Introducing: The libflame Library for Dense Matrix Computations

Field G. Van Zee, Ernie Chan,
and Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{field,echan,rvdg}@cs.utexas.edu

Enrique S. Quintana-Ortí and
Gregorio Quintana-Ortí
Departamento de Ingeniería y
Ciencia de Computadores
Universidad Jaime I
12.071–Castellón, Spain
{quintana,gquintan}@icc.uji.es

Abstract

As part of the FLAME project, we have been dilligently developing new methodologies for analyzing, designing, and implementing linear algebra libraries. While we did not know it when we started, these techniques appear to solve many of the programmability problems that now face us with the advent of multicore and many-core architectures. These efforts have culminated in a new library, `libflame`, which strives to replace similar libraries that date back to the late 20th century. With this paper, we introduce the scientific computing community to this library.

1 Why a New Library

How do we convince people that in programming simplicity and clarity — in short: what mathematicians call "elegance" — are not a dispensable luxury, but a crucial matter that decides between success and failure?

Edsger W. Dijkstra

During the past decade, the FLAME project, a collaborative effort between The University of Texas at Austin and Universidad Jaime I de Castellon, has developed a unique methodology, notation, tools, and set of application programming interfaces (APIs) for deriving and representing linear algebra libraries. In an effort to better promote the techniques characteristic to the FLAME project, we have implemented a functional library that demonstrates findings and insights from the last decade of research. We call this library `libflame`.

The primary purpose of `libflame` is to provide the scientific and numerical computing communities with a modern, high-performance dense linear algebra library that is extensible, easy to use, and available under an open source license. Its developers have published two books, numerous papers, and even more working notes over the last decade documenting the challenges and motivations that led to the APIs and implementations present within the `libflame` library [11]. Seasoned users within scientific and numerical computing circles will quickly recognize the general set of functionality targeted by `libflame`. In short, in `libflame` we wish to provide not only a framework for developing dense linear algebra solutions, but also a ready-made library that is, by almost any metric, easier to use and offers competitive (and in many cases superior) real-world performance when compared to the more traditional Basic Linear Algebra Subprograms (BLAS) [8, 9, 16] and Linear Algebra PACKage (LAPACK) libraries [1].

The purpose of this article is to briefly introduce both the philosophy that underlies the library and the library itself. Evidence of how easily it can be retargeted to what are often considered "hostile" environments is presented in form of performance results on different architectures.

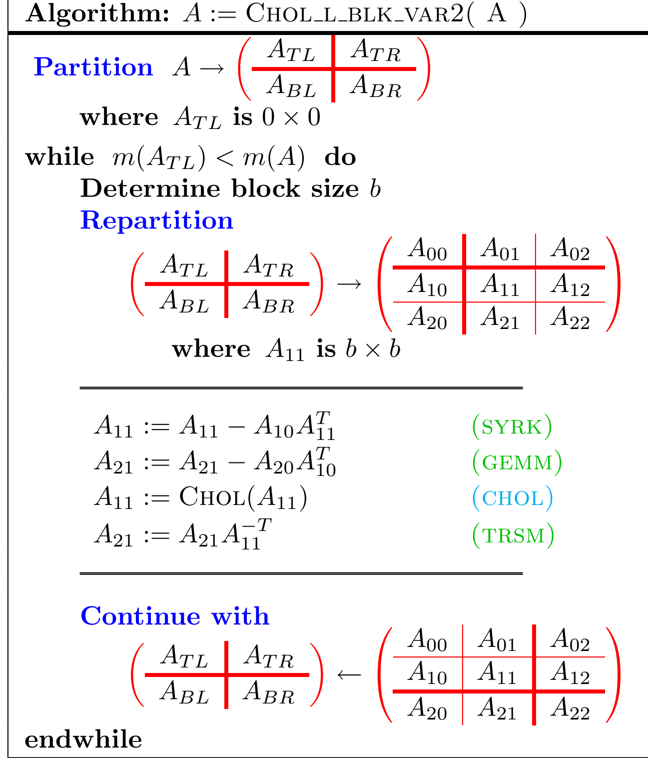

Figure 1: Blocked Cholesky factorization (variant 2) expressed as a FLAME algorithm. Subproblems annotated as SYRK, GEMM, and TRSM correspond to level-3 BLAS operations, and CHOL is the recursive Cholesky factorization subproblem.

