FLAMES2S: From Abstraction to High Performance

Richard Veras
Jonathan Monette
Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Enrique S. Quintana-Ortí
Departamento de Ingeniería
y Ciencia de Computadores,
Universidad Jaume I
12.071 - Castellón (Spain)

FLAME Working Note 35
December 14, 2008

Abstract

The FLAME project advocates representing matrix algorithms, whether typeset or in code, at the same level of abstraction that is used to explain such algorithms on a chalk board. While this has allowed us to modernize development of libraries with functionality that includes the level-3 BLAS and much of LAPACK, performance has been an issue for algorithms that cast most computation in terms of level-1 and level-2 BLAS. We show how this can be overcome with a source-to-source transformer that takes a representation that uses the FLAME/C API as input and yields code in a more traditional style (indexed loops and direct calls to BLAS routines). Impressive performance is demonstrated even when level-2 BLAS kernels are implemented in terms of optimized level-1 BLAS kernels. Thus, developers can finally fully embrace these new abstractions and tools for the development of linear algebra libraries.

1 Introduction

Over the last decade, as part of the FLAME project a large body of work has been published that advocates raising the level of abstraction at which one expresses algorithms and codes for the domain of linear algebra [27]. Many benefits have been documented: (1) a new notation allows algorithms to be presented in a way that captures the pictures that often accompany an explanation [21]; (2) this notation allows systematic derivation of proven correct families of loop-based algorithms [13, 3, 28, 12]; (3) this derivation process can be made mechanical [2]; (4) Application Programming Interfaces (APIs) can be defined for various languages so that representations in code mirror the algorithms [5]; (5) numerical error analyses can be made similarly systematic [2]; (6) the effort for development and maintenance of libraries is greatly reduced [16, 29]; and (7) new architectures can be easily accommodated [23, 22]. The only reason not to abandon more traditional approaches to library development [1, 6] has been that some algorithms, namely those that do not cast most computation in terms of level-3 Basic Linear Algebra Subprograms (BLAS) [9], or matrix-matrix operations, suffer a performance penalty from the way the API is implemented. In this paper, we show that this can be overcome via a relatively simple source-to-source transformer, FLAMES2S hereafter.

This paper is organized as follows: In Section 2 we illustrate how the FLAME notation and FLAME/C API represent algorithms for matrix computations, using the symmetric matrix-vector multiplication operation (SYMV) as an example. In that section, we also highlight the performance penalty due to the overhead associated with the API. In Section 3 we describe FLAMES2S, the source-to-source transformer that takes an implementation coded at a high level of abstraction to indexed loops with calls to low-level kernels. In Section 4, we show how even the performance of high-performance blocked algorithms is helped by the employment of FLAMES2S. We review related work in Section 5, followed by concluding remarks in Section 6.

2 A Motivating Example

We now describe one of the level-2 BLAS operations [10] (matrix-vector operations), the symmetric matrix-vector multiplication. This example is the primary vehicle with which we will illustrate the issues and solutions of this paper. We chose a matrix-vector operation on purpose: when the matrix involved is $n \times n$, the operation requires $O(n^2)$ floating point operations (flops) while the cost of indexing using the FLAME/C API is $O(n)$. When n is relatively small, the overhead of the API is inherently very noticeable. This means that if we do well for an operation like SYMV, we will do even better for an operation like the Cholesky factorization (discussed in Section 4), for which an unblocked algorithm incurs $O(n)$ indexing overhead for $O(n^3)$ flops and a blocked algorithm that incurs $O(n/b)$ overhead for $O(n^3)$ flops, where b is the block size. SYMV is fairly representative of the other level-2 BLAS and thus insights gained here should be readily applicable to the remaining level-2 operations.

Let $A \in \mathbb{R}^{n \times n}$, $x, y \in \mathbb{R}^n$, and $\alpha, \beta \in \mathbb{R}$. If A is symmetric, only the upper or lower triangular part (including the diagonal) needs to be stored. The SYMV operation computes $y := \alpha Ax + \beta y$. In our discussion $\alpha = \beta = 1$. The FLAME methodology can be used to systematically derive eight different algorithmic variants for computing this operation. Four of these are given, using FLAME notation, in Figure 1. There, $m(A)$ stands for the number of rows of a matrix. We believe the rest of the notation to be intuitive; for more details, see, e.g., [3]. Four more variants sweep through the matrix and vectors in the opposite direction. The second algorithmic variant, coded with the FLAME/C API, is given in Figure 1 (right). The experienced reader will recognize calls like `FLA_Dot_s` and `FLA_Axpy` as wrappers to level-1 BLAS operations [15] (vector-vector operations).

