Code Generation of Optimized Distributed-Memory Dense Linear Algebra Kernels

Bryan Marker Don Batory
Robert van de Geijn
The University of Texas at Austin
{bamarker,batory,rvdg}@cs.utexas.edu

November 30, 2012

Abstract

Design by Transformation (DxT) is an approach to software development that encodes domain-specific programs as graphs and expert design knowledge as graph transformations. The goal of DxT is to mechanize the generation of highly optimized code. This paper demonstrates how DxT can be used to transform sequential specifications of an important set of *Dense Linear Algebra* (DLA) kernels, the *level-3 Basic Linear Algebra Subprograms* (BLAS3), into high-performing library routines targeting distributed-memory architectures. Getting good BLAS3 performance for such platforms requires deep domain knowledge, so their implementations are left to experts. The problem is that there are few experts and developing the full variety of BLAS3 implementations takes a lot of repetitive effort. A prototype tool, DxTer, automates this tedious task. Performance results on a BlueGene/P massively parallel supercomputer show that the generated code meets or beats implementations that are hand-coded by a human expert and outperforms the widely used ScaLAPACK library. Since the BLAS3 are representative of a much broader class of DLA operations, our study becomes a powerful example of the potential of DxT for this and other domains.

1 Introduction

Computers perform tasks that are systematic, that can be encoded, and are time consuming for a human to perform. For this reason, optimizing compilers are employed for the laborious task of customizing software for a specific architecture. Still, human experts often hand-optimize large software packages including widely-used libraries, often at a higher level of abstraction than is targeted by traditional optimizing compilers.

BLAS3	# of Variants	# Optimizations generated per variant	Compared to hand optimization
`Gemm`	12	378	Added transpose
`Hemm`	8	16,884	Same
`Her2k`	4	552,415	Same
`Herk`	4	1,252	Same
`Symm`	8	16,880	Same
`Syr2k`	4	295,894	Same
`Syrk`	4	1,290	Same
`Trmm`	16	3,352	Better algorithms
`Trsm`	16	1,012	Added transpose; new implementations

Figure 1: DxTer code generation statistics for the BLAS3s.

Design by Transformation (DxT) is an approach to software development that encodes domain-specific programs as graphs and expert design knowledge as graph transformations. Doing so enables experts to focus on discovering and encoding algorithms and domain knowledge, and deferring to a tool, DxTer, the laborious task of applying this knowledge to synthesize efficient code. This paper presents the application of DxT and DxTer to an important set of functionality for *Dense Linear Algebra* (DLA), the *level-3 Basic Linear Algebra Subprograms* (BLAS3).

Many scientific computing libraries and applications cast computation in terms of high-performing DLA interfaces such as the BLAS [5], LAPACK [1], and `libflame` [19]. By providing efficient implementations for their interfaces, portable high performance can be achieved as an application is moved to new hardware architectures. DLA libraries implementing these interfaces must provide users many operations and many variants of each operation that leads to a large code base to develop and maintain.

To provide high-performance implementations of all variants, an engineer must be or become a domain expert. (S)he must consider many algorithms for each operation variant and many implementations of each algorithm (e.g. different parallelization schemes), so (s)he must have the knowledge to explore the options. There are very few experts that have such knowledge. Without it, correct high-performance code cannot be written. Further, the application of this knowledge is both difficult and tedious.

Applications invoke BLAS3 functions. By linking to a library that implements the BLAS3 in sequential or shared-memory parallel code, the application yields efficient performance on those hardware architectures. When applications compute with very large dense matrices, *distributed memory architectures* (clusters) are employed, requiring DLA libraries to be ported to distributed memory architectures. Ideally, such porting should be automatic. Unfortunately, such parallel libraries have been painstakingly hand-crafted by experts.

A large portion of the knowledge needed to implement DLA libraries resides

in the BLAS3 operations, listed in Figure 1. We explore how DxT [8, 15] encodes expert knowledge of the BLAS3 as program transformations. These transformations are used to generate automatically highly-efficient distributed-memory BLAS3 implementations. We target code for Elemental [13, 12], a DLA library for clusters, and use Elemental specifications as our *Domain-Specific Language* (DSL).

We explain how DxTer automatically generates a large number of BLAS3 variants (shown in the second column of Figure 1) for distributed memory using a small knowledge base. While BLAS3 algorithms, by themselves, do not offer many opportunities for optimization, we show that automatic generation is useful to alleviate the tedious work of an expert and even to find better implementations than those created by an expert. Further, knowledge of BLAS3 operations is essential to build more complex DLA algorithms. While the approach described in this paper is a starting point, the true power of DxT is apparent when the BLAS3 transformations of this paper are applied to more complicated algorithms built from combinations of BLAS3 operations. Some examples were explored in [8, 10], and more will be explored in the future. With complex combinations of BLAS3 operations, expert knowledge becomes even more important as there are more opportunities for optimization. This is when automated code generation shines. We show how DxT shows promise to achieve such automation. We begin with a brief overview of DxT.