2 What is Different

We believe there exist numerous reasons for users to adopt `libflame` into their software solutions:

A solution based on fundamental computer science. The FLAME project advocates a new approach to developing linear algebra libraries. It starts with a more stylized notation for expressing loop-based linear algebra algorithms [12, 13, 18, 25]. This notation closely resembles how matrix algorithms are naturally illustrated with pictures (Figure 1). The notation facilitates rigorous formal derivation of algorithms [2, 12, 20, 25], which guarantees that the resulting algorithms are correct. Moreover, it yields a family of algorithms so that the best option for a given situation (e.g., problem size or architecture) can be chosen.

Object-based abstractions and API. The BLAS, LAPACK, and ScaLAPACK [5] projects place backward compatibility as a high priority, which hinders progress towards adopting modern software engineering principles such as object abstraction. `libflame` is built around opaque structures that hide implementation details of matrices, such as data layout, and exports object-based programming interfaces to operate upon these structures [4]. Likewise, FLAME algorithms are expressed (and coded) in terms of smaller operations on sub-partitions of the matrix operands. This abstraction facilitates programming without array or loop indices, which allows the user to avoid painful index-related programming errors altogether. Figure 2 compares the coding styles of `libflame` and LAPACK, highlighting the inherent elegance of FLAME code and its striking resemblance to the corresponding FLAME algorithm shown in Figure 1. This similarity is quite intentional, as it preserves the clarity of the original algorithm as it would be illustrated on a white-board or in a publication.

libflame

```
FLA_Error FLA_Chol_l_blk_var2( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,       A00, A01, A02,
          ABL, ABR,       A10, A11, A12,
                          A20, A21, A22;
  int b, value;

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

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

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

    /* ------------------------------------------------------ */

    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A10, FLA_ONE, A11 );

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              FLA_MINUS_ONE, A20, A10, FLA_ONE, A21 );

    value = FLA_Chol_unb_external( FLA_LOWER_TRIANGULAR, A11 );

    if ( value != FLA_SUCCESS )
      return ( FLA_Obj_length( A00 ) + value );

    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );

    /* ------------------------------------------------------ */

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

  return value;
}
```

LAPACK

```
      SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )

      CHARACTER          UPLO
      INTEGER            INFO, LDA, N
      DOUBLE PRECISION   A( LDA, * )

      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
      LOGICAL            UPPER
      INTEGER            J, JB, NB
      LOGICAL            LSAME
      INTEGER            ILAENV
      EXTERNAL           LSAME, ILAENV
      EXTERNAL           DGEMM, DPOTF2, DSYRK, DTRSM, XERBLA
      INTRINSIC          MAX, MIN

      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DPOTRF', -INFO )
         RETURN
      END IF

      INFO = 0
      UPPER = LSAME( UPLO, 'U' )

      IF( N.EQ.0 )
     $   RETURN

      NB = ILAENV( 1, 'DPOTRF', UPLO, N, -1, -1, -1 )
      IF( NB.LE.1 .OR. NB.GE.N ) THEN
         CALL DPOTF2( UPLO, N, A, LDA, INFO )
      ELSE
         IF( UPPER ) THEN
*  Upper triangular case omitted for purpose of fair comparison.
         ELSE
            DO 20 J = 1, N, NB
               JB = MIN( NB, N-J+1 )
               CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
     $                     A( J, 1 ), LDA, ONE, A( J, J ), LDA )
               CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
               IF( INFO.NE.0 )
     $            GO TO 30
               IF( J+JB.LE.N ) THEN
                  CALL DGEMM( 'No transpose', 'Transpose',
     $                        N-J-JB+1, JB, J-1, -ONE,
     $                        A( J+JB, 1 ), LDA, A( J, 1 ),
     $                        LDA, ONE, A( J+JB, J ), LDA )
                  CALL DTRSM( 'Right', 'Lower', 'Transpose',
     $                        'Non-unit', N-J-JB+1, JB, ONE,
     $                        A( J, J ), LDA, A( J+JB, J ), LDA )
               END IF
   20       CONTINUE
         END IF
      END IF
      GO TO 40
   30 CONTINUE
      INFO = INFO + J - 1
   40 CONTINUE
      RETURN
      END
```

Figure 2: The algorithm shown in Figure 1 implemented with the FLAME/C API (left) and Fortran-77 (right). The FLAME/C code represents the style of coding found in `libflame` while the Fortran-77 code was obtained from LAPACK.

