A Case Study in Mechanically Deriving Dense Linear Algebra Code

Bryan Marker
bamarker@cs.utexas.edu

Don Batory
batory@cs.utexas.edu

Robert van de Geijn
rvdg@cs.utexas.edu

Abstract

Design by Transformation (DxT) is a top-down approach to mechanically derive high-performance algorithms for dense linear algebra. We use DxT to derive the implementation of a representative matrix operation, `two- sided Trmm`. We start with a knowledge base of transformations that were encoded for a simpler set of operations, the level-3 BLAS, and add only a few transformations to accommodate the more complex `two- sided Trmm`. These additions explode the search space of our prototype system, DxTer, requiring the novel techniques defined in this paper to eliminate large segments of the search space that contain suboptimal algorithms. Performance results for the mechanically optimized implementations on 8,192 cores of a BlueGene/P architecture are given.

1 Introduction

Experts compose Dense Linear Algebra (DLA) libraries from a common set of matrix/vector and data rearrangement operations. For a particular target architecture, in this study a distributed-memory architecture (cluster), sequential primitives found in libraries like the Basic Linear Algebra Subprograms (BLAS) [5, 6, 10] and LAPACK [2] or `libflame` [25] are composed with data redistribution operations like the collective communications found in the Message-Passing Interface (MPI) [22]. An expert relies on his/her knowledge about the algorithms and the target architectures to hand-parallelize and optimize each operation to be supported. The good news is that many DLA algorithms are sufficiently similar in that knowledge gained from optimizing one operation can be applied to many others. The bad news is that the functionality that is expected from a DLA library is vast. More bad news: as new knowledge arises due, for instance, to architectural changes or a new insight, optimizations must be re-applied across entire libraries. Bottom-line: we are in a period of great change. We believe the way forward is automation.

Design by Transformation (DxT) (pronounced "dext") is a top-down approach to mechanically derive high-performance algorithms for dense linear algebra; DxTer (pronounced "dexter") [11] is our prototype implementation of DxT. From an input graph that specifies a sequence of DLA operations, DxTer automatically generates a high-performance implementation using DLA knowledge that is encoded as graph rewrite rules (transformations). For clusters, it generates code for Elemental [15], a library with functionality similar to ScaLAPACK [4], that we use as a Domain-Specific Language (DSL).

Previously, we demonstrated success with DxT and DxTer for a variety of DLA algorithms [12, 13][1] and in other domains [18] [2]. In this paper, we apply previously encoded knowledge to a new operation, `two-sided Trmm` ($L^H AL$, where L and A are a lower-triangular and general matrix, respectively) [16].

[1] DxT and DxTer were introduced in [13]; we derived efficient algorithms for Cholesky decomposition and SPD inversion. [12] extended this work to all level-3 BLAS functions, which form the building blocks of much of DLA functionality.

[2] In [18], we should how DxT can be effectively applied to other domains.

This operation is important when reducing a generalized Hermitian symmetric positive-definite eigenvalue problem to a standard Hermitian problem; it is illustrative of a non-trivial algorithm in DLA in general, and Elemental in particular.

We previously reported success with a related operation, `two-sided Trsm` $(L^{-1}AL^{-H})$ [13]. For `two-sided Trmm`, we demonstrate how existing distributed-memory DLA knowledge encoded in DxTer can be leveraged when a new operation is to be optimized. The only additional transformations we needed to generate algorithms for `two-sided Trmm` come from new instantiations of an existing transformation template and a transformation to support the new `two-sided Trmm` subproblem. Unfortunately, rather than incrementally enlarging the space, these additional transformations explode the search space, rendering our existing implementation of DxTer incapable of producing a solution.

The main contributions of this paper are new methods to prune the space of suboptimal algorithms, thereby allowing DxTer to produce efficient solutions for a wider range of DLA operations. These methods to limit the search space will become more important as DxTer implements and optimizes code through more layers of architecture (e.g. shared-memory memory). Here, we present important first steps toward that goal that enabled DxTer to implement `two-sided Trmm`. Without these developments, DxTer would not have been successful.

2 Design by Transformation

2.1 What is DxT?

DxT is a top-down, domain-agnostic approach to the mechanical generation of algorithms. It uses pipe-and-filter graphs and graph transformations to codify the fundamental algorithms and domain expertise [21]. DxT enables us to automate the activities of experts: selecting and composing algorithm implementations and applying optimizations.

2.2 Graph Representation

We use a Directed Acyclic Graph (DAG) as an input representation of the algorithm to be implemented. Each node—also called a box or operation—represents a function call. Box inputs are indicated by incoming edges and box outputs by outgoing edges.

Each box (node) represents a fundamental operation in a domain. There are two kinds of boxes: interfaces and primitives. A box with no implementation details, other than preconditions and postconditions, is an interface. A primitive is a box whose implementation is given to us.

DxT uses refinements to map a graph containing only interconnected interfaces to a graph that contains only interconnected primitives. A refinement is a transformation that replaces an interface with a subgraph that exposes an implementation using a specific algorithm. These subgraphs reference lower-level interfaces and/or calls to primitives. DxT applies refinements until all interfaces are removed and only primitives remain.

Graphs that result exclusively from refinements generally do not attain the best performance because refinements can introduce and cannot remove redundant computations and data movements. Therefore, DxT uses optimizations: transformations that replace one subgraph with another that implements the same functionality in a different way. Despite the name "optimizations", such transformations do not necessarily guarantee improved performance since multiple optimizations are often combined to achieve a better-performing algorithm.

2.3 Performance Estimation

DxTer takes as input a graph and a knowledge base of transformations. It explores every sequence of transformations on the input graph, generating a huge space of semantically-equivalent graphs. A generated graph that has no interfaces is a valid implementation of the input graph because it only references primitives. Mapping a graph containing only primitives to source code, called a Model-to-Text (M2T) transformation, is straightforward, as there is a 1-1 correspondence between primitives and C++ statements that are part of Elemental [13]. Among the many thousands, tens of thousands, or hundreds of thousands of graphs, some represent the most efficient implementation of the input graph.

