Beautiful Parallel Code: Evolution vs. Intelligent Design

FLAME Working Note #34

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rvdg@cs.utexas.edu

November 21, 2008

Abstract

The individual node of a massively parallel distributed memory cluster is now itself a complex parallel computer with multiple processors, each of which may have multiple cores and/or hardware accelerators. We focus on the question of how to effectively code such nodes targeting the problem domain of dense matrix computations. We argue that the traditional way of programming leads to a level of complexity that stiffles productivity, performance, and flexibility. We review lessons learned from our own FLAME project that appear to point to what we consider a more intelligent and desirable design.

1 Introduction

The most widely used linear algebra library, LAPACK, has evolved from the 1970s LINPACK library, and itself subsequently evolved into the ScaLAPACK library for distributed memory architecture, as detailed in the recent paper "How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination" [1, 10, 14]. In the conclusion of that paper and recent talks by one of its authors [13], it is pointed out that with the advent of multicore architectures, a major rewrite of LAPACK is again in order. In this paper, we detail an alternative design for such libraries that not only has demonstrated itself to address the same environments as did LAPACK and ScaLAPACK, but appears to be portable to new architectures that now frequently appear at the heart of the nodes of parallel supercomputers: multi-socket and/or multicore architectures with or without hardware accelerators (such as multiple GPUs and/or B.E. Cell processors).

2 First Impressions

First, let us address the question of beauty without defining it, by considering the codes in Fig. 1. The first is representative of code in the Linear Algebra Package (LAPACK) library. The second is representative of code in our libFLAME library [23] that uses the FLAME/C API [18, 5, 30]. Both compute the LU factorization with partial pivoting of a matrix, which is equivalent to the familiar Gaussian elimination with rearrangement of rows as the computation unfolds. In this paper, we will refer to these styles of coding as LAPACK-style and FLAME-style, respectively.

```
        DO 20 J = 1, MIN( M, N ), NB
           JB = MIN( MIN( M, N )-J+1, NB )
*
*          Factor diagonal and subdiagonal blocks and test for exact
*          singularity.
*
           CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
*
*          Adjust INFO and the pivot indices.
*
           IF( INFO.EQ.0 .AND. IINFO.GT.0 )
     $        INFO = IINFO + J - 1
           DO 10 I = J, MIN( M, J+JB-1 )
              IPIV( I ) = J - 1 + IPIV( I )
   10      CONTINUE
*
*          Apply interchanges to columns 1:J-1.
*
           CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )
*
           IF( J+JB.LE.N ) THEN
*
*             Apply interchanges to columns J+JB:N.
*
              CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1,
     $                     IPIV, 1 )
*
*             Compute block row of U.
*
              CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                    N-J-JB+1, ONE, A( J, J ), LDA, A( J, J+JB ),
     $                    LDA )
              IF( J+JB.LE.M ) THEN
*
*                Update trailing submatrix.
*
                 CALL DGEMM( 'No transpose', 'No transpose', M-J-JB+1,
     $                       N-J-JB+1, JB, -ONE, A( J+JB, J ), LDA,
     $                       A( J, J+JB ), LDA, ONE, A( J+JB, J+JB ),
     $                       LDA )
              END IF
           END IF
   20   CONTINUE
```

```
FLA_Part_2x2( A,    &ATL, &ATR,
                    &ABL, &ABR,     0, 0, FLA_TL );

FLA_Part_2x1( p,    &pT,
                    &pB,            0, FLA_TOP );

while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
  b = min( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ), nb_alg );

  FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,      &A00, /**/ &A01, &A02,
                       /* ************* */  /* ******************** */
                                             &A10, /**/ &A11, &A12,
                         ABL, /**/ ABR,      &A20, /**/ &A21, &A22,  b, b, FLA_BR );

  FLA_Repart_2x1_to_3x1( pT,                 &p0,
                       /* ** */            /* ** */
                                             &p1,
                         pB,                 &p2,        b, FLA_BOTTOM );
  /*--------------------------------------------------------------------------*/
  FLA_Merge_2x1( A11,
                 A21,      &AB1 );

  FLA_LU_piv_unb_var5( AB1, p1 );

  FLA_Apply_multiple_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p1, A10,
                                                          A20 );

  FLA_Apply_multiple_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p1, A12,
                                                          A22 );

  FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
            FLA_ONE, A11, A12 );

  FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
            FLA_MINUS_ONE, A21, A12, FLA_ONE, A22 );
  /*--------------------------------------------------------------------------*/
  FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                   A10, A11, /**/ A12,
                          /* ************** */  /* ****************** */
                            &ABL, /**/ &ABR,       A20, A21, /**/ A22,  FLA_TL );

  FLA_Cont_with_3x1_to_2x1( &pT,                 p0,
                                                 p1,
                          /* ** */             /* ** */
                            &pB,                 p2,     FLA_TOP );
}
```

Figure 1: Code segments for blocked LU factorization with partial pivoting. Left: LAPACK. Right: FLAME/C.