Educational value. Aside from the potential to introduce students to formal algorithm derivation, FLAME has been successfully used for teaching linear algebra algorithms in a classroom setting. The clean abstractions afforded by the API also make FLAME ideally suited for instruction of high-performance linear algebra courses at the undergraduate and graduate level. Historically, the BLAS/LAPACK style of coding has been used in these pedagogical settings. However, we believe that coding in that manner obscures the algorithms; students often get bogged down debugging the frustrating errors that often result from indexing directly into arrays that represent the matrices. As we like to put it, using FLAME in the classroom greatly reduces the line of students needing help with coding during office hours.

A complete dense linear algebra framework. Like LAPACK, `libflame` provides ready-made implementations of common linear algebra operations. The implementations found in `libflame` mirror many of those found in the BLAS and LAPACK packages. However, `libflame` differs from LAPACK in two impor-

tant ways. First, as mentioned, it provides families of algorithms for each operation so that the best can be chosen for a given circumstance [3]. Second, it provides a framework for building complete custom linear algebra codes. We believe this makes it a more useful environment as it allows the user to quickly choose and/or prototype a linear algebra solution to fit the needs of the application.

High performance. In our publications and performance graphs, we do our best to dispel the myth that user- and programmer-friendly linear algebra codes cannot yield high performance. Our FLAME implementations of operations such as Cholesky factorization and triangular matrix inversion often outperform the corresponding implementations currently available in LAPACK [3]. In Section 3 and numerous papers, we give numerous examples of how performance is facilitated by the FLAME methodology.

Currently, `libflame` relies only on the presence of a core set of highly optimized unblocked routines to perform the small subproblems found in implementations of linear algebra algorithms. However, we have recently demonstrated that algorithms represented with the FLAME/C API (Figure 2 (left)) can be mechanically translated to implementations that use a more traditional style of code (Figure 2 (right)) and thus attain the same performance as traditional code [26]. A source-to-source translator, FLAMES2S, has been prototyped and is being perfected. As a result, there is no valid reason left for continuing to code linear algebra algorithms in the traditional style of BLAS/LAPACK as opposed to a high level of abstraction.

Dependency-aware thread-level parallelism. Until recently, the authors of the BLAS and LAPACK advocated exploiting shared-memory parallelism from LAPACK routines by simply linking to multithreaded BLAS. This low-level solution requires no changes to LAPACK code but also suffers from sharp limitations in terms of efficiency and scalability for small- and medium-sized matrix problems, as we will see in Section 3. The fundamental bottleneck to introducing parallelism directly within many algorithms is the web of data dependencies that frequently exists between subproblems. The `libflame` project has developed a runtime system to detect and analyze dependencies found within algorithms-by-blocks (algorithms whose subproblems operate only on block operands) [6, 7, 21, 24]. Once data dependencies are known, the system schedules sub-operations to independent threads of execution. This system is completely abstracted from the algorithm that is being parallelized and requires virtually no change to the algorithm code, but at the same time exposes abundant high-level parallelism.

Support for hierarchical storage-by-blocks. Storing matrices by blocks often yields performance gains through improved spatial locality [10]. Instead of representing matrices as a single linear array of data with a prescribed leading dimension as legacy libraries require (for column- or row-major order), the storage scheme is encoded into the matrix object. Here, internal elements refer recursively to child objects that represent submatrices. Currently, `libflame` provides a subset of the conventional API that supports hierarchical matrices, allowing users to create and manage such matrix objects as well as convert between storage-by-blocks and conventional "flat" storage schemes [17, 24].

Advanced build system. From its early revisions, `libflame` distributions have been bundled with a robust build system, featuring automatic makefile creation and a configuration script conforming to GNU standards (allowing the user to run the `./configure; make; make install` sequence common to many open source software projects). Without any user input, the configure script searches for and chooses compilers based on a pre-defined preference order for each architecture. The user may request specific compilers via the configure interface, or enable other non-default features of `libflame` such as custom memory alignment, multithreading (via POSIX threads or OpenMP), compiler options (debugging symbols, warnings, optimizations), and memory leak detection. The reference BLAS and LAPACK libraries provide no configuration support and require the user to manually modify a makefile with appropriate references to compilers and compiler options depending on the host architecture.

Cross-platform support. The `libflame` distribution includes build systems for both GNU/Linux and Microsoft Windows.

