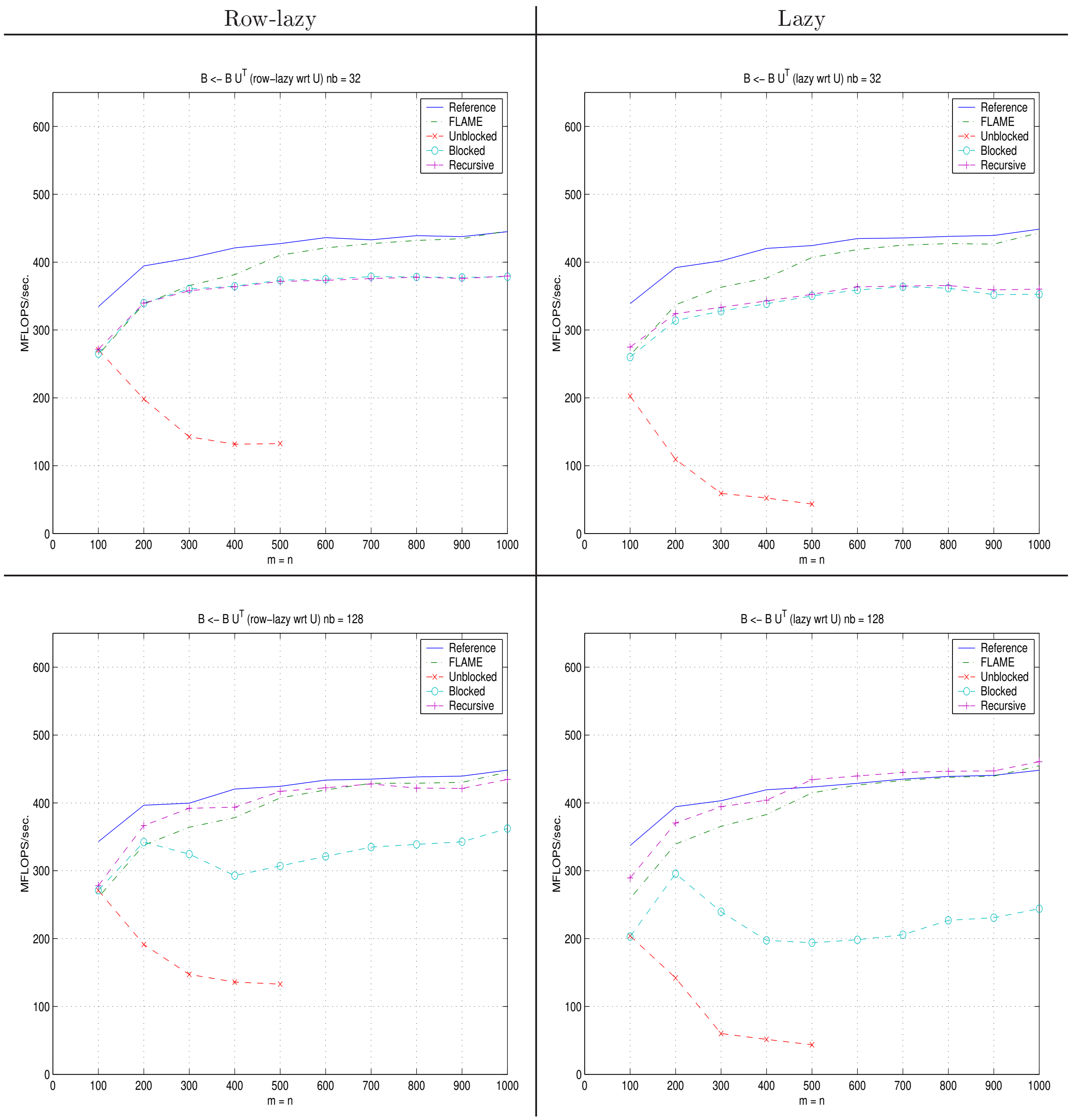


Figure 14.6: Performance of the row-lazy (left) and lazy (right) triangular matrix-matrix multiplication algorithms for a block size of 32 (top) and 128 (bottom).

Chapter 15

Triangular Solve (with Multiple RHSs) $B \leftarrow L^{-T}B$

by

Trey Henninger

Miguel Vázquez García

In this chapter, we discuss the computation of the solution of a triangular system with multiple right-hand-sides:

$$B \leftarrow L^{-T}B$$

where L is an $m \times m$ upper triangular matrix and B is $m \times n$.

Let us consider

$$D = L^{-T}B$$

keeping in mind that D will overwrite B. Notice that this is equivalent to solving

$$L^T D = B \tag{15.1}$$

for matrix D. Thus the name *triangular solve with multiple right-hand-sides* (RHSs).

15.1 Algorithms that start by splitting L

Partition

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

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

$$D \rightarrow \left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$$

where D_T and B_T have k rows.

Now, (15.1) becomes

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) T \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) = \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$$

Multiplying out the right-hand-side yields

$$\left(\begin{array}{c} L_{TL}^T D_T + L_{BL}^T D_B \\ \hline L_{BR}^T D_B \end{array}\right) = \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} L_{TL}^T D_T + L_{BL}^T D_B & = & B_T \\ L_{BR}^T D_B & = & B_B \end{array}$$

which must hold. Notice that these equalities are also equivalent to

$$\begin{array}{rcl} D_T & = & L_{TL}^{-T}(B_T - L_{BL}^T D_B) \\ D_B & = & L_{BR}^{-T} B_B \end{array}$$

Possible contents of D at this intermediate step are given in Fig. 15.1. Considering the comments in that

D contains	Comments	Viable?
$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$	This condition indicates no progress has been made.	NO
$\left(\begin{array}{c} B_T \\ \hline L_{BR}^{-T} B_B \end{array}\right)$		YES
$\left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline B_B \end{array}\right)$	This one is not a viable option since it requires D_B without actually storing D_B.	NO
$\left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline L_{BR}^{-T} B_B \end{array}\right)$		YES
$\left(\begin{array}{c} L_{TL}^{-T}(B_T - L_{BL}^T D_B) \\ \hline B_B \end{array}\right)$	This one is not a viable option since it requires D_B without actually storing D_B.	NO
$\left(\begin{array}{c} L_{TL}^{-T}(B_T - L_{BL}^T D_B) \\ \hline L_{BR}^{-T} B_B \end{array}\right)$	This condition indicates that the computation has completed.	NO

Figure 15.1: Possible contents of D at an intermediate step.

table, only two viable conditions are left, the ones for which there are no comments.

15.1.1 Row-lazy algorithm (relative to L)

Consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline L_{BR}^{-T} B_B \end{array}\right) \tag{15.2}$$

Notice that this indicates that L_{BL}^T and L_{BR}^T have been used to update D, which will 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

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right)$$

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

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

where b_1^T is a row and λ_{11} is a scalar.

Notice that the double lines have meaning:

$$(15.3) \quad \left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline d_1^T \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right) & L_{BR} = L_{22} \end{array}\right)$$

Considering (15.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) - \left(\begin{array}{c} L_{20}^T \\ \hline l_{21}^T \end{array}\right) L_{22}^{-T} B_2 \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} B_0 - L_{20}^T L_{22}^{-T} B_2 \\ \hline b_1^T - l_{21}^T L_{22}^{-T} B_2 \end{array}\right) \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right)$$

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

$$(15.4) \left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \end{array}\right)$$

Once the double lines move (in other words, at the top of the loop during the next iteration), the contents of D must equal

$$\left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) = \left(\begin{array}{c} B_0 - \left(\begin{array}{c|c} l_{10} & L_{20}^T \end{array}\right) \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right)^{-1} \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} B_0 - \left(\begin{array}{c|c} l_{01} & L_{20}^T \end{array}\right) \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \\ \hline\hline \left(\begin{array}{c|c} 1/\lambda_{11} & -l_{21}^T L_{22}^{-T}/\lambda_{11} \\ \hline 0 & L_{22}^{-T} \end{array}\right) \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right)$$

$$= \left(\begin{array}{c} B_0 - \left(\begin{array}{c|c} l_{01} & L_{02}^T \end{array}\right) \left(\begin{array}{c} d_1^T \\ \hline D_2 \end{array}\right) \\ \hline\hline \left(\begin{array}{c} d_1^T = b_1^T/\lambda_{11} - l_{21}^T L_{22}^{-T}/\lambda_{11} B_2 \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} B_0 - l_{01} d_1^T - L_{20}^T L_{22}^{-T} B_2 \\ \hline\hline \left(\begin{array}{c} d_1^T = (b_1^T - l_{21}^T L_{22}^{-T} B_2)/\lambda_{11} \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right)$$

since

$$\left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right)^{-1} = \left(\begin{array}{c|c} 1/\lambda_{11} & -l_{21}^T L_{22}^{-T}/\lambda_{11} \\ \hline 0 & L_{22}^{-T} \end{array}\right)$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} B_0 - L_{20}^T L_{22}^{-T} B_2 \\ \hline b_1^T - l_{21}^T L_{22}^{-T} B_2 \end{array}\right) \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 - l_{10} d_1^T - L_{20}^T L_{22}^{-T} B_2 \\ \hline\hline \left(\begin{array}{c} d_1^T = (b_1^T - l_{21}^T L_{22}^{-T} B_2)/\lambda_{11} \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right)$$

We conclude that an algorithm that maintains the condition in (15.2) is given in Fig. 15.2 (left). Notice that the algorithm overwrites matrix B with the result $L^{-T} B$.

