Mechanizing the Expert Dense Linear Algebra Developer

FLAME Working Note #58

Bryan Marker [*]
Andy Terrel [†]
Jack Poulson [‡]
Don Batory [*]
Robert van de Geijn [*‡]

April 25, 2011

Abstract

Sustained high performance on the fastest computers in the world has traditionally been accomplished by experts who carefully hand-code key routines. This quickly becomes unmanageable for large bodies of software and/or as architectures change enough that entire libraries need to be rewritten. We believe the problem is that a typical library for scientific computing exists as a set of routines that are manually instantiated in a specific programming language. We advocate developing libraries of the future as algorithms and fundamental kernels that encode computation and data movement, together with expert knowledge. From this a tool will synthesize efficient, platform-specific implementations by composing, transforming, and optimizing library algorithms, accomplishing automatically what today's experts do manually. This paper illustrates how this can be achieved for the domain of dense linear algebra and gives preliminary results for the automatic customization and optimization of dense linear algebra algorithms targeting large distributed memory cluster architectures.

1 Introduction

Parallelizing and optimizing *dense linear algebra (DLA)* algorithms for distributed memory machines has historically been done by domain experts who are very familiar with both linear algebra and the oddities of a target class of machines. When a DLA expert has no experience with a new architecture and wants to implement an algorithm, (s)he must live with an existing library, learn a lot about that architecture, or find an experienced developer. This is inefficient and, as we argue, unnecessary because the work of an expert can be very mechanical and systematic, and therefore automated.

We apply *Model Driven Engineering (MDE)*, which fosters the codification of fundamental algorithms and domain-specific expertise, to this domain. MDE enables us to automate the activities of experts: selecting algorithms, composing algorithms, and applying optimizations to achieve customized and high-performance implementations in code. In this paper, *we show how expert-tuned, high-performance code for a Cholesky factorization for distributed memory architectures can be automatically produced by a tool.* Furthermore, we show how layering code in a way that is amenable to mechanical transformations not only makes a library more maintainable but also writable by other software. Since Cholesky factorization is a prototypical example of a broad class of dense linear algebra operations (e.g., the commonly used matrix operations in the BLAS [10, 11, 17] and operations supported by libraries like LAPACK [2] and `libflame` [29]) and

[*]Department of Computer Science, The University of Texas at Austin,Austin, TX, 78712
[†]Texas Advanced Computing Center, The University of Texas at Austin, Austin, TX, 78712
[‡]Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX, 78712

```
Algorithm: $A := \textsc{Chol\_blk}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Determine block size $b$
  Repartition
    $\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
    where $A_{11}$ is $b \times b$

  Variant 1
    $A_{10} := A_{10}\mathrm{tril}(A_{00})^{-T}$
    $A_{11} := A_{11} - \mathrm{tril}(A_{10}A_{10}^T)$
    $A_{11} := \mathrm{chol}(A_{11})$
  Variant 2
    $A_{11} := A_{11} - \mathrm{tril}(A_{10}A_{10}^T)$
    $A_{11} := \mathrm{chol}(A_{11})$
    $A_{21} := A_{21} - A_{20}A_{10}^T$
    $A_{21} := A_{21}\,\mathrm{tril}(A_{11})^{-T}$
  Variant 3
    $A_{11} := \mathrm{chol}(A_{11})$
    $A_{21} := A_{21}\,\mathrm{tril}(A_{11})^{-T}$
    $A_{22} := A_{22} - \mathrm{tril}(A_{21}A_{21}^T)$

  Continue with
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

Figure 1: Blocked algorithms for computing the Cholesky factorization. $m(B)$ stands for the number of rows of B and $\mathrm{tril}(B)$ indicates the lower triangular part of B. The '$\star$' symbol denotes entries that are not referenced.

communication on a distributed memory architecture is an example of data movement between memory layers, we believe our approach can be extended to target other algorithms and other architectures (such as multi-core processors, GPGPUs, and many-core processors) to create the DLA libraries of the future. Since DLA library development has often introduced new software engineering techniques to the broader scientific computing community [2, 8, 9, 25, 28, 30], our approach may influence the broader scientific computing programmer community.

We expect the insights in this paper to have a profound impact on our FLAME project [30]. This project encompasses a formalism for deriving DLA algorithms, notation for expressing these as algorithms, and APIs for implementation in code. Two library instantiations exist: the `libflame` library that targets sequential, multicore, and (multi-)GPU architectures, and Elemental, which targets distributed memory architectures. The proposed approach would allow us to instead support a single encoding of algorithms and knowledge, with libraries like `libflame` and Elemental being the products (outputs) of applying our methodology. Because of the breadth of potential impact, we expect this paper to be the first of many to explore this area research.

2 What an Expert Does

In this section, we discuss what steps an expert follows in order to produce *by hand* a highly-optimized, parallel implementation of a dense matrix operation. We use Cholesky factorization, an operation that is simple yet prototypical of this class of operations, targeting a cluster architecture as a vehicle to illustrate expert activities.

```
template<typename T>
void
elemental::lapack::internal::CholLVar3( DistMatrix<T,MC,MR>& A )
{
    const Grid& g = A.Grid();
    // Matrix views
    DistMatrix<T,MC,MR> ATL(g), ATR(g),  A00(g), A01(g), A02(g),
                        ABL(g), ABR(g),  A10(g), A11(g), A12(g),
                                         A20(g), A21(g), A22(g);

