Designing Linear Algebra Algorithms by Transformation: Mechanizing the Expert Developer

Regular Paper

Bryan Marker[1], Jack Poulson[2], Don Batory[1], and Robert van de Geijn[1]

[1] Dept. of Computer Science
The Univ. of Texas at Austin
{bamarker,batory,rvdg}@cs.utexas.edu
[2] Institute for Computational Engineering and Sciences
The Univ. of Texas at Austin
poulson@cs.utexas.edu

Abstract. To implement dense linear algebra algorithms for distributed-memory computers, an expert applies knowledge of the domain, the target architecture, and how to parallelize common operations. This is often a rote process that becomes tedious for a large collection of algorithms. We have developed a way to encode this expert knowledge such that it can be applied by a system to generate mechanically the same (and sometimes better) highly-optimized code that an expert creates by hand. This paper illustrates how we have encoded a subset of this knowledge and how our system applies it and searches a space of generated implementations automatically.

1 Introduction

Parallelizing and optimizing dense linear algebra (DLA) algorithms for distributed-memory machines is typically accomplished by domain experts very familiar with both linear algebra and the oddities of the target machine. When a DLA expert has no experience with distributed-memory code 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 is very mechanical and systematic, and therefore automatable.

We use pipe-and-filter graphs and graph transformations to codify the fundamental algorithms and distributed-memory expertise used in Elemental [20], a domain-specific language with functionality similar to ScaLAPACK [8] and PLAPACK [26]. Doing so enables us to automate the activities of experts: selecting algorithms, composing algorithms, and applying optimizations. In this paper, *we show how expert-tuned, high-performance parallel code for a handful of prototypical examples for distributed-memory architectures can be mechanically produced by a tool.*

We call this approach Design by Transformation (DxT) [23,19], pronounced "dext". We explain the basic ideas behind DxT that were developed while studying distributed-memory examples. Further, we describe how we automatically generate code using DxT and show performance results. Lastly, we explain our future ambitions for DxT.

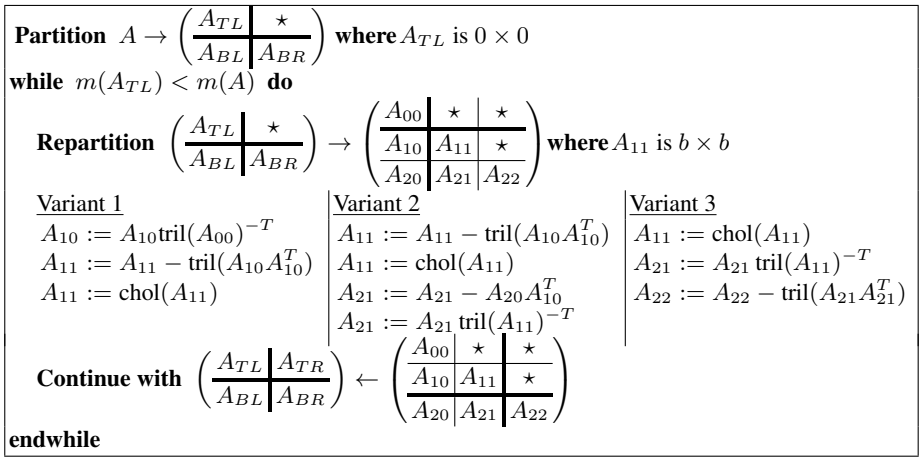

Fig. 1: Blocked algorithms for computing the Cholesky factorization. $m(A)$ stands for the number of rows of A and tril(A) indicates the lower triangular part of A. The '$\star$' symbol denotes entries that are not referenced.

2 What an Expert Does

To appreciate our work, let us examine what steps an expert follows in order to produce *by hand* a highly-optimized, parallel implementation of a dense matrix operation in Elemental. We choose 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. Section 3 explains how we automate this process.

