Transforming Linear Algebra Libraries: From Abstraction to Parallelism

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

Jim Nagle
LabVIEW Math and Signal Processing
National Instruments
Austin, TX 78759
jim.nagle@ni.com

***Abstract*—We have built a body of evidence which shows that, given a mathematical specification of a dense linear algebra operation to be implemented, it is possible to mechanically derive families of algorithms and subsequently to mechanically translate these algorithms into high-performing code. In this paper, we add to this evidence by showing that the algorithms can be statically analyzed and translated into directed acyclic graphs (DAGs) of coarse-grained operations that are to be performed. DAGs naturally express parallelism, which we illustrate by representing the DAGs with the G graphical programming language used by LabVIEW. The LabVIEW compiler and runtime execution system then exploit parallelism from the resulting code. Respectable speedup on a sixteen core architecture is reported.**

I. Introduction

The advent of multi-core and many-core architectures has brought the concern that these new architectures have to be programmed. How are we going to evolve our existing code base to these emerging environments? Programmers are considered to be ill-equipped to meet this challenge.

We believe that, for the domain of dense linear algebra, part of the answer is to take the programmer out of the picture. Instead, we are focusing on making the entire process mechanical by starting with a specification of the operation to be implemented and then mapping algorithms to a specific architecture. Over the last decade, we have systematically built a body of work that together provide evidence that this goal is achievable [3], [4], [15], [16], [32].

The current paper brings the following new contributions to the forefront:

- It shows that, from a high-level specification, a directed acyclic graph (DAG) of coarse-grained operations on coarse-grained data can be statically generated.
- It demonstrates that the methodology can target non-traditional languages such as LabVIEW's graphical programming language [20], yet this methodology can also be applied to imperative languages such as C.
- The approach works whether the matrix is originally stored as a "flat" matrix (e.g., in column-major format) or by blocks to improve locality.
- It illustrates how the LabVIEW compiler and runtime execution system exploit parallelism from a DAG.
- It reports speedup when a DAG is executed on a sixteen core architecture.

Together these contributions move us ever closer to making the entire process of programming high-quality linear algebra libraries entirely mechanical for a broad range of target architectures and languages.

The rest of the paper is organized as follows. We build the paper around a motivating example, inversion of a triangular matrix, in Section II. This operation allows us to discuss the essential information necessary to describe the algorithm at a language-independent level of abstraction in Section III. In Section IV we describe the process that analyzes the algorithm and statically generates a DAG. Section V provides performance results, and we discuss related work in Section VI. In Section VII, we give concluding remarks on how our work fits into the bigger picture of generating libraries entirely mechanically and then point out the tantalizing possibility that the methodology might also be able to eliminate the software stack altogether, generating hardware instead.

II. Inversion of a Triangular Matrix

We use inversion of a triangular matrix (TRINV) $R := U^{-1}$ where U is upper triangular as a motivating example in this paper.[1] It is a dense linear algebra operation that is highly representative of the most commonly used level-3 Basic Linear Algebra Subprograms (BLAS) [11] and operations supported by, for example, LAPACK [2] and libFLAME [31].[2]

It is well understood that in order to attain high performance, matrix algorithms of this kind must be cast in terms of blocked computations so that the bulk of the computation resides in matrix-matrix multiplication [14]. In Figure 1, we present a blocked algorithm for computing TRINV using the Formal Linear Algebra Method Environment (FLAME) notation for expressing linear algebra algorithms [16]. The thick and thin partition lines have semantic meaning and capture how algorithms move through the matrices, exposing submatrices on which computation occurs. Here, the algorithm overwrites the upper triangular part of the original matrix A.

In many of the operations, it is implicitly assumed that a matrix is upper triangular and/or only the upper triangular

[1] Similarly, we can compute $R := L^{-1}$ where L is lower triangular.

[2] We have also applied this methodology to the Cholesky factorization.