2 Design by Transformation

Abstractions, Refinements, and Optimizations We use *directed acyclic graphs (DAGs)* to encode DLA algorithms [17]. 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.

We start with a simple DAG that encodes a sequence of one or more BLAS3 operations.[1] For our application, these are the statements in the body of a loop. There are no implementation details for these operations other than preconditions and postconditions.[2] Nodes without implementation details are called *abstractions*.

A *refinement* is a transformation that replaces an abstraction with a subgraph. This subgraph exposes details of a specific algorithm that implements the abstraction (e.g. for distributed memory), maintaining the abstraction's preconditions and postconditions. These subgraphs contain nodes that are lower-level abstractions or calls to *primitive* functions whose implementations are given to us. The process of refinement continues with the newly revealed abstractions until no abstractions remain (i.e. all boxes are primitives).

At this point, experts transition to another mode of programming with the

[1] Generally, our starting DAG encodes a sequence of one or more DLA operations, but here we focus only on BLAS3 operations.

[2] We do not present formal preconditions and postconditions in this paper as informal descriptions are sufficient for our purposes.

goal of program optimization. In effect, experts optimize a DAG by repeatly replacing subgraphs with other subgraphs that implement the same functionality in a different, usually more efficient, way. Such transformations are called *optimizations*. Preconditions and postconditions of the replaced subgraph are preserved, as required for correctness. Each rewrite does not guarantee improved performance, but the result of multiple optimizations is a better-performing algorithm.

Performance Estimation After applying a sequence of refinements and optimizations, we produce a graph that references only primitives. This graph expresses a concrete algorithm. Since many such algorithms result from different choices of transformations, the question of which performs best needs to be answered.

Here again we exploit knowledge of the target domain and again mimic the activities of domain experts. A domain expert uses a rough idea of cost to estimate the benefit of using a refinement or optimization during algorithm design. In DLA, a cost function is used to estimate performance (or time-to-completion). Algorithms for DLA on distributed memory are often bulk-synchronous, making cost estimation a matter of adding the costs of the primitives, which implement collective communication or computation. Cost functions for these primitive that are accurate enough to rank-order implementations are well-understood [4].

How DxT Works It was observed in [7] that all BLAS3 can be implemented as a loop around calls to `Gemm` and other BLAS3 operations with smaller submatrices. For each BLAS3 operations, there are many such loop-based algorithms. We start with one such algorithm, and focus on optimizing the sequence of operations that appear in the loop body. This sequence can be encoded as a DAG that is architecture-independent, as all operations are abstractions. We then use refinements and optimizations to map this DAG to calls of primitive sequential BLAS3 operations and primitive data redistributions that are used to express parallel DLA algorithms in code.

There are a combinatorial number of ways refinements and optimizations can be applied to an initial DAG. Currently, all possibilities are exhaustively examined to generate a space of semantically equivalent algorithms. Cost functions are then used to rank the implementations of this space. The implementation that is top-ranked (meaning most efficient) is the output of our process. It is possible that several implementations rise to the top, each "best" for a different range of problem sizes. The selected algorithm is then synthesized by translating its graph into Elemental code, which is relatively straightforward. The tool that accomplishes this all is called *DxTer* [9] (pronounced "dexter").

Why Does DxT Work? Readers will see that DLA transformations are simple, suggesting there might be additional, hidden magic to get the results that we present. (There isn't.) Readers may also ask why does DxT work and

why have similar approaches failed (if they indeed have failed)? We have four answers:
Experience. We are leveraging highly-polished layers of abstraction in the Elemental library. This library, the refinements and abstractions that it encodes, reflects decades of work in DLA software development. Even so, the job of an Elemental expert is not simple. There is a combinatorial number of refinements and optimizations from which to choose, and knowing which to use, as we have said, requires considerable experience.
Prior Success. Spiral [14] is a mature tool that automatically generates algorithms for *digital signal processing (DSP)* operations. It takes a slightly different approach, using empirical timing to differentiate between the performance of implementation choices. Similar to DxT with DLA, Spiral has shown success because the software in the domain of DSP is cleanly layered. There are a finite number of primitives and a finite number of implementations that need to be encoded for each. Further, Spiral's rules are roughly as complex as ours.[3]
Bounded Scope. DxT does not try to do everything. Our scope is limited to well-understood software layers within DLA. This stops the generation/search at the given primitives, which are natural to DLA experts; algorithms that implement primitives require different expertise, which – for now – we do not intend to generate. Effectively, our work stops where Spiral begins. This scoping makes automated generation of high-performance DLA code practical, with approximately 15 transformations being a rough upper bound on the number of steps it takes to generate an efficient implementation. Merging layers together would open-up astronomical search spaces that would likely be unconquerable.