While the beauty of the routine in Figure 1 (right) is in the eye of the beholder, its performance is not. In Figure 2 we plot the performance of the code in Figure 1 (right), in terms of GFLOPS (10^9 flops per second), with the line labeled `FLAME/C var 2`. The calls to `FLA_Dot_s` and `FLA_Axpy` are wrappers to routines `ddot` and `daxpy` from the GotoBLAS. We only show performance for Variant 2 because it outperforms the other variants since it brings matrix A into memory only once and traverses the matrix by columns. Routine `dsymv` in GotoBLAS 1.18 is essentially the same algorithm, but hand-code and optimized. The routine `dsymv` in GotoBLAS 1.26 improves upon this by fusing the `ddot` and `daxpy` operations into a single loop, in assembly-code. What we will see is that the performance penalty is due primarily to the overhead of the FLAME/C API. The topic of the next section is how this overhead can be overcome by employing a source-to-source transformer, yielding the performance indicated by `FLAMES2S var 2`.

3 FLAMES2S: A Source-to-Source Transformer for FLAME/C Code

Conventional wisdom has been that, in scientific computing, we must sacrifice programmability for the sake of performance and therefore abstractions like the FLAME/C API are "not allowed". But does one have to choose between code like that given in Figure 1 (right) and code that explicitly exposes indices? In this section, we discuss a source-to-source transformer that translates code implemented using the FLAME/C API to more traditional code, yielding the best of both worlds.

FLAMES2S is remarkably simple by powerful: it comprises essentially of a set of rewrite rules that takes the high-level description of the algorithm to code.

How we write FLAME/C code. We start by very briefly reviewing the process by which we produce a routine like the one in Figure 1 (right). This will help us better explain some of the design decisions behind FLAMES2S.

The process by which all algorithmic variants for SYMV are derived is described in [28]. Here, we only remark that the process is systematic to the point where it has been made mechanical [2]. Indeed, this mechanical system can almost automatically produce the code in Figure 1 (right). However, here we describe how we use a web-based tool to produce a code skeleton which is then completed manually.

Algorithm: $y := Ax + y$ (SYMV)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & * \\ \hline A_{BL} & A_{BR} \end{array}\right)$,
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
where A_{TL} is 0×0, x_T, y_T have 0 elements

while $m(A_{BR}) > 0$ **do**

Repartition

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

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$$

where $\alpha_{11}, \chi_1, \psi_1$ are scalars

Variant 1:
$\psi_1 := a_{10}^T x_0 + \chi_1 \alpha_{11} + a_{21}^T x_2 + \psi_1$ (dot, dot)

Variant 2:
$\psi_1 := \chi_1 \alpha_{11} + a_{21}^T x_2 + \psi_1$ (dot)
$y_2 := \chi_1 a_{21} + y_2$ (axpy)

Variant 3:
$y_0 := \chi_1 a_{10} + y_0$ (axpy)
$\psi_1 := a_{10}^T x_0 + \chi_1 \alpha_{11} + \psi_1$ (dot)

Variant 4:
$y_0 := \chi_1 a_{10} + y_0$ (axpy)
$\psi_1 := \chi_1 \alpha_{11} + \psi_1$
$y_2 := \chi_1 a_{21} + y_2$ (axpy)

Continue with

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

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$$

endwhile

```
void Symv_unb_var2( FLA_Obj A, FLA_Obj x, FLA_Obj y )
{
  FLA_Obj
    ATL, ATR,  A00,  a01,     A02,   xT,  x0,    yT,  y0,
    ABL, ABR,  a10t, alpha11, a12t,  xB,  chi1,  yB,  psi1,
               A20,  a21,     A22,         x2,        y2;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,     0, 0, FLA_TL );
  FLA_Part_2x1( x,    &xT,
                      &xB,            0, FLA_TOP );
  FLA_Part_2x1( y,    &yT,
                      &yB,            0, FLA_TOP );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){
    FLA_Repart_2x2_to_3x3(
      ATL, /**/ ATR,     &A00,  /**/ &a01,     &A02,
   /* ************ */ /* ************************* */
                        &a10t, /**/ &alpha11, &a12t,
      ABL, /**/ ABR,     &A20,  /**/ &a21,     &A22,
      1, 1, FLA_BR );
    FLA_Repart_2x1_to_3x1( xT,            &x0,
                         /* ** */       /* **** */
                                         &chi1,
                           xB,           &x2,
                           1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( yT,            &y0,
                         /* ** */       /* **** */
                                         &psi1,
                           yB,           &y2,
                           1, FLA_BOTTOM );
    /*-----------------------------------------------*/

    FLA_Mult_add( chi1, alpha11, psi1 );
    FLA_Dot_s( FLA_ONE, a21, x2, FLA_ONE, psi1 );
    FLA_Axpy( chi1, a21, y2 );

    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
      &ATL, /**/ &ATR,       A00,  a01,     /**/ A02,
                             a10t, alpha11, /**/ a12t,
   /* ************** */  /* ********************** */
      &ABL, /**/ &ABR,       A20,  a21,     /**/ A22,
      FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &xT,                x0,
                                                  chi1,
                            /* ** */            /* **** */
                              &xB,                x2,
                              FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &yT,                y0,
                                                  psi1,
                            /* ** */            /* **** */
                              &yB,                y2,
                              FLA_TOP );
  }
}
```