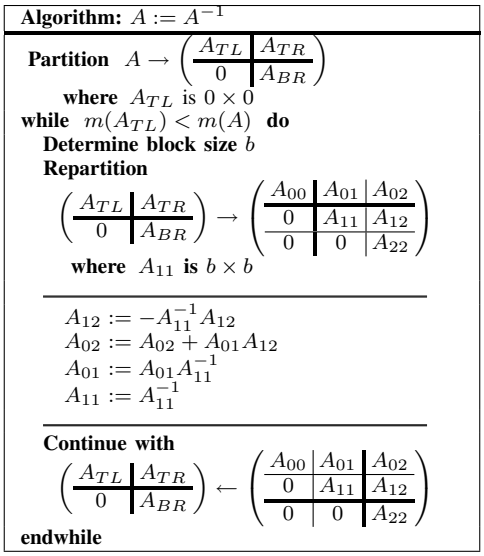

Figure 1. A blocked algorithm for computing the inverse of an upper triangular matrix where $m(B)$ stands for the number of rows of B.

```
1    DO J = 1, N, NB
2      JB = MIN( NB, N-J+1 )
3      CALL DTRSM( 'Left', 'Upper', 'No transpose', 'Non-unit',
4     $                JB, N-J-JB+1, -ONE, A( J, J ), LDA,
5     $                A( J, J+JB ), LDA )
6      CALL DGEMM( 'No transpose', 'No transpose',
7     $                J-1, N-J-JB+1, JB, ONE, A( 1, J ), LDA,
8     $                A( J, J+JB ), LDA, ONE, A( 1, J+JB ), LDA )
9      CALL DTRSM( 'Right', 'Upper', 'No transpose', 'Non-unit',
10    $                J-1, JB, ONE, A( J, J ), LDA,
11    $                A( 1, J ), LDA )
12     CALL DTRTI2( 'Upper', 'Non-unit',
13    $                 JB, A( J, J ), LDA, INFO )
14     ENDDO
```

Figure 2. LAPACK-style implementation of the blocked algorithm in Figure 1.

part of a matrix is updated. For any operation of the form $Y := B^{-1}Y$, it is implicitly assumed that B is upper triangular and that Y is updated by the solution of $BX = Y$, also known as a triangular solve with multiple right-hand sides (TRSM). A similar comment holds for $Y := YB^{-1}$.

Four blocked algorithms exist for computing TRINV, one of which is numerically unstable. We present the third variant since the bulk of its computation lies with general matrix-matrix multiplication (GEMM), $A_{02} := A_{02} + A_{01}A_{12}$. For a more thorough discussion, we refer to [5] where the inversion of a symmetric positive definite matrix (SPD-INV) is used to illustrate that the ideal choice of algorithm for a given operation is greatly affected by the characteristics of the target platform.

Traditional programming languages force users to express an algorithm within the syntax of the language, which often obscures the algorithm altogether. In Figure 2, we show a LAPACK-style implementation of the blocked algorithm of TRINV. It is apparent that nearly all of the semantic information found in the algorithm shown in Figure 1 has been lost in translation to this Fortran implementation.

The key abstraction for expressing linear algebra algorithms as DAGs lies in viewing a submatrix block as the fundamental unit of data and operations on those blocks (tasks) as the fundamental unit of computation. By storing matrices hierarchically [10], [13], [19] with one level of indirection, each submatrix block can be easily demarcated in order to determine data dependencies between each task in the resulting *algorithms-by-blocks*. As such, the nodes of the DAG represent tasks, and the edges represent data dependencies. See [28] for more details on interfacing to hierarchical matrices and formulating algorithms-by-blocks from traditional blocked linear algebra algorithms.

```
1   FLA_Part_2x2( A,     &ATL, &ATR,
2                        &ABL, &ABR,     0, 0, FLA_TL );
3
4   while( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) )
5   {
6      FLA_Repart_2x2_to_3x3(
7         ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
8      /* ************* */   /* ******************** */
9                              &A10, /**/ &A11, &A12,
10        ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
11        1, 1, FLA_BR );
12     /*----------------------------------------------*/
13     FLASH_Trsm( FLA_LEFT, FLA_UPPER_TRIANGULAR,
14                 FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
15                 FLA_MINUS_ONE, A11, A12 );
16     FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
17                 FLA_ONE, A01, A12, FLA_ONE, A02 );
18     FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
19                 FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
20                 FLA_ONE, A11, A01 );
21     FLASH_Trinv( FLA_UPPER_TRIANGULAR,
22                  FLA_NONUNIT_DIAG, A11 );
23     /*----------------------------------------------*/
24     FLA_Cont_with_3x3_to_2x2(
25        &ATL, /**/ &ATR,         A00, A01, /**/ A02,
26                                 A10, A11, /**/ A12,
27      /* ************* */   /* ****************** */
28        &ABL, /**/ &ABR,         A20, A21, /**/ A22,
29        FLA_TL );
30  }
```

Figure 3. FLASH implementation of the corresponding blocked algorithm in Figure 1.