From specification to algorithm to sequential code. Over the last decade, the FLAME project has developed a repeatable process by which loop-based families of algorithms for dense matrix operations, such as those in Basic Linear Algebra Subprograms (the BLAS [10,9,16]) and LAPACK [1], can be systematically derived [14]. FLAME uses formal derivation [5] and yields a number of algorithmic variants for each operation so the best for a given situation can be chosen[3]. In Figure 1, we show the three known blocked algorithmic variants that result when applied to Cholesky factorization. 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 $b = 1$. Henceforth, we use Variant 3 of Cholesky factorization as our running example.

The FLAME project has produced a library, `libflame` [27], 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 [4], that allows the code to closely resemble the algorithm of Figure 1. Code

[3] This expert task of deriving algorithms has been mechanized [3].

```
Chol( Lower, A11 );

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

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

(a) Original code.

```
A11_Star_Star = A11;
LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
LocalTrsm
   ( Right, Lower, Transpose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
A21 = A21_VC_Star;

A21_MC_Star = A21;
A21_MR_Star = A21;
LocalTriangularRankK
   ( Lower, Transpose,
     (T)-1, A21_MC_Star, A21_MR_Star,
     (T)1, A22 );
```

(b) Inline routines.

```
A11_Star_Star = A11;
LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
LocalTrsm
   ( Right, Lower, Transpose, 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_MR_Star = A21;
A21_VC_Star = A21;
A21_MR_Star = A21_VC_Star;
LocalTriangularRankK
   ( Lower, Transpose,
     (T)-1, A21_MC_Star, A21_MR_Star,
     (T)1, A22 );
```

(c) Inline communication.

```
A11_Star_Star = A11;
LocalChol( Lower, A11_Star_Star );
A11 = A11_Star_Star;

A21_VC_Star = A21;