Algorithm 44 $B \leftarrow L^{-T}B$
(unblocked row-lazy w.r.t. L)

partition
$B \rightarrow \left(\frac{B_T}{B_B}\right)$
where B_B **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array}\right)$
where L_{BR}^T **is** 0×0
do until L_{TL}^T **is** 0×0

repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right)$
where b_1^T **is a row**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22}^T \end{array}\right)$
where λ_{11} **is a scalar**

$b_1^T \leftarrow b_1^T / \lambda_{11}$
$B_0 \leftarrow B_0 - l_{10} b_1^T$

---

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

Algorithm 45 $B \leftarrow L^{-T}B$
(blocked row-lazy w.r.t. U)

partition
$B \rightarrow \left(\frac{B_T}{B_B}\right)$
where B_B **has** 0 **rows**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array}\right)$
where L_{BR}^T **is** 0×0
do until L_{TL}^T **is** 0×0
determine block size b
repartition
$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$
where B_1 **has** b **rows**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22}^T \end{array}\right)$
where L_{11}^T **is** $b \times b$

$B_1 \leftarrow L_{11}^{-T} B_1$
$B_0 \leftarrow B_0 - L_{10}^T B_1$

---

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

Figure 15.2: Unblocked (left) and blocked (right) row-lazy triangular solution with multiple right-hand-sides algorithms.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline\hline D_2 \end{array}\right)$$

where D_1 is a block of b rows. Similarly, we repartition

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

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

Again, the double lines have meaning:

$$\left(\begin{array}{c} D_T = \left(\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right) \\ \hline\hline D_B = D_2 \end{array}\right), \left(\begin{array}{c} B_T = \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) \\ \hline\hline B_B = B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right) & L_{BR} = L_{22} \end{array}\right)$$

Considering (15.2) and these repartitionings, D currently contains

$$\left(\begin{array}{c} B_T - L_{BL}^T D_B \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) - \left(\begin{array}{c} L_{20}^T \\ \hline L_{21} \end{array}\right) L_{22}^{-T} B_2 \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} B_0 - L_{20}^T L_{22}^{-T} B_2 \\ \hline B_1 - L_{21} L_{22}^{-T} B_2 \end{array}\right) \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right)$$

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

$$\left(\begin{array}{c} D_T = D_0 \\ \hline\hline D_B = \left(\begin{array}{c} D_1 \\ \hline D_2 \end{array}\right) \end{array}\right), \left(\begin{array}{c} B_T = B_0 \\ \hline\hline B_B = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} = L_{00} & 0 \\ \hline\hline L_{BL} = \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array}\right) & L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \end{array}\right)$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} B_0 - L_{20}^T L_{22}^{-T} B_2 \\ \hline B_1 - L_{21} L_{22}^{-T} B_2 \end{array}\right) \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 - L_{10} D_1 - L_{20}^T L_{22}^{-T} B_2 \\ \hline\hline \left(\begin{array}{c} D_1 = {L_{11}}^{-T}(B_1 - L_{21} L_{22}^{-T} B_2) \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right)$$

We conclude that an algorithm that maintains the condition in (15.2) is given in Fig. 15.2 (right). Notice that the algorithm overwrites matrix B with the result $L^{-1}B$.

15.1.2 Lazy algorithm (relative to L)

Next, consider the condition that currently

$$D \quad \text{contains} \quad \left(\begin{array}{c} B_T \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) \tag{15.5}$$

Notice that this indicates that only L_{BR}^{-T} 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

$$\left(\begin{array}{c} D_T \\ \hline\hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline d_1^T \\ \hline\hline D_2 \end{array}\right)$$

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

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

where b_1^T is a row and λ_{11} is a scalar.

Again, the double lines have the same meaning as illustrated in (15.3). Considering (15.5) and these repartitionings, D currently contains

$$\left(\begin{array}{c} B_T \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) = \left(\begin{array}{c} B_B \\ \hline\hline L_{22}^{-T} B_2 \end{array}\right)$$

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the meaning in (15.4) and D must be updated to contain

$$\begin{array}{rcl} \left(\begin{array}{c} B_T \\ \hline\hline L_{BR}^{-T} B_B \end{array}\right) & = & \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline \lambda_{21} & L_{22} \end{array}\right)^{-T} \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \\ & = & \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c|c} 1/\lambda_{11} & l_{21}^T L_{22}^{-T}/\lambda_{11} \\ \hline 0 & L_{22} \end{array}\right)^{-T} \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \end{array}\right) \\ & = & \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c} (b_1^T - l_{21}^T L_{22}^{-T} B_2)/\lambda_{11} \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right) \end{array}$$

Thus the contents of D must be updated like

$$\left(\begin{array}{c} \left(\begin{array}{c} B_0 \\ \hline b_1^T \end{array}\right) \\ \hline\hline L_{22}^{T} B_2 \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline \left(\begin{array}{c} (b_1^T - l_{21} L_{22}^{-T} B_2)/\lambda_{11} \\ \hline L_{22}^{-T} B_2 \end{array}\right) \end{array}\right)$$

Thus we conclude that an algorithm that maintains the condition in (15.5) is given in Fig. 15.3 (left). Again, the algorithm overwrites matrix B with the result.

Blocked algorithm

We leave the derivation of the lazy unblocked algorithm, given in Fig. 15.3 (right), as an exercise for the reader.

15.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trsm_llt/`

15.3 Performance

Algorithm 46 $B \leftarrow L^{-T} B$
(unblocked lazy w.r.t. L)

partition
$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$
where B_B **has** 0 **rows**
partition
$$L \rightarrow \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array} \right)$$
where L_{BR}^T **is** 0×0
do until L_{TL}^T **is** 0×0

repartition
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array} \right)$$
where b_1^T **is a row**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22}^T \end{array} \right)$$
where λ_{11} **is a scalar**

$b_1^T \leftarrow b_1^T - l_{21}^T B_2$
$b_1^T \leftarrow b_1^T / \lambda_{11}$

---

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

Algorithm 47 $B \leftarrow L^{-T} B$
(blocked lazy w.r.t. L)

partition
$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right)$$
where B_B **has** 0 **rows**
partition
$$L \rightarrow \left(\begin{array}{c||c} L_{TL}^T & L_{BR}^T \\ \hline\hline 0 & L_{BR}^T \end{array} \right)$$
where L_{BR}^T **is** 0×0
do until L_{TL}^T **is** 0×0
determine block size b
repartition
$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array} \right)$$
where B_1 **has** b **rows**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22}^T \end{array} \right)$$
where L_{11} **is** $b \times b$

$B_1 \leftarrow B_1 - L_{21}^T B_2$
$B_1 \leftarrow L_{11}^{-T} B_1$

---

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

Figure 15.3: Unblocked (left) and blocked (right) lazy triangular matrix-matrix multiplication algorithms.

```
#include "FLAME.h"

int Trsm_left_lower_trans_unb ( int variant, FLA_Obj A, FLA_Obj B )

{
  FLA_Obj     ATL, ATR,     A00, a01, A02,          BT,           B0,
              ABL, ABR,     a10t, alpha11, a12t,    BB,           b1t,
                            A20, a21, A22,                       B2;
  int         b;

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

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

  while ( 0 != FLA_Obj_length( ATR )){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &a01, /**/ &A02,
                                /**/              &a10t, &alpha11, /**/ &a12t,
                        /* ************* */    /* ********************* */
                           ABL, /**/ ABR,         &A20, &a21, /**/ &A22,
                           1, 1, /* A11 from */ FLA_TL );

    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                                                 &b1t,
                          /**/                   /**/
                           BB,                   &B2,
                           1, /* length B1 from */ FLA_TOP );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemv(FLA_TRANSPOSE, MINUS_ONE, B2,a21, ONE, b1t );

    FLA_Inv_scal( alpha11,b1t );

    if ( variant == FLA_VARIANT_ROWLAZY )
      FLA_Ger(MINUS_ONE, a10t, b1t, B0 );

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

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00, /**/ a01, A02,
                            /* ************** */    /* ****************** */
                                    /**/               a10t, /**/ alpha11, a12t,
                              &ABL, /**/ &ABR,         A20, /**/ a21, A22,
                              /* A11 added to */ FLA_BR );

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                             /***/                    /**/
                                                       b1t,
                              &BB,                     B2,
                              /* B1  added to */ FLA_BOTTOM );
  }
}
```

Figure 15.4: Unblocked lazy and row-lazy (w.r.t. L) triangular solve with multiple right-hand-sides using FLAME.

```
#include "FLAME.h"

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

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,
                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_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, &A01, /**/ &A02,
                                /**/              &A10, &A11, /**/ &A12,
                        /* ************* */    /* ******************** */
                           ABL, /**/ ABR,         &A20, &A21, /**/ &A22,
                           b, b, /* A11 from */ FLA_TL );

    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                                                 &B1,
                          /**/                   /**/
                           BB,                   &B2,
                           b, /* length B1 from */ FLA_TOP );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, B2, ONE, B1 );

    if ( rec == FLA_RECURSIVE && b > 8 )
      Trsm_left_lower_trans_blk ( variant, rec, A11, B1, nb_alg/2 );
    else
      Trsm_left_lower_trans_unb ( variant, A11, B1 );

    if ( variant == FLA_VARIANT_ROWLAZY )
      FLA_Gemm( FLA_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, B1, ONE, B0 );

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

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

    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                             /***/                   /**/
                                                       B1,
                              &BB,                     B2,
                              /* B1  added to */ FLA_BOTTOM );
  }
}
```

Figure 15.5: Blocked lazy and row-lazy (w.r.t. L) triangular solve with multiple right-hand-sides using FLAME.

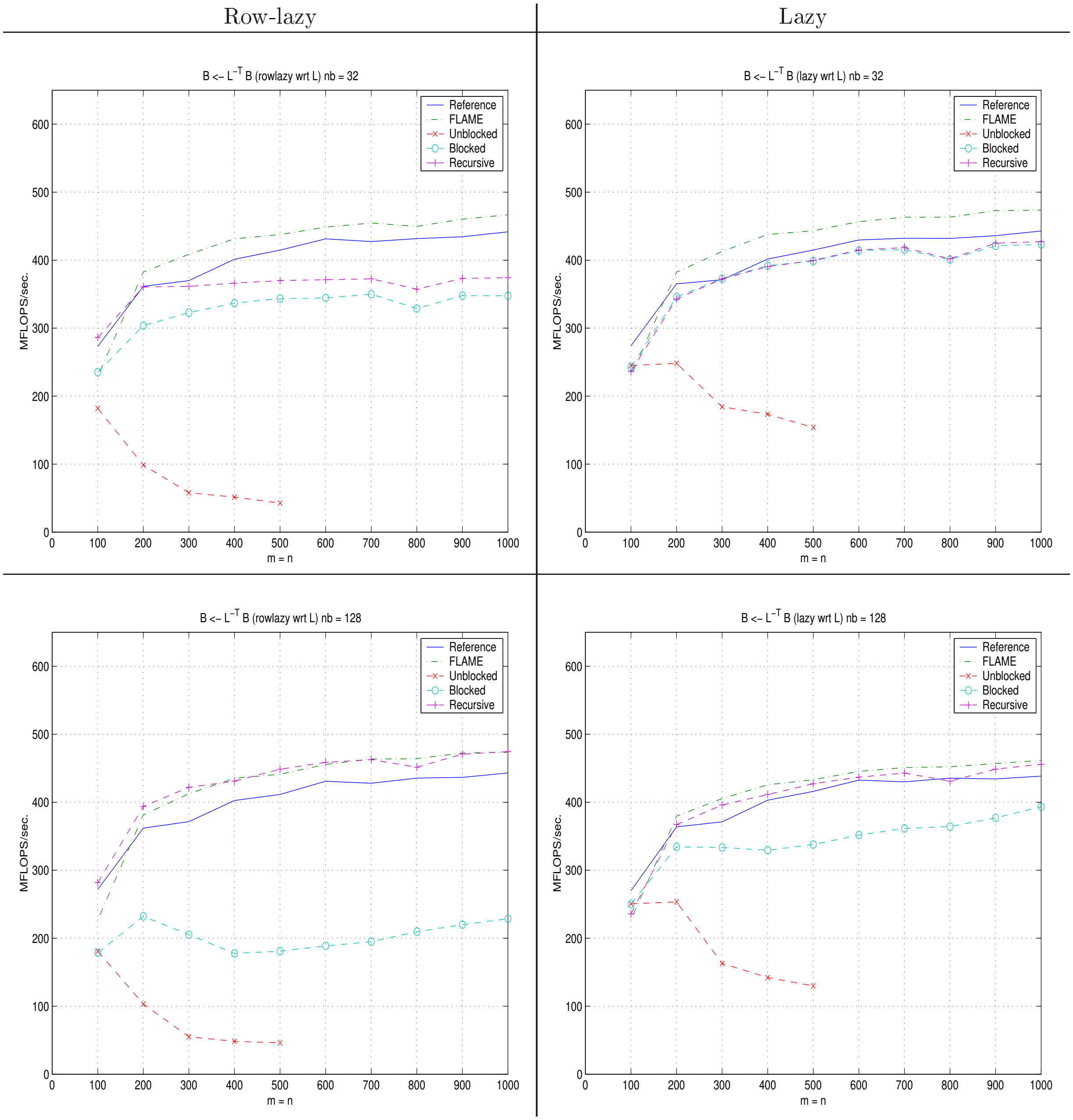


Figure 15.6: Performance of the row-lazy (left) and lazy (right) (w.r.t. C) triangular solve with multiple right-hand-sides for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Chapter 16