    PartitionDownDiagonal( A, ATL, ATR,
                              ABL, ABR, 0 );
    while( ABR.Height() > 0 )
    {
        RepartitionDownDiagonal( ATL, /**/ ATR,  A00, /**/ A01, A02,
                                /*************/ /******************/
                                      /**/       A10, /**/ A11, A12,
                                 ABL, /**/ ABR,  A20, /**/ A21, A22 );
        //--------------------------------------------------//
        Chol( Lower, A11 );
        Trsm( Right, Lower, ConjugateTranspose, NonUnit, (T)1, A11, A21 );
        TriangularRankK( Lower, ConjugateTranspose, (T)-1, A21, A21, (T)1, A22 );
        //--------------------------------------------------//
        SlidePartitionDownDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
                                         /**/       A10, A11, /**/ A12,
                                   /*************/ /******************/
                                    ABL, /**/ ABR,  A20, A21, /**/ A22 );
    }
}
```

Figure 2: Parallel code directly derived from sequential algorithm by hiding all parallelism in the calls to `Chol`, `Trsm`, and `TriangularRankK`.

2.1 From specification to algorithm

Over the last decade, the FLAME project has developed a repeatable process by which loop-based families of algorithms for dense matrix operations can be systematically derived [16]. FLAME uses formal derivation and yields a number of algorithms for each operation so that the best algorithm for a given situation can be chosen. Many papers have been written on this subject; interested readers should consult [5] for details related to Cholesky factorization and other operations. In Figure 1 we show the three known blocked algorithmic variants that result. Blocked algorithms cast most computation in terms of matrix-matrix operations (level-3 BLAS [10]), which can attain high performance on cache-based architectures. Unblocked algorithms can be obtained by setting the block size $b = 1$. This expert task of deriving algorithms has been mechanized [4].

Henceforth, we use Variant 3 as our running example.

2.2 From algorithm to sequential code

The FLAME project has produced a library, `libflame` [29], with functionality comparable to that of the widely-used LAPACK library [1]. The algorithms encoded in FLAME were systematically derived and then represented in code using an API, FLAME/C [6], that allows the code to closely resemble the algorithm of Figure 1. This code is similar to that in Figure 2, which shows a distributed-memory parallel code for Variant 3 in the style of the Elemental library [20]. Note that the code appears to be just a sequential implementation because, for now, all parallelism is hidden in the individual operations. Producing sequential code from FLAME-specifications is now largely mechanized [4].

```
Chol( Lower, A11 );




Trsm( Right, Lower, ConjugateTranspose, NonUnit,
      (T)1, A11, A21 );





TriangularRankK( Lower, ConjugateTranspose,
                 (T)-1, A21, A21, (T)1, A22 );
```

(a) Original code.

```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
A21 = A21_VC_Star;

A21_MC_Star = A21;
A21_MR_Star = A21;
blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

(b) Inline routines.

```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
\\ A21 = A21_VC_Star;
A21_MC_Star = A21_VC_Star;
A21 = A21_MC_Star;

\\ A21_MC__Star = A21;
A21_VC_Star = A21;
A21_MC_Star = A21_VC_Star;

\\ A21_MC__Star = A21;
A21_VC_Star = A21;
A21_MR_Star = A21_VC_Star;

blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

(c) Inline communication.

```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;

blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );

A21_MC_Star = A21_VC_Star;
A21 = A21_MC_Star;








A21_MR_Star = A21_VC_Star;

blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

(d) Remove redundant communication.

Figure 3: Sequence of optimizations of the loop-body in Figure 2.

2.3 Elemental

In order to understand the hidden parallelism in Figure 2, we must explain a bit about how Elemental works [20]. Elemental is a new dense linear algebra library for distributed-memory architectures that uses a 2-dimensional, cyclic distribution of data with blocksize of 1 over a 2-dimensional grid of processors. Specifically, it views the p processes as an $r \times c = p$ grid, and the data is stored, by default, in a distribution that cyclically wraps the rows and columns of the matrix around the process grid. As a result, element (i, j) of a matrix is stored on process $(i\%r, j\%c)$.

Besides this 2-dimensional distribution, Elemental supports other data distributions and ways to switch between them. This allows a programmer to parallelize an algorithm and its sub-operations in many ways. Elemental is implemented in C++ and matrices are stored in classes that know about distributions. Switching between distributions in the code is accomplished by overloading the '=' operator in the matrix classes, meaning that the ‘=’ operator hides specifics about the communication required to switch between distributions. Behind ‘=’ is code to re-format the data into buffers and call MPI collective communication routines for all combinations of distributions.

We use Elemental because it provides a domain-specific language within C++ in which we can start with a sequential algorithm and apply expertise about distributed-memory systems to parallelize and optimize an implementation. Two key insights an expert uses that must be codified are the management of redistributions (an operation that represents pure overhead due to the communication involved) and the parallelization of sub-operations. In effect an expert trades more communication (which increases overhead) for more parallelism (which allows useful computation to complete sooner). We explain these considerations below using the Cholesky example, but we remind readers they are representative of a large class of operations in this domain.

Elemental's code lends itself well to identifying and codifying those optimizations because all common operations are abstracted and layered to be modular. The Elemental library uses these common operations across codes. For example there are a finite number of data redistribution functions that are used repeatedly, hidden behind the ‘=’ operator. That code includes the MPI layer, data storage information, etc. The local (sequential BLAS and LAPACK-like) functions are called using the familiar APIs and are wrapped to work with Elemental's matrix class. Elemental's distributed BLAS and LAPACK functionality is built on top of these layers. On top of that layer is Elemental's solver functionality. Lastly, user applications are built on top of the Elemental library. *All of this layering and modularity makes mechanizing expert selections of algorithms and optimizations easier because the inherent structure of the domain is exposed.* Because of the modularity, call-site specific implementations of fucntionality are kept at a minimum. Instead of repeatedly including implementations of functionality directly in Elemental code, the modular functions are called. This results in common patterns of function calls, so the optimizations that need to be applied to these patterns are also common across codes. An expert would know this set of optimizations and apply them repeatedly across codes for different applications.

While we believe this paper is written in a way that someone who does not know the details of the Elemental API and its distributions can at least understand what the system described in Section 3 tries to automate, further details are given in [20].