To choose a best implementation, DxTer employs a different type of expert knowledge: cost estimates. An expert manually searches a space of implementations for the best-performing one. Experience (heuristics based on cost estimates) guide him/her to an efficient implementation. DxTer in its crudest form generates all implementations and then uses cost estimates to choose the best for a given problem size.

The measure of cost to use is domain-specific, but runtime (time to completion) is common to many domains. For DLA, DxTer sums the cost of primitive boxes on implementation DAGs to estimate their runtime, similar to how an expert judges a design by hand. DxTer outputs (i.e. selects as "best") the DAG that is least costly for the problem size under consideration.

What constitutes the "best" implementation is typically a function of the sizes of the operands (e.g., the sizes of the matrices) since these affect performance issues like load balance and communication overhead relative to useful computation. Since distributed-memory architectures are typically used to solve large problems that push the limits of available memory, the choice is typically skewed towards what is "best" for these large problems. Conveniently, for these large problems, low order terms in cost functions tend to become unimportant. [3]

3 Applying DxT to DLA

DLA algorithms are almost always loop-based [12, 13, 24]. The reason is that computation is orchestrated so that data in caches can be optimally reused. Figure 1 shows algorithms for a pair of non-trivial and representative operations: `two-sided Trsm` and `two-sided Trmm`. In each iteration, the algorithm exposes submatrices of the inputs on which computation is performed in what we call update statements, which are implementation-agnostic DxT interfaces. Their functionality is implemented for various architectures in libraries like the BLAS, LAPACK, `libflame`, and Elemental, which build on each other. For `two-sided Trmm` and `two-sided Trsm`, we only need to focus on the BLAS and Elemental libraries [15], from which we get the DLA primitive operations, local computations, and data redistributions that we use to implement higher-level DLA operations. In parentheses in Figure 1, we specify which BLAS interface is used for each update statement. The remaining two operations are themselves `two-sided Trsm` and `two-sided Trmm` on submatrices, described in Section 3.2. Their implementations are given to us as primitives and/or refinements in our knowledge base.

We input to DxTer a DAG that encodes the loop body of the algorithm to be implemented. In Figure 1, the DAG for each algorithm is given to the right of the loop body. DxTer applies refinements and optimizations on boxes that represent BLAS and Elemental operations as described in Sections 3.2 and 3.3. Runtime cost functions for DLA primitives that are used by experts are well-known [12, 13, 20] and are encoded in DxTer for each primitive box. This works well for block-synchronous parallel algorithms (a characteristic of Elemental). They are sufficient for DxTer (or a human expert) to choose between refinements and optimizations in this domain (i.e., to rank-order implementation options and choose the "best"). A strength of automation is that, if necessary, more

[3] The implementation choice is not sensetive to the exact blocksize and process grid selection as long as they are reasonble choices. These can be tuned for a particular architecture after an implementation is chosen.

Algorithm: $A := L^{-1}AL^{-H}$ (`two-sided Trsm`) and
$A := L^H AL$ (`two-sided Trmm`)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$

where A_{TL} and L_{TL} are 0×0.

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

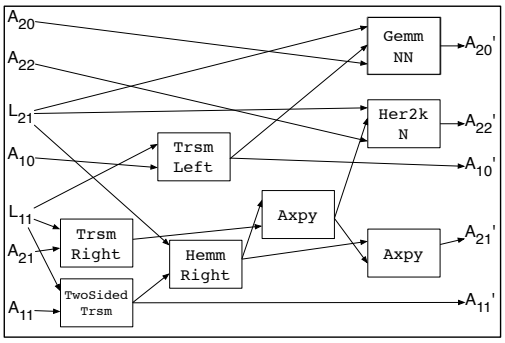
Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

where A_{11} **and** L_{11} **are** $b \times b$

Variant 4 for $L^{-1}AL^{-H}$

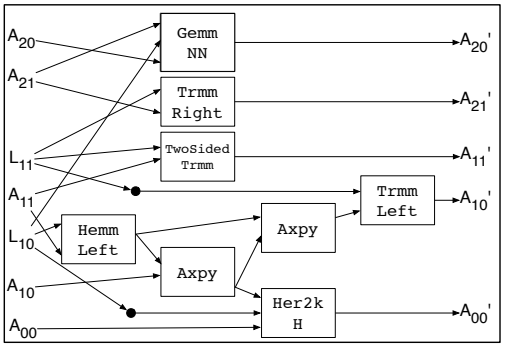
$A_{10} := L_{11}^{-1}A_{10}$ (`Trsm Left`)
$A_{20} := A_{20} - L_{21}A_{10}$ (`Gemm NN`)
$A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ (`two-sided Trsm`)
$Y_{21} := L_{21}A_{11}$ (`Hemm Right`)
$A_{21} := A_{21}L_{11}^{-H}$ (`Trsm Right`)
$A_{21} := A_{21} - \frac{1}{2}Y_{21}$ (`Axpy`)
$A_{22} := A_{22} - (L_{21}A_{21}^H + A_{21}L_{21}^H)$ (`Her2k N`)
$A_{21} := A_{21} - \frac{1}{2}Y_{21}$ (`Axpy`)



Loop body graph input to DxTer for `two-sided Trsm`.

Variant 4 for $L^H AL$

$Y_{10} := A_{11}L_{10}$ (`Hemm Left`)
$A_{10} := A_{10} + \frac{1}{2}Y_{10}$ (`Axpy`)
$A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ (`Her2k H`)
$A_{10} := A_{10} + \frac{1}{2}Y_{10}$ (`Axpy`)
$A_{10} := L_{11}^H A_{10}$ (`Trmm Left`)
$A_{11} := L_{11}^H A_{11}L_{11}$ (`two-sided Trmm`)
$A_{20} := A_{20} + A_{21}L_{10}$ (`Gemm NN`)
$A_{21} := A_{21}L_{11}$ (`Trmm Right`)



Loop body graph input to DxTer for `two-sided Trmm`.

Continue with

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

endwhile

Figure 1: Blocked Variant 4 for computing `two-sided Trsm` and `Trmm`. To the right of the update statements, the graph inputs to DxTer are given. Here $\star$ indicates the symmetric part of A that is neither stored nor updated.

accurate cost functions can be used that would be unmanageable for a human expert. So far, simple cost functions have been sufficient.