Triangular Solve (with Multiple RHSs) $B \leftarrow BL^{-T}$

by

C. Brandon Forehand

Christopher C. Johnson

David J. Patschke

In this chapter, we discuss the computation of the solution of a triangular system with multiple right-hand-sides:

$$B \leftarrow BL^{-T}$$

where L is a $m \times m$ lower triangular matrix and B is $n \times m$.

Let us consider

$$D = BL^{-T} \tag{16.1}$$

keeping in mind that D will overwrite B. Notice that this is equivalent to solving

$$DL^T = B \tag{16.2}$$

for matrix D. Thus the name *triangular solve with multiple right-hand-sides* (RHSs).

16.1 Algorithms that start by splitting L

Partition

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

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

$$D \rightarrow \left(\begin{array}{c||c} D_L & D_R \end{array} \right), \quad \text{and} \quad B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

where D_L and B_L have k cols.

Now, (16.2) becomes

$$\left(\begin{array}{c||c} D_L & D_R \end{array} \right) \left(\begin{array}{c||c} L_{TL}^T & L_{BL}^T \\ \hline\hline 0 & L_{BR}^T \end{array} \right) = \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$$

Multiplying out the left-hand-side yields

$$\left(\begin{array}{c\|c} D_L L_{TL}^T & D_L L_{BL}^T + D_R L_{BR}^T \end{array} \right) = \left(\begin{array}{c\|c} B_L & B_R \end{array} \right)$$

This in turn exposes the equalities

$$\begin{array}{rcl} D_L L_{TL}^T & = & B_L \\ D_L L_{BL}^T + D_R L_{BR}^T & = & B_R \end{array}$$

which must hold. Notice that these equalities are also equivalent to

$$\begin{array}{rcl} D_L & = & B_L L_{TL}^{-T} \\ D_R & = & (B_R - D_L L_{BL}^T) L_{BR}^{-T} \end{array}$$

Possible contents of D at this intermediate step are given in Fig. 16.1. Considering the comments in that table, only two viable conditions are left, the ones for which there are no comments.

D contains	Comments
$\left(\star \,\|\, \star \right)$	This indicates no progress has been made. Clearly not a desirable state.
$\left(B_L L_{TL}^{-T} \,\|\, \star \right)$	
$\left(\star \,\|\, B_R - D_L L_{BL}^T \right)$	This one is not a viable option since it requires D_L without actually storing D_L.
$\left(\star \,\|\, (B_R - D_L L_{BR}^T) L_{BR}^{-T} \right)$	This one is not a viable option since it requires D_L without actually storing D_L.
$\left(B_L L_{TL}^{-T} \,\|\, B_R - D_L L_{BL}^T \right)$	

Figure 16.1: Possible contents of D at an intermediate step.

16.1.1 Column-lazy algorithm (relative to L)

Consider the condition that currently

(16.3) $$D \quad \text{contains} \quad \left(\begin{array}{c\|c} B_L L_{TL}^{-T} & B_R - D_L L_{BL}^T \end{array} \right)$$

Notice that this indicates that L_{BL} and L_{TL} have been used to update D, which we will call a column-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 right. Thus, this algorithm naturally moves through matrices D and B in the "right" direction.

Unblocked algorithm

Repartition

$$\left(\begin{array}{c\|c} D_L & D_R \end{array} \right) \to \left(\begin{array}{c\|c|c} D_0 & d_1 & D_2 \end{array} \right)$$

where d_1 is a column. Similarly, we repartition

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

where b_1 is a column and λ_{11} is a scalar.

Notice that the double lines have meaning:

(16.4) $$\left(D_L = D_0 \,\|\, D_R = \left(\begin{array}{c|c} d_1 & D_2 \end{array} \right) \right), \left(B_L = B_0 \,\|\, B_R = \left(\begin{array}{c|c} b_1 & B_2 \end{array} \right) \right)$$

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

Considering (16.3) and these repartitionings, D currently contains

$$\begin{aligned} \left(\, B_L L_{TL}^{-T} \;\|\; B_R - D_L L_{BL}^T \,\right) &= \left(\, B_0 L_{00}^{-T} \;\|\; \left(\, b_1 \mid B_2 \,\right) - D_0 \left(\, l_{10} \mid L_{20}^T \,\right) \,\right) \\ &= \left(\, B_0 L_{00}^{-T} \;\|\; \left(\, b_1 - B_0 L_{00}^{-T} l_{10} \mid B_2 - D_0 L_{20}^T \,\right) \,\right) \end{aligned}$$

since $D_0 = B_0 L_{00}^{-T}$.

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

$$\left(\, D_L = \left(\, D_0 \mid d_1 \,\right) \;\|\; D_R = D_2 \,\right), \left(\, B_L = \left(\, B_0 \mid b_1 \,\right) \;\|\; B_R = B_2 \,\right) \tag{16.5}$$

$$\text{and} \quad \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right) & 0 \\ \hline\hline L_{BL} = \left(\, L_{20} \mid l_{21} \,\right) & L_{BR} = L_{22} \end{array}\right)$$

After the double lines move (in other words, at the top of the loop during the next iteration), the contents of D must equal

$$\begin{aligned} \left(\, B_L L_{TL}^{-T} \;\|\; B_R - D_L L_{BL}^T \,\right) &= \left(\, \left(\, B_0 \mid b_1 \,\right) \left(\begin{array}{c|c} L_{00}^T & l_{10} \\ \hline 0 & \lambda_{11} \end{array}\right)^{-1} \;\Bigg\|\; B_2 - \left(\, D_0 \mid d_1 \,\right) \left(\begin{array}{c} L_{20}^T \\ \hline l_{21}^T \end{array}\right) \,\right) \\ &= \left(\, \left(\, B_0 \mid b_1 \,\right) \left(\begin{array}{c|c} L_{00}^{-T} & -L_{00}^{-T} l_{10}/\lambda_{11} \\ \hline 0 & 1/\lambda_{11} \end{array}\right) \;\Bigg\|\; B_2 - \left(\, D_0 \mid d_1 \,\right) \left(\begin{array}{c} L_{20}^T \\ \hline l_{21}^T \end{array}\right) \,\right) \\ &= \left(\, \left(\, B_0 L_{00}^{-T} \mid -B_0 L_{00}^{-T} l_{10}/\lambda_{11} + b_1/\lambda_{11} \,\right) \;\Bigg\|\; B_2 - \left(\, D_0 \mid d_1 \,\right) \left(\begin{array}{c} L_{20}^T \\ \hline l_{21}^T \end{array}\right) \,\right) \\ &= \left(\, \left(\, B_0 L_{00}^{-T} \mid d_1 = -B_0 L_{00}^{-T} l_{10}/\lambda_{11} + b_1/\lambda_{11} \,\right) \;\|\; B_2 - (D_0 L_{20}^T + d_1 l_{21}^T) \,\right) \\ &= \left(\, \left(\, B_0 L_{00}^{-T} \mid d_1 = (b_1 - B_0 L_{00}^{-T} l_{10})/\lambda_{11} \,\right) \;\|\; (B_2 - D_0 L_{20}^T) - d_1 l_{21}^T \,\right) \end{aligned}$$

since

$$\left(\begin{array}{c|c} L_{00}^T & l_{10} \\ \hline 0 & \lambda_{11} \end{array}\right)^{-1} = \left(\begin{array}{c|c} L_{00}^{-T} & -L_{00}^{-T} l_{10}/\lambda_{11} \\ \hline 0 & 1/\lambda_{11} \end{array}\right)$$

Thus, the contents of D must be updated like:

$$\begin{gathered} \left(\, B_0 L_{00}^{-T} \;\|\; \left(\, b_1 - D_0 l_{10} \mid B_2 - D_0 L_{20}^T \,\right) \,\right) \rightarrow \\ \left(\, \left(\, B_0 L_{00}^{-T} \mid d_1 = (b_1 - B_0 L_{00}^{-T} l_{10})/\lambda_{11} \,\right) \;\|\; (B_2 - D_0 L_{20}^T) - d_1 l_{21}^T \,\right) \end{gathered} \tag{16.6}$$

Thus we conclude that an algorithm that maintains the condition in (**??**) is given in Fig. 16.2 (left). Notice that the algorithm overwrites matrix B with the result BL^{-T}.

Blocked algorithm

In order to derive a blocked algorithm, instead repartition like

$$\left(\, D_L \;\|\; D_R \,\right) \rightarrow \left(\, D_0 \;\|\; D_1 \mid D_2 \,\right)$$

where D_1 is a block of b columns. Similarly, we repartition

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

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

Algorithm 48 $B \leftarrow BL^{-T}$
(unblocked col.-lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0

repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,|\, B_2 \right)$
where b_1 **is a column**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \right)$$
where λ_{11} **is a scalar**

$b_1 \leftarrow b_1 / \lambda_{11}$
$B_2 \leftarrow B_2 - b_1 l_{21}^T$

continue with
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, b_1 \,\|\, B_2 \right)$
enddo

Algorithm 49 $B \leftarrow BL^{-T}$
(blocked col.-lazy w.r.t. L)

partition
$B \rightarrow \left(B_L \,\|\, B_R \right)$
where B_L **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0
determine block size b
repartition
$\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$
where B_1 **has** b **columns**
repartition
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$
where L_{11} **is** $b \times b$

$B_1 \leftarrow B_1 L_{11}^{-T}$
$B_2 \leftarrow B_2 - B_1 L_{21}^T$

continue with
$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array} \right)$$
continue with
$\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$
enddo

Figure 16.2: Unblocked (left) and blocked (right) column-lazy triangular solution with multiple right-hand-sides algorithms.