LocalTrsm
   ( Right, Lower, Transpose, 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;
LocalTriangularRankK
   ( Lower, Transpose,
     (T)-1, A21_MC_Star, A21_MR_Star,
     (T)1, A22 );
```

(d) Remove redundant communication.

Fig. 2: Sequence of optimizations of the loop-body in Variant 3 of Cholesky.

in the style of the Elemental library [20] follows this approach; the code closely resembles the algorithm.

Elemental. If one were to code the algorithm in Figure 1 directly in Elemental code, the loop body would look like Figure 2(a). To see the hidden parallelism for these operations, we must review Elemental [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 (denoted $[M_C, M_R]$) [4]. 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 combinations of distributions.

We use Elemental because it provides a domain-specific language 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. 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 insights about these components 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.* Further, this results in common patterns of function calls. An expert knows the optimizations to apply to these repeated patterns across codes for different applications. From the expert's perspective, this layering and separation of concerns improves productivity and makes the code easier to port [18]. See [20] for more details on Elemental.

How an expert optimizes for distributed-memory architectures. With this background on 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 1. It can achieve very good performance on sequential machines, but it is only implicitly parallel if routines `Chol`, `Trsm`, and `TriangularRankK` are parallelized.

[4] We provide the Elemental notation for distributions so the graphs below can be understood, but we will not fully explain what the notation means. Please see [20] for more information.

To optimize this code, an expert focuses on the loop body, which we show again in Figure 2(a), and inlines the choices of parallelized implementations of its three operations to yield Figure 2(b):

- A_{11} is distributed among processes ($[M_C, M_R]$) and `Chol(Lower, A11)` represents a small part of all computation. Thus, a convenient way to perform this operation is to bring all data to all processes ($[\star, \star]$), 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 the processes, and `A11 = A11_Star_Star` places the updated data back in `A11` (with no communication).
- Next, consider the update $A_{21} := A_{21}\text{tril}(A_{11})^{-T}$. If one partitions A_{21} into rows as $A_{21} = \begin{pmatrix} a_{21,0}^T \\ a_{21,1}^T \\ \vdots \end{pmatrix}$ and redistributes A_{21} so that rows are assigned to processes in a cyclic order ($[V_C, \star]$), then the processes can compute

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

  locally in parallel *if* A_{11} is duplicated on all nodes ($[\star, \star]$). `A21_VC_Star = A21` redistributes A_{21}. `A11_Star_Star = A11` duplicates, again, A_{11}. The local computation is performed by `LocalTrsm(...)`. 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` and `A21_MR_Star = A21` ($[M_C, \star]$ and $[M_R, \star]$, respectively) 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: the statements `A11 = A11_Star_Star` and `A11_Star_Star = A11` can be replaced by the more efficient single `A11 = A11_Star_Star`, which eliminates unnecessary communication.

Further, an Elemental expert knows that redistributions like `A21_MC_Star = A21` can be composed from two or more redistributions via intermediate distributions. Some choices of possible substitutions are exposed in Figure 2(c). In most instances, this simply inlines intermediate distributions that were previously hidden. For example replacing `A21 = A21_VC_Star` by

`A21_MC_Star = A21_VC_Star; A21 = A21_MC_Star.`

An expert knows this 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 2(d). For example the redundant `A21_MC_Star = A21_VC_Star` is removed.

Summary. An expert performs (consciously or subconsciously) the previously described steps to optimize the code of Figure 2(a). 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 graph transformations. The keys are (1) layer algorithms, and (2) explicitly codify implementation knowledge about (i.e. options for) 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 uses 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 DxT mechanizes this process.

Using graphs to model algorithms and code. The classic (and arguably greatest to date) example of automated software development is relational query optimization (RQO) [24,25]. 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 a data-flow graph setting using graph transformations. The starting point for our optimization is the loop-body in Figure 2(a) or, equivalently, Figure 1, which is represented as the data-flow graph of Figure 3. The inputs to this graph are the submatrices of A and the results of the loop body (outputs of the graph) are the updated submatrices of A. The boxes represent the update operations of the loop body (e.g. `Chol`, `TriangularRankK`, `Trsm`). These operations, also called abstractions, have no implementation details. Just as in the starting algorithm, abstraction boxes only have precondition and postconditions to specify their functionality (we omit these details here).