In Figure 3, we present an algorithm-by-blocks for TRINV implemented with the FLASH extension to the FLAME/C API [22]. All the implementation details about the matrix hierarchy are encapsulated in the object-oriented matrix data structure. This API was formulated to closely mimic FLAME notation, which allows for the simple translation from a language-independent representation to FLASH implementation, and back again if desired.

III. Requisite Semantic Information for Static Dependence Analysis

We build upon the work in [32] where the authors developed a source-to-source translator that converts linear algebra algorithms implemented using the FLAME/C API [6] into code with explicit indexing and direct calls to BLAS routines [21]. This effort reduces the overhead associated with dereferencing object-oriented matrix objects where the greatest performance gains lie with level-2 BLAS [12] and unblocked LAPACK routines [2]. This feat was accomplished by translating high-level FLAME/C implementations into descriptions of the linear algebra algorithms with FLAME/XML, a programming language-independent representation using the Extensible Markup

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <Function name="FLA_Trinv" type="blk" variant="3">
3     <Option type="uplo">FLA_UPPER_TRIANGULAR</Option>
4     <Declaration>
5        <Operand type="matrix" direction="TL->BR" inout="both">
6           A
7        </Operand>
8     </Declaration>
9     <Loop>
10       <Guard>A</Guard>
11       <Update>
12          <Statement name="FLA_Trsm">
13             <Option type="side">FLA_LEFT</Option>
14             <Option type="uplo">FLA_UPPER_TRIANGULAR</Option>
15             <Option type="trans">FLA_NO_TRANSPOSE</Option>
16             <Option type="diag">FLA_NONUNIT_DIAG</Option>
17             <Parameter>FLA_MINUS_ONE</Parameter>
18             <Parameter partition="11">A</Parameter>
19             <Parameter partition="12">A</Parameter>
20          </Statement>
21          <Statement name="FLA_Gemm">
22             <Option type="trans">FLA_NO_TRANSPOSE</Option>
23             <Option type="trans">FLA_NO_TRANSPOSE</Option>
24             <Parameter>FLA_ONE</Parameter>
25             <Parameter partition="01">A</Parameter>
26             <Parameter partition="12">A</Parameter>
27             <Parameter>FLA_ONE</Parameter>
28             <Parameter partition="02">A</Parameter>
29          </Statement>
30          <Statement name="FLA_Trsm">
31             <Option type="side">FLA_RIGHT</Option>
32             <Option type="uplo">FLA_UPPER_TRIANGULAR</Option>
33             <Option type="trans">FLA_NO_TRANSPOSE</Option>
34             <Option type="diag">FLA_NONUNIT_DIAG</Option>
35             <Parameter>FLA_ONE</Parameter>
36             <Parameter partition="11">A</Parameter>
37             <Parameter partition="01">A</Parameter>
38          </Statement>
39          <Statement name="FLA_Trinv">
40             <Option type="uplo">FLA_UPPER_TRIANGULAR</Option>
41             <Option type="diag">FLA_NONUNIT_DIAG</Option>
42             <Parameter partition="11">A</Parameter>
43          </Statement>
44       </Update>
45    </Loop>
46 </Function>
```

Figure 4. FLAME/XML representation of the blocked algorithm in Figure 1.

Language (XML). The FLAME/XML description of the algorithm is then translated into code with explicit indexing.

Using XML frees us from burying the algorithm underneath layers of syntactic clutter and allows us to represent the algorithm by storing only language-independent features.[3] FLAME/XML was also designed to closely resemble FLAME notation so as to preserve the natural readability of FLAME algorithms. We present the FLAME/XML representation of the blocked algorithm of TRINV in Figure 4.

In [32], the authors identified five properties common to all typical linear algebra algorithms: (1) the name of the operation; (2) how the algorithm proceeds through the operands; (3) whether it is a blocked or unblocked algorithm; (4) the condition for remaining in the loop (loop-guard); and (5) the updates within the loop body. In Figure 4, each of these five semantic properties can be clearly identified.

A. Additional Semantic Properties

Even though those five semantic properties are sufficient to perform source-to-source translation of FLAME/C implementations, statically generating a DAG requires two additional properties: (1) the problem size; and (2) input and output parameters of each operation.

[3]By representing a wide range of dense linear algebra operations in XML, we can view the collection of algorithms as a repository of knowledge. As a result, we can potentially mine data from this repository to understand and exploit the nature of computation expressed in linear algebra algorithms.