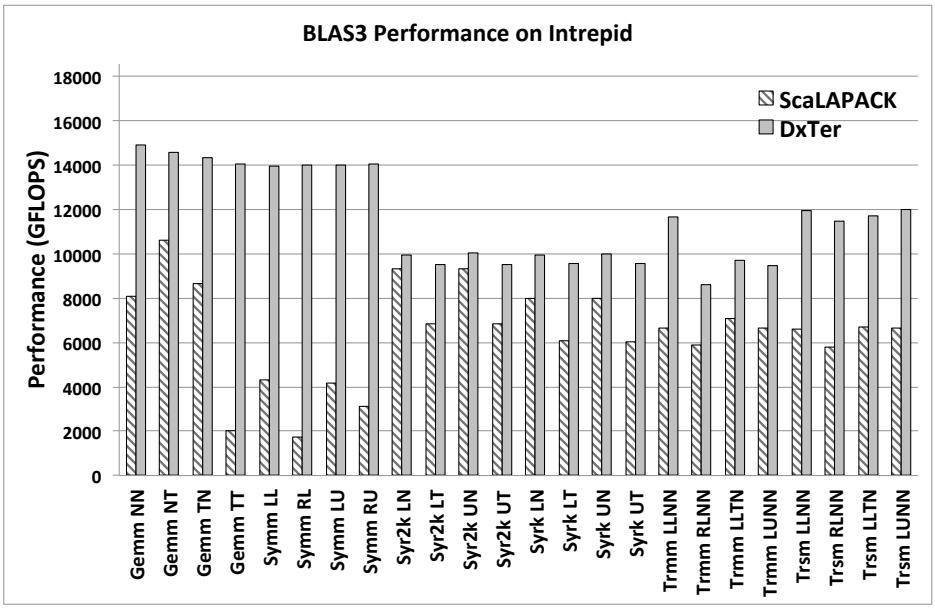


Figure 2: Performance of level-3 BLAS operation variants, generated by DxTer and implemented in ScaLAPACK [12]. These results for problem size 50,000 were taken on 8,192 cores of Argonne's BlueGene/P machine (Intrepid); the top of the graph is two-thirds of of peak performance.

Distribution	Location of data in matrix
$[*, *]$	All processes store all elements
$[M_C, M_R]$	Process $(i\%r, j\%c)$ stores element (i, j)
$[M_C, *]$	Row i of data stored redundantly on process row $i\%$r
$[M_R, *]$	Row i of data stored redundantly on process col. $i\%$c
$[*, M_C]$	Column i of data stored redundantly on process row $i\%$r
$[*, M_R]$	Column i of data stored redundantly on process col. $i\%$c
$[V_C, *]$	Rows wrapped around proc. grid in col.-major order
$[V_R, *]$	Rows wrapped around proc. grid in row-major order
$[*, V_C]$	Columns wrapped around proc. grid in col.-major order
$[*, V_R]$	Columns wrapped around proc. grid in row-major order

Figure 3: Examples of distributions on a $p = r \times c$ process grid.

Figure 2 (taken from [12]) shows that DxTer-generated code noticeably outperforms ScaLAPACK for a variety of level-3 BLAS operations.[4] Why is this important? Elemental is the state-of-the-art DLA library; few people on earth can write such code. The fact that we can now automate the development of such code is significant, because it is strong evidence that mechanical generation of high-performance code is possible and that it need not involve the few experts that do exist.

As said in the introduction, this paper describes the challenge to generate code for yet another DLA operation, `two-sided Trmm`. Only a few new rules are needed to generate an efficient implementation for it. However, these rules explode the search space, requiring a more "intelligent", rather than brute-force, search than what we have used so far.

We have found the following approach useful to explain and illustrate DxT: show the derivation of a high-performance algorithm for some Elemental operation, in this paper `two-sided Trmm`. This derivation is one of many thousands of algorithms that DxT generates; we single out the algorithm that was automatically determined to be the most efficient. In effect, we explain the reasoning behind the derivation—precisely the reasoning that an Elemental expert would use to produce this algorithm. While Elemental experts do not necessarily develop code in terms of transformations, it becomes evident that encoding design knowledge as transformations to replicate expert decision processes is indeed possible. In turn, we illustrate how DxT works and its generality.

3.1 Elemental Basics

While Elemental is mainly used by computational scientists as a library of DLA operations, for us it is a framework for parallelizing DLA algorithms. We use it as a DSL for which we generate code. We provide a minimal background for Elemental, needed to understand its use in DxT.

In Elemental, the p MPI [22] processes of a cluster are viewed as a two-dimensional grid, $p = r \times c$. The default distribution of matrices is a 2D element-wise cyclic distribution, labeled $[M_C, M_R]$[5].

Besides this 2D distribution, Elemental supports other 1D and 2D distributions, with examples listed in Figure 3. These options allow a programmer to parallelize an algorithm and its suboperations in many ways. It suffices to understand that to parallelize a computation, submatrices are redistributed from the default distribution to enable parallel local computation (by calling a locally-sequential function on each process), after which the result is placed back into the original distribution, possibly with a reduction operation.

Elemental is written in C++ and encodes matrices and attributes (including distribution) in objects. Redistribution from one distribution to another is accomplished using the overloaded C++ "=" oper-

[4] DxTer outperforms or does as well as Elemental.

[5] We will not go into explanations for distribution names. Details are in [15, 20].

ation, which hides the (MPI) collective communication required to perform data redistribution. By hiding such lower-level details, experts can focus on important implementation decisions: how to distribute data to parallelize computation and how to redistribute data.

An expert tries to balance the expense of collective communication with improvement in parallel computation (enabled by the communication). We next explain how parallelization choices are made and how optimizations can reduce communication.

3.2 Parallelization

The knowledge base of [12] encodes refinements to parallelize level-3 BLAS operations. We reuse these refinements to parallelize the `two-sided Trmm` algorithm in Figure 1.

With the exception of $A_{11} := L_{11}^H A_{11} L_{11}$ (`two-sided Trmm` itself, which we consider below), the update statements are standard BLAS operations. For each, there are many ways to parallelize their computation based on the size of operands, the parameters, and the surrounding code. An expert must explore these options to choose the best implementation. For each operation, input submatrices are redistributed from $[M_C, M_R]$ to some other distribution so that local computation can be performed after which the result is redistributed back to $[M_C, M_R]$.