Figure 1: Left: Four algorithmic variants for computing SYMV, typeset with FLAME notation. Right: Implementation of Variant 2 coded with the FLAME/C API.

Once the algorithm has been derived, a code skeleton is generated using a webpage-based tool, Spark [25], depicted in Figure 3[1]. The idea is that by filling out a simple form, most of the code can be generated automatically, leaving only the updates between the `/*------*/` lines to be filled in manually with subroutine calls that perform the necessary computations. We note that one of the "output language" options of the tool yields a representation that typesets a skeleton for algorithms with LaTeX as depicted in Figure 1 (left).

The essence of a typical linear algebra algorithm. What is important is that the high-level description of the algorithm requires a few parameters (those chosen in the Spark tool) together with the computations

[1]We encourage the reader to visit this website and to try it before continuing.

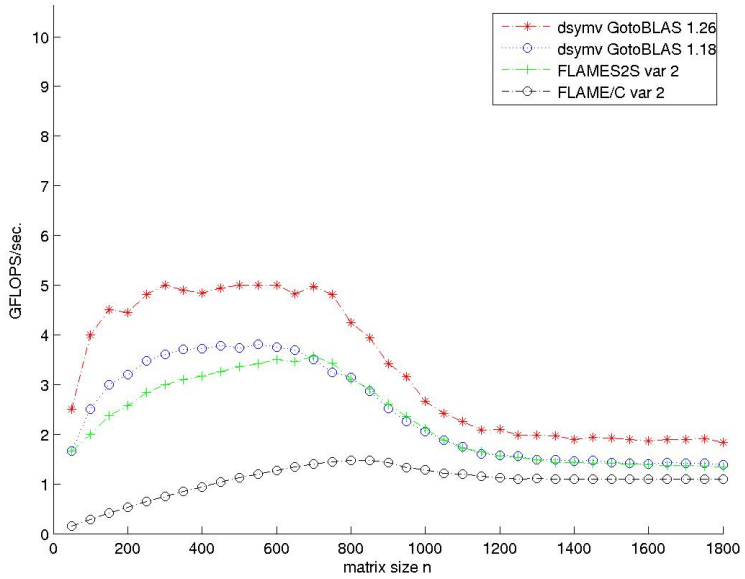

Figure 2: Performance of various implementations of SYMV using a single core of an Intel Xeon x5335 (2.66 GHz). Here, and in all our graphs, the top of the graph represents the theoretical peak that can be attained. For an operation that performs $O(n^2)$ computation with $O(n^2)$ data, it cannot be expected that near-peak performance can be reached.

that form the body of the loop. Thus, the essence of a typical linear algebra algorithm can be described by a very small number of parameters, for each of which there are very few choices: (1) the name of the operation; (2) how the algorithm proceeds through the operands, or in other words, the direction in which the operands are traversed; (3) whether it is a blocked or unblocked algorithm; (4) the condition for remaining in the loop (the loop-guard), which sometimes must be modified from the default choice; and (5) the updates within the loop body. Sometimes, there are additional initialization and finalization commands that occur before and after the loop, respectively.

From a FLAME/C routine back to the essence of the algorithm. It is generally the case that if one expresses information at a high level of abstraction, it becomes easier to take advantage of that information. In our case, it is easier to convert an algorithm expressed in terms of its essential properties, as discussed above, to a code expressed with loops and direct calls to BLAS. We have already created a large library implemented using the FLAME/C API and the fact that the FLAME/C code closely resembles the typeset algorithm retains a certain attraction. Fortunately, the FLAME/C implementation is rigidly structured so that this essence of the algorithm can be easily extracted. Thus, we chose to have FLAMES2S start by converting FLAME/C code to an XML description of the essence of the algorithm. Given the input in Figure 1 (right), this step yields the FLAME/XML description of the underlying algorithm given in Figure 4.