3 What Makes Code Desirable?

Beauty is to a large degree in the eye of the beholder, so let us instead focus on what makes the FLAME code in Fig. 1(right) desirable in the domain of dense matrix computations.

Code is a Representation of an Algorithm Ideally code is a natural representation of the underlying algorithm for computing the operation being implemented. We sketch how the given blocked algorithm for computing LU factorization with partial pivoting is often explained and present the algorithm in a notation that reflects that explanation. This then reveals the implementation in Fig. 1(right) as a translation of the algorithm to C code, using an API implemented as a collection of library routines.

The *blocked algorithm* without pivoting that underlies the codes in Fig. 1 is typically explained as follows: Consider $A = LU$ and partition the matrices into quadrants so that

$$\left(\begin{array}{c|c} A_{11} & A_{01} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} U_{11} & U_{01} \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11}U_{11} & L_{11}U_{12} \\ \hline L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{array}\right),$$

where A_{11}, L_{11} and U_{11} are $b \times b$ submatrices. Then

$$\begin{array}{c|c} A_{11} = L_{11}U_{11} & A_{12} = L_{11}U_{12} \\ \hline A_{21} = L_{21}U_{11} & A_{22} = L_{21}U_{12} + L_{22}U_{22}. \end{array}$$

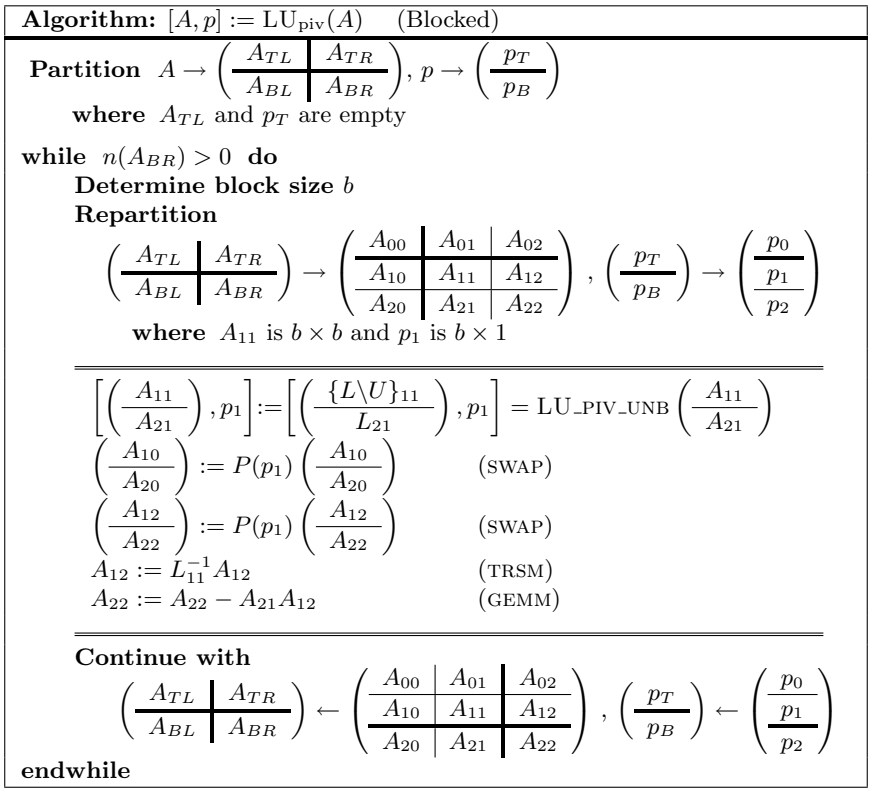

Figure 2: Right-looking blocked LU factorization algorithms with pivoting.