Methodology DxT transformations are acquired from a variety of sources. Most refinements can be found in technical papers, but low-level optimizations are found only by reverse engineering source code written by experts. From our experience, 45% of the encoded rules for DLA are refinements. The remaining 55% are optimizations, which are templatized to represent many more transformations.

3 Parallelizing for Elemental

We now review basics about the Elemental library and explain how an Elemental expert manually developed an algorithm for a BLAS3 operation optimized for distributed-memory. We document the steps that an expert took in terms of transformations. While the expert did not necessarily view his/her task with transformations in mind, the resulting code can be forward-engineered by transformations. Further, these transformations are reusable, understandable, and independent pieces of DLA knowledge.

[3]DxT is compared to other automatic code generation approaches, including telescoping compilers, ATLAS, Broadway, and TCE in [8].

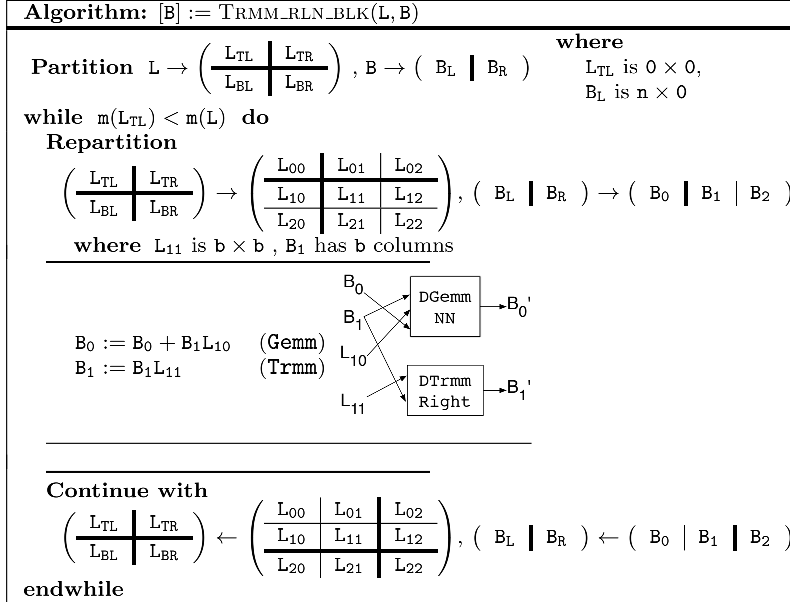

Figure 2: Variant of Trmm: right, lower, non-transposed. The graph representation of the loop body (the DxT encoding) is shown.

A prototypical BLAS3 algorithm Figure 2 shows a prototypical BLAS3 algorithm in FLAME notation [18], which, given a lower triangular matrix L and general matrix B, overwrites B with BL. This is known as a triangular matrix-matrix multiply (Trmm). What it shows is that this operation can be implemented as a loop around operations with submatrices, which we call *update statements*. This is a *blocked algorithm* because the loop body operates on blocks (submatrices) as opposed to vectors or scalars. If L is $n \times n$, then L_{11} is $b \times b$ with blocksize $b \ll n$ so that most computation is in the Gemm operation, $B_0 := B_0 + B_1 L_{10}$ (defined in Figure 2).

This is a prototypical example of how all BLAS3 can be implemented by casting most computation in terms of Gemm [7]. The primary concern then is to get maximal parallelism from $B_0 := B_0 + B_1 L_{10}$, while a secondary concern is to parallelize $B_1 := B_1 L_{11}$ (see Figure 2) and to minimize necessary communication.

It is well-known that hiding all parallelism within the separate update statements can introduce redundant communication and/or synchronization. Our goal is for DxT to encode this algorithm, the knowledge to parallelize its abstract update statements, and the knowledge to optimize the resulting algorithm. Further, we want this knowledge to be reusable for other DLA algorithms.

Elemental Basics Elemental is a framework for parallelizing DLA algorithms as well as a library for DLA operations. In Elemental code, the p MPI processes on a distributed-memory system are viewed as a two-dimensional grid, $p = r \times c$.

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

Figure 3: Distributions on a $\mathtt{p}=\mathtt{r}\times\mathtt{c}$ process grid.

For the default distribution, Elemental uses a 2D element-wise cyclic distribution, labeled $[\mathtt{M_C},\mathtt{M_R}]$ where $\mathtt{M_C}$ and $\mathtt{M_R}$ represent partitionings of the index space that provide a filter to determine which row and column indices are assigned to a given process.[4] There are a handful of other one and two-dimensional distributions of matrices, examples listed in Figure 3, that are used to redistribute data so that efficient local computation can be utilized. Elemental is written in C++ and encodes matrices and attributes (including distribution) in objects. In order to parallelize a computation, matrices are redistributed from the default distribution to enable parallel local computation, after which the result is placed back into the original distribution. In Elemental, this is accomplished using the overloaded "=" operation in C++, which hides the (MPI) collective communication required to perform data redistribution efficiently.