The FLAME/XML description is broken up into four blocks, a declaration block that contains the details regarding the matrices being processed, an optional pre-loop block for operations that need to be performed before the update, the loop block containing the sequence of suboperations that execute until the guard fails, and finally an optional post-loop block to perform any clean-up functions that may be necessary.

From essence to performance. Having converted the code back to its essence, the next step is to translate the code to one that explicitly exposes indices and calls the standard BLAS interface. Given the

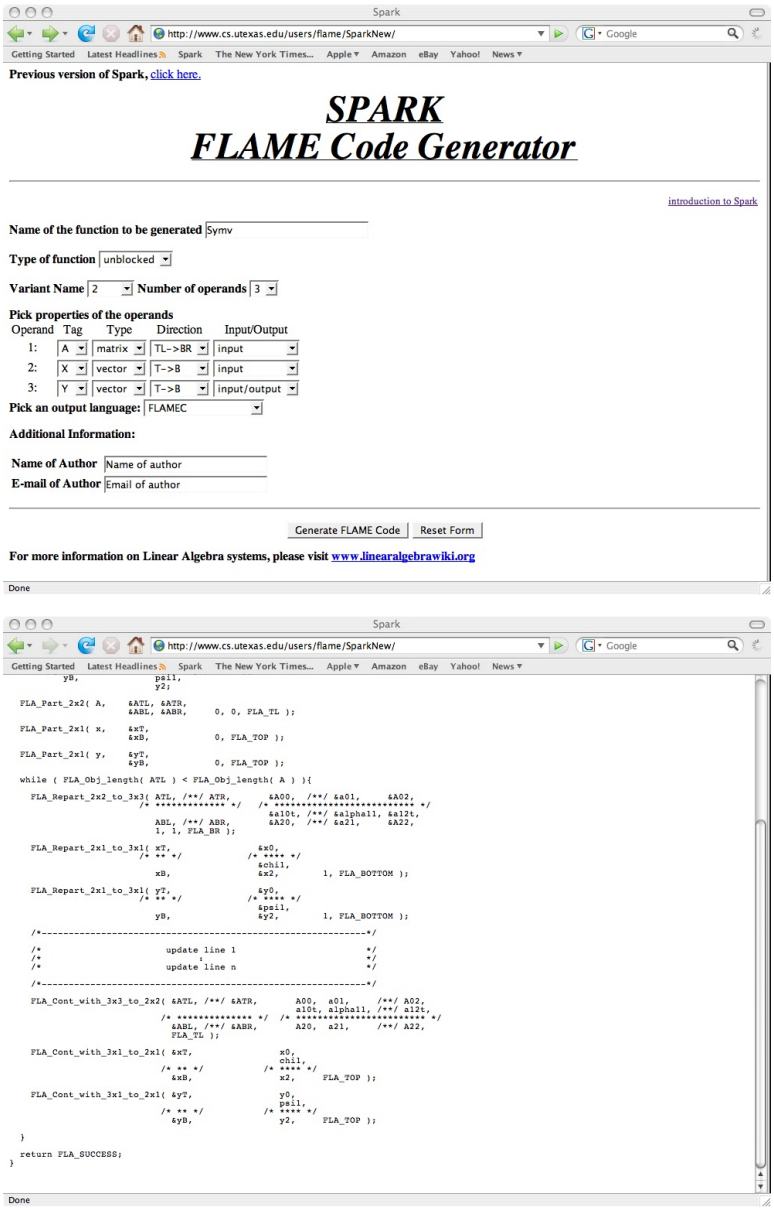

Figure 3: The Spark tool for generating code skeletons (`http://www.cs.utexas.edu/users/flame/Spark/`).

FLAME/XML description in Figure 4, FLAMES2S yields the code in Figure 5[2], which attains the performance presented in Figure 2 as `FLAMES2S var 2`. Note that Variant 2 still does not match the performance of GotoBLAS 1.26. However, it *does* almost match the performance of the equivalent handcoded routine that is part of GotoBLAS 1.18. As mentioned, the version in GotoBLAS 1.26 is assembly-coded and employs an algorithm that "fuses" the calls to `ddot` and `daxpy`, in addition to other optimizations. Thus, we cannot possibly expect to match its performance. It is significant that we can match the performance of this very high quality library before recent additional optimizations were performed. It would be interesting to see what performance would result if FLAMES2S had access to a fused `ddot` and `daxpy` routine.