If L and U overwrite the original matrix A as part of the factorization, then these equations suggest that (1) A_{11} be overwritten with its LU factors L_{11} and U_{11}; (2) A_{21} be overwritten with the solution of $X_{21}U_{11} = A_{21}$; (3) A_{12} be overwritten with the solution of $L_{11}X_{12} = A_{12}$; (4) A_{22} be updated with $A_{22} - L_{21}U_{12}$. The algorithm proceeds by computing the LU factorization of the updated A_{22}. Pivoting can be added to this process to yield the algorithm in Fig. 2, which uses a notation that we employ in our papers.

By comparing the algorithm in Fig. 2 with the code in Fig. 1(right) it becomes obvious that that code is a direct translation. With the aid of the webpage illustrated in Fig. 3 that generates a code skeleton, it takes minutes to covert an algorithm expressed as in Fig. 2 to a C implementation as given in Fig. 1(right).

Correctness Code that cannot be determined to be correct is not desirable. The FLAME/C API allows the algorithm in Fig. 2 to be translated directly into the code in Fig. 1(right). Because of this, *if* the algorithm is correct, *then* we can assert a high degree of confidence in the code. This leaves us to discuss how to ensure that the algorithm is correct.

In a large number of papers [20, 18, 3], two dissertations [17, 2], and a book [30], we have shown that the algorithm notation used in Fig. 2 also supports the systematic derivation of these algorithms hand-in-hand with their proof of correctness. From the mathematical specification of the matrix operation, a family of loop-based algorithms for computing that operation can be derived to be correct. The methodology is sufficiently systematic that it has been made mechanical [2]. This allows us to assert an unprecedented level of confidence in the correctness of our algorithms and, consequently, our code.

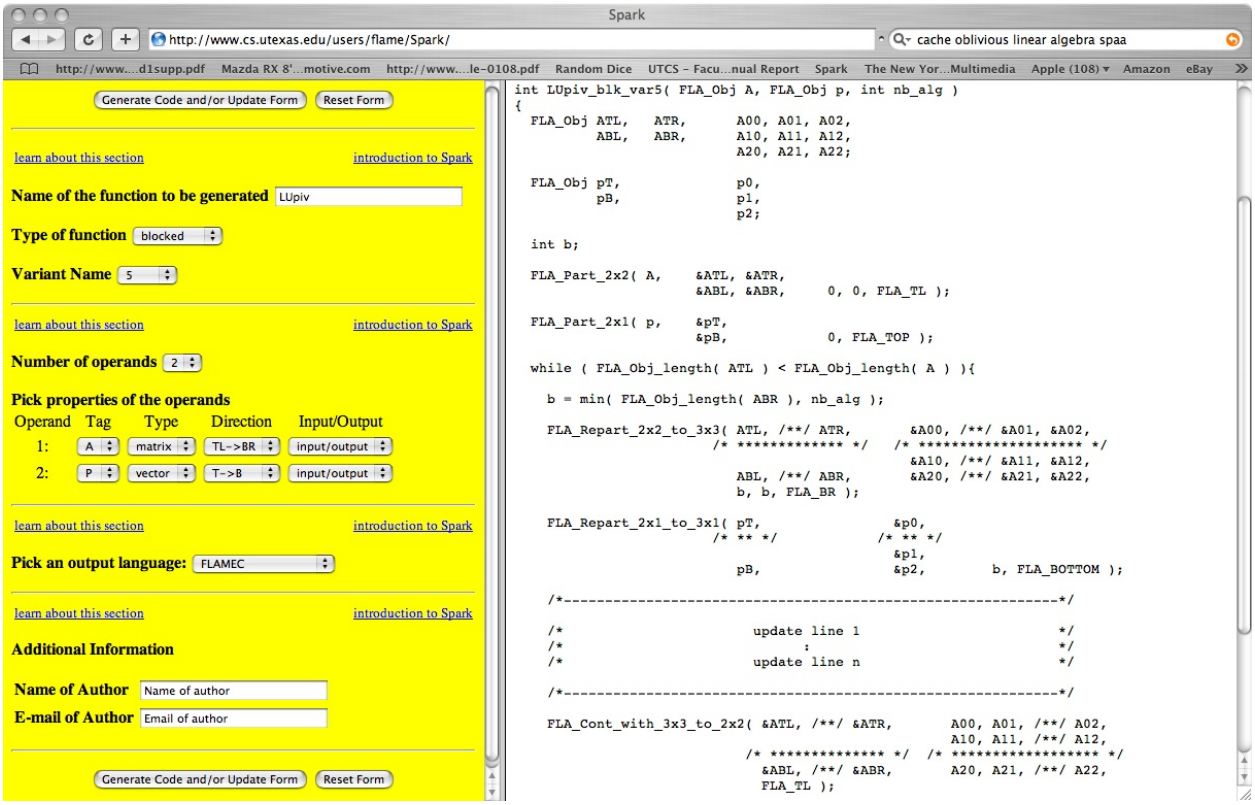

Figure 3: Spark webpage for generating code skeletons [28].