```
1  <?xml version=''1.0'' encoding=''ISO-8859-1''?>
2  <Function name=''FLA_Trsm''>
3     <Declaration>
4        <Operand type=''scalar'' inout=''in''>alpha</Operand>
5        <Operand type=''matrix'' inout=''in''>A</Operand>
6        <Operand type=''matrix'' inout=''both''>B</Operand>
7     </Declaration>
8  </Function>
9  <Function name=''FLA_Gemm''>
10    <Declaration>
11       <Operand type=''scalar'' inout=''in''>alpha</Operand>
12       <Operand type=''matrix'' inout=''in''>A</Operand>
13       <Operand type=''matrix'' inout=''in''>B</Operand>
14       <Operand type=''scalar'' inout=''in''>beta</Operand>
15       <Operand type=''matrix'' inout=''both''>C</Operand>
16    </Declaration>
17 </Function>
```

Figure 5. Input and output parameters for TRSM and GEMM using the FLAME/XML representation.

Even though the problem size is not explicitly expressed within a linear algebra algorithm, it induces the number of iterations executed by each loop.

In order to identify data dependencies, we need the input and output parameters of each operation, which is shown on Line 5 in Figure 4 for TRINV and Figure 5 for TRSM and GEMM. When dealing with algorithms encoded in typical programming languages, this information involving input and output parameters is all but lost. For more details on how to systematically identify flow (read-after-write), anti (write-after-read), and output (write-after-write) data dependencies in linear algebra algorithms, see [9].

IV. Static Generation of a Directed Acyclic Graph

Our methodology consists of two phases: source code translation and DAG generation. We first start from a simplified algorithmic description expressed in FLAME/XML and then translate that representation into an intermediate FLASH implementation. The main difference between this intermediary and the implementation shown in Figure 3 is that the translated source code generates a separate set of source code that builds a directed acyclic graph. This intermediate FLASH implementation steps through execution of the algorithm and detects data dependencies through annotations that specify input and output information for each operation.

The DAG generation phase, which is described further in Figure 6, can be viewed as unrolling loops within a linear algebra algorithm where multiple loops can be nested. For example, in the blocked algorithm for computing TRINV, the operations TRSM and GEMM are called, both of which are computed using similar loop-based algorithms, so unrolling these nested loops involves elements of interprocedural analysis.

Also notice that the blocked algorithm for TRINV also consists of a recursive subproblem. Here, the TRINV subproblem is implemented via an unblocked algorithm. This issue opens the question as to whether we should completely unroll all loops for both blocked and unblocked algorithms, which reflects the need to specify data granularity. Since we

Data: Linear algebra algorithm S, block size b
Result: Directed acyclic graph (V, E)
$V := \emptyset$; $E := \emptyset$;
foreach $A \in \mathbb{R}^{m \times n}$ *accessed by* S **do**
  for i *to* $\frac{m}{b}$ **do**
    for j *to* $\frac{n}{b}$ **do**
      $\hat{A}_{i,j} := \emptyset$;
    end
  end
end
while *Execute* S **do**
  Store Task: $V := V \cup \{t\}$;
  foreach $A_{i,j}$ *accessed by* t **do**
    $\hat{i} := \frac{i}{b}$; $\hat{j} := \frac{j}{b}$;
    if $\hat{A}_{\hat{i},\hat{j}} \neq \emptyset$ **and** $\hat{A}_{\hat{i},\hat{j}} \neq t$ **then**
      Store Dependency: $E := E \cup \{(\hat{A}_{\hat{i},\hat{j}}, t)\}$;
    end
    if $A_{i,j}$ *is overwritten by* t **then**
      $\hat{A}_{\hat{i},\hat{j}} := t$;
    end
  end
end

Figure 6. The process that statically generates a DAG from a linear algebra algorithm and block size.

have formulated these operations as algorithms-by-blocks, we restrict ourselves to only unroll blocked subproblems, which is specified on Line 2 in Figure 4. As a result, each task mainly consists of level-3 BLAS operations.

As stated in Section III, the problem size is required to generate the DAG, but actually another piece of information is assumed: the algorithmic block size. The problem size and block size together determine the number of iterations that are executed and thus the *loop unrolling factor*.