Figure 4 shows the refinement for each update statement in `two-sided Trmm` that leads to the best implementation. Many, many others are encoded and explored. We now explain a few of them to give a feeling of how one parallelizes operations in Elemental and how to encode that knowledge with DxT.

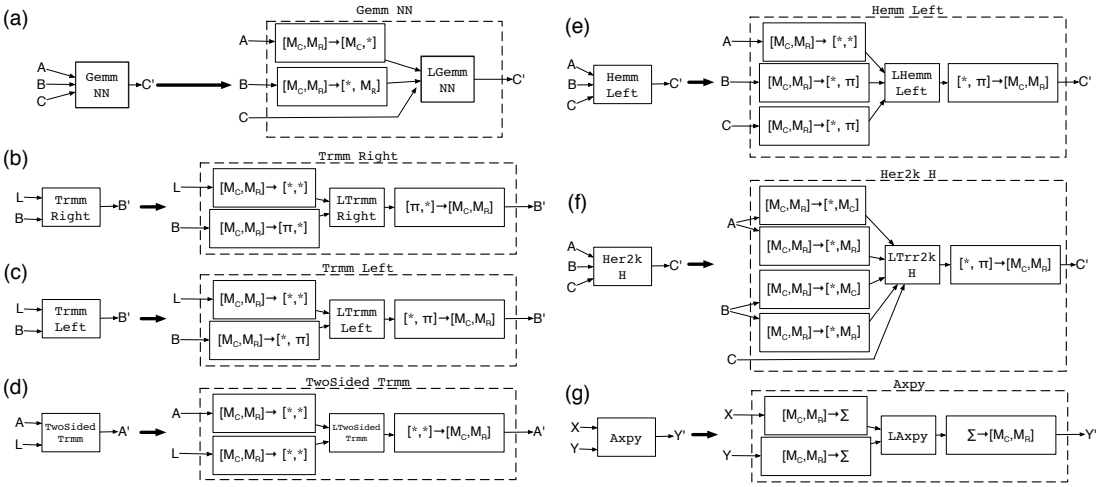


Figure 4: Refinements used for `two-sided Trmm`. $\pi \in \{*, M_C, M_R, V_C, V_R\}$ and Σ is any Elemental distribution. `LTrr2k` is a local computation kernel in Elemental that implements `Her2k` efficiently using additional distributions of the inputs.

First, we focus on the version of `Gemm` with operands A and B Normal ($C := C + AB$, where each operand is a general matrix) as opposed to (Hermitian) transposed or Transposed. We call this operation `Gemm NN`, and it shows up in the algorithm as $A_{20} := A_{20} + A_{21} L_{10}$. The three classes of parallelization schemes keep one of the A, B, or C input matrices stationary, meaning that the matrix is not communicated, avoiding costly redistribution from $[M_C, M_R]$. The best choice generally keeps the largest matrix (in terms of data) stationary.

In this case, input A_{20} is the largest. To parallelize `Gemm NN` with a stationary A_{20}, we redistribute L_{10} (to $[*, M_R]$) and A_{21} (to $[M_C, *]$), after which a local `Gemm` can be performed in parallel on all processes, calculating with disjoint portions of A_{20}. Figure 4 (a) shows this refinement.

Redistribution operations are represented with a node labeled "A $\rightarrow$ B" where A is the starting distribution and B is the resulting distribution. This corresponds to Elemental's C++ "=" operator. `LGemm NN` is a Local call to sequential `Gemm NN`, found in the BLAS library. Refinements propagate coefficient and parameter information from interfaces to refinement nodes, but we do not show that information here because there is no novelty in doing so and would clutter our graphical representation.

For `Axpy` ($Y := \alpha X + Y$, where α is a scalar and X and Y are matrices), the two inputs can be in any identical distribution (the same holds for both) to enable local computation, as shown in the template of Figure 4 (g). Some choices (e.g. $\Sigma = [*, V_R]$) would result in efficiently distributed work performed in parallel. Others result in some redundancy of computation (e.g. $\Sigma = [M_C, *]$) or complete redundancy (e.g. $\Sigma = [*, *]$). For this algorithm, both `Axpy` operations are best implemented with a $[*, V_R]$ distribution, but in fact any distribution with little communication could be used since this operations contributes very little to the overall cost, as an expert would know.

In our previous study of `two-sided Trsm` [13], we only included a few `Axpy` refinements, which was sufficient for that algorithm. For `two-sided Trmm`, we needed additional refinements to produce the most efficient code due to the different combination of update statements. The template of Figure 4 (g) allowed for many options; we instantiated two in [13]. Now we needed to instantiate it completely, for all distributions in Figure 3 except $[*, *]$ (seven more for a total of nine). As `LAxpy` computes with the locally-stored portions of the input data, this enables seven new ways to compute `Axpy` on the process grid.

The subproblem $A_{11} := L_{11}^H A_{11} L_{11}$ is itself an instance of `two-sided Trmm`. It works on smaller inputs, so it is implemented differently than the algorithm in Figure 1. Here, both operands can be redistributed such that they are stored redundantly ($[*, *]$) and local computation (`LTwoSidedTrmm`) can be performed in parallel, redundantly on all processes. This is shown in Figure 4 (d). One cannot use this refinement to implement the operation on the full A and L input matrices because a massive amount of memory would be required and no parallelism would be achieved. For the suboperation, though, this is the best refinement. *The seven new `Axpy` refinements and this `two-sided Trmm` refinement were the only new transformations added to DxTer for this work.* We return to this point later.

3.3 Optimizations

Each refinement in Figure 4 replaces an interface with a subgraph, meaning that applying a series of refinements can ultimately produce a graph containing only primitives, which maps directly to functional Elemental code. With refinements, `two-sided Trmm` would be parallelized, but it would not perform particularly well because of unnecessarily high communication overhead.

