The Science of Programming High-Performance Linear Algebra Libraries

Paolo Bientinesi* John A. Gunnels† Fred G. Gustavson† Greg M. Henry‡
Margaret E. Myers* Enrique S. Quintana-Ortí§ Robert A. van de Geijn*

Abstract

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

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

Over the last decade, our research in the area of the implementation of high performance sequential and parallel linear algebra libraries has made the wisdom of the above observations obvious.

In this paper, we show how to apply formal derivation methods to linear algebra operations. By combining this approach with a high-level abstraction for coding such algorithms, a linear algebra library that is more desirable, as measured by the metric provided by the above observations, can be achieved. Moreover, such a library can include a richer array of algorithms and is, at the same time, more easily maintained and extended. Most surprisingly, the methodology results in more efficient implementations. The approach is sufficiently systematic that much of it can be, and has been, automated.

*Department of Computer Sciences, The University of Texas, Austin, TX 78712

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

‡Intel Corporation, 5350 NE Elam Young Pkwy, Hillsboro, OR 97124-6461

§Dept. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12080 Castellón, Spain

1. Introduction

The title of this paper was derived from the title of Gries' undergraduate text *The Science of Programming* [8]. That text introduces students to the concept of proving programs correct. It is based on the early work of Floyd [7], Dijkstra [2, 3], and Hoare [14], among others. The proofs are constructive so that the derivation of the program and its proof are fundamentally intertwined. In this paper we apply this methodology to high-performance algorithms for dense linear algebra operations.

This paper is one of a sequence of papers that we hope will illustrate to the HPC community the benefits of the formal derivation of algorithms. In the first set of papers [10, 13], we outlined the concept of formal derivation and its application to dense linear algebra algorithms. In that paper, we also show that by, introducing an Application Programming Interface (API) for coding the provably correct algorithms, claims about the correctness of the algorithms allow claims about the correctness of the implementation to be made. Finally, we show that excellent performance can be attained. While in the first set of papers we give an overview by exploring the example of the LU factorization, we show that the method applies to more complex operations, in particular the solution of the triangular Sylvester Equation, in [18].

This paper illustrates the primary attributes of the derivation method, namely that it provides a step-by-step "recipe" that novice and veteran alike can use to rapidly derive algorithms. We apply the methodology to symmetric matrix multiplication, which is in some sense more complex than the LU factorization used in previous papers. This demonstrates that the techniques are generally applicable to a set of matrix multiplication operations, the level-3 Basic Linear Algebra Subprograms [5] (BLAS) when cast to be rich in matrix multiplication as advocated in [15].

The techniques in this paper apply to linear algebra operations for which there are algorithms that consist of a simple initialization followed by a loop. While this may appear to be extremely restrictive, the linear algebra libraries community has made tremendous strides towards modularity. As a consequence, almost any operation can be decomposed into operations (linear algebra building blocks) that, on the one hand, are themselves meaningful linear algebra operations and, on the other hand, whose algorithms have the structure given by the algorithm in Fig. 1. At this time, we do not have a clean characterization of the operations that fall into this category. However, over the last few years, we have shown that it includes all BLAS (levels 1, 2, and 3) [16, 6, 5, 12], all major factorization algorithms (LU, Cholesky, and QR) [10], matrix inversion (of general, symmetric, and triangular matrices) [17], and a number of matrix equations that arise in control theory [18].

This paper is organized as follows: In Section 2 we show that the derivation of families of algorithms for a large class of linear algebra operations is reduced to a systematic set of steps. In Section 3 we demonstrate the methodology by applying it to a typical example. How to translate a derived algorithm into code is discussed in Section 4. High-performance is demonstrated in Section 5. Concluding remarks are given in the final section.

2. A worksheet for deriving linear algebra algorithms

In Fig. 1, we give a generic "worksheet" for deriving a large class of linear algebra algorithms. Expressions in curly-brackets denote predicates that describe the state of the variables at the given point of the algorithm. For this paper, it suffices to realize that the statements between the assertions in the curly-brackets must hold at the indicated points in the algorithm.