Backward compatibility with LAPACK. We understand that you may have already invested a lot of time in your current dense linear algebra application. That is why we provide a set of compatibility routines that map conventional LAPACK invocations to their corresponding implementations within `libflame`. By simply linking to the `liblapack2flame` compatibility layer, you can take advantage of the performance benefits offered by FLAME with virtually no changes to your application.[1]

A detailed users manual. A detailed reference manual is available that details the library [27]. This document is kept up-to-date as library functionality is changed or added.

3 Performance

The scientific community expects performance. For decades it has been assumed that the price for solving the programmability problem is deminished performance and thus the kinds of abstractions that underlie `libflame` are "not allowed". We now show that these abstractions in fact yield performance in the same conditions under which traditional libraries thrive and flexibly support high performance under what are generally considered more hostile circumstances.

Sequential performance via optimized BLAS. Traditional libraries like LAPACK are layered upon the BLAS for portable performance. In its simplest mode, `libflame` is merely an alternative way of programming families of algorithms for operations, such as the LU, QR, and Cholesky factorizations, still in terms of traditional BLAS operations and linked to traditional BLAS libraries.

In Figure 3 (left), we show performance of three sequential implementations of LU factorization with partial pivoting: `dgetrf` from Intel's MKL library, `FLA_LU_piv` from `libflame`, and `dgetrf` from the reference implementation of LAPACK available from [15]. For this experiment, the target platform is a single Itanium2 processor. The implementation of MKL is highly optimized and attains excellent performance. Yet the other implementations, which are available under open source license, are highly competitive.

Traditional parallelism via multithreaded kernels. With the advent of parallel computers like symmetric multiprocessing (SMP), non-uniform memory access (NUMA), and multicore architectures, the simplest approach to parallelizing libraries like LAPACK and `libflame` is to link to multithreaded BLAS kernels so that the parallel implementations remain identical to the original sequential code.

In Figure 3 (right), we again show performance of three implementations of LU factorization with partial pivoting from MKL, `libflame`, and LAPACK. For this experiment, the target platform is a 16 Itanium2 CPU system. We notice that extracting parallelism only within the BLAS limits the performance attained by both the LAPACK and `libflame` implementations.

Scheduling algorithms-by-blocks to multiple threads. The problems with extracting parallelism only via multithreaded BLAS are numerous: (1) Each call to a BLAS operation ends with a synchronization of the threads (so-called fork-and-join parallelism); (2) The factorization of the "current panel" introduces an operation in the critical path during which there is a reduced opportunity for parallelism; (3) Computing operations out-of-order to facilitate more parallelism is difficult to implement without greatly complicating the code.

To overcome these problems, a number of abstractions have been added to `libflame`:

- The API has been extended to support storage of matrices by blocks. While traditional linear algebra libraries enforce column-major storage both in how the matrices are mapped to memory and in how the algorithm code is explicitly written, the object-based FLAME/C API allows elements in a matrix themselves to be descriptors of matices, thus providing a convenient way of expressing matrices that are hierarchically stored by blocks [17].
- The concept of algorithms-by-blocks was introduced. These algorithms view each element of a matrix to be a block and express the computation to be performed in terms of operations between those blocks.

[1] Currently, any operation called through the `liblapack2flame` compatibility layer will execute sequentially. In order to invoke our parallelized implementations, you must use native FLAME interfaces.