We conclude by noting that very similar performance results were attained on other platforms, which is

[2]Here we describe a prototype implementation. The exact details of the code in Figure 5 will likely change. Indeed, the alert reader will immediately see many ways in which the code can be simplified and optimized.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Function name="FLA_Symv" type="unb" variant="2">
    <Option type="uplo">FLA_LOWER_TRIANGULAR</Option>
    <Declaration>
        <Operand type="matrix" direction="TL-&gt;BR" inout="both">A</Operand>
        <Operand type="vector" direction="T-&gt;B" inout="in">x</Operand>
        <Operand type="vector" direction="T-&gt;B" inout="in">y</Operand>
    </Declaration>
    <Loop>
        <Guard>A</Guard>
        <Update>
            <Statement name="FLA_Axpy">
                <Parameter partition="1">x</Parameter>
                <Parameter partition="11">A</Parameter>
                <Parameter partition="1">y</Parameter>
            </Statement>
            <Statement name="FLA_Dot_s">
                <Parameter>FLA_ONE</Parameter>
                <Parameter partition="21">A</Parameter>
                <Parameter partition="2">x</Parameter>
                <Parameter>FLA_ONE</Parameter>
                <Parameter partition="1">y</Parameter>
            </Statement>
            <Statement name="FLA_Axpy">
                <Parameter partition="1">x</Parameter>
                <Parameter partition="21">A</Parameter>
                <Parameter partition="2">y</Parameter>
            </Statement>
        </Update>
    </Loop>
</Function>
```

Figure 4: Output of first stage of FLAMES2S. It produces an intermediate XML representation given the code in Figure 1 (right).

why we do not provide additional performance graphs.

4 Impact on Unblocked and Blocked Algorithms

A typical level-3 BLAS or "LAPACK-level" operation is implemented using a "blocked" algorithm that casts most computation in terms of matrix-matrix multiplication [14]. As part of the blocked algorithm, a smaller subproblem has to be solved, which represents a lower-order term in the cost of the algorithm. Such subproblems are then often implemented via an "unblocked" algorithm that casts most operations in terms of level-2 BLAS operations. While the overhead of the FLAME/C API is not as noticeable for unblocked algorithms, since it constitutes $O(n)$ overhead for $O(n^3)$ operations, it still affects how fast a routine ramps up to high performance. We illustrate how FLAMES2S solves this problem, using the Cholesky factorization as an example.

Details on the Cholesky factorization, presented with FLAME notation, can be found in many of our publications, most notably [28, 4] We give the three unblocked and three blocked algorithms for this operation in Figure 6. Implementations for all these algorithms can be easily created with the Spark tool.

In Figure 7 we report the performance of various implementations. We only consider the cases where the blocked algorithm uses Variant 3 and the unblocked algorithm, employed to factor A_{11} in the blocked algorithm, uses Variant 2. Details of why these choices are usually best go beyond the scope of this paper. We report performance for four different choices. In the legend, `FLAME/C blk var3` indicates that for the blocked algorithm Variant 3 was employed. For the suboperation, one of two implementations of Variant 2 (which typically attains slightly better performance for small subproblems) was used: `FLAME/C unb var2` (an implementation that uses the FLAME/C API) or `FLAMES2S unb var2` (the result of FLAMES2S). In addition, we report the performance of LAPACK blocked routine `dpotrf` for Cholesky factorization. In all

```
#include "FLAME.h"

#define AA(i,j) buff_A[ (j-1)*ldim_A + (i-1) ]
#define xx(i,j) buff_x[ (j-1)*ldim_x + (i-1) ]
#define yy(i,j) buff_y[ (j-1)*ldim_y + (i-1) ]

int Symv_unb_var2_opt1( FLA_Obj A,FLA_Obj x,FLA_Obj y ){

  int ldim_A, n_A, m_A, n_A_min_j, m_A_min_j, ldim_x, n_x, m_x,
    n_x_min_j, m_x_min_j, ldim_y, n_y, m_y, n_y_min_j, m_y_min_j, j, i_one=1;

  double *buff_A, *buff_x, *buff_y;

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

  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 );
  ldim_x   = FLA_Obj_ldim( x );

  m_y      = FLA_Obj_length( y );
  n_y      = FLA_Obj_width( y );
  ldim_y   = FLA_Obj_ldim( y );

  for (j=1; j<=m_A; j++)
  {

    int m_A_min_j = m_A - j ;

    daxpy_( &i_one, &xx( j , 1 ) , &AA( j , j ) , &i_one, &yy( j , 1 ) , &i_one );