Parallelizing `Trmm` We now examine the actions of an Elemental expert to develop an optimized parallel algorithm for `Trmm`. We do so in terms of transformations, first explaining the refinements that parallelize suboperations and then optimizations that are subsequently applied.

`Trmm` could be any of the following set of operations:

$$\mathtt{B}=\mathtt{LB},\mathtt{B}=\mathtt{L}^{\mathtt{T}}\mathtt{B},\mathtt{B}=\mathtt{UB},=\mathtt{U}^{\mathtt{T}}\mathtt{B},\mathtt{B}=\mathtt{BL},\mathtt{B}=\mathtt{BL}^{\mathtt{T}},\mathtt{B}=\mathtt{BU},\mathtt{B}=\mathtt{BU}^{\mathtt{T}},$$

where $\mathtt{L}$ and $\mathtt{U}$ are lower and upper triangular matrices, respectively. Each of these eight possibilities is implemented separately with different algorithms. We focus on $\mathtt{B}=\mathtt{BL}$ for which Figure 2 gives one of several algorithms that an expert considers.

The input matrices, $\mathtt{L}$ and $\mathtt{B}$, have the default $[\mathtt{M_C},\mathtt{M_R}]$ distribution. To par-

[4] We do not further explain the reason for different distribution names because they are out of the scope of this paper. Details are in [13, 16].

allelize the algorithm, the updates `Trmm` and `Gemm` in Figure 2 are parallelized by redistributing submatrices, performing local computation (via calls to sequential BLAS3 routines) on each (MPI) process, and (if necessary) reducing and/or communicating the result.

An expert would need to consider the various ways to parallelize the sub-operations. The three classes of parallelization schemes for the `Gemm` update statement keep the `A`, `B` or `C` matrix *stationary*, avoiding costly redistribution from $[M_C, M_R]$. The best choice generally keeps the largest matrix stationary. In this case, the B_0 matrix (defined in Figure 2) is the largest. To parallelize `Gemm` with a stationary B_0, we must redistribute L_{10} (to $[*, M_R]$) and B_1 (to $[M_C, *]$), after which a local `Gemm` can be performed in parallel on all processes, calculating disjoint portions of B_0.

To parallelize $B_1 := B_1 L_{11}$, an expert understands that if L_{11} is duplicated to all processes (distribution $[*, *]$) and B_1 is redistributed so that any one process owns complete rows of this matrix (e.g., distribution $[V_C, *]$), then $B_1 L_{11}$ can be computed in parallel by locally calling a sequential `Trmm` with local data. But the expert would also consider many other distributions, given in Figure 3, for L_{11} and B_1 before arriving at this particular refinement of the abstract operation. There are many refinements to consider, each of which distributes computation differently, requiring different communication and different local computation, offering a balance between communication (overhead) and parallelism in computation (useful computation). For large problems, one refinement may be best because the cost of communication is amortized over more computation. We focus on large problem sizes here, but an expert would serve the user best by providing a set of optimized implementation variants for a range of problem sizes. We use the refinement with a $[V_C, *]$ distribution of B_1 in subsequent discussions.

Encoding the algorithm with Elemental Elemental variable declarations and loop code are straight-forward and uninteresting, so we do not show it here. The code of the Elemental update statements, once parallelized with the above choices of refinements, are given by: [5]

```
B1_MC_STAR = B1;                                        }
L10_STAR_MC = L10;                                      } B0 := B0 + B1L10
LocalGemm( NORMAL, NORMAL, 1.0, B1_MC_STAR,             }
           L10_STAR_MC, 1.0, B0 );                      }

L11_STAR_STAR = L11;                                    }
B1_VC_STAR = B1;                                        }
LocalTrmm( RIGHT, LOWER, NORMAL, NON_UNIT, 1.0,         } B1 := B1L11
           L11_STAR_STAR, B1_VC_STAR );                 }
B1 = B1_VC_STAR;                                        }
```

This is close to the code found in the Elemental library. The "=" operation in Elemental hides MPI collective communication calls. An expert would consider alternate ways to perform the same communication and would notice an opportunity for optimization in the above code. Data (B_1) is redistributed from

[5] By Elemental convention, variables are named by the submatrix stored, appended with the distribution name for readability.

$[\mathtt{M_C},\mathtt{M_R}]$ to $[\mathtt{M_C},*]$ (denoted $[\mathtt{M_C},\mathtt{M_R}] \to [\mathtt{M_C},*]$) and then $[\mathtt{M_C},\mathtt{M_R}] \to [\mathtt{V_C},*]$. The $[\mathtt{M_C},\mathtt{M_R}] \to [\mathtt{V_C},*]$ redistribution can be implemented with an `AllToAll` or it can be implemented in terms of the two redistributions, $[\mathtt{M_C},\mathtt{M_R}] \to [\mathtt{M_C},*] \to [\mathtt{V_C},*]$, which is an `AllGather` followed by a memory copy. Although this redistribution is not as efficient, it allows an expert to remove the extra redistribution to $[\mathtt{M_C},*]$, which results in the best performance.