Some redistribution operations are implemented via a series of intermediate redistributions, which the Elemental C++ operation "=" hides. Further, redistributions can be implemented in alternate, suboptimal ways that may expose opportunities for subsequent optimizations. We use optimizations to replace a redistribution box to expose intermediate redistributions (when they are used) and alternate implementations. Experts explore these options so they can reduce the overall overhead due to communication. Inverse (e.g., a gather followed by a scatter, which is together equivalent to a no-op) and redundant redistributions are commonly found after parallelizing algorithm update statements, so optimizing transformations encode expert knowledge to remove such inefficiencies.

We now explore a typical example of how an optimization improves performance. After applying the best refinements (Figure 4), the result of the first `Axpy` in the algorithm is redistributed from $[*, V_R]$ to $[M_C, M_R]$ (with an `AllToAll` within process columns) and then redistributed from $[M_C, M_R]$ to $[*, M_R]$ (with an `AllGather` within process columns) for the `Her2k` operation. The top of Figure 5 (c)

shows the graph representing this.

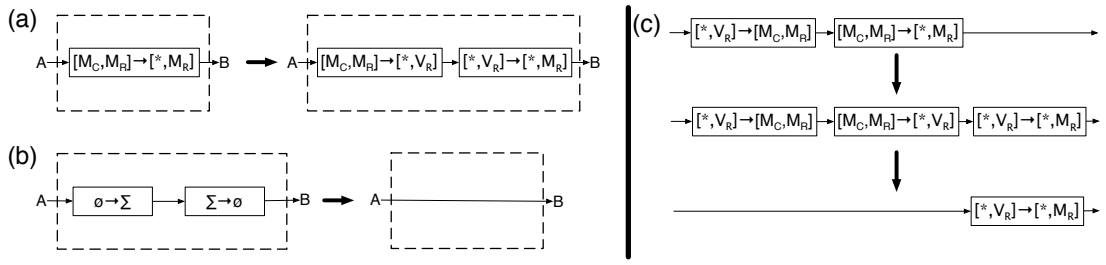


Figure 5: Optimizations used in `two-sided Trmm`. ϕ and Σ can be any Elemental distribution.

An alternate implementation of $[M_C, M_R] \rightarrow [*, M_R]$ uses an intermediate distribution: $[M_C, M_R] \rightarrow [*, V_R] \rightarrow [*, M_R]$. This is an optimization of the operation that internally uses `AllToAll` followed by `AllGather` collectives. This implementation performs worse than the default, but applying it exposes a series of redistributions: $[*, V_R] \rightarrow [M_C, M_R] \rightarrow [*, V_R]$, shown in the middle Figure 5 (c). These are inverse redistributions, so we can apply an optimization to remove the inverse operations and end with the less costly redistribution implementation $[*, V_R] \rightarrow [*, M_R]$ (an `AllGather` within process columns). The first transformations increase runtime, but the second transformation removes a redistribution that is implemented with an expensive `AllToAll`, yielding a net gain.[6]

Figure 5 (a) and (b) show the optimizations to explore an alternate implementation and remove an inverse operation, respectively. The templatized optimization is used often in DxTer. For another example application, the output of the `Hemm` operation $Y_{10} := A_{11}L_{10}$ is distributed as $[*, V_R]$ and is redistributed to $[M_C, M_R]$. The $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ (`Axpy`) refinement then redistributes it back to $[*, V_R]$ (see `Hemm` and `Axpy` refinements in Figure 4 (e) and (g) with $\pi = V_R$ and $\Sigma = [*, V_R]$). The optimization of Figure 5 (b) removes the unnecessary redistribution before `LAxpy`.

DxTer has many optimizations that are all roughly as complicated as these examples. They encode deep expert knowledge about redistributions, so we do not show many. The key is that they encode this knowledge in a reusable and simple way (as the discussion shows).

3.4 Transpose Optimizations

One of the benefits of DxT is that new optimizations and refinements can be added and automatically applied to all algorithms. The rules in Sections 3.2 and 3.3 and others like them are sufficient to generate code for the algorithms of Figure 1. Further, this code performs well on many clusters. On some, though, these rules are insufficient. When the expert developer of Elemental tested his code on an IBM BlueGene/P supercomputer, he discovered that further optimizations were needed to improve memory access. He reviewed all existing code to apply these optimizations repeatedly. While these optimizations are especially helpful on BlueGene/P, they are also beneficial on other architectures, so this optimization is used for all targets. With DxTer, new optimizations can be added to the knowledge base and automatically applied, relieving the expert's burden. Comparing DxTer's generated code to the expert's manually-created code, we found many instances where the expert missed optimizations for the algorithms in this paper and others [12, 13].

When MPI collectives are called behind the C++ "=" operation, data is packed into and unpacked from send and receive buffers. The memory access pattern during these data permutations is some-

[6] One might ask why we use transformations to optimize the communication pattern in this example from two "=" operators to three to one. With only two redistributions, this example is a simple case. More often, there are many redistributions to optimize in concert and these transformations and others like them enable that.

times more favorable if the matrices being communicated are transposed relative to the original matrix. This results in more unit-stride memory access while packing and unpacking data [12]. To enable this, Elemental redistributions were added to transpose data during communication. Many local DLA computation functions support transposed inputs, so data transposed during redistribution is un-transposed during computation, allowing the expert, or DxT, to choose whether to leave the data in transposed form.

This is a DxT optimization that requires deep knowledge of Elemental operations as only some redistributions can be transposed. Further, knowledge of computation functions is needed to identify which inputs can be transposed to undo redistribution transposition. Figure 6 shows a transpose optimization on one input to `LGemm`. This case shows up in the `DGemm` refinement of Figure 4 (a) and many other cases. Here, the `B` input to the `LGemm` box comes from a redistribution that can be transposed. This optimization transposes the redistribution box and changes the `LGemm` box to undo that transposition (i.e., `NN` to `NT`).

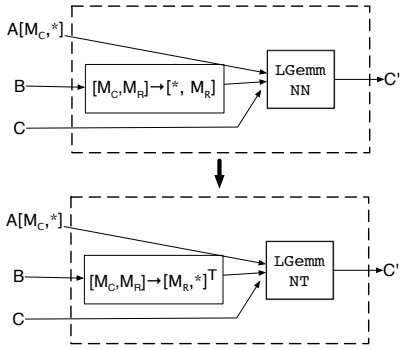


Figure 6: Optimization to transpose communication.

