The Science of Programming High-Performance Linear Algebra Libraries

Enrique S. Quintana-Ortí

Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I de Castellón (Spain)
quintana@icc.uji.es

December 12, 2003

In collaboration with
Robert A. van de Geijn (UT-Austin) et al

Motivation

[Dijkstra 1976]:

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

Usual approach in Linear Algebra (LA):

Prove correctness by exhaustive testing

Example: LAPACK

Is it possible to formally prove correctness?

At least for dense LA, the answer seems to be yes

How do we derive an algorithm while, simultaneously, proving correctness?

Outline

- The FLAME approach to deriving LA libraries
- Impact on the Implementation of LA Algorithms
- Impact on Performance
- Impact on Library Development and Maintainability
- Open Questions

The Formal Linear Algebra Methods Environment:
(or FLAME for short)

– A systematic approach for deriving provably correct algorithms

  Algorithms are proved to be correct at the same time they are derived

– Different APIs (C, Matlab) for the implementation

Some notation and preliminaries:

- Predicate. Example:

  $\gamma := 1$

  $P: \ \{\gamma = 1\}$

- Precondition and postcondition. Example:

  $P_{\text{pre}}: \ \{\gamma = \beta\}$

  $\gamma := \gamma + 1$

  $P_{\text{post}}: \ \{\gamma = (\beta + 1)\}$

- Loop guard G and loop-invariant P_{inv}

  Consider the loop

  while G **do**

  S

  enddo

  P_{inv} is an assertion that holds before and after each iteration

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

A very simple example: Vector scaling

Level-1 BLAS routine `xSCAL`

$$x := \alpha \cdot x$$

where α is a scalar and x is $n \times 1$

Step 1. Determine the P_{pre} and P_{post}

P_{pre} specifies the conditions before the operation commences:

$$x \text{ is } n \times 1, \quad \hat{x} = x$$

P_{post} specifies the desired state upon completion:

$$x = \alpha \cdot \hat{x}$$

Step 2. Determine P_{inv}

Start by partitioning the operands:

$$x = \left(\frac{x_T}{\overline{x_B}} \right), \quad \hat{x} = \left(\frac{\hat{x}_T}{\overline{\hat{x}_B}} \right)$$

and plug the partitioned vector into P_{post},

$$\left(\frac{x_T}{\overline{x_B}} \right) = \alpha \cdot \left(\frac{\hat{x}_T}{\overline{\hat{x}_B}} \right)$$

Thus,

$$\left(\frac{x_T = \alpha \cdot \hat{x}_T}{\overline{x_B = \alpha \cdot \hat{x}_B}} \right)$$

At an intermediate state, only one of the operations

$$x_T := \alpha \cdot \hat{x}_T$$
$$x_B := \alpha \cdot \hat{x}_B$$

will have been performed

These two possibilities lead us to two different loop-invariants

Computed?		$P_{\text{inv}} : \left(\frac{x_T}{x_B}\right) =$	Comment
$\alpha \cdot \hat{x}_T$	$\alpha \cdot \hat{x}_B$		
yes	no	$\left(\frac{\alpha \cdot \hat{x}_T}{\hat{x}_B}\right)$	Variant 1
no	yes	$\left(\frac{\hat{x}_T}{\alpha \cdot \hat{x}_B}\right)$	Variant 2

In our example, we will assume that, at the intermediate stage, the state is given by

$$\left(\frac{x_T}{x_B}\right) = \left(\frac{\alpha \cdot \hat{x}_T}{\hat{x}_B}\right)$$

(variant 1) which becomes P_{inv} in our worksheet

Step 3. Determine the loop-guard G

After the loop completes, $P_{\rm inv} \wedge \neg G$ holds. Thus, we should select G such that $(P_{\rm inv} \wedge \neg G) \Rightarrow P_{\rm post}$