The generic linear algebra operation is given by $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$. Notice that some operands may be both input and output parameters. Constraints on these parameters, such as structure or original contents, are given by the predicate P_{pre}, the *precondition*. The *postcondition*, P_{post}, is the predicate that describes the desired state upon completion of the algorithm.

We will require that the state given by the predicate P_{inv}, the *loop-invariant*, is maintained at the top of the loop. P_{inv} must hold before the loop is entered, it must hold at the end of the loop so it will again hold at the top of the loop, and it will hold upon completion of the loop. This is indicated in Fig. 1 at the points where P_{inv} occurs in the assertions.

The *loop-guard*, G, is the condition under which the program remains in the loop. Thus, after the loop completes, $\neg G$ must hold. If $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$ then we can conclude, by induction, that the loop computes the desired result.

Since the loop-invariant must hold before the loop commences, the *initialization*, Step 4 in Fig. 1, must have the property that starting in the state P_{pre}, the initialization leaves us in a state where P_{inv} holds.

In order to make progress towards the condition under which the loop is completed, we will see that regions of operands to be used and/or updated must be identified and appended to regions that have already been updated in a consistent manner. It is this identification of submatrices and shifting of boundaries that occurs in Steps 5a and 5b in Fig. 1.

The exposure of submatrices to be used and/or updated dictates the state Q_{before} before any updates have occurred. The update itself, S_U, must be such that the state Q_{after} holds after its application. This state must be such that after the shifting of the boundaries the loop-invariant again holds at the bottom of the loop.

In the next section, we illustrate how the worksheet allows us to derive common linear algebra operations. In particular, the systematic derivation of loop-invariants for a given linear algebra operation is central to the process.

3. Example: Symmetric matrix multiplication

Let us consider the example $C = AB + C$ where C and B are $m \times n$ matrices and A is an $m \times m$ symmetric matrix. This is similar to the operation provided by the level-3 BLAS routine DSYMM. Due to symmetry we will assume that only the lower triangular part of A is stored. One way to implement this operation is to copy matrix A and "symmetrize" it by copying the lower triangular part of the matrix to the upper triangular part, after which a regular matrix multiplication can be used. However, this involves an $m \times m$ temporary matrix, which is undesirable when m is large.

Step 1: Determine P_{pre} and P_{post}. The conditions before the operation commences (the precondition) can be described by the predicate indicated in Step 1a in Fig. 2. Here $m(X)$ and $n(X)$ return the row and column dimensions of X, respectively, while Symm(A) is a predicate that returns *true* iff A is a symmetric matrix and InLower(A) indicates that only the lower triangular part of A is stored. Finally, $\hat{C}$ indicates the original contents of matrix C.

The predicate in Step 1b in Fig. 2 indicates the desired state upon completion (the postcondition).

Step 2: Determine P_{inv}. In order to determine possible intermediate contents of the matrix C, we start by partitioning one of the operands, say A. The idea is that we will assume that, in a consistent manner, different regions of A will have been used to update C. Since A is symmetric, it becomes important to partition it into four quadrants:

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

Step	Annotated Algorithm: $[D, E, F, \ldots] = \text{op}(A, B, C, D, \ldots)$
1a	$\{P_{\text{pre}}\}$
4	**Partition** **where**
2	$\{P_{\text{inv}}\}$
3	**while** G **do**
2,3	$\{(P_{\text{inv}}) \wedge (G)\}$
5a	**Repartition** **where**
6	$\{Q_{\text{before}}\}$
8	S_U
7	$\{Q_{\text{after}}\}$
5b	**Continue with**
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{(P_{\text{inv}}) \wedge \neg (G)\}$
1b	$\{P_{\text{post}}\}$

Figure 1. Worksheet for developing linear algebra algorithms.

where A_{TL} and A_{BR} are square so that they themselves are symmetric. Here T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

We now plug this partitioned matrix into the postcondition in order to determine how the other operands can be conformally partitioned:

$$\begin{aligned}&(\text{ some partitioning of } C \text{ }) = \\ &\quad \left(\begin{array}{c\|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right) (\text{ some partitioning of } B \text{ }) \\ &\quad +(\text{ some partitioning of } \hat{C} \text{ })\end{aligned}$$