Again, the double lines have meaning:

$$(16.7)\qquad \left(\, D_L = D_0 \;\Big\|\; D_R = \left(\, D_1 \;\big|\; D_2 \,\right) \,\right), \left(\, B_L = B_0 \;\Big\|\; B_R = \left(\, B_1 \;\big|\; B_2 \,\right) \,\right)$$

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

Considering (16.3) and these repartitionings, D currently contains

$$\begin{array}{rcl} \left(\, B_L L_{TL}^{-T} \;\big\|\; B_R - D_L L_{BL}^T \,\right) & = & \left(\, B_0 L_{00}^{-T} \;\big\|\; \left(\, B_1 \;|\; B_2 \,\right) - D_0 \left(\, L_{10} \;|\; L_{20} \,\right)^T \,\right) \\ & = & \left(\, B_0 L_{00}^{-T} \;\big\|\; \left(\, B_1 - D_0 L_{10}^T \;|\; B_2 - D_0 L_{20}^T \,\right) \,\right) \end{array}$$

since $D_0 = B_0 L_{00}^{-T}$.

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

$$(16.8)\qquad \left(\, D_L = \left(\, D_0 \;\big|\; D_1 \,\right) \;\Big\|\; D_R = D_2 \,\right), \left(\, B_L = \left(\, B_0 \;\big|\; B_1 \,\right) \;\Big\|\; B_R = B_2 \,\right)$$

$$\text{and} \left(\begin{array}{c||c} L_{TL} = \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array} \right) & 0 \\ \hline\hline L_{BL} = \left(\, L_{20} \;|\; L_{21} \,\right) & L_{BR} = L_{22} \end{array} \right)$$

Thus, the contents of D must be updated like:

$$\begin{array}{rcl} \left(\, B_L L_{TL}^{-T} \;\big\|\; B_R - D_L L_{BL}^T \,\right) & = & \left(\left(\, B_0 \;|\; B_1 \,\right) \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array} \right)^{-T} \;\Bigg\|\; B_2 - \left(\, D_0 \;|\; D_1 \,\right) \left(\, L_{20} \;|\; L_{21} \,\right)^T \right) \\ & = & \left(\left(\, B_0 \;|\; B_1 \,\right) \left(\begin{array}{c|c} L_{00}^T & L_{10}^T \\ \hline 0 & L_{11}^T \end{array} \right)^{-1} \;\Bigg\|\; B_2 - \left(\, D_0 \;|\; D_1 \,\right) \left(\begin{array}{c} L_{20}^T \\ \hline L_{21}^T \end{array} \right) \right) \\ & = & \left(\left(\, B_0 \;|\; B_1 \,\right) \left(\begin{array}{c|c} L_{00}^{-T} & -L_{00}^{-T} L_{10}^T L_{11}^{-T} \\ \hline 0 & L_{11}^{-T} \end{array} \right) \;\Bigg\|\; B_2 - (D_0 L_{20}^T + D_1 L_{21}^T \right) \\ & = & \left(\left(\, B_0 L_{00}^{-T} \;\big|\; -B_0 L_{00}^{-T} L_{10}^T L_{11}^{-T} + B_1 L_{11}^{-T} \,\right) \;\big\|\; B_2 - D_0 L_{20}^T - D_1 L_{21}^T \,\right) \\ & = & \left(\left(\, B_0 L_{00}^{-T} \;\big|\; -D_0 L_{10}^T L_{11}^{-T} + B_1 L_{11}^{-T} \,\right) \;\big\|\; B_2 - D_0 L_{20}^T - D_1 L_{21}^T \,\right) \\ & = & \left(\left(\, B_0 L_{00}^{-T} \;\big|\; (B_1 - D_0 L_{10}^T) L_{11}^{-T} \,\right) \;\big\|\; (B_2 - D_0 L_{20}^T) - D_1 L_{21}^T \,\right) \end{array}$$

Thus, the contents of D must be updated like:

$$(16.9)\qquad \left(\, B_0 L_{00}^{-T} \;\big\|\; \left(\, B_1 - D_0 L_{10}^T \;\big|\; B_2 - D_0 L_{20}^T \,\right) \,\right) \rightarrow$$

$$\left(\left(\, B_0 L_{00}^{-T} \;\big|\; (B_1 - D_0 L_{10}^{-T}) L_{11}^{-T} \,\right) \;\big\|\; (B_2 - D_0 L_{20}^T) - D_1 L_{21}^T) \,\right)$$

Thus we conclude that an algorithm that maintains the condition in (**??**) is given in Fig. 16.2 (left). Notice that the algorithm overwrites matrix B with the result BL^{-T}.

16.1.2 Lazy algorithm (relative to L)

Next, consider the condition that currently

$$(16.10)\qquad D \quad \text{contains} \quad \left(\, B_L L_{TL}^{-T} \;\big\|\; * \,\right)$$

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

Again, this algorithm naturally moves through matrices D and B in the "right" direction.

Unblocked algorithm

Repartition

$$\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,\|\, d_1 \mid D_2 \right)$$

where d_1 is a column. Similarly, we repartition

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

where b_1 is a column and λ_{11} is a scalar.

Again, the double lines have the same meaning as illustrated in ???. Considering, ??? and these repartitionings, D currently contains

$$\left(B_L L_{TL}^{-T} \,\|\, * \right) = \left(B_0 L_{00}^{-T} \,\|\, * \right)$$

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the same meaning in ??? and D must be updated to contain

$$\begin{aligned}
\left(B_0 L_{00}^{-T} \,\|\, * \right) &= \left(\left(B_0 \mid b_1 \right) \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right)^{-T} \,\Big\|\, * \right) \\
&= \left(\left(B_0 \mid b_1 \right) \left(\begin{array}{c|c} L_{00}^T & l_{10} \\ \hline 0 & \lambda_{11} \end{array}\right)^{-1} \,\Big\|\, * \right) \\
&= \left(\left(B_0 \mid b_1 \right) \left(\begin{array}{c|c} L_{00}^{-T} & -L_{00}^{-T} l_{10}/\lambda_{11} \\ \hline 0 & 1/\lambda_{11} \end{array}\right) \,\Big\|\, * \right) \\
&= \left(\left(B_0 L_{00}^{-T} \mid -B_0 L_{00}^{-T} l_{10}/\lambda_{11} + b_1/\lambda_{11} \right) \,\|\, * \right) \\
&= \left(\left(B_0 L_{00}^{-T} \mid (b_1 - B_0 L_{00}^{-T} l_{10})/\lambda_{11} \right) \,\|\, * \right)
\end{aligned}$$

Thus, the contents of D must be updated like:

$$\left(B_0 L_{00}^{-T} \,\|\, \left(* \mid * \right) \right) \rightarrow \left(\left(B_0 L_{00}^{-T} \mid (b_1 - B_0 L_{00}^{-T} l_{10})/\lambda_{11} \right) \,\|\, * \right)$$

Thus we conclude that an algorithm that maintains the condition in (**??**) is given in Fig. 16.2 (left). Notice that the algorithm overwrites matrix B with the result BL^{-T}.

Blocked algorithm

Repartition

$$\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,\|\, D_1 \mid D_2 \right)$$

where D_1 is a block of b columns. Similarly, we repartition

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

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

Again, the double lines have the same meaning as illustrated in ???. Considering, ??? and these repartitionings, D currently contains

$$\left(B_L L_{TL}^{-T} \,\|\, * \right) = \left(B_0 L_{00}^{-T} \,\|\, * \right)$$

After moving the double lines ahead, in preparation of the next iteration, the partitioning takes on the same meaning in ??? and D must be updated to contain

$$
\begin{aligned}
\left(\begin{array}{c||c} B_0 L_{00}^{-T} & * \end{array} \right) &= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array} \right)^{-T} & * \end{array} \right) \\
&= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} L_{00}^T & L_{10}^T \\ \hline 0 & L_{11}^T \end{array} \right)^{-1} & * \end{array} \right) \\
&= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right) \left(\begin{array}{c|c} L_{00}^{-T} & -L_{00}^{-T} L_{10}^T L_{11}^{-T} \\ \hline 0 & L_{11}^{-T} \end{array} \right) & * \end{array} \right) \\
&= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 L_{00}^{-T} & -B_0 L_{00}^{-T} L_{10}^T L_{11}^{-T} + B_1 L_{11}^{-T} \end{array} \right) & * \end{array} \right) \\
&= \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 L_{00}^{-T} & (B_1 - D_0 L_{10}^T) L_{11}^{-T} \end{array} \right) & * \end{array} \right)
\end{aligned}
$$

Thus, the contents of D must be updated like:

$$
\left(\begin{array}{c||c} B_0 L_{00}^{-T} & \left(\begin{array}{c|c} * & * \end{array} \right) \end{array} \right) \rightarrow \left(\begin{array}{c||c} \left(\begin{array}{c|c} B_0 L_{00}^{-T} & (B_1 - D_0 L_{10}^{-T}) L_{11}^{-T} \end{array} \right) & * \end{array} \right)
$$

Thus we conclude that an algorithm that maintains the condition in (**??**) is given in Fig. 16.2 (left). Notice that the algorithm overwrites matrix B with the result BL^{-T}.

Algorithm 50 $B \leftarrow BL^{-T}$
(unblocked lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$
where B_L **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0

repartition
$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} B_0 & b_1 & B_2 \end{array} \right)$
where b_1 **is a column**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array} \right)$
where λ_{11} **is a scalar**

$b_1 \leftarrow (b_1 - B_0 l_{10}) / \lambda_{11}$

continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array} \right)$
continue with
$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} B_0 & b_1 & B_2 \end{array} \right)$
enddo

Algorithm 51 $B \leftarrow BL^{-T}$
(blocked lazy w.r.t. L)

partition
$B \rightarrow \left(\begin{array}{c||c} B_L & B_R \end{array} \right)$
where B_L **has** 0 **columns**
partition
$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$
where L_{TL} **is** 0×0
do until L_{BR} **is** 0×0
determine block size b
repartition
$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} B_0 & B_1 & B_2 \end{array} \right)$
where B_1 **has** b **columns**
repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$
where L_{11} **is** $b \times b$

$B_1 \leftarrow (B_1 - B_0 L_{10}^T) L_{11}^{-T}$

continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array} \right)$
continue with
$\left(\begin{array}{c||c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} B_0 & B_1 & B_2 \end{array} \right)$
enddo

Figure 16.3: Unblocked (left) and blocked (right) lazy triangular solution with multiple right-hand-sides algorithms.