    {
      double ddot_();
      double *delta_temp;
      double *alpha_temp;
      double *beta_temp;
      double ddot_temp;
      delta_temp = &yy( j , 1 ) ;
      alpha_temp = FLA_DOUBLE_PTR( FLA_ONE ) ;
      beta_temp = FLA_DOUBLE_PTR( FLA_ONE ) ;
      ddot_temp = ddot_( &m_A_min_j, &AA( j+1 , j ) , &i_one, &xx( j+1 , 1 ) , &i_one );

      ddot_temp *= *alpha_temp;
      *delta_temp *= *beta_temp;
      *delta_temp += ddot_temp;
    }

    daxpy_( &m_A_min_j, &xx( j , 1 ) , &AA( j+1 , j ) , &i_one, &yy( j+1 , 1 ) , &i_one );
  }
}
```

Figure 5: Output of the second stage of FLAMES2S given the input in Figure 4. (Clearly there are further opportunities for optimization. The above is essentially a line-by-line translation of the FLAME/C code.)

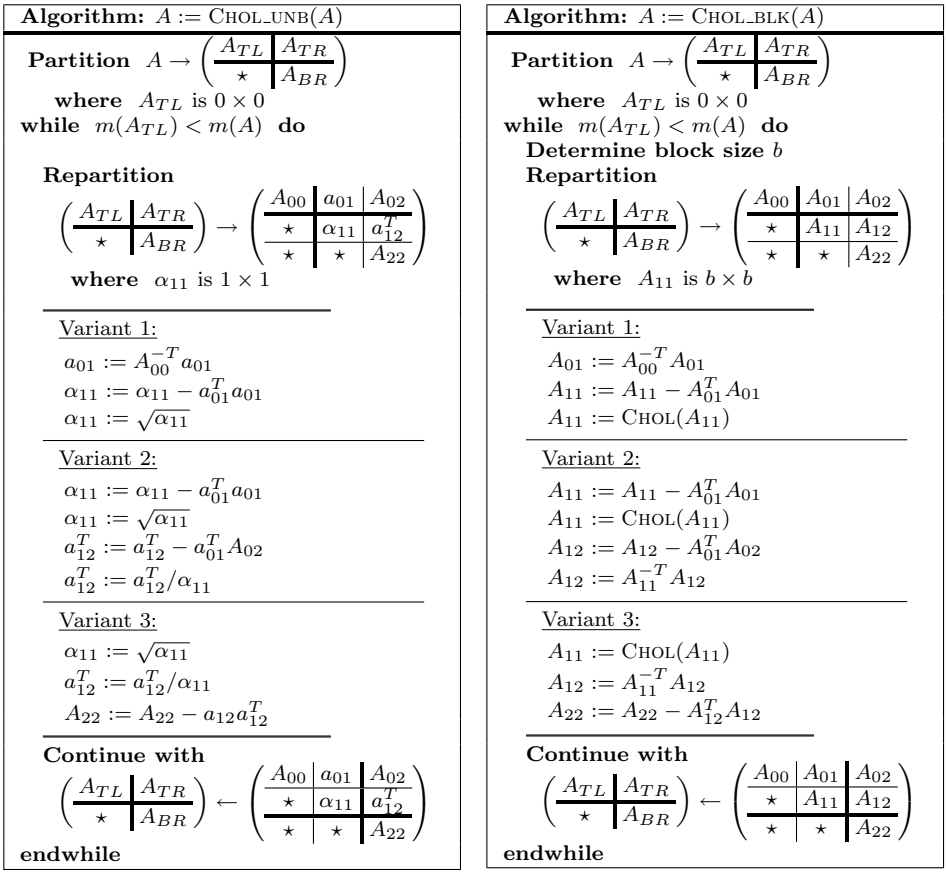

Figure 6: Unblocked and blocked algorithms for computing the Cholesky factorization.

cases the block size that was used was taken to equal 128, which levels the playing field, and all were linked to the GotoBLAS (1.26) BLAS library. Clearly, when both the blocked and unblocked routines are coded with the FLAME/C API, performance ramps up more slowly, due to the overhead of the API. The performance of the other three implementations is essentially identical. The LAPACK `dpotrf` routine eventually performs worse because it employs algorithmic Variant 2 for the blocked algorithm, a suboptimal choice. Qualitatively similar results were obtained on other platforms.

This experiment illustrates that the FLAMES2S transformer allows routines that are coded with the FLAME/C API to achieve essentially the same performance as implementations that are coded at a lower level of abstraction.

5 Related Work

It can be argued that there have been many efforts to express linear algebra libraries at a high level of abstraction, starting with the original MATLAB environment developed in the 1970s [19]. Mapping from MATLAB-like descriptions to high-performance implementations was already pursued by the FALCON project [8, 17, 7]. MATLAB itself now has a compiler that does so. More recently, a project to produce "Build to Order Linear Algebra Kernels" [24] takes an approach similar in philosophy to what we are proposing: input is a MATLAB-like description of linear algebra algorithms and output is optimized code.

What sets our approach apart from these efforts is that the FLAME project has managed to greatly

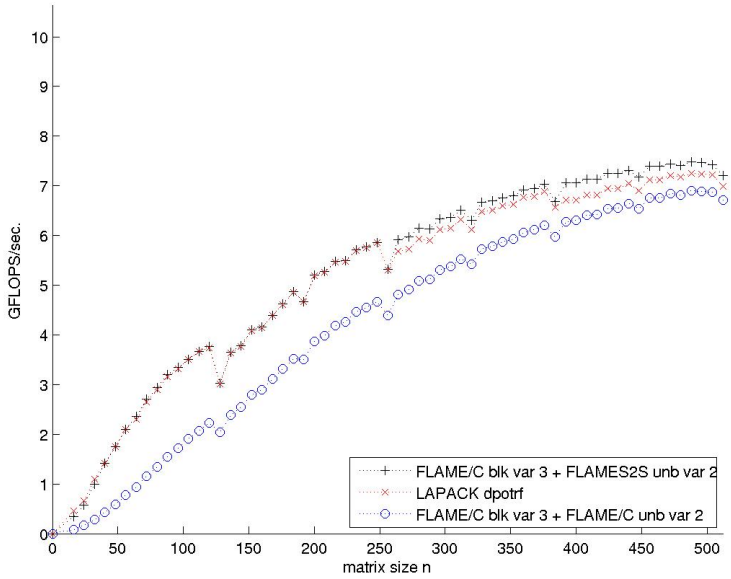

Figure 7: Performance of various implementations of Cholesky factorization on the same architecture as was used for Figure 2.

simplify how algorithms are expressed at a high level of abstraction. A solution close to ours was proposed in John Gunnels' dissertation [12], a participant in what became the FLAME project. He proposed a domain-specific language for expressing PLAPACK code for distributed-memory architectures, PLAWright, and a translator, implemented with Mathematica, was developed that yielded PLAPACK code and a cost analysis of its execution. In many ways, PLAWright code resembles what later became the FLAME/C and FLAME@lab APIs for the C and M-script (MATLAB) languages, and thus, FLAMES2S merely targets a different architecture, namely a single processor. There are, however, differences: The translation of PLAWright to PLAPACK code was simpler by virtue of the fact that PLAPACK code itself still exhibits a level of abstraction very similar to that of PLAWright. In other words: it is a simpler translation from PLAWright to PLAPACK code (like translating the XML description back to FLAME/C code) and there was no need to produce lower-level code because the overhead of the PLAPACK API was minute compared with the cost of starting a communication on a distributed-memory architecture. FLAMES2S instead takes high-level abstraction to low-level code.