We conclude that it is consistent to partition C, B, and $\hat{C}$ into 2×1 blocked matrices:

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

By multiplying the right-hand-side out and equating the blocks on the left and the right, we find that ultimately the following equalities must hold:

$$\begin{array}{c} C_T = A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T \\ \hline C_B = A_{BL}B_T + A_{BR}B_B + \hat{C}_B \end{array}$$

The idea now is that at an intermediate stage only some of the operations $A_{TL}B_T$, $A_{BL}^T B_B$, $A_{BL}B_T$, and $A_{BR}B_B$ will have been performed. All possibilities for this partitioning of the operands are tabulated in Table. 1.

In the remainder of this section we will use at the intermediate stage the state where

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + \hat{C}_T \\ \hline \hat{C}_B \end{array}\right)$$

which becomes P_{inv} in our worksheet in Fig. 2.

Step 3: Determine loop-guard G. Notice that the assumption is that after the loop completes, $P_{\text{inv}} \wedge \neg G$ holds. Thus, by choosing a loop-guard G such that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$, we guarantee that the loop completes in a state that implies that the desired result has been computed. When A_{TL} equals all of A, this implies that C_T, B_T and $\hat{C}_T$ equal all of C, B, and $\hat{C}$, respectively, and therefore

$$\left(\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL}B_T + \hat{C}_T \\ \hline \hat{C}_B \end{array}\right) \wedge \text{SameSize}(A, A_{TL})\right) \Rightarrow (C = AB + \hat{C})$$

Here the predicate SameSize(A, A_{TL}) is *true* iff the dimensions of A and A_{TL} are equal. Thus, the iteration should continue as long as $\neg$SameSize(A, A_{TL}), the loop-guard G in the worksheet.

Step 4: Determine the initialization. The loop-invariant must hold before entering the loop. Ideally, only the partitioning of operands is required to attain this state. Notice that the initial partitionings given in Step 4 in Fig. 2 have the effect that C contains the desired contents, without requiring any update to the contents of C.

Step 5: Determine how to move boundaries. Notice that as part of the initialization, C_T, $\hat{C}_T$, and B_T have no rows and A_{TL} is 0×0, while upon completion of the loop these parts of the matrices should correspond to the complete matrices. Thus, the boundaries, denoted by the double lines, must be moved forward as part of the body of the loop,