16.2 Implementation

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

`http://www.cs.utexas.edu/users/flame/materials/trsm_rlt/`

16.3 Performance

```
#include "FLAME.h"

void Trsm_right_lower_trans_unb( int variant, FLA_Obj A, FLA_Obj B )
{
  FLA_Obj     ATL, ATR,     A00,  a01,     A02,    BL, BR,   B0, b1, B2,
              ABL, ABR,     a10t, alpha11, a12t,
                            A20,  a21,     A22;
  int         b;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                   /* ************** */
                      &ABL, /**/ &ABR,
                 0, 0,     /* submatrix */ FLA_TL );

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



  while ( 0 != FLA_Obj_length( ABR ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/  &a01,     &A02,
                        /* ************* */    /* ************************** */
                                 /**/              &a10t, /**/ &alpha11, &a12t,
                           ABL, /**/ ABR,         &A20, /**/  &a21,     &A22,
                           1, 1, /* alpha11 from */ FLA_BR );

    FLA_Repart_1x2_to_1x3( BL, /**/ BR,           &B0,  /**/ &b1,  &B2,
                           1, /* width B1 from */ FLA_RIGHT );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemv( FLA_NO_TRANSPOSE, MINUS_ONE, B0, a10t, ONE, b1 );

    FLA_Inv_scal( alpha11, b1 );

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Ger( MINUS_ONE, b1, a21, B2 );

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

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,        A00,  a01,     /**/ A02,
                                     /**/              a10t, alpha11, /**/ a12t,
                            /* ************** */   /* ********************* */
                              &ABL, /**/ &ABR,        A20,  a21,     /**/ A22,
                              /* alpha11 added to */ FLA_TL );

    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,           B0,  b1,  /**/ B2,
                              /* b1  added to */ FLA_LEFT );
  }
}
```