Many computation boxes have transformations similar to this, so the optimization is encoded in DxTer in a versatile way. Knowledge is encoded on which redistributions and computation inputs can be transposed. DxTer queries this information to explore transposition. Optimizations like Figure 5 (a) expose many opportunities for transpositions by exposing redistributions that can be transposed.

4 Limiting the Search Space

Every domain has a structure that experts intuitively understand and exploit. Expert insights, known as tactics or heuristics, direct the search process. It is too early for us to identify tactics that can be shared by different domains. We have, however, discovered tactics for the DLA domain that enable DxTer to find solutions. We now review those that we now use and have invented—Cost Prioritization, Merged Transformations, and Simplifiers—to address `two-sided Trmm` (though they are general purpose).

DxTer is a code generator that takes as input three forms of knowledge: 1) a set of transformations, 2) cost estimates for primitives, and 3) a DAG of the algorithm to be implemented. Section 4.1 describes a way we changed how DxTer applies a subset of the transformations. The heuristics described in Sections 4.2 and 4.3 change the transformations input to DxTer. We are studying how to apply such a heuristic automatically and more generally for other refinements, limiting DxTer's search time to the most promising options.

4.1 Cost Prioritization

DxTer previously used brute force to generate the space of implementations. It applied all refinements and optimizations it could to the input graph, analyzed the cost of each generated implementation, and selected the "best." While all generated graphs implemented the input algorithm, few were efficient. To apply DxTer to `two-sided Trmm`, we mentioned earlier that we added seven new refinements to the knowledge base. `Axpy` now has nine refinements instead of two, so there are $\frac{9}{2} = 4.5$-times as many implementation options. Note that the `Trmm` algorithm of Figure 1 has two `Axpy` boxes; this means that there will be roughly $4.5 \times 4.5 \approx 20$-times as many options to explore. The actual number of additional implementations depends on what optimizations are enabled by the new refinements, but it would take about 20-times as long to generate and analyze all possibilities.

We knew DxTer took 20 minutes to generate about 200,000 implementations for `two-sided Trmm` with two `Axpy` refinements. With nine refinements, we expected $20 * 20 = 400$ minutes or 6.7 hours of runtime. DxTer abruptly stopped after several hours (producing more than four million implementations) once it ran out of free memory. While brute force had been sufficient previously, it is now necessary to limit the refinements to consider in an intelligent way to make DxTer computations finish in a reasonable time.

An expert recognizes that `Axpy` is a $O(n^2)$ cost operation, so he/she does not worry about perfect parallelization as much as minimizing the communication to enable it, which is more expensive. As such, we can limit refinement choices to those that require little additional redistribution (as this is the dominant cost). For `Axpy`, we use a heuristic to explore only refinements that re-use input or output distributions that are already available from refinements in surrounding code. That is, DxTer only explores refinements of `Axpy` that are best in the operation's local context. This makes the search space tractable and only removes implementation options that are almost always suboptimal since they generally require more communication. Further, if a removed option is best, the performance penalty is relatively small because `Axpy` is only a $O(n^2)$ cost operation found in algorithms that are $O(n^3)$ cost. We expect this heuristic to work in any algorithm for which `Axpy` is a relatively small cost. In a library of level-3 BLAS operations, this is commonly the case.

Inherent in using such a heuristic is a need to order transformations. All non-`Axpy` operations must be refined first, and only then is `Axpy` refined so DxTer can "consider" the surrounding implementation details. We already use two phases in DxTer, one to refine and the next to optimize, and DxTer removes graphs between the first and second phase that contain non-primitive boxes, which significantly speeds up the search without removing any viable implementations [11]. This ordering comes naturally: parallelize (refine) first and then optimize [14]. In future work, as we add more heuristics, we will study transformation orderings, but for now it is much simpler than that found in compilers [1, 9].

4.2 Merged Transformations

Some optimizations that explore alternate redistributions are only useful when combined with other optimizations. Only applying the first optimization results in a worse implementation until the second optimization is applied.

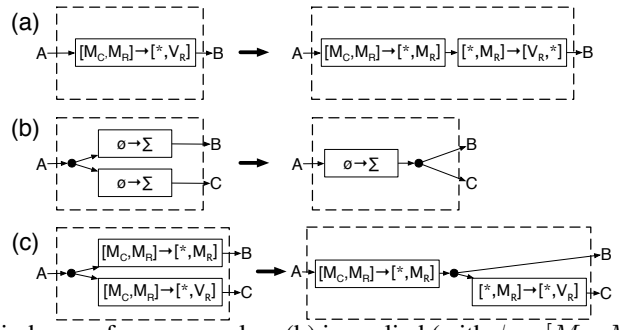


Figure 7: (a) always hinders performance unless (b) is applied (with $\phi = [M_C, M_R]$ and $\Sigma = [*, M_R]$). (c) merges the two optimizations to remove from the search space implementations that are always bad.

The optimizations in Figure 7 (a)-(b) are examples of transformations that should only be applied together. The transformation of Figure 7 (a) always hinders performance unless the input "A" is additionally redistributed to $[*, M_R]$ in the surrounding graph and the transformation of Figure 7 (b) can remove the extraneous $[M_C, M_R] \rightarrow [*, M_R]$ redistribution. The result of both transformations together generally improves performance and they are necessary to get the best implementation of `two-sided Trmm`. Allowing DxTer to explore the first transformation without the second is wasteful since all imple-

mentations that result will be suboptimal. Other optimizations to explore redistribution implementations are of this type. They are only worth applying to find combinations of intermediate redistributions that can be optimized away by a second transformation.

In listing transformation rules, it is beneficial to keep both optimizations separate. Each is an understandable piece of domain knowledge. When an expert applies transformations, an expert recognizes that the first transformation is only useful when the second is also applied. Therefore, we encode such combinations in a single merged transformation. By using one transformation instead of two, the search space is significantly reduced because suboptimal implementations are pruned. Figure 7 (c) shows the merged form of the other two transformations, which is used for `two-sided Trmm`. The optimization of Figure 7 (b) is still included in DxTer by itself (with ϕ and Σ being any Elemental distribution) since it improves performance even when not combined with Figure 7 (a) or similar transformations.