Step	Annotated Algorithm: $C := AB + C$
1a	$\left\{ C = \hat{C} \wedge \text{Symm}(A) \wedge \text{InLower}(A) \wedge m(C) = m(A) = n(A) = m(B) \wedge n(C) = n(B) \right\}$
4	**Partition** $C = \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$, $\hat{C} = \left(\begin{array}{c} \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right)$, $B = \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$, and $A = \left(\begin{array}{ccc} A_{TL} & \Vert & A_{BL}^T \\ \hline\hline A_{BL} & \Vert & A_{BR} \end{array}\right)$ **where** C_T, $\hat{C}_T$, and B_T have 0 rows and A_{TL} is 0×0
2	$\left\{ \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL} B_T + \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \right\}$
3	**while** $\neg\text{SameSize}(A, A_{TL})$ **do**
2,3	$\left\{ \left(\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL} B_T + \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \right) \wedge (\neg\text{SameSize}(A, A_{TL})) \right\}$
5a	**Determine block size** b **Repartition** $\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline\hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\begin{array}{c} \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \hat{C}_0 \\ \hline\hline \hat{C}_1 \\ \hline \hat{C}_2 \end{array}\right)$, $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$, and $\left(\begin{array}{ccc} A_{TL} & \Vert & A_{BL}^T \\ \hline\hline A_{BL} & \Vert & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{ccccc} A_{00} & \Vert & A_{10}^T & \vert & A_{20}^T \\ \hline\hline A_{10} & \Vert & A_{11} & \vert & A_{21}^T \\ \hline A_{20} & \Vert & A_{21} & \vert & A_{22} \end{array}\right)$ **where** C_1, $\hat{C}_1$, and B_1 have b rows and A_{11} is $b \times b$
6	$\left\{ \left(\begin{array}{c} C_0 \\ \hline\hline \left(\begin{array}{c} C_1 \\ \hline C_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} A_{00} B_0 + \hat{C}_0 \\ \hline\hline \left(\begin{array}{c} \hat{C}_1 \\ \hline \hat{C}_2 \end{array}\right) \end{array}\right) \right\}$
8	$C_0 := A_{10}^T B_1 + C_0$ $C_1 := A_{10} B_0 + A_{11} B_1 + C_1$
7	$\left\{ \left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline C_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} A_{00} B_0 + A_{10}^T B_1 + \hat{C}_0 \\ \hline A_{10} B_0 + A_{11} B_1 + \hat{C}_1 \end{array}\right) \\ \hline\hline \hat{C}_2 \end{array}\right) \right\}$
5b	**Continue with** $\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline\hline C_2 \end{array}\right)$, $\left(\begin{array}{c} \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \hat{C}_0 \\ \hline \hat{C}_1 \\ \hline\hline \hat{C}_2 \end{array}\right)$, $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$, and $\left(\begin{array}{ccc} A_{TL} & \Vert & A_{BL}^T \\ \hline\hline A_{BL} & \Vert & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{ccccc} A_{00} & \vert & A_{10}^T & \Vert & A_{20}^T \\ \hline A_{10} & \vert & A_{11} & \Vert & A_{21}^T \\ \hline\hline A_{20} & \vert & A_{21} & \Vert & A_{22} \end{array}\right)$
2	$\left\{ \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL} B_T + \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \right\}$
	enddo
2,3	$\left\{ \left(\left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right) = \left(\begin{array}{c} A_{TL} B_T + \hat{C}_T \\ \hline\hline \hat{C}_B \end{array}\right) \right) \wedge \neg(\neg\text{SameSize}(A, A_{TL})) \right\}$
1b	$\left\{ C = AB + \hat{C} \right\}$

Figure 2. Worksheet for developing an algorithm for symmetric matrix multiplication.

Computed?					
$A_{TL}B_T$	$A_{BL}^T B_B$	$A_{BL}B_T$	$A_{BR}B_B$	$P_{\text{inv}} : \left(\frac{C_T}{C_B}\right) =$	Comment
NO	NO	NO	NO	$\left(\frac{\hat{C}_T}{\hat{C}_B}\right)$	No loop-guard exists so that $P_{\text{inv}} \wedge \neg G \Rightarrow P_{\text{post}}$
YES	NO	NO	NO	$\left(\frac{A_{TL}B_T + \hat{C}_T}{\hat{C}_B}\right)$	Example in Figs. 2 and 3. (Variant 1)
NO	YES	NO	NO	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{\hat{C}_B}\right)$	No loop-guard exists so that $P_{\text{inv}} \wedge \neg G \Rightarrow P_{\text{post}}$.
YES	YES	NO	NO	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{\hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 2)
NO	NO	YES	NO	$\left(\frac{\hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	No loop-guard exists so that $P_{\text{inv}} \wedge \neg G \Rightarrow P_{\text{post}}$.
YES	NO	YES	NO	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 3)
NO	YES	YES	NO	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	No loop-guard exists so that $P_{\text{inv}} \wedge \neg G \Rightarrow P_{\text{post}}$.
YES	YES	YES	NO	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 4)
NO	NO	NO	YES	$\left(\frac{\hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 5)
YES	NO	NO	YES	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
NO	YES	NO	YES	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 6)
YES	YES	NO	YES	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
NO	NO	YES	YES	$\left(\frac{\hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 7)
YES	NO	YES	YES	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
NO	YES	YES	YES	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	Leads to an alternative algorithm. (Variant 8)
YES	YES	YES	YES	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.

Table 1. Potential loop-invariants for symmetric matrix multiplication for the specific partitioning in Section 3.