Figure 16.4: Unblocked lazy and column-lazy (w.r.t. L) triangular solve with multiple right-hand-sides using FLAME.

```
#include "FLAME.h"

void Trsm_right_lower_trans_blk( int variant, int rec, FLA_Obj A, FLA_Obj B, int nb_alg )
{
  FLA_Obj     ATL, ATR,     A00, A01, A02,    BL,  BR,  B0,  B1,  B2,
              ABL, ABR,     A10, A11, A12,
                            A20, A21, A22;
  int         b;
  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,
                0, 0,     /* submatrix */ FLA_TL );

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



  while ( b = min( FLA_Obj_length( ABR ), nb_alg ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                        /* ************* */    /* ********************* */
                                /**/              &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
                           b, b, /* A11 from */ FLA_BR );

    FLA_Repart_1x2_to_1x3( BL, /**/ BR,           &B0,  /**/ &B1,  &B2,
                           b, /* width B1 from */ FLA_RIGHT );

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

    if ( variant == FLA_VARIANT_LAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, MINUS_ONE, B0, A10, ONE, B1 );

    if ( rec == FLA_RECURSIVE && b > 8 )
        Trsm_right_lower_trans_blk( variant, rec, A11, B1 , nb_alg/2 );
    else
      Trsm_right_lower_trans_unb( variant, A11, B1 );

    if ( variant == FLA_VARIANT_COLLAZY )
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, MINUS_ONE, B1, A21, ONE, B2 );

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

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

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

Figure 16.5: Blocked lazy and column-lazy (w.r.t. L) triangular solve with multiple right-hand-sides using FLAME.

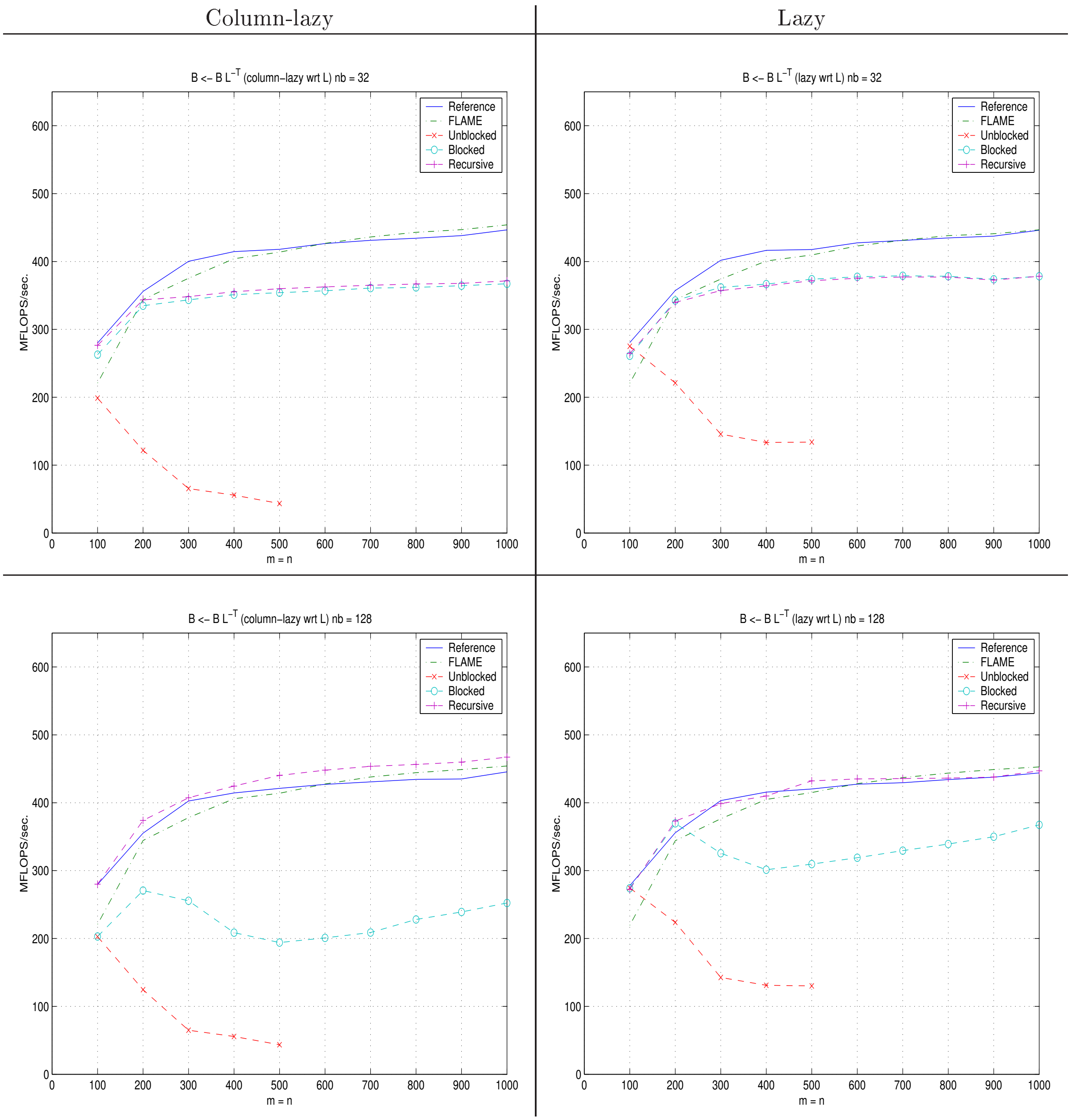


Figure 16.6: Performance of the column-lazy (left) and lazy (right) (w.r.t. C) triangular solve with multiple right-hand-sides for a block size of 32 (top) and 128 (bottom). For these experiments, the ITXGEMM matrix-matrix multiplication kernel was used.

Appendix A

Basic Linear Algebra Operations

A.1 Operations on vectors and matrices

Scaling of a vector

$$\alpha x = \alpha \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix} = \begin{pmatrix} \alpha\chi_1 \\ \alpha\chi_2 \\ \vdots \\ \alpha\chi_n \end{pmatrix}$$

Adding vectors

$$x + y = \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix} + \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix} = \begin{pmatrix} \chi_1 + \eta_1 \\ \chi_2 + \eta_2 \\ \vdots \\ \chi_n + \eta_n \end{pmatrix}$$

Transposition

Definition 1 *Let $A \in \mathbf{R}^{m \times n}$ and $B \in \mathbf{R}^{n \times m}$. Then B is said to be the* transpose *of A if $\beta_{ij} = \alpha_{ji}$ for all $i, i = 1, \ldots, n$ and $j, j = 1, \ldots, m$. Notations:*

$$B = A^T$$

Notice: if $B = A^T$ then $A = B^T$.

Inner (dot) product

The *inner product* of two vectors $x, y \in \mathbf{R}^n$ is given by

$$\sum_{k=1}^{n} \chi_i \eta_i = \chi_1\eta_1 + \chi_2\eta_2 + \cdots + \chi_n\eta_n$$

Scalar times a matrix

$$\beta A = \beta \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{pmatrix} = \begin{pmatrix} \beta\alpha_{11} & \beta\alpha_{12} & \cdots & \beta\alpha_{1n} \\ \beta\alpha_{21} & \beta\alpha_{22} & \cdots & \beta\alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \beta\alpha_{m1} & \beta\alpha_{m2} & \cdots & \beta\alpha_{mn} \end{pmatrix}$$

Matrix times a vector

$$
\begin{aligned}
Ax &= \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{pmatrix} \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix} \\
&= \chi_1 \begin{pmatrix} \alpha_{11} \\ \alpha_{21} \\ \vdots \\ \alpha_{n1} \end{pmatrix} + \chi_2 \begin{pmatrix} \alpha_{12} \\ \alpha_{22} \\ \vdots \\ \alpha_{n2} \end{pmatrix} + \cdots + \chi_n \begin{pmatrix} \alpha_{1n} \\ \alpha_{2n} \\ \vdots \\ \alpha_{nn} \end{pmatrix} \\
&= \begin{pmatrix} \alpha_{11}\chi_1 + \alpha_{12}\chi_2 + \cdots + \alpha_{1n}\chi_n \\ \alpha_{21}\chi_1 + \alpha_{22}\chi_2 + \cdots + \alpha_{2n}\chi_n \\ \vdots \\ \alpha_{m1}\chi_1 + \alpha_{m2}\chi_2 + \cdots + \alpha_{mn}\chi_n \end{pmatrix}
\end{aligned}
$$

Matrix times a matrix

$$
\begin{aligned}
AB &= \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1k} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2k} \\ \vdots & \vdots & & \vdots \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mk} \end{pmatrix} \begin{pmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1n} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2n} \\ \vdots & \vdots & & \vdots \\ \beta_{k1} & \beta_{k2} & \cdots & \beta_{kn} \end{pmatrix} \\
&= \left(\begin{array}{c|c|c|c} A \begin{pmatrix} \beta_{11} \\ \beta_{21} \\ \vdots \\ \beta_{k1} \end{pmatrix} & A \begin{pmatrix} \beta_{12} \\ \beta_{22} \\ \vdots \\ \beta_{k2} \end{pmatrix} & \cdots & A \begin{pmatrix} \beta_{1n} \\ \beta_{2n} \\ \vdots \\ \beta_{kn} \end{pmatrix} \end{array} \right) \\
&= \left(\begin{array}{c|c|c|c} \alpha_{11}\beta_{11} + \cdots + \alpha_{1k}\beta_{k1} & \alpha_{11}\beta_{12} + \cdots + \alpha_{1k}\beta_{k2} & \cdots & \alpha_{11}\beta_{1n} + \cdots + \alpha_{1k}\beta_{kn} \\ \hline \alpha_{21}\beta_{11} + \cdots + \alpha_{2k}\beta_{k1} & \alpha_{21}\beta_{12} + \cdots + \alpha_{2k}\beta_{k2} & \cdots & \alpha_{21}\beta_{1n} + \cdots + \alpha_{2k}\beta_{kn} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m1}\beta_{11} + \cdots + \alpha_{mk}\beta_{k1} & \alpha_{m1}\beta_{12} + \cdots + \alpha_{mk}\beta_{k2} & \cdots & \alpha_{m1}\beta_{1n} + \cdots + \alpha_{mk}\beta_{kn} \end{array} \right)
\end{aligned}
$$

Note: $c_{ij} = \sum_{l=1}^{k} \alpha_{il}\beta_{jl}$

Note: A (column) vector of length n can be viewed as a $n \times 1$ matrix. A column vector of length n can be viewed as a $1 \times n$ matrix. The innerproduct of two vectors of length n, x and y, can thus be written as

$$x^T y = \sum_{i=1} \chi_i \eta_i$$

Blocked matrix-matrix multiplication

A 2×2 blocked partitioning of a given $m \times n$ matrix A is given by

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right)$$

where

$$
\begin{aligned}
A_{11} &\in \mathbf{R}^{m_1 \times n_1} \\
A_{21} &\in \mathbf{R}^{m_2 \times n_1} \\
A_{12} &\in \mathbf{R}^{m_1 \times n_2} \\
A_{22} &\in \mathbf{R}^{m_2 \times n_2}
\end{aligned}
$$

with $m_1 + m_2 = m$ and $n_1 + n_2 = n$.

Let A, B, and C all be partitioned so that

$$C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right), \quad A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \quad \text{and} B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

with

$$\begin{array}{rcl rcl rcl} C_{11} & \in & \mathbf{R}^{m_1 \times n_1} & A_{11} & \in & \mathbf{R}^{m_1 \times k_1} & C_{11} & \in & \mathbf{R}^{k_1 \times n_1} \\ C_{21} & \in & \mathbf{R}^{m_2 \times n_1} & A_{21} & \in & \mathbf{R}^{m_2 \times k_1} & C_{21} & \in & \mathbf{R}^{k_2 \times n_1} \\ C_{12} & \in & \mathbf{R}^{m_1 \times n_2} & A_{12} & \in & \mathbf{R}^{m_1 \times k_2} & C_{12} & \in & \mathbf{R}^{k_1 \times n_2} \\ C_{22} & \in & \mathbf{R}^{m_2 \times n_2} & A_{22} & \in & \mathbf{R}^{m_2 \times k_2} & C_{22} & \in & \mathbf{R}^{k_2 \times n_2} \end{array}$$

and $m_1 + m_2 = m$, $n_1 + n_2 = n$, and $k_1 + k_2 = k$. Then

$$C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right)$$

Note: Blocked matrix-matrix multiplication is just like regular matrix-matrix multiplication, except with blocks instead of scalars.

Generalization

A $M \times N$ blocked partitioning of a given $m \times n$ matrix A is given by

$$A = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \cdots & A_{1N} \\ \hline A_{21} & A_{22} & \cdots & A_{2N} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M1} & A_{M2} & \cdots & A_{MN} \end{array}\right)$$

where

$$A_{ij} \in \mathbf{R}^{m_i \times n_j}$$

with $m_1 + m_2 + \cdots + m_M = m$ and $n_1 + n_2 + \cdots + n_N = n$.

Let A, B, and C partitioned so that C is blocked into $M \times N$ blocks, A is blocked into $M \times K$ blocks, andn B is blocked into $K \times N$ blocks, with C_{ij} of size $m_i \times n_i$, A_{ij} of size $m_i \times k_j$ and B_{ij} of size $k_i \times n_j$. Then

$$C_{ij} = \sum_{l=1}^{K} A_{il} B_{lj}$$

Again, blocked matrix-matrix multiplication is just like regular matrix-matrix multiplication, with scalars replaced by matrix blocks.

Note: Given two matrices A and B, it is generally **NOT** the case that $AB = BA$.

Blocked matrix-vector multiplication

Notice that a vector of length n can be viewed as a $n \times 1$ matrix, and thus we can apply the blocked matrix-matrix multiplication to matrix-vector multiplication: Let A be an $m \times n$ matrix, and x be a vector of length n. Block A into $M \times N$ blocks, $\{A_{ij}\}$ and x into N subvectors $\{x_j\}$. Let A_{ij} be $m_i \times n_j$ and x_j be of length n_j. Then

$$Ax = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \cdots & A_{1N} \\ \hline A_{21} & A_{22} & \cdots & A_{2N} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M1} & A_{M2} & \cdots & A_{MN} \end{array}\right) \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_n \end{array}\right) = \left(\begin{array}{c} A_{11}x_1 + \cdots + A_{1N}x_N \\ \hline A_{21}x_1 + \cdots + A_{2N}x_N \\ \hline \vdots \\ \hline A_{M1}x_1 + \cdots + A_{MN}x_N \end{array}\right)$$

A special case is when one partitions an $m \times n$ matrix by columns ($M = 1$ and $N = n$):

$$A = \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array}\right)$$

where a_j represents the *vector* that makes up the jth column. Notice for example that

$$\begin{array}{rcl} Ax & = & \left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right) \left(\begin{array}{c} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{array} \right) \\ & = & \chi_1 a_1 + \chi_2 a_2 + \cdots + \chi_n a_n \end{array}$$

Similarly, it will be convenient to partition an $m \times n$ matrix by rows ($M = m$ and $N = 1$):

$$A = \left(\begin{array}{c} \hat{a}_1^T \\ \hline \hat{a}_n^T \\ \hline \vdots \\ \hline \hat{a}_m^T \end{array} \right)$$

where $\hat{a}_i^T$ represents the *row vector* that makes up the ith row. Notice for example that

$$Ax = \left(\begin{array}{c} \hat{a}_1^T \\ \hline \hat{a}_n^T \\ \hline \vdots \\ \hline \hat{a}_m^T \end{array} \right) x = \left(\begin{array}{c} \hat{a}_1^T x \\ \hline \hat{a}_n^T x \\ \hline \vdots \\ \hline \hat{a}_m^T x \end{array} \right)$$

Rank-1 update

The operation $A \leftarrow \alpha x y^T + A$ is often called a rank-1 update, where A is $m \times n$ and x and y are vectors of length m and n, respectively. (The operation xy^T is known as an outer-product.)

Notice that

$$\begin{array}{rcl} A & \leftarrow & xy^T + A = x \left(\begin{array}{c|c|c} \eta_1 & \cdots & \eta_n \end{array} \right) + \left(\begin{array}{c|c|c} a_1 & \cdots & a_n \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} \eta_1 x + a_1 & \cdots & \eta_n x + a_n \end{array} \right) = \left(\begin{array}{c|c|c} \eta_1 \chi_1 + \alpha_{11} & \cdots & \eta_n \chi_1 + a_{1n} \\ \hline \vdots & & \vdots \\ \hline \eta_1 \chi_m + \alpha_{m1} & \cdots & \eta_n \chi_m + a_{mn} \end{array} \right) \end{array}$$

Notice that the name "rank-1 update" comes from the fact that matrix A is updated with a matrix with rank (less than or equal to) one, since each column of xy is a multiple of x.

Also,

$$\begin{array}{rcl} A \leftarrow xy^T + A & = & \left(\begin{array}{c} \chi_1 \\ \hline \vdots \\ \hline \chi_m \end{array} \right) y^T + A = \left(\begin{array}{c} \chi_1 y^T \\ \hline \vdots \\ \hline \chi_m y^T \end{array} \right) + \left(\begin{array}{c} \hat{a}_1^T \\ \hline \vdots \\ \hline \hat{a}_m^T \end{array} \right) \\ & = & \left(\begin{array}{c} \chi_1 y^T + \hat{a}_1^T \\ \hline \vdots \\ \hline \chi_m y^T + \hat{a}_m^T \end{array} \right) = \left(\begin{array}{c|c|c} \eta_1 \chi_1 + \alpha_{11} & \cdots & \eta_n \chi_1 + a_{1n} \\ \hline \vdots & & \vdots \\ \hline \eta_1 \chi_m + \alpha_{m1} & \cdots & \eta_n \chi_m + a_{mn} \end{array} \right) \end{array}$$

Appendix B

Describing FLAME algorithms in LaTeX

The following LaTeXcommands and environments can be found at

`http://www.cs.utexas.edu/users/flame/LaTeX/`

We encourage others to use them when stating linear algebra algorithms similar to the ones discussed in this document. We ask as a courtesy that when they are used for a paper or report, this document be referenced.

Partitioned matrices

<table>
<tr><td><code>$ \FlaTwoByTwo{ A_{TL} }{ A_{TR} }
             { A_{BL} }{ A_{BR} } $</code></td><td>$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaTwoByTwoSingleLine{ A_{TL} }{ A_{TR} }
                       { A_{BL} }{ A_{BR} } $</code></td><td>$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaTwoByOne{ A_{T} }
             { A_{B} }$</code></td><td>$\left(\begin{array}{c} A_{T} \\ \hline\hline A_{B} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaTwoByOneSingleLine{ A_{T} }
                       { A_{B} }$</code></td><td>$\left(\begin{array}{c} A_{T} \\ \hline A_{B} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaOneByTwo{ A_{L} }{ A_{R} } $</code></td><td>$\left(\begin{array}{c||c} A_{L} & A_{R} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaOneByTwoSingleLine{ A_{L} }{ A_{R} } $</code></td><td>$\left(\begin{array}{c|c} A_{L} & A_{R} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaThreeByThreeBR{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}} $</code></td><td>$\left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaThreeByThreeTL{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}} $</code></td><td>$\left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaThreeByOneB{A_{0}}
                 {A_{1}}
                 {A_{2}} $</code></td><td>$\left(\begin{array}{c} A_{0} \\ \hline\hline A_{1} \\ \hline A_{2} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaThreeByOneT{A_{0}}
                 {A_{1}}
                 {A_{2}} $</code></td><td>$\left(\begin{array}{c} A_{0} \\ \hline A_{1} \\ \hline\hline A_{2} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaOneByThreeR{A_{0}}{A_{1}}{A_{2}} $</code></td><td>$\left(\begin{array}{c||c|c} A_{0} & A_{1} & A_{2} \end{array}\right)$</td></tr>
<tr><td><code>$ \FlaOneByThreeL{A_{0}}{A_{1}}{A_{2}} $</code></td><td>$\left(\begin{array}{c|c||c} A_{0} & A_{1} & A_{2} \end{array}\right)$</td></tr>
</table>

Flame algorithm environment