2.4 How an expert optimizes for distributed-memory architectures

With this basic understanding of Elemental, we now give a high-level explanation of how an expert takes a sequential algorithm and optimizes it for a distributed-memory architecture. Doing so motivates the codification of domain expertise. Consider the sequential, Variant 3, lower-triangular Cholesky algorithm of Figure 2. It can achieve very good performance on sequential machines, but it is only implicitly parallel if routines `Chol`, `Trsm`, and `TriangularRankK` are parallelized, which is the case in the Elemental code of Figure 2.

To optimize this code, an expert starts with the loop body, which we show again in Figure 3(a), and inlines the implementations of its three operations to yield Figure 3(b):

- A_{11} is distributed among the processes and
  `Chol( Lower, A11 )`
  represents a small part of the total computation. Thus, a convenient way to perform this operation is to bring all data to all processes, and to then perform the operation redundantly.
  `A11_Star_Star = A11;`
  performs the allgather that duplicates data to all processes.
  `LocalChol(...)`
  then locally performs the factorization on each process, and
  `A11 = A11_Star_Star;`
  locally places the updated values back in `A11` (requiring no communication).

- Next, consider the update $A_{21} := A_{21}\text{tril}(A_{11})^{-T}$. If one partitions A_{21} into rows,

$$A_{21} = \begin{pmatrix} a_{21,0}^T \\ a_{21,1}^T \\ \vdots \end{pmatrix},$$

  and redistributes A_{21} so that rows are assigned to processes in a cyclic order, then the processes can can perform

$$\begin{pmatrix} a_{21,k}^T \\ a_{21,k+p}^T \\ \vdots \end{pmatrix} := \begin{pmatrix} a_{21,k}^T \\ a_{21,k+p}^T \\ \vdots \end{pmatrix} \text{tril}(A_{11}^{-T})$$

  locally in parallel *if* A_{11} is also duplicated on all nodes. The assignment that redistributes A_{21} is
```
    A21_VC_Star = A21;.
```
  The assignment
```
    A11_Star_Star = A11;
```
  duplicates, again, A_{11}.

  The local computation is performed by
```
    LocalTrsm(...);
```
  and the data is placed back in `A21` by
```
    A21 = A21_VC_Star;.
```

- Similarly, the call to
```
    TriangularRankK(...)
```
  is parallelized by redistributions of data
```
    A21_MC_Star = A21;
    A21_MR_Star = A21;
```
  followed by a local computation.

Details of what the distributions are and how exactly they are accomplished are not crucial to our discussion [20]. The resultant code provides a hint as to why optimizations are needed: clearly the statements
```
    A11 = A11_Star_Star;
    A11_Star_Star = A11;
```
can be replaced by the more efficient
```
    A11 = A11_Star_Star;
```
which eliminates unnecessary communication.

An Elemental expert knows that redistributions like
```
    A21_MC_Star = A21;
```
themselves can be composed from two or more redistributions via intermediate distributions. One choice of possible substitutions that use intermediate distributions are exposed in Figure 3(c). In most instances, this simply inlines intermediate distributions that were previously hidden. One instance is the replacement of
```
    A21 = A21_VC_Star;
```
by
```
    A21_MC_Star = A21 = A21_VC_Star;
    A21 = A21_MC_Star;
```
which an expert knows is inefficient as data is distributed from one distribution to another and back to the original distribution instead of redistributing only as necessary. However, the astute reader may notice redundant redistributions which, when removed, yield the code in Figure 3(d). For example the redundant line of
```
    A21_MC_Star = A21_VC_Star;
```
is removed because it is an unnecessary communication.

2.5 Summary

To optimize, an expert in parallelizing dense matrix operations performs (consciously or subconsciously) the previously described steps. Machine-specific details influence how updates in the loop body are parallelized and which operations are expensive and can/need to be optimized. We show in the next section how to mechanize these steps by codifying knowledge about distributed-memory computing and related optimizations using MDE. The keys are to (1) continue layering algorithms, and (2) explicitly codify implementation knowledge about algorithms, layers, communication, and target architectures – details that were inlined in this section.

3 Toward a Mechanical Expert

The previous section showed, step-by-step, the process a domain expert goes through to parallelize and optimize a sequential algorithm. The process is not only systematic but also applies to a broad class of operations in the domain of dense matrix computations. In this section, we discuss how the process can be mechanized.

3.1 The vision

The classic (and arguably greatest to date) example of automated software development is *relational query optimization (RQO)* [24, 27]. A *query evaluation program (QEP)* is represented by a relational algebra expression. A query optimizer rewrites this expression, using relational algebra identities, to an equivalent expression (program) that has better performance. The optimized expression is then translated to code, thereby synthesizing an efficient QEP implementation. The keys to RQO are (a) representing the design of QEPs as relational algebra expressions and (b) optimizing these expressions to produce efficient programs.

We follow the same paradigm but in an MDE setting. The starting point for our optimization is the loop-body in Figure 3(a). We map operations (e.g. `Chol`, `TriangularRankK`, `Trsm`, `=`) to implementing algorithms; the pairing of operations with their algorithms form the algebraic identities (a.k.a. *transformations* or *refinements*) of the DLA domain. Algorithms can reference lower-level operations, which have their own implementing algorithms, and this recurses. *Optimizations* are identities of the form $exp_1 = exp_2$, which allows us to replace one expression (DLA subprogram) exp_1 with another, often more efficient, expression (DLA subprogram) exp_2. By exploring the space of equivalent expressions for a given DLA application and selecting the expression with the best performance characteristics, an efficient implementation is synthesized. Source is produced by translating the optimized expression to code. We illustrate these ideas shortly.

Given a portfolio of basic, local (sequential) operations and redistribution primitives, cost functions for each primitive, and a target sequence of DLA operation (e.g. as given in Figure 3(a)), a mechanical system employs transformations that an expert would apply by hand. Doing so produces all implementations that have merit (meaning they are best by some measure for some subset of operands) and a mechanism by which to choose from these implementations (e.g. cost functions for the implementations).