By using hierarchical matrix storage, we have abstracted away the need to specify a block size within the algorithm since it manifests as the size of each contiguously stored submatrix block. When striding over hierarchical matrices, we use a unit block size because the top-level data structure refers to a matrix whose elements are pointers to contiguously stored submatrix blocks, which is highlighted on Line 11 in Figure 3.

A. Determining Block Size and/or Loop Unrolling Factor

Though we cannot statically determine the problem size due to the dynamic allocation of matrices, we can adjust two related variables: block size and loop unrolling factor.

If we keep the block size constant, the loop unrolling factor will grow as a function of the problem size. With more code being unrolled, the instruction footprint increases, and thus we expect performance to suffer from instruction cache misses.[4] On the other hand, BLAS implementations are tuned for specific block sizes which depend on the size of the data cache, so the execution of individual tasks would attain higher performance with a constant block size.

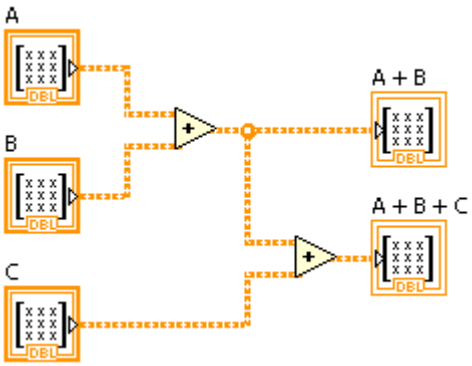

Figure 7. G code for the summation of three matrices.

If we keep the loop unrolling factor constant, fewer instruction cache misses would result, but the block size would have to vary. For large problem sizes, individual blocks would exceed the size of the data cache, resulting in data cache misses.

In order to attain load balance between processors, tasks must roughly have the same weight, which can be defined via computational runtimes. To achieve this goal, a simple solution is to divide the matrices into uniform submatrix blocks. Poor data alignment might render this strategy ineffective despite the gains in load balancing if the problem size and loop unrolling factor are not perfectly divisible by the data alignment length.

Different architectural features, such as data and instruction cache sizes, will affect the strategy for statically generating DAGs. A rudimentary heuristic is to perform DAG generation for a fixed range of loop unrolling factors and vary the block size dynamically to adjust for different problem sizes.

V. Performance

In this section, we apply our methodology to a graphical programming language and show performance improvements from expressing computation as a DAG.

A. LabVIEW

We use LabVIEW [20] and its graphical programming language (G) as the testing environment for our DAG generation methodology. G is data flow language where virtual instruments (VI), which encapsulate functions, are connected via wires to represent the explicit data flow of variables. In Figure 7, we show G code that summates three

[4]Functional and data flow programming languages can store the DAG as code, but imperative languages may require the DAG to be stored separately within internal data structures.

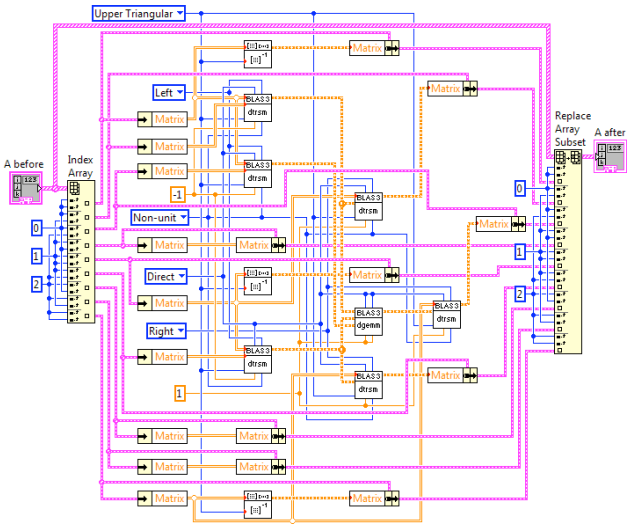

Figure 8. G code for triangular inversion on a 3×3 matrix of blocks.

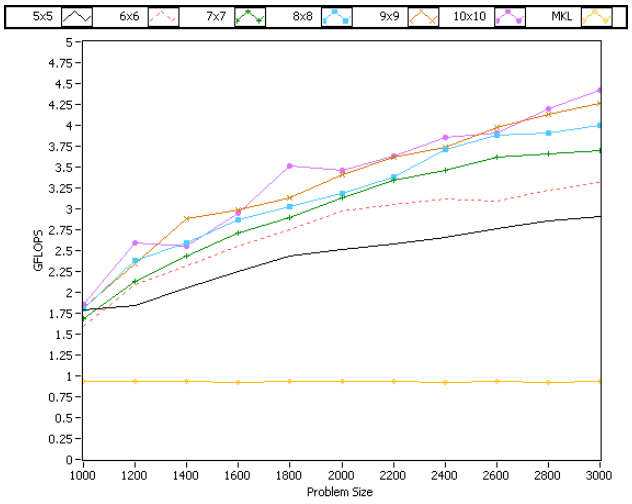

Figure 9. Performance results of single-threaded MKL versus DAG implementations of TRINV.

input matrices. The input and output operands for each VI are easily recognizable in Figure 7 where the summation VI has two matrices as inputs and one resulting output.

G allows us to easily encode a DAG of tasks where each VI represents a task and the wires represent flow dependencies between tasks. Due to the data flow nature of this programming language, anti-dependencies cannot exist because variables are never overwritten. In Figure 8, we show the automatically generated G code for TRINV given a 3×3 matrix of blocks. As you can see, handcoding this diagram for operations with larger problem sizes would be a daunting task.

The runtime execution system of LabVIEW attempts to exploit opportunities for parallelism within G diagrams, but as with imperative programming languages, complex data dependencies, interprocedural analysis, and varying control flow prevent LabVIEW from fully parallelizing many computations.

LabVIEW has an inability to link with multithreaded BLAS libraries because it cannot explicitly control the threads spawned by those external libraries. This problem became the primary motivation to develop this static DAG generation methodology in order to leverage the LabVIEW compiler and runtime execution system to exploit parallelism. As a result, a LabVIEW application that controls a large-scale telescope, which in part computes a Cholesky factorization, can greatly benefit from our approach.

B. Target Architecture