Partition $C = \left(\begin{array}{c} C_T \\ \hline\hline C_B \end{array}\right)$, $B = \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$, and $A = \left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
  where C_T, and B_T have 0 rows and A_{TL} is 0×0

while $\neg\text{SameSize}(A, A_{TL})$ **do**
  Determine block size b
  Repartition

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

  where C_1, and B_1 have b rows and A_{11} is $b \times b$

$C_0 := A_{10}^T B_1 + C_0$
$C_1 := A_{10} B_0 + A_{11} B_1 + C_1$

  Continue with

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

enddo

Figure 3. Algorithm for symmetric matrix multiplication.

adding rows to C_T, $\hat{C}_T$, and B_T, and rows and columns to A_{TL}. The approach is to identify parts of the matrix that must be moved between regions at the top of the loop, and adding them to the appropriate regions at the bottom of the loop, as illustrated in Steps 5a and 5b in Fig. 2.

Step 6: Determine Q_{before}. Notice that the loop-invariant is *true* at the top of the loop, and is thus *true* after the repartitioning that identifies parts of the matrices to be moved between regions. In Step 6 in Fig. 2 the state, in terms of the repartitioned matrices, is given. This predicate is derived by simple substitution into the loop-invariant of the submatrices that make up the different regions, as defined by the double lines, which clearly have semantic meaning.

Step 7: Determine Q_{after}. Notice that after the regions have been redefined, as in Step 5b in Fig. 2, the loop-invariant must again be *true*. Given the redefinition of the regions in Step 5b, the loop-invariant, with the appropriate substitution of what the regions will become, must be *true* before the movement of the double lines. Thus,

$$\left(\begin{array}{c} \left(\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right) \\ \hline\hline C_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c|c} A_{00} & A_{10}^T \\ \hline A_{10} & A_{11} \end{array}\right)\left(\begin{array}{c} B_0 \\ \hline B_1 \end{array}\right) + \left(\begin{array}{c} \hat{C}_0 \\ \hline \hat{C}_1 \end{array}\right) \\ \hline\hline \hat{C}_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} A_{00}B_0 + A_{10}^T B_1 + \hat{C}_0 \\ \hline A_{10}B_0 + A_{11}B_1 + \hat{C}_1 \end{array}\right) \\ \hline\hline \hat{C}_2 \end{array}\right)$$

must be *true* after the update, S_U, in Fig. 2.

Step 8: Determine the update S_U. By comparing the state in Step 6 with the desired state in Step 7, the required update, given in Step 8, can be easily determined.

Final algorithm. Finally, by noting that $\hat{C}$ was introduced only to denote the original contents of C and is never referenced in the update, the algorithm for computing $C := AB + C$ can be stated as in Fig. 3.

Note that the operation $C_1 := A_{10}B_0 + A_{11}B_1 + C_1$ is typically implemented as

$$\begin{array}{rcl} C_1 & := & A_{10}B_0 + C_1 \\ C_1 & := & A_{11}B_1 + C_1 \end{array}$$

where the second operation itself requires a symmetric matrix multiplication. When $b = 1$, this symmetric matrix multiplication becomes a scaled (row) vector addition (AXPY) operation. Thus, the so-called "blocked" version of the algorithm, where $b > 1$, could be implemented by calling an "unblocked" version, where $b = 1$ and $C_1 := A_{11}B_1 + C_1$ is implemented as a scaled vector addition.

Alternative algorithms. Notice that by applying Steps 3-8 to each of the potential loop-invariants in Table 1 one will either conclude that the potential loop-invariant does not result in an algorithm, or one will discover an alternative algorithm. We leave it as an exercise for the reader to derive all seven additional algorithms (Variants 2–8).

In addition, a second set of algorithms can be derived by partitioning C, $\hat{C}$, and B horizontally, and not partitioning A, so that

$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) = A \left(\begin{array}{c|c} B_L & B_R \end{array} \right) + \left(\begin{array}{c|c} \hat{C}_L & \hat{C}_R \end{array} \right)$$

This leads to algorithms that update C one of more columns at a time.