```
\begin{FlameAlg}
statement 1 \\
statement 2 \\
\> indent statement 3 \\
\> \> double indent statement 4 \\
statement 5 \\
\end{FlameAlg}
```

statement 1
statement 2
    indent statement 3
        double indent statement 4
statement 5

```
\begin{FlameAlgNarrow}
statement 1 \\
statement 2 \\
\> indent statement 3 \\
\> \> double indent statement 4 \\
statement 5 \\
\end{FlameAlgNarrow}
```

statement 1
statement 2
    indent statement 3
        double indent statement 4
statement 5

Partition

```
\begin{FlameAlgNarrow}
\FlaPartition { $
  A \rightarrow
  \FlaTwoByTwo{ A_{TL} }{ A_{TR} }
              { A_{BL} }{ A_{BR} }
$ }
{
  \FlaWhere{ $ A_{TL} $ is $ 0 \times 0 $ }
}
\end{FlameAlgNarrow}
```

partition

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

where A_{TL} **is** 0×0

Repartition

```
\begin{FlameAlgNarrow}
\FlaRepartition{ $
  \FlaTwoByTwo{ A_{TL} }{ A_{TR} }
              { A_{BL} }{ A_{BR} } \rightarrow
  \FlaThreeByThreeBR{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}}
$ }
{
  \FlaWhere{$ A_{11} $ is $ b \times b $}
}
\end{FlameAlgNarrow}
```

repartition

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

where A_{11} **is** $b \times b$

Continue

```
\begin{FlameAlgNarrow}
\FlaContinue{ $
  \FlaTwoByTwo{ A_{TL} }{ A_{TR} }
              { A_{BL} }{ A_{BR} } \leftarrow
  \FlaThreeByThreeTL{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}}
$ }
\end{FlameAlgNarrow}
```

continue with

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

Delimiting Computation

```
\begin{FlameAlgNarrow}
\FlaStartCompute \\  % ******************
statement 1 \\
statement 2 \\       % Important: \\
                     % before \FlaEndCompute
\FlaEndCompute\\     % ******************
\end{FlameAlgNarrow}
```

---

statement 1
statement 2

A typical example

A typical example is given Fig. B.1.

```
\begin{FlameAlgNarrow}
\FlaPartition { $
A \rightarrow
     \FlaTwoByTwo{ A_{TL} }{ A_{TR} }
                 { A_{BL} }{ A_{BR} }
$ }
{
  \FlaWhere{ $ A_{TL} $ is $ 0 \times 0 $ }
}
\\
\FlaDoUntil{$ A_{BR} $ is $ 0 \times 0 $} \\
  \FlaRepartition{ $
  \FlaTwoByTwo{A_{TL}}{A_{TR}}
              {A_{BL}}{A_{BR}} \rightarrow
  \FlaThreeByThreeBR{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}}
  $ }
  {
    \FlaWhere{$ A_{11} $ is $ b \times b $}
  } \\
  \FlaStartCompute \\  % ******************

  $ A_{11} \leftarrow \FlaUpLo{L}{U}_{11}
          =          \mbox{LU}(A_{11})  $ \\
  $ A_{21} \leftarrow L_{21}
          = A_{21}  \FlaInverse{ U_{11} } $ \\
  $ A_{12} \leftarrow U_{12}
          = \FlaInverse{ L_{11} } A_{12}  $ \\
  $ A_{22} \leftarrow A_{22}-L_{21} U_{21} $ \\

  \FlaEndCompute\\     % ******************
  \FlaContinue{ $
  \FlaTwoByTwo{A_{TL}}{A_{TR}}
              {A_{BL}}{A_{BR}} \leftarrow
  \FlaThreeByThreeTL{A_{00}}{A_{01}}{A_{02}}
                    {A_{10}}{A_{11}}{A_{12}}
                    {A_{20}}{A_{21}}{A_{22}}
  $ } \\           % Note: it is important to
\FlaEndDo \\       % have \\ before \FlaEndDo
\end{FlameAlgNarrow}
```

partition

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

where A_{TL} **is** 0×0

do until A_{BR} **is** 0×0

repartition

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

where α_{11} **is a scalar**

$\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11}$ (no-op)
$a_{21} \leftarrow l_{21} = a_{21}/\upsilon_{11}$
$a_{12} \leftarrow u_{12} = a_{12}$ (no-op)
$A_{22} \leftarrow A_{22} - l_{21} u_{12}^T$

---

continue with

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

enddo

Figure B.1: A typical example of a FLAME algorithm. This particular algorithm computes an LU factorization of a given matrix A.

Appendix C

Computational Routines (alphabetical)

In this appendix, we give a large number of routines that are part of FLAME as of this writing. Additional routines can be found at

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

as they become available.

C.1 Constants

Transposition/Conjugation (`trans`, `transa`, `transb`)	Structure (`uplo`)	Side (`side`)	Diagonal (`diag`)

`FLA_NO_TRANSPOSE`
`FLA_TRANSPOSE`
`FLA_CONJUGATE`
`FLA_NO_CONJUGATE`
`FLA_CONJUGATE_TRANSPOSE`
	`FLA_LOWER_TRIANGULAR`
	`FLA_UPPER_TRIANGULAR`
	`FLA_LOWER_TRAPEZOIDAL`
	`FLA_LOWER_TRAPEZOIDAL`
		`FLA_LEFT`
		`FLA_RIGHT`
		`FLA_TOP`
		`FLA_BOTTOM`
		`FLA_TL`
		`FLA_TR`
		`FLA_BL`
		`FLA_BR`
			`FLA_NONUNIT_DIAGONAL`
			`FLA_UNIT_DIAGONAL`
			`FLA_ZERO_DIAGONAL`

C.2 Routines

```
void FLA_Axpy   (              FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
void FLA_Axpy_x ( int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```

Purpose: Update $B \leftarrow \alpha A + B$, $B \leftarrow \alpha A^T + B$, $B \leftarrow \alpha \texttt{conj}(A) + B$, or $B \leftarrow \alpha A^H + B$.

`trans`	indicates whether to transpose and/or conjugate the data
`alpha, A, B`	descriptors for α, A, and B

```
void FLA_Copy   (              FLA_Obj A, FLA_Obj B )
void FLA_Copy_x ( int trans, FLA_Obj A, FLA_Obj B )
```

Purpose: $B \leftarrow A$, optionally transposing and/or conjugating the data.

`trans`	indicates whether to transpose and/or conjugate the data
`A, B`	descriptors for A and B

```
void FLA_Dot   (                FLA_Obj x, FLA_Obj y,               FLA_Obj rho )
void FLA_Dot_x ( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho )
```

Purpose: Compute $\rho \leftarrow x^T y$ or $\rho \leftarrow \alpha x^T y + \beta\rho$.

`alpha, x, y, beta, rho`	descriptors for α, x, y, β, and ρ

```
void FLA_Dotc   (                FLA_Obj x, FLA_Obj y,               FLA_Obj rho )
void FLA_Dotc_x ( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho )
```

Purpose: Compute $\rho \leftarrow x^H y$ or $\rho \leftarrow \alpha x^H y + \beta\rho$.

`alpha, x, y, beta, rho`	descriptors for α, x, y, β, and ρ

```
void FLA_Gemm  ( int transA, int transB, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                                         FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha AB + \beta C$, optionally transposing and/or conjugating A and/or B.

`trans`	indicates whether to transpose and/or conjugate A and/or B
`alpha, A, B, beta, C`	descriptors for α, A, B, β, and C

```
void FLA_Gemm_x  ( int side, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```

Purpose: Update $B \leftarrow \alpha AB$ or $B \leftarrow \alpha BA$, optionally transposing and/or conjugating A.

`side`	indicates side from which A multiplies B
`trans`	indicates whether to transpose and/or conjugate A
`alpha, A, B`	descriptors for α, A, and B

```
void FLA_Gemv  ( int transA, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                             FLA_Obj beta,  FLA_Obj y )
```

Purpose: Update $y \leftarrow \alpha Ax + \beta y$, optionally transposing and/or conjugating A.

`transA`	indicates whether to transpose and/or conjugate A
`alpha, A, x, beta, y`	descriptors for α, A, x, β, and y

```
void FLA_Ger   ( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha x y^T + A$.

`alpha, x, y, A`	descriptors for α, x, y, and A

```
void FLA_Gerc   ( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha x y^C + A$.

`alpha, x, y, A`	descriptors for α, x, y, and A

```
void FLA_Iamax ( FLA_Obj x, FLA_Obj k )
```

Purpose: Compute index k such that $|\chi_k| = \|x\|_\infty$.
Note: This operation only works when x has unit row or column dimension.

`x, k`	descriptors for x and k

```
void FLA_Invert ( FLA_Obj alpha )
```

Purpose: Update $\alpha \leftarrow \frac{1}{\alpha}$.

`alpha`	descriptor for α

```
void FLA_Inv_scal ( FLA_Obj alpha, FLA_Obj A )
```

Purpose: Update $A \leftarrow \frac{1}{\alpha} A$.

`alpha, A`	descriptors for α and A

```
void FLA_LU ( FLA_Obj A, FLA_Obj ipiv )
```

Purpose: Compute the LU factorization with partial pivoting of matrix A, overwriting A with factors L and U. The pivot matrix P is stored as a vector of indices, in `ipiv`.

`A`	descriptor for A
`ipiv`	descriptor for vector in which P is stored

```
void FLA_Negate ( FLA_Obj A )
```

Purpose: Negate the data described by `A`: $A \leftarrow -A$.

`A`	descriptor of data to be updated

```
void FLA_Nrm1    ( FLA_Obj A, FLA_Obj alpha )
void FLA_Nrm_inf ( FLA_Obj A, FLA_Obj alpha )
```

Purpose: Compute $\alpha \leftarrow \|A\|_1$ or $\alpha \leftarrow \|A\|_\infty$.

`A, alpha`	descriptor for A and α

```
void FLA_Nrm2 ( FLA_Obj x, FLA_Obj alpha )
```

Purpose: Compute $\alpha \leftarrow \|x\|_2$.
Note: while the two-norm of a matrix is a commonly used quantity in theory, it is extremely expensive to compute in practice. This operation only works when x has unit row or column dimension.

`x, alpha`	descriptor for x and α

```
void FLA_Obj_set_to_one( FLA_Obj A )
```

Purpose: Set all elements of A to one: $A \leftarrow 1'$s.

`A`	descriptor for A

```
void FLA_Obj_set_to_zero( FLA_Obj A )
```

Purpose: Set all elements of A to zero: $A \leftarrow 0$.