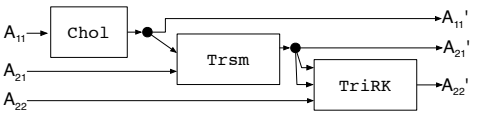

Fig. 3: Cholesky variant 3 algorithm loop body in data-flow graph. This uses abstractions (solid boxes) for component operations; implementations must be chosen.

Abstractions are implemented by algorithms; the pairing of operations with their algorithms form algebraic identities (a.k.a. refinements) of the domain. Algorithms can reference lower-level operations, which have their own implementing algorithms, and this recurses. This codifies the layering of software libraries. There could potentially be many refinements for an operation, each depending on, for example, architecture-specific details, the method of parallelization, or numerical stability characteristics. An expert explores such options or instinctively chooses one out of experience. In

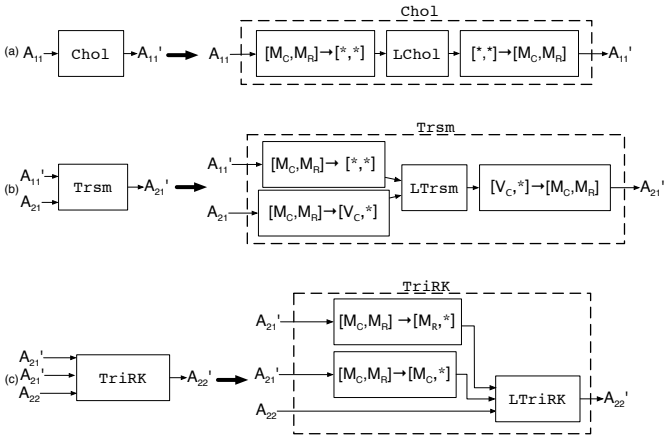

Fig. 4: Sample refinements to implement some operations for distributed-memory. The boxes with $\rightarrow$ are redistribution operations in Elemental (i.e. "=" operators) from one distribution, represented on the left of the arrow, to another, represent on the right of the arrow. The other solid boxes represent local computation.

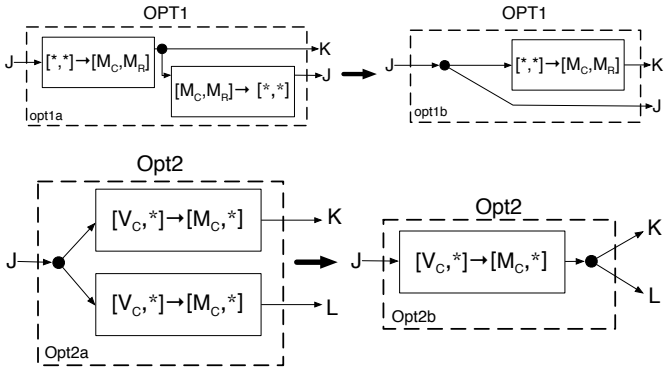

Fig. 5: Sample optimizing graph transformations to remove unnecessary redistribution.

the Cholesky algorithm, an expert replaces abstractions with implementation details as shown in Figure 2(b), which are represented as graph transformations in Figure 4. That figure only shows the best refinement for each abstraction in this algorithm; keep in mind there are others. The boxes represent either redistribution operations (i.e. the "=" operator in the code) or a sequential LAPACK or BLAS function call. Some of the redistribution boxes are abstractions that can be further refined with implementation options like an expert changing code from Figure 2(b) to Figure 2(c). These refinements are also represented as graph transformations.

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 exp_2. An expert developer recognizes the inefficient redistribution code patterns in Figure 2(c) and optimizes the code to end with Figure 2(d). We can represent these optimizations as graph transformations such as those shown in Figure 5. The top is equivalent to the optimization needed to replace `A11 = A11_Star_Star` and `A11_Star_Star = A11` with just `A11 = A11_Star_Star` and the bottom removes the redundant `A21_MC_Star = A21_VC_Star` in the Cholesky example.

Using a graph representation. By refining the operations of the Cholesky algorithm in Figure 2(a), the top layer of code is flattened to expose redistribution in Figure 2(b). These redistributions are another layer of operations that can be refined in various ways. By refining some of them as in Figure 2(c), we can break through this layer to expose inefficient redistributions that can be removed to create the optimized implementation in Figure 2(d). All of these steps can be encoded as graph transformations that can be applied to more algorithms than just this Cholesky example.

By exploring the space of equivalent graphs (implementations) for a given DLA application and selecting the graph with the best performance characteristics, an efficient DLA program is synthesized. Source is produced by translating the optimized graph to code. These refinements and optimizations are the same as those experts know well and currently apply repeatedly by hand; they can be thought of as transformations that incrementally elaborate DLA programs. One goal of our work is to enable experts to identify and encode these transformations so they can be mechanically applied. Just as functions are modularized for re-use, these transformations can be modularized for re-use, hence the name 'Design by Transformation'.

Given a portfolio of basic, local (sequential) operations and redistribution primitives, cost functions for each primitive, and a target sequence of DLA operations (e.g. as given in Figure 2(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 implementations).