The utility of merged transformations is not a function of the specific algorithm (e.g. `two-sided Trmm`). We only merge transformations in this way when it is always worth doing. Here, for example, it is never beneficial to have the optimization of Figure 7 (a) in isolation, no matter the algorithm, because it increases cost in all contexts.

Figure 5 (a)-(b) is not an example of transformations that can be merged because the first optimization enables many other transformations. We could create a merged transformation for each pair, but this would require more effort to study all transformations. It is future work to determine how to pre-process automatically a set of transformations to optimize them for search.

4.3 Simplifiers

Finally, some optimizations are always worth applying. For example, optimizations that remove redundant (Figure 7 (b)) or inverse (Figure 5 (b)) redistributions always reduce implementation cost. An expert recognizes, for example, that there is never a benefit to having duplicate communication (i.e. it hinders performance and never enables a useful follow-up transformation). Therefore, (s)he always removes redundant communication. We call such optimizations simplifiers because DxTer always applies them [11] and they reduce the search space as they prune implementations that are always suboptimal. Similar to merged transformations, simplifiers are always beneficial, no matter the algorithm.

5 Experimental Results

5.1 DxTer Search Space and Search Time

We tested DxTer on a quad-core Intel Core i7 processor with hyper-threading running 8 threads. For `two-sided Trmm`, DxTer generated a total of 302,010 implementations. DxTer took one second to find 64 ways to parallelize the algorithm and then it took 45 minutes to explore different ways to optimize redistributions. It took 20 seconds to estimate and compare each implementation's cost and choose a "best." Without the heuristics described in Section 4, DxTer would have run out of memory after hours of graph generation.

To illustrate how much more complex `two-sided Trmm` is, consider that for `two-sided Trsm` DxTer generated 90,090 implementations (including the new inventions and `Axpy` refinements described above). DxTer took one second to find 72 parallelized implementations and 8 minutes to optimize redistributions. Cost analysis and comparison took 6 seconds.

The loop-body generated for each operation is roughly 20 lines of code. This code specifies how to parallelize the computation and how to redistribute data. DxT is successful mainly because the programs are small and it is useful because those few lines are hard to write. Using "core" or "fundamental" interfaces enables a developer to be productive, focusing on the important issues and not on implementation minutia. They also limit the size of graphs that are being generated and, therefore, the size of the

search space. Elemental satisfies this interface requirement using sequential BLAS or LAPACK-level computation operations and redistribution operations (via the C++ "=" operator).

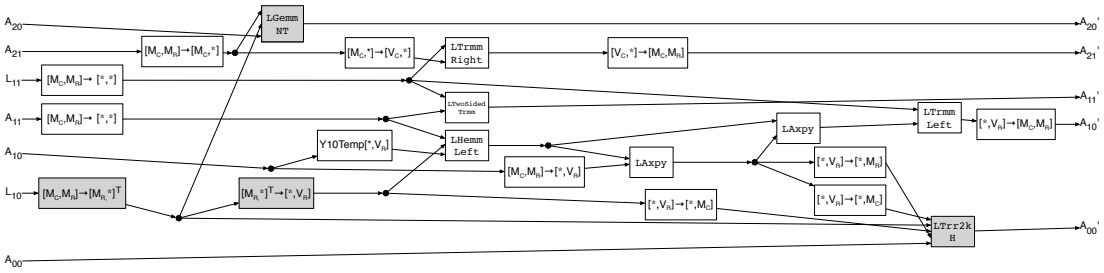


Figure 8: Final implementation graph for `two-sided Trmm`.

5.2 Generated Implementation

The implementation DxTer generated for each operation was slightly better than the version in Elemental, developed by an expert (Jack Poulson). DxTer refined all operations just as the expert had, but DxTer improved the communication slightly using transposition optimizations the expert missed. These improvements have since been incorporated into Elemental. In Figure 8, we show the final implementation graph for `two-sided Trmm`. The grayed boxes are those that were affected by the transposed optimizations that the expert missed. Even with the additional refinements of `Axpy` and some new optimizations that were added to DxT since the results of [13], the same "best" code was chosen for `two-sided Trsm` as before, but DxTer explored the additional options. We believe it is an important success for the DxT approach that it can generate better implementations of two complicated algorithms than the expert developer of Elemental.

In Figure 9, we compare the performance of DxTer-generated code to that of ScaLAPACK code. The unoptimized DxTer code is the result of only applying refinements (i.e. the loop body operations are parallelized). The optimized DxTer code is the result of DxTer applying refinements and optimizations, demonstrating the need to both implement and optimize algorithms.

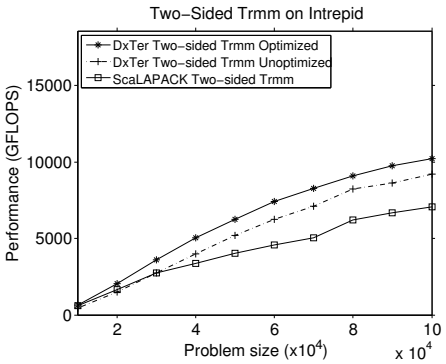


Figure 9: Performance of DxTer-generated and ScaLAPACK implementations. The top of the graph is two-thirds of peak performance.

We tested on the BlueGene/P cluster Intrepid at Argonne National Lab. We ran on 8192 cores (2 racks), which have a combined theoretical peak of over 27 TFLOPS. The top of the graphs is two-thirds of peak performance. Double precision arithmetic was used in all computations. For each problem, we tested a range of algorithmic block sizes and a set of process grid configurations and show the best results. The DxTer-generated optimized code performs much better than the ScaLAPACK code and somewhat better than the Elemental code developed by hand.[7] For the largest problem size, DxTer-generated code is about 45% better than ScaLAPACK code.

[7] DxTer performance is slightly better than Elemental, but the difference is not noticeable on our graph.

6 Related Work

As said earlier, DxT is a general approach to software generation.[8] Here, we describe the work most closely related to DxT applied to DLA. For work related to software engineering, see [19].

Autotuning is an important way to improve performance of code automatically [26]. DxT is different and complementary in that it generates a space of semantically equivalent implementations from a high-level understanding of how algorithms can be derived. Also, we use cost estimates that guide DxTer to the best implementation(s) instead of performance testing. We envision a comprehensive process that includes a DxTer-like tool to generate code followed by an autotuning step to then choose the best parameters like, for example, the algorithmic block size and process grid configuration.