`A`	descriptor for A

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

Purpose: Print the contents of A.

`string1`	string to be printed before contents
`A`	descriptor for A
`format`	format to be used to print each individual element
`string2`	string to be printed after contents

```
void FLA_Random_matrix( FLA_Obj A )
```

Purpose: Fill A with random values in the range $(-1, 1)$: $A \leftarrow$ `random matrix`.

`A`	A

```
void FLA_Scal     ( FLA_Obj alpha, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha A$.

`alpha, A`	descriptor for α and A

```
void FLA_Scal_x     ( int uplo, FLA_Obj alpha, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha A$, scaling only the upper or lower triangular part of the matrix.

`uplo`	indicates whether to scale upper or lower part of matrix
`alpha, A`	descriptor for α and A

```
void FLA_Set_diagonal( FLA_Obj sigma, FLA_Obj A )
```

Purpose: Set the diagonal of A to σI: $\text{diag}(A) \leftarrow \sigma I$. All other values in A are unaffected.

`sigma, A`	descriptor for σ and A

```
void FLA_Shift_diagonal( FLA_Obj sigma, FLA_Obj A )
```

Purpose: Shift the diagonal by σ: $A \leftarrow A - \sigma I$.

`sigma, A`	descriptor for σ and A

```
void FLA_Sqrt ( FLA_Obj alpha )
```

Purpose: Update the data described by `A` with its square root: $\alpha \leftarrow \sqrt{\alpha}$. Note: `A` must describe a scalar, i.e., its row and column dimension must equal one.

`A`	descriptor of data to be updated

```
void FLA_Swap   (            FLA_Obj A, FLA_Obj B )
void FLA_Swap_x ( int trans, FLA_Obj A, FLA_Obj B )
```

Purpose: $B \leftrightarrow A$, optionally transposing and/or conjugating the data.

`trans`	indicates whether to transpose and/or conjugate the data
`A, B`	descriptors for A and B

```
FLA_Symm  ( int side, int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                                FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where `side` indicates the side from which A multiplies B. `uplo` indicates whether A is stored in the upper or lower triangular part of `A`.

`side`	indicates side from which A multiplies B
`uplo`	indicates where A is stored
`alpha, A, B, beta, C`	descriptors for α, A, B, β, and C

```
void FLA_Symmetrize( int uplo, int conj, FLA_Obj A )
```

Purpose: $A \leftarrow \text{symm}(A)$ or $A \leftarrow \text{herm}(A)$.

`uplo`	indicates where A is stored.
`conj`	indicates whether to create a symmetric or hermitian matrix.
`A`	descriptor for matrix A.

```
FLA_Symv  ( int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                      FLA_Obj beta,  FLA_Obj y )
```

Purpose: Update $y \leftarrow \alpha Ax + \beta y$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo` .

`uplo`	indicates where A is store
`alpha, A, x, beta, y`	descriptors for α, A, x, β, and y

```
FLA_Syr   ( int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha x x^T + A$.

`uplo`	indicates in which triangle symmetric A is stored
`alpha, x, A`	descriptors for α, x, and A

```
void FLA_Syr2  ( int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A )
```

Purpose: Update $A \leftarrow \alpha x y^T + \alpha y x^T + A$.

`uplo`	indicates in which triangle symmetric A is stored
`alpha, x, y, A`	descriptors for α, x, y, and A

```
void FLA_Syr2k  ( int uplo, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                                       FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha(AB^T + BA^T) + \beta C$ (if `trans == FLA_NO_TRANSPOSE`) or $C \leftarrow \alpha(B^T A + A^T B) + \beta C$ (if `trans == FLA_TRANSPOSE`). `uplo` indicates whether C is stored in the upper or lower triangular part of `C`.

`uplo`	indicates where C is stored
`trans`	indicates which whether to transpose and/or conjugate A
`alpha, A, B, beta, C`	descriptors for α, A, B, β, C

```
void FLA_Syrk  ( int uplo, int trans, FLA_Obj alpha, FLA_Obj A,
                                      FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha A A^T + \beta C$ (if `trans == FLA_NO_TRANSPOSE`) or $C \leftarrow \alpha A^T A + \beta C$ (if `trans == FLA_TRANSPOSE`). `uplo` indicates whether C is stored in the upper or lower triangular part of `C`.

`uplo`	indicates where A is stored
`trans`	indicates which whether to transpose and/or conjugate A
`alpha, A, beta, C`	descriptors for α, A, β, C

```
void FLA_Triangularize( int uplo, int diag, FLA_Obj A )
```

Purpose: $A \leftarrow \text{lower}(A)$ or $A \leftarrow \text{upper}(A)$. If `uplo == FLA_UPPER_TRIANGULAR`, set the lower triangular part of A to zero. If `uplo == FLA_LOWER_TRIANGULAR`, set the lower triangular part of A to zero. Parameter `diag` indicates whether to set the diagonal to zero (`FLA_ZERO_DIAGONAL`), one (`FLA_UNIT_DIAGONAL`), or leave it alone (`FLA_NONUNIT_DIAGONAL`).

`uplo`	indicates whether whether A should be upper of lower triangular.
`diag`	indicates whether set diagonal to zero or one.
`A`	descriptor for matrix A.

```
void FLA_Trmm  ( int side, int uplo, int transa, int diag,
                 FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```

Purpose: Update $B \leftarrow \alpha AB$ or $B \leftarrow \alpha BA$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `side` indicates whether to multiply B by A from the left or right. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`side`	indicates side from which A multiplies B
`uplo`	indicates whether A is upper or lower triangular
`trans`	indicates which whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, B`	descriptors for α, A, and B

```
void FLA_Trmm_x  ( int side, int uplo, int transa, int diag,
                   FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                   FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `side` indicates whether to multiply B by A from the left or right. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`side`	indicates side from which A multiplies B
`uplo`	indicates whether A is upper or lower triangular
`trans`	indicates which whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, B, beta, C`	descriptors for α, A, B, β, C

```
void FLA_Trmv  ( int uplo, int transA, int diag, FLA_Obj A, FLA_Obj x )
```

Purpose: Update $x \leftarrow Ax$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`uplo`	indicates whether A is upper or lower triangular
`transA`	indicates whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`A, x`	descriptors for A and x

```
void FLA_Trmv_x  ( int uplo, int transA, int diag,
                   FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                   FLA_Obj beta,  FLA_Obj y )
```

Purpose: Update $y \leftarrow \alpha Ax + \beta y$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`uplo`	indicates whether A is upper or lower triangular
`transA`	indicates whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, x, beta, y`	descriptors for α, A, β, and y

```
void FLA_Trsm  ( int side, int uplo, int transa, int diag,
                 FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```

Purpose: Update $B \leftarrow \alpha A^{-1}B$ or $B \leftarrow \alpha BA^{-1}$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed and/or conjugated. `side` indicates whether to multiply B by A^{-1} from the left or right. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`side`	indicates side from which A^{-1} multiplies B
`uplo`	indicates whether A is upper or lower triangular
`trans`	indicates which whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, B`	descriptors for α, A, and B

```
void FLA_Trsm_x  ( int side, int uplo, int transa, int diag,
                   FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                   FLA_Obj beta,  FLA_Obj C )
```

Purpose: Update $C \leftarrow \alpha A^{-1} B + \beta C$ or $C \leftarrow \alpha B A^{-1} + \beta C$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `side` indicates whether to multiply B by A^{-1} from the left or right. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`side`	indicates side from which A^{-1} multiplies B
`uplo`	indicates whether A is upper or lower triangular
`trans`	indicates which whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, B, beta, C`	descriptors for α, A, B, β, C

```
void FLA_Trsv  ( int uplo, int transA, int diag, FLA_Obj A, FLA_Obj x )
```

Purpose: Update $x \leftarrow A^{-1} x$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`uplo`	indicates whether A is upper or lower triangular
`transA`	indicates whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`A, x`	descriptors for A and x

```
void FLA_Trsv_x  ( int uplo, int transA, int diag,
                   FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                   FLA_Obj beta,  FLA_Obj y )
```

Purpose: Update $y \leftarrow \alpha A^{-1} x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`, and is optionally transposed. `diag` indicated whether the diagonal of A is to be taken to equal unity or equal to the stored values.

`uplo`	indicates whether A is upper or lower triangular
`transA`	indicates whether to transpose and/or conjugate A
`diag`	indicates whether A has unit diagonal
`alpha, A, x, beta, y`	descriptors for α, A, β, and y

Bibliography

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), Sept. 1994.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[3] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0*. SIAM, 1994.

[5] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert van de Geijn. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98*, 1998.

[6] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*. ACM SIGARC, July 1997.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[9] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience*, to appear.

[10] Edsger Wybe Dijkstra. Under the spell of Leibniz's dream. Technical Report EWD1298, The University of Texas at Austin, April 2000. `http://www.cs.utexas.edu/users/EWD/`.

[11] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[13] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[14] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer Verlag, 1992.

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

[16] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[17] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer, 2001.

[18] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.

[19] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. Parallel out-of-core cholesky and qr factorizations with pooclapack. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2001.

[20] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science 1541, pages 195–206. Springer-Verlag, 1998.

[21] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar GEMM-based level 3 BLAS – the on-going evolution of a portable and high-performance library. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science 1541, pages 207–215. Springer-Verlag, 1998.

[22] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[23] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

[24] B. Kågström, P. Ling, and C. Van Loan. Gemm-based level 3 blas: High-performance model, implementations and performance evaluation benchmark. LAPACK Working Note #107 CS-95-315, Univ. of Tennessee, Nov. 1995.

[25] B. Kågström, P. Ling, and C. Van Loan. Gemm-based level 3 blas: Installation, tuning and use of the model implementations and the performance evaluation benchmark. LAPACK Working Note #108 CS-95-316, University of Tennessee, Nov. 1995.

[26] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *TOMS*, 24(3):268–302, 1998.

[27] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, 2nd edition, 1994.

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

[29] Enrique S. Quintana-Ortí and Robert van de Geijn. Fast parallel kernels for selected problems in control theory. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing 1999*, 1999.

[30] Enrique S. Quintana-Ortí and Robert van de Geijn. Fast parallel kernels for selected problems in control theory. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[31] Wesley C. Reiley. Efficient parallel out-of-core implementation of the Cholesky factorization. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[32] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

[33] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[34] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, Orlando, Florida, 1973.

[35] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[36] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[37] Y.-J. J. Wu, P. A. Alpatov, C. Bischof, and R. A. van de Geijn. A parallel implementation of symmetric band reduction using PLAPACK. In *Proceedings of Scalable Parallel Library Conference, Mississippi State University*, 1996. PRISM Working Note 35.

Index

Author Index

Constant Index

Function Index

Operation Index