Numerical stability Perhaps the most compelling reason for evolving an existing code as new hardware arrives is that the numerical properties of a given implementation have been painstakingly established and that these properties are inherited by a code that is evolved. We note, however, that stability is a property of the algorithm rather than the implementation. Expressing the algorithm using different notation and the code using a different API does not change the numerical properties. Moreover, in a recent dissertation [2], it has been shown that the mentioned systematic derivation of algorithms for dense matrix computations can be extended to yield a systematic derivation of the numerical stability analysis of the algorithms.

High performance Conventional wisdom dictates that raising the level of abstraction of a code adversely impacts performance, which is unacceptable to the computational science community.

In the case of dense matrix computations, this conventional wisdom does not hold for a number of reasons. In blocked algorithms, the cost of tracking submatrices as in Fig. 1(right) is ammortized over enough computation that performance is virtually unaffected. Also, the FLAME derivation methodology yields multiple algorithms so that the best algorithm for a given architecture can be chosen (accommodating multiple algorithms in traditional code is cumbersome). A compelling example of the benefits of families of algorithms is given in [4]. Finally, in yet unpublished work we have shown that FLAME-style code can be translated to LAPACK-style code via source-to-source translators, thus removing the last reason not to code at a higher level of abstraction.

4 Evolution and Intelligent Design in a Changing Environment

We now discuss how the resulting design, built upon insight and hindsight, supports different situations.

Recursive algorithms Some researchers believe that recursive algorithms can naturally adapt to architectures with multilevel memories since eventually submatrices will fit in a given layer of memory [11]. There is strong evidence that loop-based algorithms are inherently more efficient in the domain of dense linear algebra [31, 19]. Regardless, there will likely be a benefit to combining recursion with loop-based algorithms. For example, the factorization of the current panel in the algorithms in Fig. 2(right) can itself be implemented as an invocation of a blocked algorithm (not necessarily the same algorithm) with a smaller block size.

The paper "How Elegant Code Evolves ..." shows how a dramatic change in LAPACK-style code is needed to implement a recursive algorithm. By contrast, the code in Fig. 1(right) can implement a purely recursive algorithm by choosing the block size to be half of the width of matrix `A` and calling the routine recursively until the matrix consists of only one column. But it also can combine recursion with iteration by instead calling itself (or a routine that implements another algorithmic variant of LU factorization) with a smaller block size. Indeed, our libFLAME library already supports a mechanism which allows arbitrary recursion, whereby algorithmic variants and block sizes are encoded *a priori* within the nodes of a control tree and then decoded by the operation as subproblems execute.

Algorithms-by-blocks It has long been speculated that the time would come when it would become beneficial to store dense matrices by blocks (e.g., by submatrices that are mapped to contiguous memory) rather than the traditional row- or column-major order [21]. Coding algorithms when matrices are stored by blocks greatly complicates implementations when the LAPACK-style of coding is employed.

Adapting the code in Fig. 1(right) to storage-by-blocks is relatively straight forward. The details related to the matrix `A` are already encoded in a descriptor (object) and that easily supports a "matrix of matrices" by allowing elements of the matrix themselves to be matrices. This is then supports (hierarchical) storage-by-blocks similar to what was proposed in [12].

Details on the extension of the FLAME API that supports matrices stored (hierarchically) by blocks can be found in [24, 27].