All experiments were performed on a 16-core AMD machine consisting of four 1.9 GHz quad-core Opteron processors, each with 4 GB of general-purpose physical memory. We used LabVIEW 8.6 running on Windows XP, which links to single-threaded Intel MKL 7.2 for BLAS and LAPACK functionality.

C. Implementations

We report the performance in GFLOPS (10^9 floating point operations per second) of two TRINV implementations: DAG representations in G and the single-threaded `dtrtri` routine present in MKL.[5] We performed static loop unrolling for 5×5 through 10×10 matrix of blocks, allowing the block size to vary proportionally as a function of the overall problem size. The DAG results also included the time to copy a matrix from flat row-major storage to a hierarchical matrix and also back to flat storage.

D. Results

Performance results are reported in Figure 9. Several comments are in order:

- Due to LabVIEW's ability to exploit the opportunities for parallelism exposed by a DAG, larger loop unrolling factors provided incremental improvements in performance. Larger loop unrolling factors create DAGs with a larger number of tasks, which leads to more opportunities for parallelism among the tasks. For this range of problem sizes, 1000×1000 through 3000×3000 in increments of 200×200, performance gains reach an upper limit with a 10×10 matrix of blocks.
- LabVIEW is able to provide significant speedup versus the single-threaded MKL implementation. However, the LabVIEW implementations exhibited poor efficiency; despite having access to sixteen cores, the speedup observed is roughly five.

[5]MKL does not provide a multithreaded implementation of `dtrtri` even within MKL 10.2.

The compiler and runtime execution system that exist within LabVIEW are designed as a general purpose G code execution mechanism and therefore must be flexible enough to handle complex control flow constructs. On the other hand, domain-specific runtime systems might only handle computation expressed as a DAG without any control flow variations, so more complex scheduling heuristics can be employed to attain better efficiency on different parallel architectures.

- The use of hierarchical matrix storage allows for better spatial locality of submatrix blocks. LabVIEW forces internal copies when a wire is split, creating an inherent bottleneck in performance. Even though this overhead is amortized across all computation within the DAG, temporal locality is lost through the allocation of new submatrix blocks despite the inherent reuse of data. Additional memory management is required, which also adversely affects performance.

  Anti-dependencies do not exist in G because of the implicit copies. Despite this extra overhead, the lack of these dependencies potentially creates additional opportunities for parallelism that are not present when variables are explicitly read and overwritten.
- On top of the hidden copy of submatrices at forked wires, an explicit memory copy is required to convert flat matrices to hierarchical ones. Storing submatrices contiguously creates overhead up-front, but this storage scheme preserves spatial locality.

  Even though we used TRINV as a motivating example, the DAG generation methodology can be applied to multiple linear algebra operations in sequence. In [9], parallelism is extracted from the inversion of a symmetric positive definite matrix, which may be implemented by computing the Cholesky factorization $A \rightarrow U^T U$, followed by triangular inversion $R := U^{-1}$, and then triangular matrix multiplication by its transpose $A^{-1} := RR^T$. Since each of these constituent operations of SPD-INV are computed via loop-based algorithms, we can apply this methodology to generate a DAG for all three operations together. Since the computational complexity $O(n^3)$ grows faster than the data storage $O(n^2)$, the overhead of copying the matrix to a hierarchical matrix becomes a smaller fraction of the overall amount of computation.

As a result of using our DAG generation methodology, we have attained modest performance gains using the existing tools provided by LabVIEW without incorporating complex scheduling heuristics into the general purpose runtime execution system.

VI. Related Work

Modern superscalar computer architectures have long used out-of-order execution techniques, akin to Tomasulo's algorithm [30], to exploit instruction-level parallelism [18]. Data dependencies exist between the register and memory locations of different scalar instructions. Once all data dependencies are fulfilled for each operand, instructions are dispatched and scheduled out-of-order to execute in parallel on separate functional units.