An expert explores this option in code by replacing (refining) the line: `B1_VC_STAR = B1;`with `B1_VC_STAR = B1_MC_STAR = B1;` and optimizing the inefficient code by removing one of the redundant redistributions in the two lines `B1_MC_STAR = B1; B1_MC_STAR = B1;` The resulting optimized code, which *is* in the Elemental library, is:

```
B1_MC_STAR = B1;
L10_STAR_MR = L10;
LocalGemm( NORMAL, NORMAL, 1.0, B1_MC_STAR, L10_STAR_MR,
           1.0, B0 );

L11_STAR_STAR = L11;
B1_VC_STAR = B1_MC_STAR;
LocalTrmm( RIGHT, LOWER, NORMAL, NON_UNIT, 1.0,
           L11_STAR_STAR, B1_VC_STAR );
B1 = B1_VC_STAR;
```

The above code is the result of two parallelizing refinements, one refinement to explore an alternate implementation of $[\mathtt{M_C},\mathtt{M_R}] \to [\mathtt{V_C},*]$, and one optimization to remove a redundant redistribution. Each transformation is easy to understand individually, but learning and then manually exploring all of the options and choosing the correct combination is not easy and/or is tedious. It takes considerable knowledge and experience with Elemental to do this well.

4 Encoding Knowledge of the BLAS3

With a basic understanding of DxT and Elemental, we can show prototypical transformations that enable DxTer to generate implementations automatically for all BLAS3 variants in Figure 1. We now describe the primitive operations and transformations used.

Graph and Code Operations High-performance parallel DLA software is coded in terms of loops, sequential DLA function calls, and communication operations. There are other operations, but these are the main operations to be considered in well-layered code thanks to decades of software engineering in this field [1, 19, 13, 3].

General rules attaining high performance are that communication should be minimized and the portion of time spent in high-performing computation kernels should be maximized. On a single (many-core) CPU, communication is data movement between cache layers. With GPUs communication is data movement between devices and the host computer. With clusters, communication is movement between processes.

The important design decisions for Elemental deal with a small number of computation operations. For the parallel BLAS3, high-performance implemen-

tations call sequential BLAS3 kernels for suboperations. Further, Elemental code requires redistribution operations (collective communication) between a finite number of supported distributions. Only knowledge regarding these redistributions needs to be encoded, and much of that, as shown below, is repetitive. These are the primitives in terms of which DxT graphs will ultimately be defined.

The best implementations come down to the right combination of a small number of operations. The transformations to generate those implementations can be very simple. The rest of this section demonstrates this point.

Algorithms to Explore The FLAME project has developed a repeatable process by which loop-based families of algorithms for DLA operations can be systematically derived [6]. Using formal derivation, a person or a mechanical system [2] can derive multiple correct algorithmic variants, expressed similarly to Figure 2, for a target operation. This is useful because there is generally no single algorithm that works best for all architectures, so with a family of algorithms for an operation, the best variant can be chosen.

BLAS3 operations and their FLAME-derived algorithms are mathematical in nature (e.g., Figure 2) and are architecture invariant, so different optimizations and transformations are needed to yield efficient implementations for a specific architecture.

We represent BLAS3 in a graph with nodes named after the operations they represent (e.g., if we are interested in optimizing the `Trmm` operation, we start with a single node labeled `Trmm`). These are purely mathematical abstractions — they have no implementation details. Abstract operations can be combined in a graph with other nodes to compose higher-level functionality, but in this paper we focus just on implementations of the BLAS3 functions in isolation, and hence start with a graph with one node.

For each BLAS3 operation (e.g. `Trmm`), a refinement for each known algorithmic variant is encoded in DxTer. These refinements replace the abstract node with a graph representing the algorithmic loop and loop body operations. For blocked algorithms like in Figure 2, the update statements in the loop are BLAS3 operations themselves, operating on smaller submatrices.

The refinement of node `Trmm` for the algorithm of Figure 2 is a loop with abstract update statements `Trmm` and `Gemm`. The part of the loop that does not include the update statements we call the *loop skeleton*, which can be specified at a very high level of abstraction and is often identical for all variants.

To differentiate between top-level BLAS3 operations that need to be implemented by a loop algorithm and the update statement BLAS3 operations that are implemented differently (described below), the update statements are not abstract BLAS3 nodes (e.g. with the label `Trmm`). Instead, they are architecture-specific, which we call `DTrmm` and `DGemm` (where the `D` stands for *distributed*, not to be confused with the subroutine `DGemm` where the `D` stands for *double precision* [5]). Boxes that start with `D` (`D*` boxes) are BLAS3 operations implemented in distributed-memory parallel code via redistribution, local computation, and

redistribution of the result. When targeting other architectures, the loop body operations are the same, but D may be replaced with, say, GPU flavors of the same operations. In this way, algorithm transformations are reusable across architectures; only the implementation of the suboperations changes, with different architecture-specific refinements. To transform the loop body operations to architecture-specific implementations, there are distributed-memory refinements, described below.