3.2 Model-driven engineering

MDE advocates the creation of domain-specific models of software and the application of transformations to map models to code artifacts [19]. Models are initially built out of domain-specific *operations*, which represent the desired application functionality. An operation in the loop body of Cholesky factorization is defined by the requirements on the input data (i.e. a single input that is a symmetric, positive-definite matrix) and the output (i.e. the result is the Cholesky factor of the input). All operations have explicit preconditions and postconditions.

There are no implementation details associated with operations, only precondition and postcondition specifications. Implementation details are chosen when an operation is replaced with one of its implementing algorithms, called a *refinement*. MDE allows potentially many refinements of each operation to be defined. Each variation can depend on, for example, different architecture-specific details, parallelization, or numerical stability characteristics. Refinements can be models themselves (i.e. inter-connected collections of operations), which allows models to be layered just as software is layered in libraries.

Refinements preserve operation (or abstraction) boundaries. But refinements are not enough to guarantee efficient implementations. There must also be optimizing transformations, which break operation (abstraction) boundaries and replace inefficient compositions of algorithms or operations with more efficient compositions.

It is the combination of refinement and optimizations that enable efficient applications can be synthesized.

3.3 Applying MDE to dense linear algebra

We use MDE to refine and optimize the operations in the loop body of Cholesky factorization. The loop body in Figure 1 can be modeled with MDE using operations defined for each update function. Refinements of those operations could be calls to a sequential function or complex code to parallelize the operations for SMP or distributed-memory architectures. Here we focus on the latter.[1]

We start by representing each operation in Figure 3(a) as an operation in a model. An expert would replace each of those operations with one of its refinements as in Figure 3(b). The refinement chosen for the `Chol` operation communicates the data and then redundantly calls the sequential function on all processes. Alternatively, another refinement might communicate the data to one process and perform the factorization on it and then distribute the result to all other processes. An expert would explore such options or would instinctively choose one out of experience. Our mechanical system must similarly explore the search space.

By refining the operations of Figure 3(a), the top layer of code is flattened to expose redistribution in Figure 3(b). These redistributions are another layer of operations that can be refined in various ways. By refining some of them as in Figure 3(c), we can break through this layer to expose inefficient redistributions that can be removed to create the optimized model in Figure 3(d). This illustrates how MDE provides a way to model algorithms as connected functionalities and to systematically choose implementation details for that functionality.

3.4 A prototype system

Our MDE models are data-flow graphs. For the same reason that compilers generate data-flow graphs from textual code, we use them to explicitly capture the data dependencies in the sequence of operations that make up the loop body. By only using refinements that preserve data dependencies in the graph, we can maintain the correctness of the algorithm at each step of refinement so that the generated code is as correct as the input sequential algorithm. The same holds for optimizing transformations. (This is called *correct-by-construction* [3, 14, 26, 18].) Proofs of correctness for each refinement and optimization are outside the scope of this paper. Proofs could be constructed, or refinements and optimizations could be validated through extensive testing, which is common for software like this.

To optimize from Figure 3(c) to Figure 3(d), an expert looks for inefficient patterns such as redundant communication and replaces that code with better code. Thought of a different way, an expert replaces the refinement/implementation details of some abstraction/functionality with a different refinement that is better-performing. This can be accomplished in our system with graph transformations that replace sub-graphs that implement some functionality with different sub-graphs that implement the same functionality. Thus, our optimizations are similar to compiler optimizations on data-flow graphs. Inefficient sub-graphs are replaced with better sub-graphs.

The full process is similar to what an expert does in code or in an algorithm, inlining function calls with implementation code or replacing sequences of operations with better alternatives. Viewing the algorithm as a data-flow graph, though, we aim to encode the expert's knowledge more easily, respect data dependencies, and mechanize the process. To optimize for a new target architecture, new refinements would be added. The upside is that by using a well-defined collection of operations and refinements through proper layering, the number of optimizations that need to be encoded is generally small because graph patterns are repeated across algorithms. Evidence for this is the fact that hand optimizing the implementation of many algorithms usually requires the same re-writes to be performed repeatedly.

[1] MDE is truly a visual medium of design, where models and metamodels are graphically created as UML diagrams. We do indeed define our models and metamodels in a traditional MDE manner, but these details are not the thrust of this paper (e.g. see [23]).

Operation	Cost
`LocalChol` ($n \times n$)	$\gamma n^3/3$
`LocalTrsm` (Right, Lower, $n \times n$, $m \times n$)	γmnn
`A11_Star_Star = A11` ($m \times n$)	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} mn$
`A21_MC_Star = A21_VC_Star` ($m \times n$)	$\alpha \lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$
`A21_MR_Star = A21_VC_Star` ($m \times n$)	$\alpha (1 + \lceil \log_2 r \rceil) + \beta (\frac{m}{p} n + \frac{r-1}{r} \frac{m}{c} n)$

Table 1: Representative first-order approximations for the cost of operations found in the code of Figure 3(c). Here, m and n are the row and column sizes of the matrices that are being distributed, the p processes are configured as an $r \times c$ mesh, α is the communication latency, β the cost of communicating a floating point number, and γ the cost of a floating point computation.

3.5 Searching the space of implementations

An expert implementing an algorithm is guided by an understanding of the cost of operations to select refinements and optimizations to apply. To an observer, (s)he would appear to follow instincts. In fact, though, (s)he explores possibilities and assesses (implicitly or explicitly) costs.

How do we enable a mechanical system to choose the best implementation using "instincts"? We do not (*yet*). By iteratively applying all possible transformations to an input algorithm's graph, our method generates a search space of all implementations, both good and bad. By associating a cost with every operation, the best in the search space can, in principle, be picked out analytically. Thus, the prototype system employs run-time cost estimates for redistribution and computation operations in Elemental in an effort to find the best performing codes. We want the system to see that the code of Figure 3(d) is better than the code of Figure 3(a) by summing operation costs and determining which takes less time to execute. The system should then choose Figure 3(d) out of all implementations generated in the search space, just as an expert would.