Notice that when x_T equals all of x,

$$\left(\frac{x_T}{x_B} \right) = \left(\frac{\alpha \cdot \hat{x}_T}{\hat{x}_B} \right) \wedge \ \mathrm{SameSize}(x, x_T) \quad \Rightarrow \quad x = \alpha \cdot \hat{x}$$

so that a possible guard is

$$\neg \mathrm{SameSize}(x, x_T)$$

Step 4: Determine the initialization

Starting in a state where P_{pre}

$$x \text{ is } n \times 1, \quad \hat{x} = x$$

holds, the initialization must lead us to a state where P_{inv}

$$\left(\frac{x_T}{\overline{x_B}}\right) = \left(\frac{\alpha \cdot \hat{x}_T}{\overline{\hat{x}_B}}\right)$$

holds

In our case, the initial partitioning $x = \left(\frac{x_T}{\overline{x_B}}\right)$,

where x_T has 0 rows, has the desired effect

Step 5. Determine how to move boundaries

Initially, x_T has 0 rows and, upon completion, x_T should equal x

Thus, the boundary, denoted by the double lines, must be moved forward as part of the body of the loop.

This leads us to the following repartitioning of the vector

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline\hline x_1 \\ \hline x_2 \end{array}\right),$$

where x_1 has b rows

After the current iteration, the computation moves forward by b rows in x, as shown next

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline\hline x_2 \end{array}\right)$$

Step 6. Determine Q_{before}

Substitution into P_{inv}

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) = \left(\begin{array}{c} \alpha \cdot \hat{x}_T \\ \hline\hline \hat{x}_B \end{array}\right)$$

of the subvectors that make up the different regions give Q_{before}. In our case,

$$\left(\begin{array}{c} x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right)$$

Step 7. Determine Q_{after}

After the repartitioning that identifies the parts of the vectors to be moved between regions, P_{inv}

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) = \left(\begin{array}{c} \alpha \cdot \hat{x}_T \\ \hline\hline \hat{x}_B \end{array}\right)$$

must again be true, so

$$\left(\begin{array}{c} \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline \alpha \cdot x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right)$$

Step 8. Determine the update S_U

By comparing the state in Step 6,

$$\left(\begin{array}{c} x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right),$$

with the desired state in Step 7,

$$\left(\begin{array}{c} \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline \alpha \cdot x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right),$$

the required update can be easily determined:

$$x_1 := \alpha \cdot x_1$$

$\{\ x \text{ is } n \times 1, \quad \hat{x} = x\ \}$

Partition $x = \left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right)$, **where** x_T has 0 rows

while $\neg\text{SameSize}(x, x_T)$ **do**

$\left\{ \left(\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) = \left(\begin{array}{c} \alpha \cdot \hat{x}_T \\ \hline\hline \hat{x}_B \end{array}\right) \right) \wedge (\neg\text{SameSize}(x, x_T)) \right\}$

Determine block size b

Repartition

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline\hline x_1 \\ \hline x_2 \end{array}\right), \textbf{ where } x_1 \text{ has } b \text{ rows}$$

$$\left\{ \left(\begin{array}{c} x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline\hline \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) \end{array}\right) \right\}$$

$x_1 := \alpha \cdot x_1$

$$\left\{ \left(\begin{array}{c} \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} \alpha \cdot x_0 \\ \hline \alpha \cdot x_1 \end{array}\right) \\ \hline\hline x_2 \end{array}\right) \right\}$$

Continue with

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline\hline x_2 \end{array}\right)$$

enddo

$\left\{ \left(\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) = \left(\begin{array}{c} \alpha \cdot \hat{x}_T \\ \hline\hline \hat{x}_B \end{array}\right) \right) \wedge \neg\,(\neg\text{SameSize}(x, x_T)) \right\}$

Notice that

– The loop terminates

– $P_{\text{inv}} \wedge \neg G$ imply the desired result is computed by the algorithm as

$$\left(\frac{x_T}{\overline{x_B}}\right) = \left(\frac{\alpha \cdot \hat{x}_T}{\overline{\hat{x}_B}}\right) \wedge \neg(\neg \text{SameSize}(x, x_T))$$

$$\Rightarrow x := \alpha \cdot x$$

The algorithm

Algorithm: $x := \alpha \cdot x$

Partition $x = \left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array} \right)$, **where** x_T has 0 rows

while $\neg\text{SameSize}(x, x_T)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array} \right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline\hline x_1 \\ \hline x_2 \end{array} \right), \textbf{ where } \; x_1 \text{ has } b \text{ rows}$$

$x_1 := \alpha \cdot x_1$

Continue with

$$\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array} \right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline\hline x_2 \end{array} \right)$$

enddo

Can we apply FLAME to deriving provably correct algorithms for more complex LA operations?

yes!!!