6 Conclusion

On the surface, the central message of this paper seems to be simple and the results straightforward: The inherent structure in FLAME/C code allows a direct source-to-source transformer to be employed to produce code that attains the same high level of performance as traditional code. In addition, we believe that the transformer completes a metamorphosis of dense and banded linear algebra libraries that started a decade ago [13] because it demonstrates that one can embrace programmability without sacrificing performance. Earlier, a legitimate objection to abandoning what we often call "the LAPACK style" of coding has been that coding at a lower level yields high performance, especially for small problem sizes, operations that perform $O(n^2)$ computation with $O(n^2)$ data, and/or computations on banded matrices. The presented prototype source-to-source transformer, FLAMES2S, and the preliminary results neutralize this objection.

It is the very rigid format used for typesetting FLAME algorithms and representing them in code that greatly simplifies FLAMES2S. This suggests that a greater transformation is possible: one that transforms

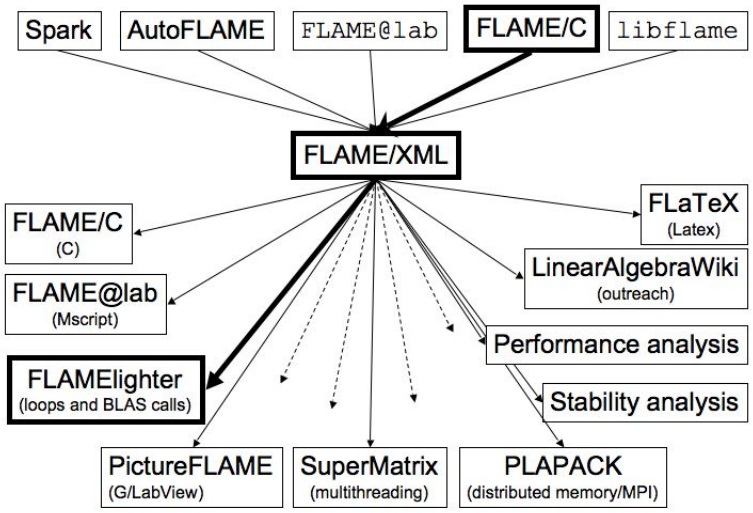

Figure 8: FLAME2S: A system for transforming algorithm specifications into various representations. This paper discusses the highlighted path of transformation.

the landscape of linear algebra libraries.

The key is the realization that a few parameters together with the updates to various submatrices and/or vectors define a typical dense linear algebra algorithm. This means that a library no longer needs to exist in a specific language instantiation. Rather, it can be represented and stored in a language-independent format. In this paper, we suggest that a collection of XML descriptions is convenient, but one could imagine other formats. To create an XML description, we have a number of options at our disposal, as illustrated in Figure 8. In this paper, we describe how input can be code that is represented with the FLAME/C API, which is transformed by the first stage of FLAMES2S. But inputs could also be fed from from our `libflame` library, from a modified Spark webpage (where a user can also enter the updates to be performed), or from a mechanical system for deriving algorithms (AutoFLAME) like the one described in [2]. A second stage of the FLAMES2S could then yield high-performance FLAMElighter code similar to the code in Figure 5. Alternatively, the transformer could yield representations using our LaTeX [28] commands for typesetting algorithms, FLAME@lab code [5] for MATLAB [18] (or Octave [11] or LabVIEW Mathscript [20]), FLAME/C code as in Figure 1 (right), PictureFLAME code (a FLAME API for LabVIEW's G programming language), PLAPACK code for distributed-memory architectures [26], or even cost [12] and/or stability analyses [2]. We illustrate this in Figure 8. While a system could be designed to simply produce such output from any of the described methods of input, we believe there is a benefit to providing a separation of concern, making the XML description (or similar representation) the intermediate representation.