Multithreaded parallelism With the arrival of multicore architectures, how to develop programs for architectures that can execute multiple threads simultaneously has become a major topic. The FLAME-style of programming elegantly supports multithreaded parallelism at multiple levels:

- Functionality supported by LAPACK implemented using the FLAME/C API can be linked to multithreaded Basic Linear Algebra Subprograms (BLAS, a standard interface for fundamental matrix and vector computations) libraries [22, 16, 15]. Since the FLAME approach yields multiple algorithms, an algorithm that casts most of its computation in terms of BLAS operations that parallelize well can be chosen [4].
- The FLAME methodology can be used to derive algorithms for the BLAS themselves and algorithms that exhibit parallelism can be chosen. Several runtime mechanisms have been designed so that minimal or no change the FLAME/C code that implement BLAS operations can be parallelized by using OpenMP or pthreads [7, 32, 8, 25]

- Algorithms-by-blocks can be used to expose operations on submatrices as DAGs that can then be scheduled out-of-order much like superscalar architectures schedule individual operations. It is shown to be very elegantly supported by the FLAME-style of coding [7, 27]. This is also recognized by LAPACK-related developers [6].

A flexible style of programming is particularly important when targeting multicore architectures since it is now knows how these architectures will change even in the next five years.

As a measure of how flexible the FLAME-style of coding is, consider that it took two of our collaborators a weekend to convert all level-3 BLAS (matrix-matrix operations) from sequential implementations implemented with the FLAME/C API to implementations of algorithms-by-blocks interfaced to a runtime system that schedules suboperations to threads, as detailed in [9].

Distributed memory architectures The design of FLAME as a development methodology and API has its roots in the Parallel Linear Algebra Package (PLAPACK) [29]. The PLAPACK API hides the details of indexing specifically to overcome the complexity that can become nearly unmanageable when programming distributed memory architectures yielding code that is very similar to that presented in Fig. 1(right).

Targeting exotic architectures There is currently a lot of interest in using architectures like the B.E. Cell processor and GPUs as linear algebra accellerators by farming out the most compute intensive operations to such processors. The donation of a system with four NVIDIA Tesla processors to our collaborators at Univ. Jaume I provided an opportunity to evaluate how easily the FLAME-style of coding can be adapted to an uncharted environment. In [26] we describe how very respectable performance was easily attained for an operation closely related to LU factorization: the Cholesky factorization of a Symmetric Positive Definite matrix.

5 Conclusion

In computer sciences, as in any science, there are periods during which discovery is incremental and periods of insight that radically change our understanding. The field is distinguished from, for example, physics in that there are no laws of programming like there are laws of nature. When we gain insight into how to program, we can change the rules we follow. This means that programs need not evolve with a change in demands. A complete or partial redesign can, and we argue should, be embraced.

Acknowledgments This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714.

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).

We thank members of the FLAME team for their support.

Further information Please visit `http://www.cs.utexas.edu/users/flame/`

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, 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 Trans. Math. Soft.*, 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] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.

[7] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007. ACM.

[8] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and practices of parallel programming (PPoPP'08)*, 2008.

[9] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proceedings of IEEE Cluster Computing 2007*, pages 91–99, September 2007.

[10] 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.

[11] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 71–80, New York, NY, USA, 2007. ACM.

[12] Timothy Collins and James C. Browne. Matrix++: An objectoriented environment for parallel high-performance matrix computations. In *Proc. of the Hawaii Intl. Conf. on Systems and Software*, 1995.

[13] Jack Dongarra. The challenges of multicore and specialized accelerators. Talk presented at the International Advanced Research Workshop on High Performance Computing and Grids, Cetraro (Italy), July 2006.

[14] Jack Dongarra and Piotr Luszczak. How elegant code evolves with hardware: The case of Gaussian elimination. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 14, pages 229–252. O'Reilly and Associates, Sebastopol, CA, 2007.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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-Verlag, 2001.

[20] 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.

[21] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

[22] 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.

[23] libflame Release Notes. `http://www.cs.utexas.edu/users/flame/ReleaseNotes/`.

[24] Tze Meng Low and Robert van de Geijn. An api for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[25] Bryan Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, LNCS 4641, pages 748–757, 2007.

[26] 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. Submitted.

[27] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert van de Geijn. Programming algorithms-by-blocks for matrix computations on multithreaded architectures. FLAME Working Note #29 TR-08-04, The University of Texas at Austin, Department of Computer Sciences, January 2008. Submitted to ACM TOMS.

[28] Spark: Code skeleton generator. `http://www.cs.utexas.edu/users/flame/Spark/`.

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

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

[31] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, New York, NY, USA, 2007. ACM.

[32] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable parallelization of flame code via the workqueuing model. *ACM Trans. Math. Soft.*, 34(2):10, March 2008. Article 10, 29 pages.