BLAS3 Distributed-Memory Refinements With knowledge of algorithmic variants encoded, we now need transformations to refine and parallelize `D*` boxes. When an expert implements abstract suboperations, (s)he chooses from the ways to redistribute the operands in to enable computations to be performed in parallel across a machine by calling locally sequential computation on each core (e.g. via a call to a sequential (local) BLAS3 function). The result then needs be re-redistributed to the default $[M_C, M_R]$ distribution if it is not already distributed as such. To encode parallelization options for each of the `D*` boxes, we add refinements that have the building blocks of the local BLAS3 calls and the Elemental redistribution operation ("="). `L*` boxes represent local computation that does not require communication with other MPI processes. For Elemental these boxes map to a call to a sequential BLAS3 kernel (with parameter checking), so `L*` boxes are graph primitives (e.g. calling `LocalGemm` in Elemental code).

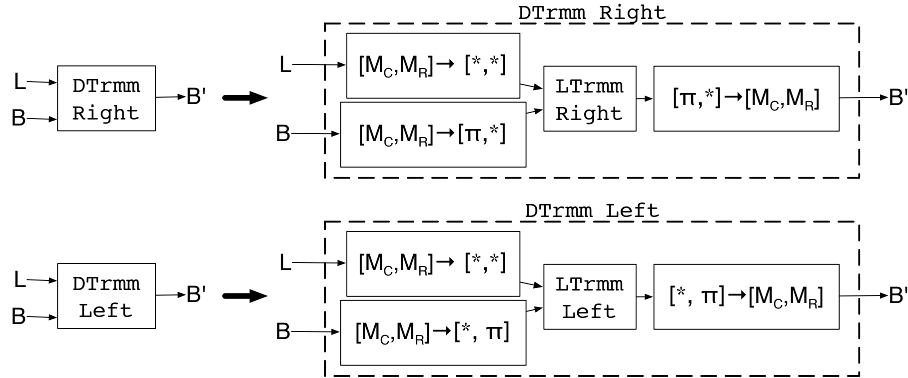

Figure 4: Templatized refinements for `DTrmm`: triangular matrix on the left or right and $\pi \in \{*, M_C, M_R, V_C, V_R\}$.

Consider `DTrmm`. In Figure 4, we show a templatized form of the refinements for `DTrmm` with the triangular matrix on the left or the right and $\pi \in \{*, M_C, M_R, V_C, V_R\}$. These options parallelize the computation over the process grid's rows or columns or over the entire grid. An expert considers these options based the other operations in the loop body, the problem size, etc. Each possible refinement is included in the DxT knowledge base. The top refinement

of Figure 4 with $\pi = \texttt{V}_\texttt{C}$ was used for the code of Section 3.

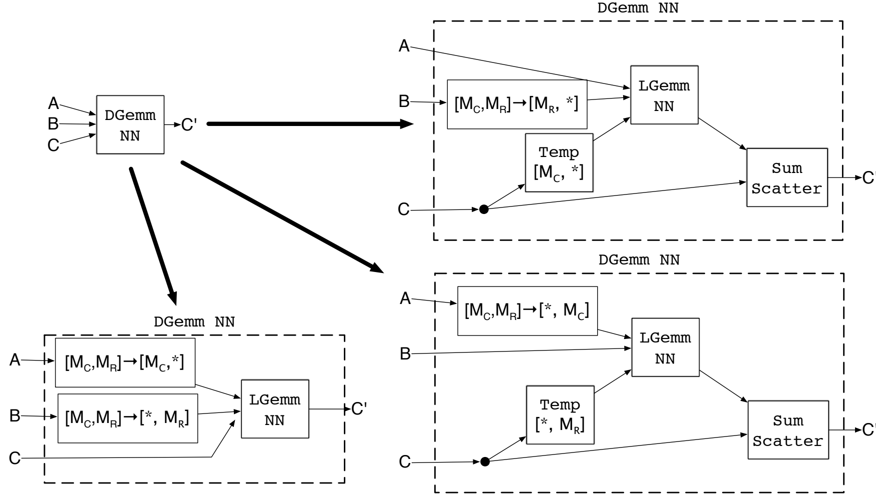

Figure 5: Three refinements for `DGemm NN` (`DGemm` without transposition), stationary `A`, `B`, and `C` from the top to bottom.

For `DGemm`, an expert again has a handful of choices to consider based on, for example, the surrounding operations and the size of operands. In Figure 5, we show three refinements for stationary `A`, `B`, and `C` for a non-transposed `DGemm NN`, which is the form of `DGemm` without transposition (i.e. `A` and `B` are both Normal instead of Transposed). `TEMP` boxes create a temporary storage matrix with the specified distribution. The input matrix provides `TEMP` with problem size information, but its data is not changed.