Searching the space of implementations. To an observer, an expert implementing an algorithm appears to follow instincts to select refinements and optimizations. In fact, though, (s)he explores possibilities and assesses cost (implicitly or explicitly). How do we enable a mechanical system to choose the best implementation using "instincts"? We do not (*yet*). Instead of encoding the heuristics and instincts of an expert, we currently use a breadth-first (or brute-force) search that works well for all of the algorithms we have studied thus far. 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. For all examples we describe in this paper, DxTer takes at most four hours to generate the search space; Cholesky takes less than 5 seconds. By associating a cost with every implementation, the best in the search space can, in principle, be picked out analytically. Thus, our prototype system employs run-time cost estimates for redistribution and computation operations in Elemental to find the best-performing codes. We want the system to see that the code of Figure 2(d) is better than the code

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` ($m \times n$)	$\alpha\lceil\log_2 c\rceil + \beta\frac{c-1}{c}\frac{m}{r}n$

Table 1: Representative first-order approximations for the cost of operations .

of Figure 2(a) by summing operation costs and determining which takes less time to execute. It should then choose Figure 2(d) out of all implementations 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, the 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 and 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. Cross-over points between the best implementations occur often. To identify these in our system, we need more accurate models of computation and communication. Fortunately, a mechanical system does not care how complicated the expressions become, which we hope to investigate in future research. An expert rarely attempts this degree of optimization and accuracy since it requires careful analysis that is too error-prone and time consuming to perform by hand. Automation overcomes this hurdle. [5]

For DLA we have reasonable cost estimates. 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) $\gamma 2mkn$ where γ is the time for a floating point operation. The cost of every computation kernel can be approximated by the operation count multiplied by γ. [6]

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 transferred, respectively. For example redistributing an $n \times n$ block of A_{11} as in line `A11_Star_Star = A11` on p processes requires an all-gather 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 to enable the prototype we describe in the next section. They only

[5] Readers may note that this is exactly the RQO paradigm (described above) applied to DLA implementations.

[6] A second-order approximation would take algorithm performance variation into account, but for now we stick to first-order approximations since this is generally good enough for an expert implementing algorithms by hand.

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, but we expect to improve them to find the best code for more complicated algorithms. 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 DxT and the cost functions described in the previous section. We call this prototype DxTer [17], pronounced "dexter". We now describe our initial findings.

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.[7] The combined theoretical peak performance of all 240 cores is 3200 GFLOPS. For each problem, we tested a range of algorithmic block sizes and a set of process grid configurations and show the best results. Two-thirds of peak performance is shown at the top of the graphs.

Cholesky Variant 3. We encoded the most useful refinements for a handful of common operations (e.g. BLAS functions) as well as Elemental redistributions to enable DxTer to implement the Cholesky example. From Figure 3, DxTer is able to mechanically produce, without human intervention, hundreds of loop body implementations including all versions in Figure 2. Each of these is Elemental code for Cholesky Variant 3. This allows DxTer to explore the space of implementations. Additionally, we encoded more complicated and subtle expert optimizations that are out of the scope of this paper. With these DxTer generates an even better implementation than that of Figure 2(d). This superior implementation is identical to that coded by the expert developer of Elemental.

In its current incarnation, DxTer applies all possible graph re-writes to enumerate the entire search space of implementations. It then uses symbolic cost functions like those those described in Section 3 to choose the best of all the mechanically generated implementations (this is a breadth-first search). Figure 6 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

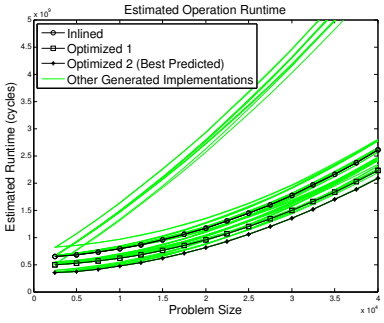

Fig. 6: Cholesky Variant 3 estimated runtime in processor cycles on 240 cores.

[7] We used versions 11.1 of the Intel compiler, 1.6 of MVAPICH2, 1.8.0 of ScaLAPACK, and 1.30 of the GotoBLAS.

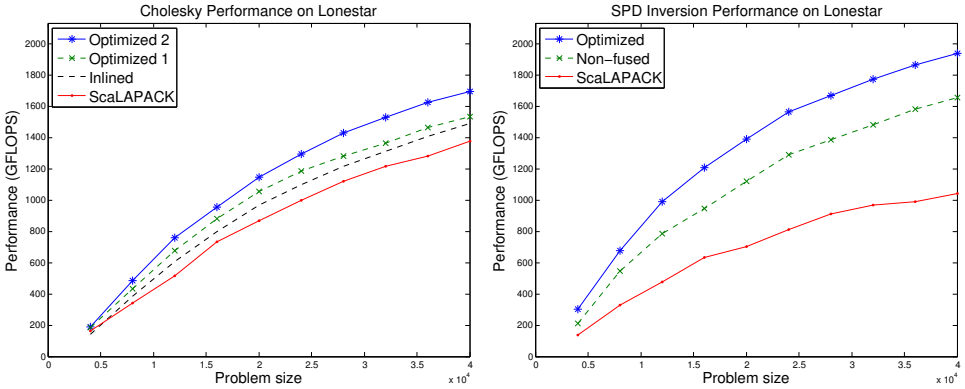

Fig. 7: Cholesky Variant 3 (left) and SPD Inversion (right) implementation performance. Two-thirds of peak performance is at the top of the graphs.

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 6, this implementation's cost estimate is at the bottom of the graph, labeled "Optimized 2."