We dub this system, which for now remains only a vision, the FLAME-to-source transformer (FLAME2S). The natural next step is to add to this intermediate description the ability to express expert knowledge that can be exploited by the transformer. This would allow one to perform automated optimizations, similar to what an optimizing compiler would do when compiling source code to machine language.

Acknowledgments

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin. Enrique S. Quintana-Ortí is also partially supported by the CICYT project TIN2008-09037-C02-02 and FEDER, and project P1B-2007-19 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. We thank the other members of the FLAME team for their support.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[2] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, 2006. Technical Report TR-06-46. September 2006.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[4] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*, 35(1), 2009.

[5] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[6] 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.

[7] Luiz DeRose and David Padua. A MATLAB to FORTRAN90 translator and its effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing*, 1996.

[8] Luiz Antonio DeRose. *Compiler Techniques for MATLAB Programs*. PhD thesis, Computer Sciences Department, The University of Illinois at Urbana-Champaign, 1996.

[9] 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.

[10] 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.

[11] GNU Octave. `www.octave.org`.

[12] John 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.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

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

[15] 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.

[16] libflame. `www.cs.utexas.edu/users/flame/libflame/`.

[17] Bret Andrew Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation.* PhD thesis, Computer Sciences Department, The University of Illinois at Urbana-Champaign, 1997.

[18] C. Moler, J. Little, and S. Bangert. *Pro-Matlab, User's Guide.* The Mathworks, Inc., 1987.

[19] Cleve B. Moler. MATLAB– an interactive matrix laboratory. Technical Report Technical Report 369, Department of Mathematics and Statistics, University of New Mexico, 1980.

[20] NI LabView MathScript. `www.ni.com/labview/mathscript.htm`.

[21] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[22] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, 2009. To appear.

[23] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software.* To appear.

[24] Jeremy G. Siek, Ian Karlin, and Elizabeth R. Jessup. Build to order linear algebra kernels. In *International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008)*, pages 1–8, 2008.

[25] Spark code skeleton generator. `www.cs.utexas.edu/users/flame/Spark/`.

[26] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[27] Robert A. van de Geijn. Beautiful parallel code: evolution vs. intelligent design. FLAME Working Note #34 TR-08-46, The University of Texas at Austin, Department of Computer Sciences, November 2008.

[28] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations.* `www.lulu.com/content/1911788`, 2008.

[29] Field G. Van Zee. `libflame`*: The Complete Reference.* `www.cs.utexas.edu/users/flame/publications/`. In preparation.