Finding the optimal implementations by cost estimates requires information about the machine such as communication costs, computation speed, and the number of processors. Further, as those familiar with manual optimization are aware, the input problem size affects which algorithm is optimal; different parallelization schemes yield varying performance based on the matrix size. We consider a range of problem sizes, find implementations that are optimal for some subset of that range, and use cost functions to then choose which implementation to employ when at run time the problem size is known, based on cross-over points for the cost functions. An expert rarely achieves this level of optimization since it requires careful analysis that is too error-prone and time consuming to perform by hand. Automation overcomes this hurdle.[2]

For the domain of DLA, we are able to generate reasonable cost estimates for the usual computations. First order approximations for sequential operations can be given in terms of the number of floating point operations that are performed as a function of the size of operands. For example the matrix multiplication $C = AB$ where C, A, and B are $m \times n$, $m \times k$ and $k \times n$, respectively, takes time (costs) $2mkn\gamma$ where γ is the time for a floating point operation. Similarly, Cholesky factorization of an $n \times n$ matrix costs $n^3/3\gamma$. The cost of every computational kernel can be approximated by the operation count multiplied by the time for performing a floating point operation. Since it is well known that not all algorithms perform at the same speed, a second-order approximation would take such variation into account. But for now, we stick to first-order approximations.

The data redistributions found in Elemental are implemented using MPI collective communication routines. Lower-bound costs of the common algorithms under idealized models of communication are known [7] in terms of coefficients α and β, which capture the latency and cost per item transfered, respectively. For example redistributing an $n \times n$ block of A_{11} as in line `A11_Star_Star = A11` on p processes requires an allgather operation, which has a lower bound cost of approximately $\alpha \log_2(p) + \beta \frac{p-1}{p} n^2$.

Sample cost functions from our Cholesky example are in Table 1. They are a subset of those necessary

[2] Readers may note that this is exactly the RQO paradigm (described in Section 3.1) applied to DLA implementations.

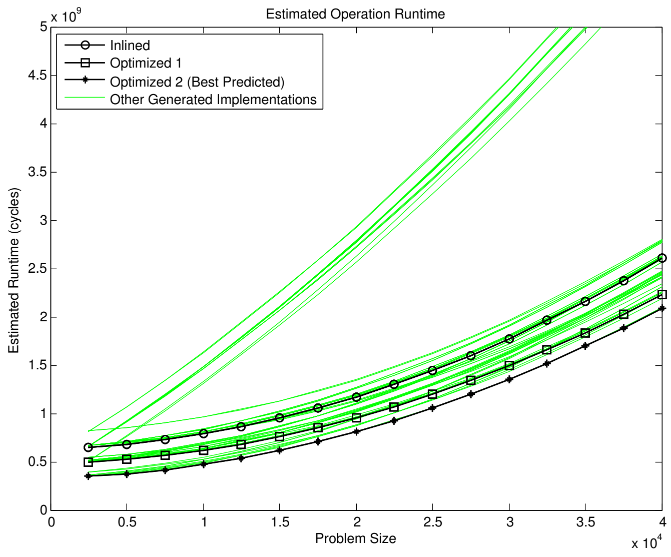

Figure 4: Cholesky Variant 3 estimated runtime in processor cycles on 240 cores.

to enable the prototype we describe in the next section. They only include higher-order terms and are first-order approximations meant to distinguish good (lower-cost) implementations of an algorithm from others that the system generates. It turns out these estimates are good enough for the examples we have studied so far. We expect to improve these estimates to find the best variants for more complicated algorithms, though. For example we have encountered situations where collective communications are suboptimally implemented on a specific architecture while some other architectures provide hardware support for such redistributions.

4 Experimental Results

We developed a prototype system to test the power of the approach and cost functions described in the previous section. We now describe our initial findings.

4.1 Platform

The results shown in this section were taken from the Lonestar cluster at the Texas Advanced Computing Center. We used 20 nodes, each with 2 Intel Xeon hexa-core processors running at 3.33 GHz. The combined theoretical peak performance of all 240 cores is 3200 GFLOPS. For each problem size, we tested a range of algorithmic block sizes and show the best results. We used versions 11.1 of the Intel compiler, 1.6 of MVAPICH2, 1.8.0 of ScaLAPACK, and 1.30 of the GotoBLAS.

4.2 Cholesky Variant 3

We encoded knowledge of a handful of common computation routines (e.g. BLAS functions) as well as Elemental redistributions to enable our prototype to implement the Cholesky example. As input to the system, we encoded the algorithm in data-flow format. The system is able to mechanically produce, without intervention by a human, hundreds of loop body implementations including all versions in Figure 3. Each of these loop body implementations is Elemental code for Cholesky Variant 3. This allows the system to explore the space of Elemental designs of the algorithm. In previous sections, we only described some of the transformations an expert performed because the remaining optimizations for Cholesky are subtle and the reasons for them are out of the scope of this paper. We were able to encode them into our system, and it generates an even better implementation of Cholesky than that of Figure 3(d). This better implementation is identical to that coded by the expert developer of Elemental.