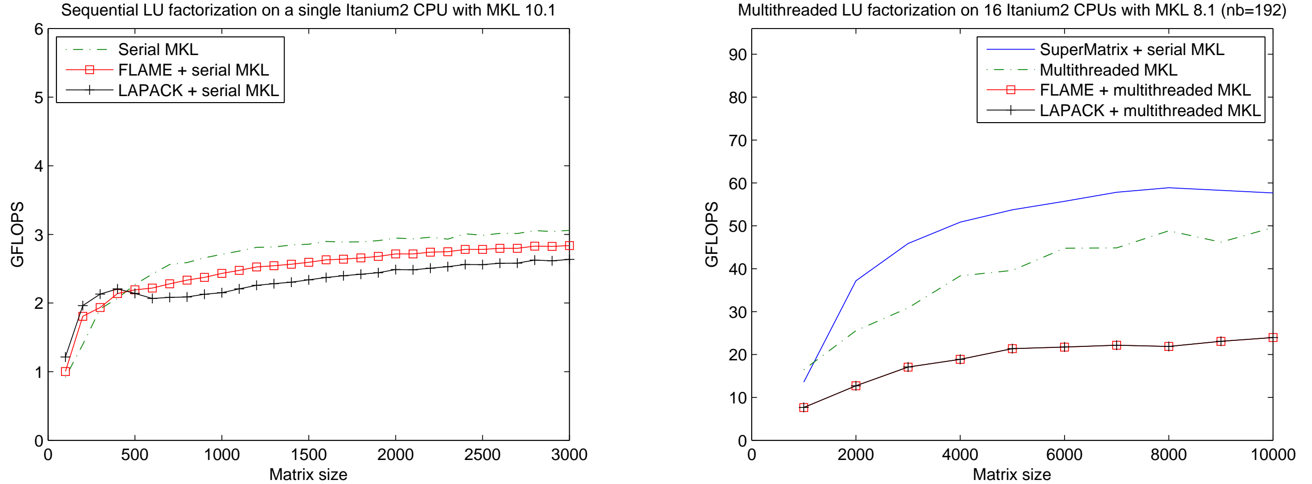

Figure 3: Representative performance of various library implementations of LU factorization with partial pivoting. Left: Sequential performance. Right: Performance on a 16 Itanium2 CPU system.

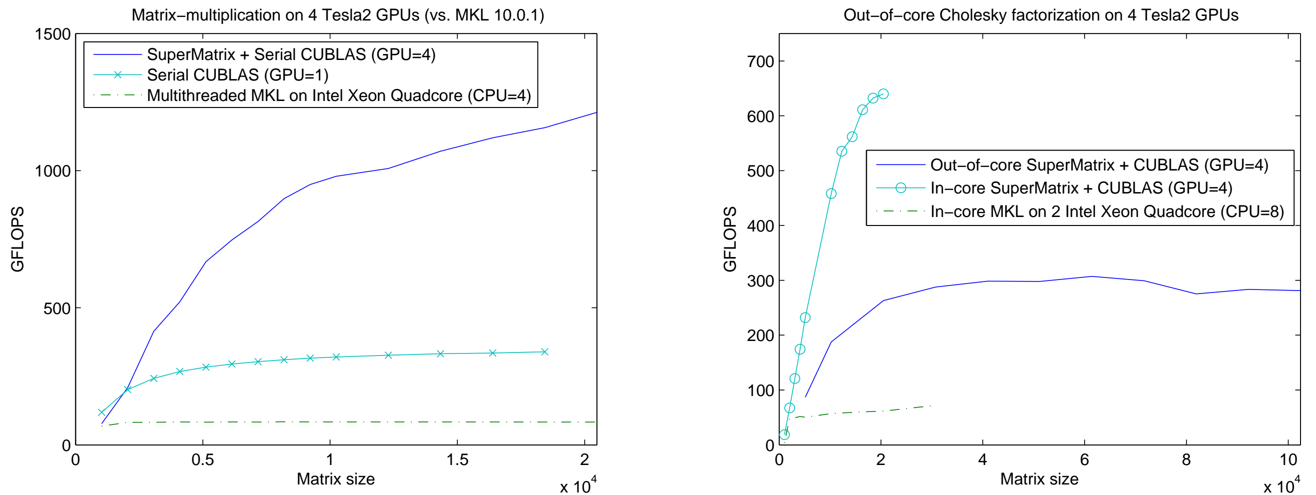

Figure 4: Representative performance on a system with four Tesla2 accelerators. Left: Performance of single precision matrix-multiplication. Right: Performance of in-core and out-of-core, single precision Cholesky factorization.

- In some cases, new algorithms had to be invented to fit the algorithms-by-blocks concept. For example, the LU factorization with partial pivoting and QR factorization via Householder transformations in their traditional incarnations require columns of blocks to collaborate in order to identify a pivot or perform an inner product. The concepts of incremental pivoting [19, 22] and incremental QR factorization [14, 23] overcome this bottleneck.

- When blocks are viewed as units of data and operations with blocks as units of computation, parallelism may be exposed by expressing the algorithm as a directed acyclic graph (DAG) of sub-operations. In sequential mode, `libflame` implementations, coded in the style illustrated in Figure 2 (left), execute sub-operations immediately. However, abstractions within the library allow a nearly identical implementation to instead build a DAG as it executes, deferring computation until the DAG is complete. This provides the opportunity for sub-operations to be dispatched to threads for parallel execution

even when many dependencies exist between subproblems [24]. Thus, `libflame` algorithms need not change as one moves from a sequential to a shared-memory parallel environment.