In Figure 7 (left), we show the performance results of the code of Figure 2(c) (labeled "Inlined"), the code of Figure 2(d) (labeled "Optimized 1"), and the further optimized code (labeled "Optimized 2"). DxTer automatically generated all of these implementations. We leave out the performance of the original code as 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 performance would be what (s)he would see. It shows what happens when (s)he calls the high-level operations, which have hidden inefficiencies. The difference between "Optimized 1" and "Optimized 2" shows the performance gained when complex operations are understood and applied. It is clear that expert optimizations are necessary to obtain good performance.

Additional operations. DxTer 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, DxTer was applied to other operations to examine how easily the methodology can be applied to new algorithms and extended with new knowledge and reusing knowledge that is already encoded.

Our first experiment was to apply DxTer to a specific algorithm for triangular solve with multiple 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 DxTer took

very little work on our part because existing transformations were re-used. DxTer's costs models point to the same implementation as the hand-tuned as the best out of the hundreds that were automatically generated.

Next, we tested DxTer'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. With this refinement encoded, DxTer was again able to produce the same optimized implementation code that an expert created. Adding such a transformation to DxTer was an easy change, and it will henceforth be explored for any algorithm with a `Gemm` operation.

DxTer was also applied to triangular matrix multiplication and triangular matrix inversion. For the former, the expert made an implementation error that *produced wrong results*; DxTer generated correct code. For the inversion operation, DxTer generated a version that had slightly better theoretical numerical properties (with equivalent performance). DxTer had a transformation encoded to use `Trsm` with a triangular matrix before inversion instead of `Trmm` after inversion. The expert developer applied this transformation to other algorithms, so it was encoded into DxTer for them, but he forgot to apply it here. Elemental has been updated with both of the differences DxTer discovered.

Complex operations. Our greatest triumphs to date came when we applied DxTer to two much more complex operations, $A := L^{-1}AL^{-H}$, which is important in reducing a generalized Hermitian symmetric positive-definite eigenvalue problem to a standard problem, and a fused symmetric, positive-definite (SPD) matrix inversion algorithm.

The parallelization of $A := L^{-1}AL^{-H}$, or two-sided triangular solve, is discussed extensively in [21]. The loop body of one of five algorithmic variants is shown in Figure 8. This algorithm is significantly more complex than those we described previously, but it is similar in style to them and many other DLA algorithms. In addition to the BLAS refinements already encoded, refinements for Axpy had to be added as well as some additional parallelization schemes for TriangularRankK. DxTer will now explore them with any algorithm that uses them. Lastly, the unblocked operation $L^{-1}AL^{-H}$, specific to this algorithm, was added. After these additions, DxTer was able to generate *tens of thousands* of implementations. The existing optimizations described above enabled much of this variety; no new optimizations were needed. DxTer's cost models were used to automatically choose a "best" implementation from those generated. The chosen version was slightly better than the optimized version implemented by the expert developer of Elemental. He had forgotten to apply an optimization that was used in other algorithms. Our tool had no such excuses and found the superior implementation.

$$
\begin{aligned}
A_{10} &:= L_{11}^{-1}A_{10} \\
A_{20} &:= A_{20} - L_{21}A_{10} \\
A_{11} &:= L_{11}^{-1}A_{11}L_{11}^{-H} \\
Y_{21} &:= L_{21}A_{11} \\
A_{21} &:= A_{21}L_{11}^{-H} \\
A_{21} &:= W_{21} = A_{21} - \tfrac{1}{2}Y_{21} \\
A_{22} &:= A_{22} - \\
&\qquad (L_{21}A_{21}^{H} + A_{21}L_{21}^{H}) \\
A_{21} &:= A_{21} - \tfrac{1}{2}Y_{21}
\end{aligned}
$$

Fig. 8: Loop body of $A := L^{-1}AL^{-H}$.

Next, we applied DxTer to a fused-loop algorithm for SPD matrix inversion. We encoded variant 2 of the algorithm described in [6], the loop body of which is shown in Figure 9. With no additional transformations, DxTer generated *hundreds of thousands* of implementations and chose the same implementation as the expert developed. Figure 7 (right) shows the resulting performance. The "Optimized" version is the implementation generated by DxTer. The "Non-fused" version is the implementation that uses the optimized Cholesky factorization, triangular matrix inversion, and triangular matrix-matrix multiply operations, described above, in succession (also generated by DxTer). This prohibits some optimizations to reduce communication allowed by the fused-loop algorithm. ScaLAPACK uses a non-fused version of the algorithm.