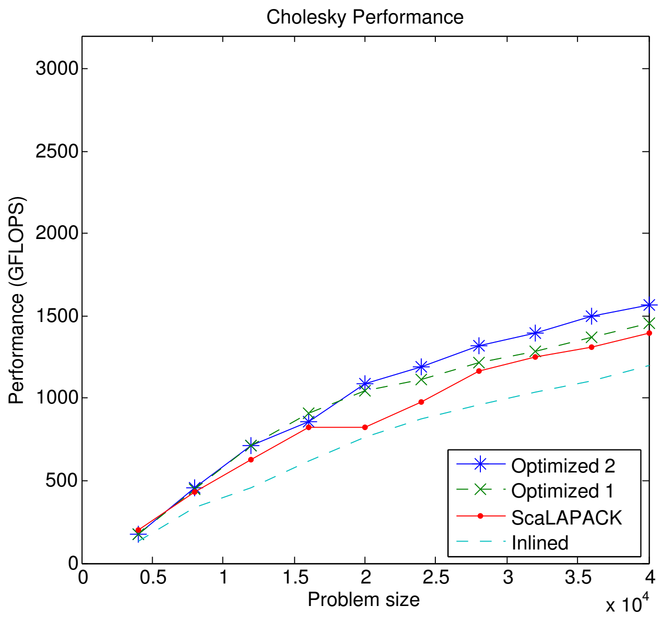

Figure 5: Cholesky Variant 3 implementation performance on 240 cores. Peak performance is at the top of the graph.

In its current incarnation, the system applies all possible graph re-writes to enumerate the entire search space of implementations. We then use symbolic cost functions like those those described in Section 3 to choose the best of all the mechanically generated implementations. Figure 4 shows the cost estimates for the most interesting generated implementations across a range of problem sizes (we omit clearly sub-optimal choices). To make the choice of which is best, we fixed the machine-specific parameters that appear in the cost functions. We take the process grid to be 16×15. γ, a measure of machine speed, is set to be 1 and the other machine parameters are set as reasonable multiples of γ. We then determined the Cholesky implementation in Elemental has a lower cost, i.e. run-time, than any of the hundreds of automatically generated implementations. In Figure 4, this implementation's cost estimate is at the bottom of the graph, labeled "Optimized 2."

In Figure 5 we show the performance results of the code of Figure 3(c) (labeled "Inlined"), the code of Figure 3(d) (labeled "Optimized 1"), and the further optimized code (labeled "Optimized 2"). Our system automatically generated all of these implementations. We leave out the performance of the original code because it is similar to that of the inlined code. Notice that if a domain expert only implemented the algorithm directly and did not optimize considering the machine, the inlined code and performance would be what she would see. It shows what happens when she calls the high-level operations, which have hidden inefficiencies. It is clear that expert optimizations are necessary to obtain good performance.

In these results, the un-optimized code performs considerably worse than the optimized code because of the inefficiencies described in Section 2. By refining abstractions to machine-specific implementations and removing those inefficiencies, the system produces an implementation that is the same as the hand-optimized code in the Elemental library.

4.3 Additional operations

While from the start the system was designed to be applied to most, if not all, of the operations supported by `libflame`, the initial development used Cholesky as the driving example. Once this worked, the system was applied to other operations to examine how easily the system can be applied to new algorithms and extended with new knowledge.

Our first experiment was to apply the system to a specific algorithm for triangular solve with multiple

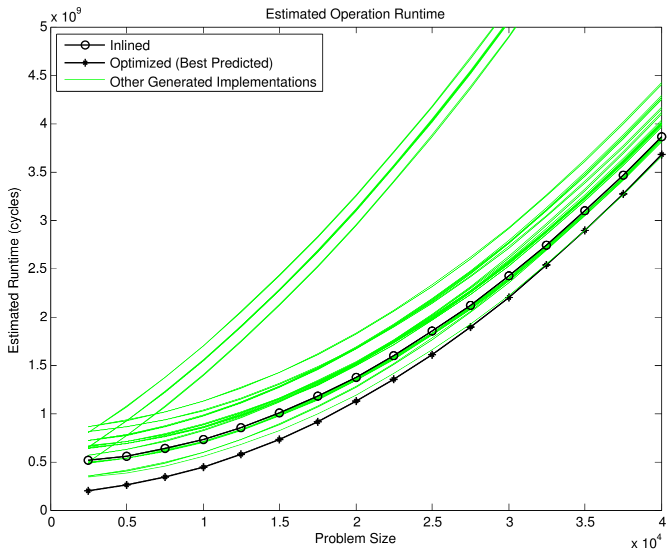

Figure 6: `Trsm` estimated runtime in processor cycles on 240 cores.

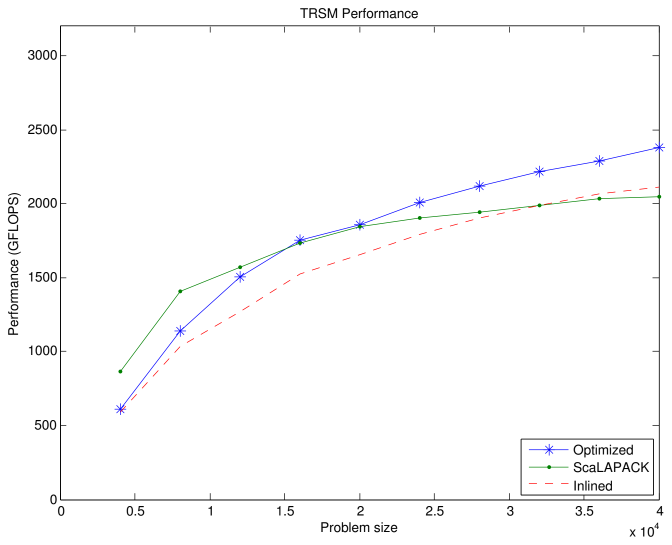

Figure 7: `Trsm` implementation performance on 240 cores. Peak performance is at the top of the graph.

right-hand sides (`Trsm`) that casts the computation in the loop-body in terms of operations that are very similar to those in the loop-body of Cholesky factorization Variant 3. As expected, getting hundreds of implementations from the system took very little work on our part. Figure 7 shows the performance of the un-optimized, inlined code and the optimized implementation generated by the system, which turned out to be identical to the hand-created version. Again, the system's costs models point to the same implementation as the hand-tuned as the best out of the hundreds that were automatically generated. Figure 6 illustrates the cost estimates of the best generated implementations and representative other implementations.