The data dependencies between scalar instructions also form a DAG where the nodes now represent individual scalar instructions as opposed to coarse-grained tasks. As such, we can view out-of-order superscalar execution as an analog to scheduling DAGs in parallel. Recent research projects, such as PLASMA [7], SMPSs [24], and SuperMatrix [8], [9], [28], have leveraged the idea of DAG scheduling in order to exploit parallelism from matrix computations, such as the Cholesky factorization [1], [17].

With the recent work on scheduling matrix computations in parallel, building a DAG from an algorithm-by-blocks is done sequentially at runtime [7], [24], [28]. This process invokes Amdahl's law where the sequential component of DAG construction limits the potential speedup of the parallelized matrix computation. The focus of this paper is how to statically generate a DAG and thus decouple the parallel scheduling of tasks in a DAG from the static dependence analysis. We can potentially adapt this methodology to interface with domain-specific schedulers such as SuperMatrix in order to eliminate this sequential overhead and provide more dramatic speedups in performance.

The traditional approach for performing static dependence analysis entails trying to recover semantic information from an implementation of an algorithm, usually instantiated in programming languages such as C or Fortran [25]. The difficulty of this approach is that semantic information is obfuscated behind many of the implementation details such as explicit indexing of matrices [23]. Relatively simple information such as the input and output parameters of each operation become almost impossible to recover because of the complexities incurred from pointer aliasing and varying interprocedural control flow.

Earlier work on DAG scheduling of matrix computations [34] also dealt with the static generation of a DAG but also performed the static scheduling of tasks for distributed-memory architectures whereas recent work is geared towards dynamic scheduling for shared-memory architectures [7], [24], [28]. C-based programming language extensions are presented in [24], [34] to construct a DAG whereas we present a methodology for encapsulating linear algebra algorithms in a simple, programming language-independent representation. As a result, we address the issue of progammability.

SPIRAL is a project that automatically generates and optimizes code starting from a mathematical specification of linear transforms [26]. Their problem domain only presents itself with a limited set of tunable parameters, so the generated code produced by SPIRAL is quite efficient. On the other hand, matrix computations provide a vastly larger

search space of such parameters and hence our using a simple heuristic for determining the block size and loop unrolling factor. Thus far, the auto-tuning of BLAS libraries has only dealt with optimizing sequential kernels [29], [33].

VII. Conclusion

The results in this paper, in and by themselves, represent an interesting case study of how "knowledge" stored as information in an XML description can be statically analyzed, yielding a DAG that can then be used to exploit parallelism. It also provides additional evidence that it is possible to change programming from an art that requires expert human understanding into a system that mechanically exploits the knowledge of an expert human in this problem domain.

In previous work, we showed that for a broad class of dense linear algebra operations it is possible to take a description of an operation and systematically transform this description into a family of algorithms that computes the operation [4]. The process often yields new algorithms even when no algorithms were previously known for the given operation [27]. Next, we showed that this process could be made mechanical [3]. The output of this process can be a high-level description and knowledge about those algorithms (e.g., a cost function or numerical properties).