Finally, a third set of algorithms can be derived by partitioning C, $\hat{C}$, A, and B, all into quadrants. This approach leads to a very large number of variants, of which the already described variants are a subset. For a hint at how these different variants relate to each other, see [18].

4. Implementation

The systematic derivation of provably correct algorithms solves only part of the problem, namely that there are no logic errors in the algorithm. So-called programming bugs are generally introduced in the translation of the algorithm into code. We solve this problem by introducing an API, the Formal Linear Algebra Methods Environment (FLAME) for coding the algorithms that closely resembles the algorithms themselves [13, 10, 12].

In Fig. 4, we show the FLAME code that implements a recursive version of the blocked algorithm given in Fig. 3. We believe the code to be self-explanatory. Notice that implementations of Variants 2–4 are identical to this code modulo the updates in the "Compute" section of the code. Variants 5–8 are also very similar.

5. Performance

In this section we illustrate how the derivation method, combined with the FLAME API, leads to high-performance algorithms and implementations for the DSYMM operation. Performance was measured on a 650 MHz Intel (R) Pentium (R) III processor-based laptop with a 256K L2 cache running the Linux (Red Hat 7.1) operating system. All computations were performed in 64-bit (double precision) arithmetic. For our implementations, the FLAME API linked to BLAS provided by Intel's Math Kernel Library (MKL) library, V5.1.1 for the Pentium III processor, *except* for the DGEMM matrix multiplication kernel, for which we used our own ITXGEMM implementation [11]. In our graphs we report the rate of computation, in millions of floating point operations per second (MFLOPS/sec.), using the accepted operation count of $2m^2n$ floating point operations. In all graphs we show the performance of the MKL and ATLAS R3.1 [19] implementations of DSYMM for comparison. We also report the performance of the naive implementation in which we copy matrix A, "symmetrize" the copy and use this symmetrized copy to perform the matrix multiplication, requiring an $m \times m$ temporary matrix.

Notice that the theoretical peak of this particular architecture is 650 MFLOPS/sec. However, due to memory bandwidth limitations, in practice the peak performance seen by DGEMM is around 525 MFLOPS/sec. [11].

In Fig. 5(a) we report the performance of various unblocked algorithms ($b = 1$). These implementations perform the bulk of their computation in the level-2 BLAS operations DGER and DGEMV [6]. It is well-known that these operations cannot attain high-performance since they perform $O(n^2)$ operations on $O(n^2)$ data, which makes the limited memory bandwidth a bottleneck. Note that Variant 2 performs most computation in DGEMV, Variants 1 and 4 perform equal amounts of computation in DGEMV and DGER, and Variant 3 performs most computation in DGER. This explains the relative performance of these implementations since high-performance implementations of DGEMV incur half the memory traffic of DGER.

In Fig. 5(b) we report the performance of blocked versions of the algorithms when the algorithmic blocksize equals $b = 128$ (`nb_alg` in Fig. 4) and an unblocked implementation of the indicated variant is used for the smaller subproblem. The reason $b = 128$ was used is that the ITXGEMM matrix multiplication achieves near optimal performance when one of the matrix dimensions is 128. Notice that performance, although much better than the unblocked implementations, is far from peak. The variation in the performance is primarily related to the variation of the respective unblocked algorithms being used for the smaller subproblems.

In Fig. 5(c) we report the performance of recursive blocked versions of the algorithms. This time, blocksizes of $b = 128, 64, 32, 16, 8$ are used as part of the recursion, where once matrix A is 8×8 or smaller the unblocked algorithm is used. While ultimately the performance of the different implementations is quite respectable, it suffers from the fact that the performance of the unblocked algorithms is not great when the matrix sizes are relatively small.

In Fig. 5(d) we report the performance of the same blocked algorithms as in Fig. 5(b), again using a blocksize of $b = 128$. This time the subproblem is implemented by "symmetrizing" a copy of A and performing the symmetric matrix multiplication using this copy. Performance improves, at the cost of a 128×128 temporary matrix.

In Fig. 5(e) we report the performance of the same recursive blocked algorithms as in Fig. 5(e), this time using blocksizes $b = 128, 64$. The subproblem is again implemented by symmetrizing a copy of A and performing the symmetric matrix multiplication using this copy. While performance improves, the size of the temporary matrix is also reduced to 64×64.