Next, we tested our system's implementation of Cholesky Variant 2. It requires a different flavor of parallelization since the bulk of computation is in the `Gemm` operation, which requires local computations to be summed (reduced) across processes. Once this distributed-memory implementation of `Gemm` was encoded in the system, it was again able to produce the same optimized implementation code that an expert created.

The greatest triumph to date for the system came when we applied it to an algorithm for a much more complex operation, $A := L^{-H} A L^{-1}$, an operation that is important when reducing a generalized Hermitian symmetric positive-definite eigenvalue problem to a standard problem and one of the most complex operations in Elemental. This operation and its parallelization with Elemental is discussed extensively in [21]. The system generated *tens of thousands* of possible implementations, from which cost models chose one that is more efficient than the optimized implementation that had been coded by the expert developer of Elemental. We will report on this experience in a future paper as there is insufficient space here to give sufficient detail to this complicated operation.

5 Related Work

In many ways, the present paper takes a giant step forward for a vision that has been part of the FLAME project since its inception. In the first dissertation that resulted from the project [15], "The Big Picture" was expressed that already captured the idea of encoding algorithms and expert knowledge and mechanically transforming it into code. There, too, optimized parallel implementations were the goal. At the time, the PLAPACK library [28] played the role of a domain specific language much like the Elemental library does in this paper. Many implementations were generated by a system coded in Mathematica and performance estimates were generated from annotations with cost functions of the algorithms. The present work benefits from an extra ten years of insights during which dozens of papers were published that slowly filled in the blanks of knowledge that now enable the current, more sophisticated, approach based on MDE.

Our approach is similar in goal to the SPIRAL project [22], which primarily focuses on the domain of Digital Signal Processing (DSP). SPIRAL aims to automatically generate high-performance kernels for target architectures. It starts with a mathematical description of the operation in a DSL and performs transformations similar to refinements to recursively replace abstract operations with specific implementation code. It uses learning techniques, online code compilation, and performance testing to explore the space of implementations. Our approach is aimed at higher-level operations, built on lower-level, architecture-specific functions like the BLAS. At this higher-level, we are able to rely on relatively accurate cost models instead of using online-search. Furthermore, our domain has limited levels of recursion when refining algorithms, which allows us to fully enumerate the search space. SPIRAL cannot do this because the search space for their operations is often too large. The lower-level, architecture-specific functions we rely on could be hand-tuned, as is the case with those used in Section 4, or could be automatically generated by an approach like SPIRAL. Thus, we can envision building upon kernels that are themselves generated by a system like SPIRAL.

Autotuning is often viewed as a way to automatically improve performance [31]. Our approach is different in that it generates the search space from a high-level understanding of how algorithms can be transformed. Also, we generate parameterized cost estimates which then guide us to the best implementation(s). We can envision adding autotuning to this approach in order to then choose the best parameters like, for example, the algorithmic block size.

6 Future Work

This paper presents initial, convincing evidence that loop-bodies for dense linear algebra algorithms can be automatically optimized. This is part of a larger picture and more general goal, as we discuss here.

6.1 The Big Picture

Our goal is to automatically generate libraries of algorithms for this domain by encoding knowledge about operations and target architectures. A system would then transform this knowledge into optimized algorithms based on cost estimates, automatically generating families of implementations. Our prototype shows promising results for operations that are highly indicative of most operations found in the domain of DLA.

Our approach will allow us to change how libraries for this domain and other domains are developed from instantiations in code to repositories of algorithms, knowledge about how to transform algorithms, and knowledge about architectures. In this volatile time when architectures are changing dramatically, this will allow libraries to be generated by providing knowledge about a new architectures as an alternative to using the existing library, modifying the existing library, or writing a new library.

6.2 Next steps

The initial investigations reported in this paper point to numerous possibilities.
Pruning the search space. The experiment with the operation $A := L^{-H} A L^{-1}$ showed how theory and/or heuristics are needed to prune the search space since the time for comparing the cost estimates becomes prohibitively expensive when tens of thousand or more implementations are generated. This is an active area of research and will be covered in more detail in a follow-up paper.
Adding knowledge. We have not yet included all possible transformations in our prototype system. This is obvious from the fact that for `Trsm` the ScaLAPACK implementation outperformed all optimizations that were generated by the system for a range of matrix sizes. Correcting this should be a matter of adding transformations to the system. Also, the cost functions that were used were first-order approximations for the true cost of the various operations. Better parameterized costs estimates can eventually be incorporated.
Other target architectures. We chose to first apply the system to the optimization of distributed memory algorithms for three reasons: (1) Since there are a lot of choices for how computations are distributed among processes and how to communicate between different distributions, we knew a large number of possible algorithms would results; (2) We suspected that first-order approximations for the cost of operations would suffice, at least when very large matrix sizes were involved; and (3) A considerable penalty would observed if a clearly wrong optimization was chosen, so the benefits of optimization would be clearly visible in the experiments. An equally important application of the current system would target, for example, optimization of sequential and multithreaded dense linear algebra libraries like those with functionality supported by the BLAS and `libflame`. There, communication would show up in the form of copying of data into contiguous buffers for performance reasons and computation would be performed by so-called inner kernels [12, 13, 31]. While in principle this is very similar to what we have described in this paper, we suspect that in practice the cost functions need to be much more accurate and sophisticated if benefits are to be shown. We will pursue this in future research.
Beyond a single loop-body. All optimizations discussed in this paper deal with a single loop-body. We envision in the future adding loop transformations, much like modern compilers perform but at a high level of abstraction. For example by merging two or more consecutive loops, the body of the resulting loop might expose more opportunities for the kinds of optimizations that are discussed in this paper. Similarly, if loops are unrolled, inter-iteration optimizations may be exposed.

6.3 Other Scientific Applications

The techniques described in this paper provide a general method for analyzing and building structured algorithms. Most scientific applications are similarly well-structured providing ample opportunities to apply these techniques.
Sparse Matrix Solvers. For sparse matrix solvers, the sparse matrix-vector product is usually a performance bottleneck. To achieve better performance, choices such as data layout and algorithm are critical. Using the refinement process, our tool could be extended to make these decisions based on the details of the solver and data distribution.