DxT is similar to SPIRAL [17]. SPIRAL largely focuses on generating high-performance Digital Signal Processing (DSP) kernels for target architectures. It starts with a mathematical description of the algorithm in a DSL and performs transformations similar to refinements and optimizations to recursively replace abstract operations with specific implementation code and to improve that code. It uses machine learning via online code compilation and performance testing to explore the space of implementations. DxT targets higher-level operations, built on lower-level primitives like the BLAS, so we can utilize relatively accurate cost models instead of empirically-based search. We envision in the future relying on SPIRAL-generated primitives instead of the hand-developed and hand-tuned implementations we use today.

The Tensor Contraction Engine (TCE) [3] aims to generate code for a tensor contraction expressed in a high-level DSL. It applies (mostly loop) transformations to reduce computational complexity, space complexity, communication cost, and then data access cost. These transformations and cost models are similar to those of DxT. TCE specifically targets tensor contractions; DxT is more general-purpose.

The Broadway compiler [8] had a similar goal as ours to encode expert knowledge to generate optimized code. Library functions were annotated, so Broadway could choose the best implementation of an interface at a call site. It was not able to optimize as DxT does, though, which prevented it from generating the "best" code. Further, it did not use a search space of implementations, which is necessary to avoid local minima.

The FLAME project is closely related to DxT. In [7], "The Big Picture" expressed the idea of encoding algorithms and expert knowledge to mechanically generate code. There, optimized parallel code was also the goal, but the PLAPACK library [23] was the targeted DSL instead of Elemental. Many implementations were generated and performance estimates were created from cost function annotations in the algorithms. Our current work benefits from extra years of insights and experience, which enable a more sophisticated approach based on graph transformations (which is more general-purpose than that DLA-specific work).

7 Conclusion

Domain knowledge of DLA implementations on clusters can be encoded as transformations. We demonstrated how transformations from previous work were reapplied automatically to generate code for a new operation, `two-sided Trmm`. Further, we detailed additional transformations (units of DLA knowledge) that were needed to optimize memory accesses by transposing data redistribution, encoding deep domain knowledge. The result of this study was automatically-generated code that is better than an expert-developed implementation because the expert missed opportunities for optimization.

The key to DxT is exposing the structure of the domain. We define fundamental semantic identities that are used in DLA as refinement rewrites (how interfaces are implemented) and as optimizations (how compositions of operations are equivalent to other compositions). We are currently studying these

[8] This section is a slight modification of a similar comparison of DxT for DLA to other works given in [13].

details for other layers of the DLA software stack, including the internal structure of sequential BLAS and LAPACK-level operations. Optimizing code through many layers of the stack will result in even better implementations at a cost of much larger search spaces.

Applying DxT to `two-sided Trmm` and `two-sided Trsm` allowed us to study how well transformations are re-used (highlighting how rote an expert's job is), which is important when studying other operations across layers of the stack. In this work, we added variations of the `Axpy` operation simply by instantiating its template over new parameters, but this greatly enlarged the search space. We developed a heuristic to limit exploration to the likely best implementations. This is an early study in how to deal with large search spaces. The `Axpy` heuristic, merged transformations, and simplifiers reduced the search space from being impossibly large to tractable. We continue to study ways to battle the combinatorial space in DxTer while still generating high-performance code.

Acknowledgments. Marker held fellowships from Sandia National Laboratories and NSF (grant DGE-1110007). This work was also partially sponsored by NSF grants CCF-0917167 and OCI-1148125 and used resources of the Argonne Leadership Computing Facility at Argonne National Lab, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We are greatly indebted to Jack Poulson for his help to understand his Elemental library. *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] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. In *In Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, 2004.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du 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.

[3] A. Auer, G. Baumgartner, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, S. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Automatic code generation for many-body electronic structure methods: The Tensor Contraction Engine. *Molecular Physics*, 2005.

[4] J. Choi, J. Dongarra, R. Pozo, and D. 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.

[5] 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), March 1990.

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

[7] J. 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.

[8] S. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE, Special issue on program generation, optimization and adaptation*, January-February 2005.

[9] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.*, 6(1):1:1–1:36, April 2009.

[10] 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, September 1979.

[11] B. Marker, D. Batory, and C.T. Shepherd. Dxter: A dense linear algebra program synthesizer. Computer Science report TR-12-17, Univ. of Texas at Austin, 2012.

[12] B. Marker, D. Batory, and R. van de Geijn. Code generation and optimization of distributed-memory dense linear algebra kernels. In *International Workshop on Automatic Performance Tuning (iWAPT)*, 2013. To Appear.

[13] B. Marker, J. Poulson, D. Batory, and R. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *International Workshop on Automatic Performance Tuning (iWAPT)*, 2012.

[14] J.M. Neighbors. Draco: a method for engineering reusable software systems. In T.J. Biggerstaff and A.J. Perlis, editors, *Software reusability: vol. 1, concepts and models*. ACM, 1989.

[15] J. Poulson, B. Marker, J. Hammond, N. Romero, and Robert van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Soft.*, 2010. accepted.

[16] J. Poulson, R. van de Geijn, and J. Bennighof. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. FLAME Working Note #56. Technical Report TR-11-05, The University of Texas at Austin, Department of Computer Sciences, February 2011.

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

[18] R. Riche, R. Goncalves, B. Marker, and D. Batory. Pushouts in Software Architecture Design. In *Generative Programming and Component Engineering (GPCE)*, 2012.

[19] T. Riché, D. Batory, R. Goncalves, and B. Marker. Software Architecture Design by Transformation. Technical Report TR-11-19, University of Texas at Austin, Dept. of CS, April 2011.

[20] M. Schatz, J. Poulson, and R. van de Geijn. Scalable universal matrix multiplication algorithms: 2D and 3D variations on a theme. *ACM Trans. Math. Soft.*, 2012. submitted.

[21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[22] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

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

[24] R. van de Geijn and E. Quintana-Ortí. *The Science of Programming Matrix Computations*. www.lulu.com, 2008.

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

[26] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 1–27, 1998.