- A run-time system, SuperMatrix, has been developed to analyze and dynamically schedule the sub-operations captured in algorithms-by-blocks. This system implements in software many of the techniques, such as out-of-order execution, incorporated at the hardware level in superscalar processors.

Together, these abstractions allow for algorithms with many dependencies to be elegantly and efficiently scheduled to exploit thread-level parallelism.

In Figure 3 (right), we illustrate the benefits of this approach. The SuperMatrix curve reports the performance of an algorithm-by-blocks for LU factorization with incremental pivoting that was implemented with the extension to `libflame` described above.

Exploiting hardware accelerators. A recent development in high-performance computing has been the arrival of hardware accelerators like IBM's Cell Broadband Engine and general-purpose graphics processing units (GPUs).

As part of a prototype extension of `libflame`, the same mechanism that was used to target shared-memory parallel architectures allowed stunning performance to be almost effortlessly achieved on systems with multiple hardware accelerators [21]. In Figure 4 (left), we show how four Tesla2 accelerators achieve a rate of more than a trillion single precision floating point operations per second (TFLOPS).

Solving medium-sized problems on a desktop. In 1991, the LINPACK benchmark executed on the fastest machine in the world achieved 13.9 GFLOPS (billions of floating point operations per second) while factoring what was then a large problem: a $25,000 \times 25,000$ matrix that fit in the combined memories of 512 processors. A modern desktop computer can now store such a matrix in its memory and solve it in seconds.

But that is now just a small problem. The same mechanism that allows algorithms-by-blocks to be scheduled to shared-memory parallel architectures can be used to schedule matrix blocks stored on hard drives. In Figure 4 (right), we show how this allows medium sized symmetric positive definite problems of size $100,000 \times 100,000$ to be factored in about 21 minutes via an out-of-core implementation using multiple hardware accelerators.

4 Conclusion

While new computer architectures have brought new challenges, for the domain of dense linear algebra it appears these challenges merely have provided a catalyst for change. Abstraction is no longer a luxury that must be avoided in order to retain high performance. Rather, it is a necessity that *enables* high performance.

Many opportunities and much work remain. We are continuously expanding the functionality of `libflame` and prototyping new ideas. We hope the scientific computing community not only benefits from `libflame` but also contributes feedback so that we may continue to improve the library.

Additional information

For additional information on FLAME visit

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

Acknowledgments

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. The researchers from the Universidad Jaime I were supported by projects CICYT TIN2008-06570-C04-01 and FEDER, and P1B-2007-19, P1B-2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. In addition, Microsoft has provided significant support. We thank NVIDIA for the donation of some of the hardware that was employed in the experimental evaluation.

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, 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.

[3] 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):3:1–3:22, July 2008.

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

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[6] 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 2007.

[7] 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 *PPoPP '08: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–132, Salt Lake City, UT, USA, February 2008.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[10] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[11] FLAME Publications. `http://www.cs.utexas.edu/users/flame/publications/`.

[12] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[13] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 193–210, Ottawa, ON, Canada, October 2000.

[14] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[15] LAPACK – Linear Algebra PACKage. `http://www.netlib.org/lapack/`.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[17] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-04-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[18] Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.

[19] Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2):11:1–11:16, July 2008.

[20] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software*, 29(2):218–243, June 2003.

[21] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 121–130, Raleigh, NC, USA, February 2009.

[22] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design and scheduling of an algorithm-by-blocks for LU factorization on multithreaded architectures. In *MTAAP '08: Proceedings of the 2008 Workshop on Multithreaded Architectures and Applications*, Miami, FL, USA, April 2008.

[23] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert A. van de Geijn, and Field G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *PDP '08: Proceedings of the Sixteenth Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 301–310, Toulouse, France, February 2008.

[24] 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*. 36(3), 2009.

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

[26] Richard M. Veras, Jonathan S. Monette, Enrique S. Quintana-Ort, and Robert A. van de Geijn. Transforming linear algebra libraries: From abstraction to high performance. *ACM Transactions on Mathematical Software*. Submitted.

[27] Field G. Van Zee. `libflame`*: The Complete Reference*. `http://www.lulu.com/content/5915632/`, 2009.