The `SumScatter` box is a form of Elemental redistribution that performs a `ReduceScatter` collective operation on the first operand and stores the result in the second operand [11]. There are small variations on these refinements for the three transposed versions of `DGemm`. An interested reader can discover them by looking at the Elemental library's `Gemm` implementations [11], which DxTer reproduces.

The other `D*` BLAS3 functions have refinements that are comparably simple, but the particular parallelization schemes are not important here. The fixed set of Elemental distributions enable the most useful (and some less useful) ways to parallelize BLAS3 operations. These schemes are encoded in our DxT knowledge base.

Redistribution Refinements Redistribution boxes map one-to-one to a single "=" operation in Elemental. This operation is implemented with MPI collective communication, but there are also alternate implementations. Exposing the implementation behind "=" and exploring alternatives enables the expert or DxTer to optimize the overall communication pattern of an implementation,

possibly combining communications exposed by refinements of different update statements.

In some cases, Elemental implements "=" as a series of redistributions. One example is $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_R},*]$, which utilizes an intermediate distribution $[\mathtt{V_C},*]$ (i.e., with $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_C},*] \rightarrow [\mathtt{V_R},*]$). Refinements like Figure 6 break through a layer of code to expose this detail. Then, optimizations can remove redundant or inverse redistributions that were hiding behind the "=" interface. In Section 3, we demonstrated why this is necessary.

Other refinements find an abstract node that cannot be directly implemented in Elemental and replace it with a specific implementation. Refinements like Figure 6, though, replace a node that represents valid Elemental code to expose internal implementation details (and hidden inefficiencies). This does not change the implementation; it just allows for subsequent optimizations.

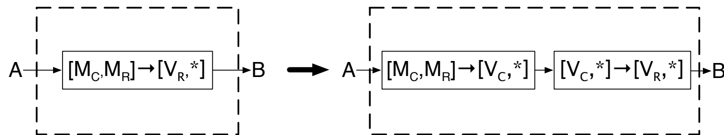

Figure 6: A redistribution refinement.

Refinements like Figure 6 can explore alternate implementations of redistributions, too. For example the $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{M_C},*]$ redistribution found in one **`DTrmm`** and **`DGemm`** refinement is implemented behind "=" with an **`AllGather`** collective among process rows. This redistribution can also be implemented as the two redistributions $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_C},*] \rightarrow [\mathtt{M_C},*]$, which requires an **`AllToAll`** followed by an **`AllGather`**, both among process rows. If code around the $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{M_C},*]$ operation already redistributes the data to $[\mathtt{V_C},*]$, then exposing the alternate redistributions enables a better overall implementation because an unnecessary redistribution to $[\mathtt{V_C},*]$ can be removed. There are four cases similar to this that are implemented with one templatized transformation. These transformations replace a node representing valid Elemental code with a subgraph that chooses a *different* implementation, which will allow DxTer to explore subsequent optimizations.

Redistribution Optimizations Refinements are sufficient to attain parallel, executable code. Combinations of costly redistribution operations need to be optimized to remove inefficient communication. For that, we use optimizing transformations for Elemental redistribution boxes.

The first optimization removes an inverse redistribution operation. The template is shown in Figure 7 (top), in which Σ and ϕ are any Elemental distributions. There is no need to perform an inverse redistribution when we can just use the original distribution. This optimization is applied often by experts and DxTer.

The second optimization removes a redundant redistribution operation. The template is shown in Figure 7 (bottom), where Σ and ϕ are any Elemental

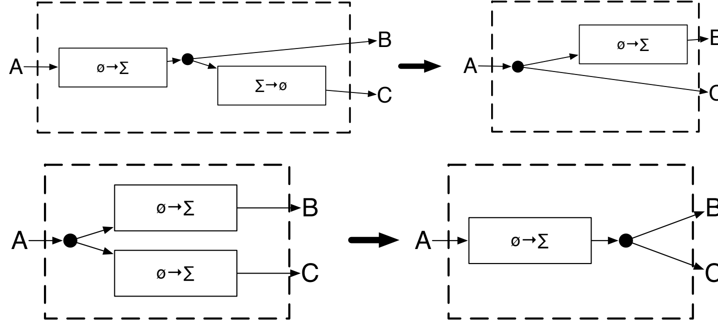

Figure 7: Templatized optimizations to remove inverse (top) and redundant (bottom) redistribution operation. Σ and ϕ can be any Elemental distribution

distributions.

As shown in Section 3, $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_C},*]$ can be implemented (suboptimally) as $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{M_C},*] \rightarrow [\mathtt{V_C},*]$ This is a refinement of the $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_C},*]$ redistribution. This enables a subsequent optimization. We encode the optimization of Figure 8, to represent both steps. This reuses an intermediate distribution $[\mathtt{M_C},*]$. There are 8 versions of this transformation that are implemented using a templatized version of Figure 8. Template parameters are limited to distributions that make sense for this optimization.