What we have shown in [32] and this paper is that this knowledge can, for example, be represented with XML. From this intermediate representation, a number of different implementations can be obtained via relatively simple rewrite rules. The analysis that yields a DAG discussed in this paper is just one of several possibilities. For example, we can transform algorithms into our FLAME/C API for sequential routines or generate code at a level similar to that used by traditional libraries like LAPACK, as described in [32] which also has a more thorough discussion about the different outputs we have explored for that system.

We finish with a final possible output, which we find particularly interesting. It should be possible to map algorithms directly to hardware. After all, the G code in Figure 8 resembles a circuit. In other words, we may be able to mechanically generate special-purpose hardware for commonly used linear algebra operations, circumventing the software stack entirely.

Additional information

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

Acknowledgment

This research was sponsored by National Instruments Corporation and 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).

References

[1] Ramesh C. Agarwal and Fred G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *SC '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 225–233, Reno, NV, USA, November 1989.

[2] 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.

[3] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, The University of Texas at Austin, 2006.

[4] 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.

[5] 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.

[6] 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.

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, January 2009.

[8] 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.

[9] 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.

[10] Timothy Scott Collins. *Efficient Matrix Computations through Hierarchical Type Specifications*. PhD thesis, The University of Texas at Austin, 1996.

[11] 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.

[12] 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.

[13] 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.

[14] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[15] John A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, The University of Texas at Austin, 2001.

[16] 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.

[17] Fred G. Gustavson, Lars Karlsson, and Bo Kagstrom. Three algorithms for Cholesky factorization on distributed memory using packed storage. *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 550–559, Springer Berlin / Heidelberg, September 2007.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2003.

[19] José Ramón Herrero. *A Framework for Efficient Execution of Matrix Computations*. PhD thesis, Polytechnic University of Catalonia, Spain, 2006.

[20] LabVIEW. `http://www.ni.com/labview/`.

[21] 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.

[22] 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.

[23] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Effectiveness of data dependence analysis. *International Journal of Parallel Programming*, 23(1):63–81, February 1995.

[24] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster '08: Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, Tsukuba, Japan, September 2008.

[25] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1121–1132, November 1996.

[26] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, Special Issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, February 2005.

[27] 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.

[28] 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):14:1–14:26, July 2009.

[29] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *POHLL '08: Proceedings of the 2008 Workshop on Performance Optimization for High-Level Languages and Libraries*, pages 1–8, Miami, FL, USA, April 2008.

[30] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[31] Field G. Van Zee, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Introducing: The `libflame` library for dense matrix computations. *IEEE Computing in Science & Engineering*, 11(6):56–62, November 2009.

[32] Richard M. Veras, Jonathan S. Monette, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. FLAMES2S: From abstraction to high performance. FLAME Working Note #35 TR-08-49, The University of Texas at Austin, Department of Computer Sciences, December 2008.

[33] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27, San Jose, CA, USA, November 1998.

[34] Tao Yang. *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, Rutgers, The State University of New Jersey, 1993.