$$
\begin{aligned}
A_{11} &:= Chol(A_{11}) \\
A_{01} &:= A_{01}A_{11}^{-1} \\
A_{00} &:= A_{00} + A_{01}A_{01}^{T} \\
A_{12} &:= A_{11}^{-T}A_{12} \\
A_{02} &:= A_{02} - A_{01}A_{12} \\
A_{22} &:= A_{22} - A_{12}^{T}A_{12} \\
A_{01} &:= A_{01}A_{11}^{-T} \\
A_{12} &:= -A_{11}^{-1}A_{12} \\
A_{11} &:= A_{11}^{-1} \\
A_{11} &:= A_{11}A_{11}^{T}
\end{aligned}
$$

Fig. 9: Loop body of SPD matrix inversion.

To recap, these examples demonstrate that it is possible to generate complex, high-performance DLA code mechanically. Indeed, the original motivation for a tool like DxTer was to simplify the burden of experts. We believe DxT is a practical basis to do this.

5 Related Work

Our 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 [13], "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 [26] 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 graph transformations.

DxT is similar in goal to the SPIRAL project [22], which primarily focuses on the domain of Digital Signal Processing (DSP). SPIRAL automatically generates high-performance kernels for target architectures. It starts with a mathematical description of the operation in a DSL and applies rewrite transformations to recursively replace operations with implementation code. It uses learning techniques and performance testing to explore the space of implementations. DxT is aimed at higher-level operations, built on lower-level, architecture-specific functions like the BLAS, which allows us to use relatively accurate cost models instead of online-search. We can envision building on SPIRAL-generated kernels, though, instead of the hand-tuned components we use today. We would need the kernels as well as cost estimates for this to work.

Autotuning is often viewed as a way to automatically improve performance [28]. DxT 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.

The Broadway compiler [15] had a similar goal as ours to encode expert knowledge to enable optimized code generation. Library function annotations enable Broadway to choose the best implementation at a call site. Broadway is not able to optimize by replacing inlined code with a better implementation, though. Further, Broadway does not search the space of implementations, which is necessary to avoid local minima.

Finally, The Tensor Contraction Engine (TCE) [2] aims to generate high-performance code for a tensor contraction expressed at a high-level using a DSL. It does so by applying transformations to reduce computational complexity, space complexity, communication cost, and then data access. The transformations and cost models of TCE are very similar in spirit and goal to DxT. TCE is geared specifically to tensor contractions while DxT is more general (i.e. not just DLA algorithms [23]). We aim to make transformations easy to understand and encode, so DxT can be applied to other domains.

6 Future Work

Our larger goal is to automatically generate libraries of algorithms for this domain by encoding knowledge about operations and many target architectures. A system would then transform this knowledge into optimized algorithms based on cost estimates, automatically generating families of implementations for more than just distributed-memory machines. Our prototype shows promising results on distributed-memory targets for operations that are indicative of most operations found in the domain of DLA. Obviously, there remain many additional problems before we can reach this goal, such as the topics below.

Adding knowledge. We have not yet included all possible transformations in our prototype system. Instead, we have incrementally added those needed by algorithms as we target them for experiments. For example similar to the more advanced optimizations an expert applies for a particular target architecture, there are likely other target-specific optimizations that should be added 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 to predict machine-specific performance oddities.

Other target architectures. We chose first to test with distributed-memory algorithms for three reasons: (1) We knew a large number of possible algorithms would result; (2) We suspected first-order approximations for the cost of operations would suffice for large matrix sizes; and (3) A considerable penalty would observed if a clearly wrong optimization was chosen, so the benefits of optimization would be clearly visible in experiments. Another important application of DxT would target optimization of sequential and multithreaded dense linear algebra libraries . 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 [11,12,28]. While in

principle this is similar to what we have described in this paper, we suspect that in practice the cost functions need to be more accurate and sophisticated. With knowledge encoded to optimize for different architectures, DLA software could be optimized at all layers of code. We will pursue this in future research.