We conclude this section by pointing out that all four

```
#include "FLAME.h"

void Symm_left_lower_var1_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,
                 /* with */ 0, /* x */ 0, /* quadrant */ FLA_TL );
  FLA_Part_2x1 ( B,            &BT,
                            /* *** */
                               &BB,
                 /* with */ 0, /* rows from */ FLA_TOP );
  FLA_Part_2x1 ( C,            &CT,
                            /* *** */
                               &CB,
                 /* with */ 0, /* rows from */ FLA_TOP );

  while ( FLA_Obj_length( A ) != FLA_Obj_length( ATL ) ) {
    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,
                            /* with */ b, /* x */ b, /* A11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1 ( BT,                &B0,
                           /**/                /**/
                                               &B1,
                            BB,                &B2,
                            /* with */ b, /* rows matrix B1 from */ FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1 ( CT,                &C0,
                           /**/                /**/
                                               &C1,
                            CB,                &C2,
                            /* with */ b, /* rows matrix C1 from */ FLA_BOTTOM );

    /* ------------ Compute ----------------------------------------------- */
    FLA_Gemm ( FLA_TRANSPOSE,    FLA_NO_TRANSPOSE, ONE, A10, B1, ONE, C0 );
    FLA_Gemm ( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A10, B0, ONE, C1 );
    if (b <= 8) Symm_ll_var1_unb( A11, B1, C1 );
    else        Symm_left_lower_var1_rec( A11, B1, C1, nb_alg/2 );
    /* -------------------------------------------------------------------- */

    FLA_Cont_with_3x3_to_2x2 ( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
                                                     A10, A11, /**/ A12,
                             /* ************** */ /* ****************** */
                               &ABL, /**/ &ABR,      A20, A21, /**/ A22,
                               /* with A11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1 ( &BT,                  B0,
                                                     B1,
                               /**/                 /**/
                               &BB,                  B2,
                               /* with B1 added to */ FLA_TOP );
    FLA_Cont_with_3x1_to_2x1 ( &CT,                  C0,
                                                     C1,
                               /**/                 /**/
                               &CB,                  C2,
                               /* with C1 added to */ FLA_TOP );
  }
}
```

Figure 4. FLAME implementation of blocked algorithm (Variant 1).