A more complex operation: LU factorization

$$A = LU$$

where A is $m \times m$

The following algorithm (classical) and four more variants (Sherman's march, Picket's charge$\times 2$, and Crout) were automatically derived using *Mathematica*
By construction they were all proved to be correct

Algorithm: $A := L\backslash U = \mathrm{LU}(A)$

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

where A_{TL} is 0×0

while $\neg\mathrm{SameSize}(x, x_T)$ **do**

Determine block size b

Repartition

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

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

$$A_{11} := L\backslash U_{11} = \mathrm{LU}(A_{11})$$
$$A_{12} := U_{12} = L_{11}^{-1} A_{12}$$
$$A_{21} := L_{21} = A_{21} U_{11}^{-1}$$
$$A_{22} := A_{22} - L_{21} U_{12}$$

Continue with

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

enddo

Can we apply FLAME to deriving provably correct algorithms for more complex LA operations?

Among others, applied to:

- All levels 1, 2, and 3 of BLAS
- All major factorizations: LU (with/without pivoting); Cholesky; QR (with/without pivoting) by Householder, Gram-Schmidt, and Givens
- Reduction to condensed forms: Hessenberg, bidiagonal
- Triangular linear matrix equations: Sylvester, Lyapunov
- Sparse linear algebra: Arnoldi and Lanczos algorithms

Impact on the Implementation of LA Algorithms

Capturing the algorithm in code: Typos, bugs, etc.?

Operations like partitionings, repartitionings, etc., are provided as part of the FLAME API; e.g., the repartitioning

$$\left(\begin{array}{c||c} A_{TL} & A_{BL}^T \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01}^T & A_{02}^T \\ \hline\hline A_{10} & A_{11} & A_{12}^T \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

is done using

```
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_Part_2x2( A,  &ATL, /**/ &ATR,
                /* ************** */
                  &ABL, /**/ &ABR, /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

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

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

  /* ************************************************************************ */
  FLA_LU_nopivot_level2( A11 );

  FLA_Trsm( FLA_SIDE_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
            ONE, A11, A21 );
  FLA_Trsm( FLA_SIDE_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
            ONE, A11, A12 );
  FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22 );
  /* ************************************************************************ */

  FLA_Cont_3x3_to_2x2( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
                             /**/               A10, A11, /**/ A12,
                     /* ************* */    /* ***************** */
                        &ABL, /**/ &ABR,        A20, A21, /**/ A22,
               /* with A11 added to submatrix */ FLA_TL );

}
```