Pruning the search space. For now our breadth-first approach to search is sufficient. It only takes hours for even the most complex operations. When optimizing at all architectural levels, though, the search space will become substantially larger, and this cost will become prohibitive. We must study how to prune the space to limit an explosion of choices and we must study more advanced searching techniques. This is an active area of research and will be covered in more detail in a follow-up paper.

7 Conclusion

Using Design by Transformation, we have demonstrated that it is possible to mechanize the actions of an expert dense linear algebra developer to parallelize an algorithm for a distributed-memory target. We presented multiple non-trivial case studies that showed we could reproduce automatically what experts today produce manually. One of the more complicated examples clearly indicates as DLA design problems become more complex, a mechanized expert can produce even better code than manual designs. DLA codes targeting distributed-memory architectures and the related optimizations have similar structure to the examples we have explored. Therefore, we believe our prototype's successes thus far indicate potential for success for a large body of DLA algorithms for distributed-memory computers and even other targets.

The key to DxT is exposing the inherent structure of the DLA domain – this is accomplished by capturing the fundamental operations of the domain using layered designs. Further, we codify fundamental algorithms that implement the operations and 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 goal. Further, we explained why we believe DxT is not limited to distributed-memory and how it can be used to optimize code beyond what is currently possible by hand. As such we expect this paper to be the first of many to explore the topics described above.

Acknowledgements

Marker was sponsored by a fellowship from Sandia National Laboratories and an NSF Graduate Research Fellowship under grant DGE-1110007. 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. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, and resources at the Texas Advanced Computing Center. *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 et al. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.
2. A. Auer et al. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. *Molecular Physics*, 2005.
3. Paolo 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.
4. Paolo Bientinesi et al. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.
5. Paolo Bientinesi et al. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 35(1), 2008.
6. Paolo Bientinesi et al. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.
7. Ernie Chan et al. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.
8. J. Choi et al. 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. Jack J. Dongarra et al. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.
10. Jack J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
11. Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.
12. Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.
13. John A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.
14. John A. Gunnels et al. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.
15. Samuel Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of the IEEE*, volume 93, pages 342–357. 2005. Special issues on program generation, optimization, and adaptation.
16. C. L. Lawson et al. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.
17. Bryan Marker et al. Dxter: An automated software generation prototype for dense linear algebra. In preparation.
18. Bryan Marker et al. Programming many-core architectures - a case study: Dense matrix computations on the intel scc processor. *Concurrency and Computation: Practice and Experience*. To appear.
19. Bryan Marker et al. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *PPoPP '12: Proceedings of the seventeenth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2012. 2 pages. To Appear.
20. Jack Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. Accepted.
21. Jack Poulson et al. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. *ACM Transactions on Mathematical Software*. submitted.

22. Markus Püschel et al. 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.L. Riche et al. Software architecture design by transformation. Computer Science report TR-11-19, Univ. of Texas at Austin, 2011.
24. P G Selinger et al. Access Path Selection in a Relational Database Management Syst em. In *ACM SIGMOD*, 1979.
25. Jeffrey D. Ullman et al. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
26. Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.
27. Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.
28. R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.