For example, changing the sparse compression technique is a common refinement. If a solver works on preconditioning separable blocks of the matrix, redistributing the matrix from a general compressed row format to the compressed block format becomes critical for speedup. Another example is the large number of orderings for the degrees of freedom (dofs) in a solver. Often a solver must reevaluate a residual on the dofs, thus redistributing the matrix for optimal memory bandwidth is necessary contrary to the more natural ordering of blocking similar dofs together.
Finite Element Methods. Finite element methods allow for a number of choices for the assembly and discretization of the continuous problem. An assembly algorithm based on a reference element can be precomputed for any given equation. During this algorithm generation, refinements can be made based on the coefficients from mesh data, number of quadrature weights and order of quadrature loops. In the case of hp adaptivity, algorithms can be generating for each reference elements based on the given 'p' with refinements such as static condensation for a large enough element matrix and efficient interpolation between different elements.
Stencil Evaluations. Stencil evaluations have been studied extensively in the context of auto-tuning. The automation of loop unrolling and jamming has produced efficient code that is hard to imagine that even an

expert would be capable of producing. Unfortunately because of the low-level approach of auto-tuning, the gains are often completely lost once variable coefficients are used with the stencil. Using our MDE approach, one could abstract over the stencil and the necessary data streams for coefficient evaluation allowing for the same automation of unrolling and jamming.

6.4 Automated Software Engineering

Twenty years ago, *knowledge-based software engineering (KBSE)* addressed a similar problem of automated software design, but achieved mixed success [3, 14, 26]. Their ideas were correct—namely that domain-knowledge can be expressed as transformations and that automated and semi-automated design tools could aid engineers to design reliable systems. The success of KBSE was limited by its era: *model-driven engineering (MDE)* and its emphasis on transformation-based designs were unknown then, *component-based software engineering (CBSE)* and *software architectures* were nascent, and experience in building large systems by transformations simply did not exist. Times have changed and our work takes a fresh look at this general approach to automated software development.

Our derivations of efficient DLA codes are not isolated case studies. We have applied our approach to recover the legacy designs of stateless parallel database joins and stateful crash fault tolerant servers in terms of refinements and optimizations – exactly as we did for DLA codes in this paper [23]. Both of these case studies had informal descriptions; we showed how transformations could be used not only to explain their designs, but also to reconstruct these applications directly from our models. In the case of database joins, we have proven that each transformation (refinement or optimization) that we used in deriving the parallel join architecture is correct. We hope our work starts a new generation of tools to make KBSE ideas and design-by-transformation practical.

7 Conclusion

In this paper, we have demonstrated that it is possible to mechanize the actions of a expert dense linear algebra developer. We presented two non-trivial case studies that showed we could reproduce automatically what experts today produce manually. And further experiments clearly indicate as DLA design problems become more complex, a mechanized expert can produce even better code than manual designs.

The key to our approach is exposing the inherent structure of the DLA domain – this is accomplished by engineering layered designs, which captures the fundamental operations of the domain, and codifying fundamental algorithms that implement these operations, and the fundamental optimizations that naturally arise in this domain.

Given this structure, we explained that the manual process that a DLA expert uses to design efficient algorithms is so systematic that we could mechanize these tasks. We presented a tool that accomplished this task. Further, we explained why our approach is not limited to DLA, and that further investigation of 'libraries of the future' shows great promise. As such we expect this paper to be the first of many to explore the topics described above.

For additional information on FLAME visit

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

Acknowledgements

We thank the other members of the FLAME team for their support. Bryan Marker was sponsored by a fellowship from Sandia National Laboratories, and Jack Poulson was sponsored by a fellowship from the Institute of Computational Engineering and Sciences. Batory is supported by the NSF's Science of Design Project CCF 0724979. This research was also partially sponsored by NSF grants OCI-0850750 and CCF-0917167 as well as by a grant from Microsoft.

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, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[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] I. Baxter. Practical issues in building knowledge-based code synthesis sys tems. *6th Annual Reuse Workshop (WISR'93)*, 1993.

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

[5] P. Bientinesi, B. Gunter, and R. A. V. de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 35(1), 2008.

[6] P. Bientinesi, E. S. Quintana-Ortí, and R. 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.

[7] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[8] 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.

[9] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[11] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[12] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[13] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[14] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. *Kestrel Institute Technical Report KES.U.83.2*, 1983.

[15] J. 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.

[16] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[17] 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.

[18] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architectural Refinement. *IEEE TSE*, 21:356–372, 1995.

[19] D. Petriu, N. Rouquette, and ystein Haugen, editors. *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010. Proceedings*, Lecture Notes in Computer Science. Springer, 2010. (to appear).

[20] J. Poulson, B. Marker, J. R. Hammond, N. A. Romero, and R. van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. FLAME Working Note #44 TR-2010-20, The University of Texas at Austin, Department of Computer Sciences, 2010. Submitted to ACM TOMS.

[21] J. Poulson, R. van de Geijn, and J. Benninghof. Parallel algorithms for reducing the generalized Hermitian-definite eigenvalue problem. *ACM Trans. Math. Soft.* submitted.

[22] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[23] T. Riche, D. Batory, R. van de Geijn, R. Goncalves, , and B. Marker. Architecture design by transformation. Computer Science report TR-11-XX, Univ. of Texas at Austin, 2011.

[24] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management Syst em. In *ACM SIGMOD*, 1979.

[25] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*. Lecture Notes in Computer Science 6. Springer-Verlag, New York, second edition, 1976.

[26] D. R. Smith and E. A. Parra. Transformational approach to transportation scheduling. *8th Knowledge-Based Software Engineering Conference*, 1993.

[27] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

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

[29] F. G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[30] F. G. Van Zee, E. Chan, R. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. Introducing: The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[31] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.