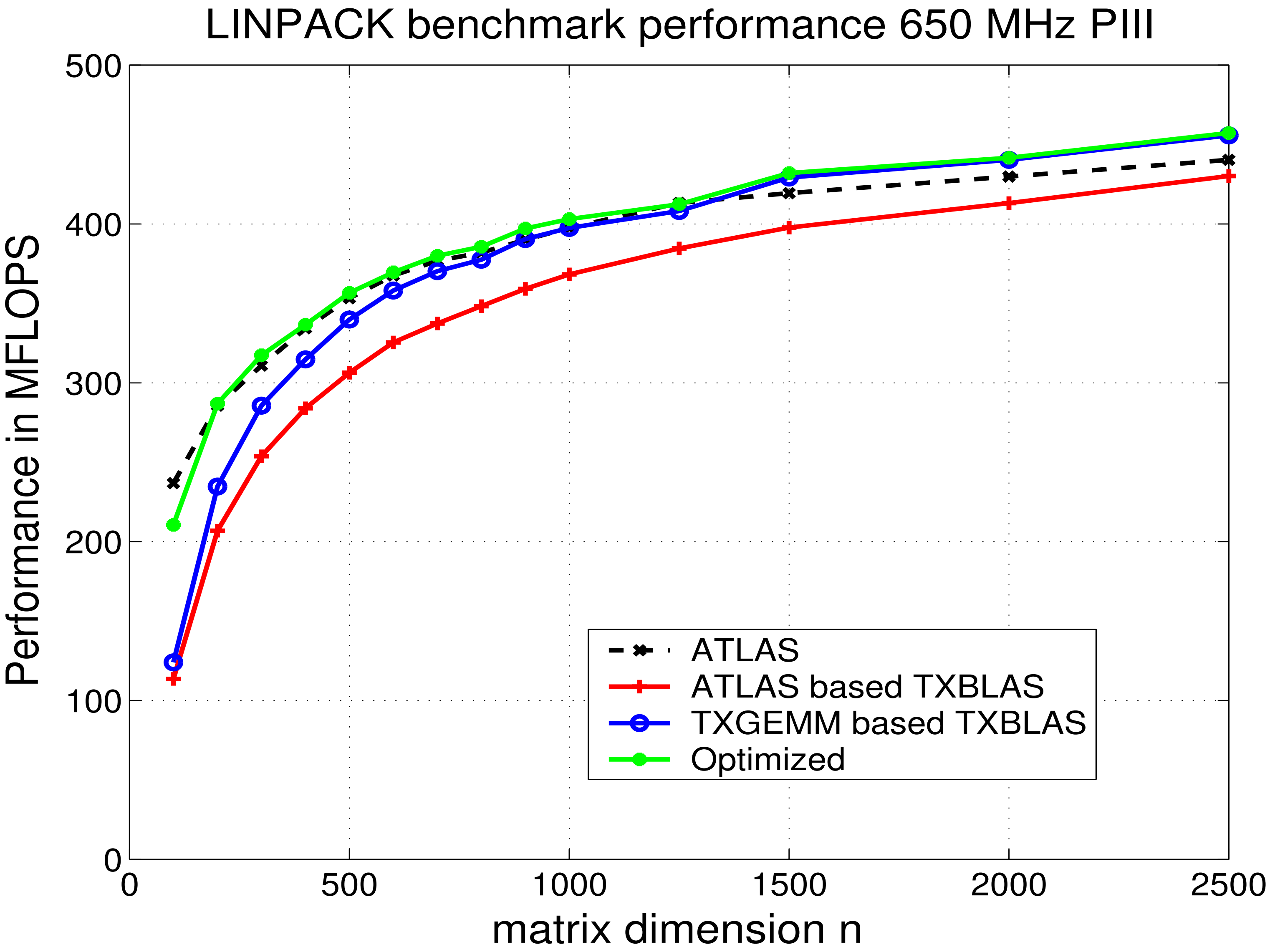
Parallel code?

Use the FLAME to PLAPACK API:

Just replace `FLA_Part_...` by `PLA_Part_...`

Impact on Performance

- IEEE double precision arithmetic
- Intel Pentium III (650 MHz)
- Linux (Redhat 6.2)
- Peak achieved for matrix-matrix multiply:
  - ⋆ Around 490 MFLOPS for ATLAS matrix-matrix multiply
- LINPACK benchmark (unofficial): LU factorization with partial pivoting plus triangular system solve


LINPACK benchmark performance 650 MHz PIII
Performance in MFLOPS
matrix dimension n
0
100
200
300
400
500
0
500
1000
1500
2000
2500
ATLAS
ATLAS based TXBLAS
TXGEMM based TXBLAS
Optimized

Impact on Library Development and Maintainability

- Productivity:

  Implementation of the level-3 BLAS by RvdG in 10 hours including testing!!!

- A recipe for deriving linear algebra libraries
- Automatic derivation of algorithms:

  Mathematica based system by P. Bientinesi (UT-Austin)

- Automatic generation of parallel libraries: John Gunnels' dissertation
- Fast translation from algorithm to code: FLAME APIs for Matlab, C, and parallel PLAPACK
- Simplicity: Implementation of the level-3 BLAS by linear-algebra-challenged undergraduates

Open questions

- Characterization of the class of operations FLAME applies to
- Automatic derivation of forward error analysis
- Criteria to choose between algorithmic variants

More information

- FLAME: `http://www.cs.utexas.edu/users/flame`
- J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM TOMS*, Dec. 2002
- P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The Science of Deriving Linear Algebra Algorithms. *ACM TOMS*, submitted
- 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, 2001. 200+ pages
- P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn. Representing Linear Algebra Algorithms in Code: The FLAME APIs. *ACM TOMS*, submitted
- Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal Derivation of Algorithms for the Triangular Sylvester Equation. ACM TOMS, July 2003