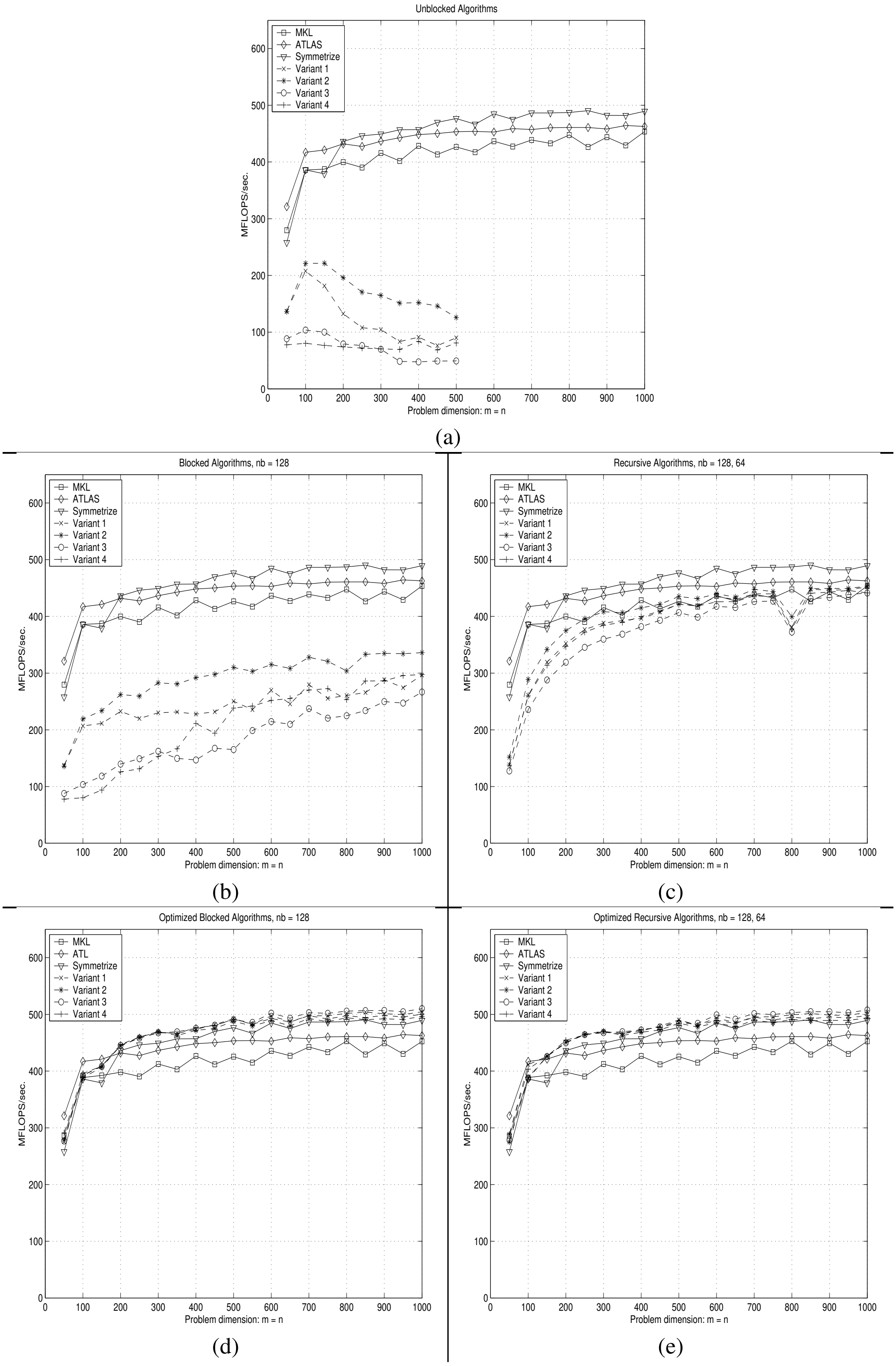

Figure 5. Performance of symmetric matrix multiplication implementations.

variants were developed, implemented, debugged, and timed in an aggregate time measured in *hours*.

6. Conclusion

In this paper, we have presented a case-study that shows how formal derivation is a useful tool when designing and implementing linear algebra libraries. While it may not be convincing by itself, a large number of such case studies, as reported in [1, 10, 12, 18], together do present considerable evidence.

We believe that the methodology is sufficiently systematic to be largely automated. In a semester project Sergey Kolos, a graduate student at UT-Austin, automated the presented steps using Mathematica [20]. In a recent dissertation, one of the authors showed how algorithms similar to the one presented in Fig. 3 can be automatically translated to high-performance parallel code, while simultaneously automatically generating a cost analysis [9].

Additional information

For additional information on FLAME visit
`http://www.cs.utexas.edu/users/flame/`

Acknowledgments

We would like to thank the undergraduate and graduate students in *CS378: High-Performance Parallel Computing* and *CS395: High-Performance and Parallel Numerical Algorithms* (Spring 2001 and Spring 2002 at The University of Texas at Austin) for their valuable feedback. In addition, we would like to thank Mark Hinga for his comments.

Preliminary research related to FLAME was partially supported by the Remote Exploration and Experimentation Project at Caltech's Jet Propulsion Laboratory, which is part of NASA's High Performance Computing and Communications Program, and is funded by NASA's Office of Space Science.

References

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

[2] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[3] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[4] E. W. 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/`.

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

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

[7] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[8] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[9] J. A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.

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

[11] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

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

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

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

[15] B. Kågström, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[16] 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.

[17] E. S. Quintana, G. Quintana, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

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

[19] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[20] S. Wolfram. *The Mathematica Book: 3rd Edition*. Cambridge University Press, 1996.