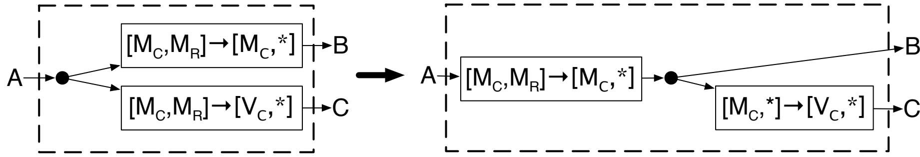

Figure 8: Reusing an intermediate redistribution.

For redistribution operations, data is copied into and out of buffers that are passed to collective communication (MPI) functions. It can be very costly to access memory with non-unit stride. With Elemental, data can be transposed in some redistribution operations. This moves the cost of non-unit stride between packing and unpacking to push the performance hit on the piece with less data to copy. Many inputs to BLAS3 functions can be transposed, so DxTer has optimizing transformations that transpose data during redistribution and untranspose it in BLAS3 function calls.

Simplicity of Transformations The graph transformations we have illustrated are no more complicated than those we have not shown here. Abstractly, they are all simple graph rewrites that capture deep domain knowledge of DLA and its encoding in Elemental. Had we chosen another distributed-memory

DLA library that did not have a cleanly-layered design, we suspect we would have been less successful or not successful at all. We can not stress enough that the key to the simplicity of our rewrite rules is that they capture relationships between fundamental levels of abstraction in DLA library design. If these abstractions are encoded in an ugly way, transformations are substantially more complex.

5 Cost Estimates

DxTer applies transformations to an input graph to generate a space of semantically equivalent graphs (implementations). It then computes a cost estimate for each implementation to choose the best (most efficient) from the space. In this section, we explain how cost estimates are made.

Node Costs The cost of a node is a function of the node's type and the size of its input matrices. DxTer calculates cost as the number of floating point operations performed multiplied by γ, an estimate of the time to complete a floating point operation on the target machine.

Redistribution boxes (Elemental "=" operations) are implemented with a packing operation, a call to an MPI collective communication routine, and an unpacking operation. Cost estimates of collectives are known for idealized models in terms of α and β, the cost of network startup and the cost to communicate a piece of data, respectively [4]. An expert balances the cost of data communication relative to the cost of computation. For this (s)he uses a rough estimate of the relative costs. α and β are set to be reasonable multiples of γ to mimic the expert's attempt to balance the factors.

Packing and unpacking operations are not directly modeled in DxT graphs (at this point), but their costs are included in communication cost models. An exact cost that reflects complex memory access is not needed. A cost function needs to mimic the decisions of an expert, who does not seem to use a very accurate estimate when manually coding. The goal is to have a cost estimate that penalizes bad (large) memory strides but does not grossly overestimate. DxTer currently uses a cost of ω, which we set equal to γ, to copy each number with unit stride and $10 \times \omega$ to copy each number with non-unit stride. The motivation is that reading and writing a number takes on the order of the time it takes to perform a floating-point operation when the stride is one and substantially more time when the stride is not one. This is indeed a rough estimate, but it is sufficient in all of our tests to choose transposed communication when it does not substantially increase the cost of communication.

The cost of an entire graph is calculated by summing the cost of all nodes for an input problem size.

Sample cost functions are shown in Figure 9. Consider when the computation of `Trmm` is parallelized such that each processor's `LTrmm` operation works on $\frac{1}{\mathtt{p}}$, $\frac{1}{\mathtt{c}}$, or $\frac{1}{\mathtt{r}}$ of the overall computation, which happens with the parallelization

refinements above (with process grid configuration $p = r \times c$). The LTrmm cost model reflects these differences in runtimes.

The two redistribution costs reflect the collective communication and packing and unpacking costs. Studying these, one can see why DxTer finds transpose optimizations to be beneficial. The same amount of data is communicated via an AllGather collective, reflected in the first line of each cost. The first and second term in the second line of each is the packing cost and unpacking cost, respectively. By transposing during redistribution, the additional cost of copying data with a non-unit stride is shifted to the packing operation. The amount of data copied while packing is c-times less than is copied while unpacking, so this reduces the overall penalty of non-unit stride.

Loops In DxTer loops are represented with a graph for the loop body. Loop inputs are split into submatrices (views of the input matrix), which are inputs to the loop body. The outputs of the body are submatrices "combined" to form the output of the loop (there is no actual combination since the submatrices are just views of the same matrix). This reflects the beginning and end of the while loop in Figure 2, where submatrices are exposed and combined. The split and combine operations are represented in the loop body by LoopSplit and LoopCombine nodes, which mark the beginning and end of a loop body in the graph.

For the results in [8], DxTer only calculated the cost of the loop body for the middle iteration. Even though submatrices are a different size at each iteration, it was sufficient to optimize for the middle iteration to reach the same design decisions as an expert (perhaps he reasoned about the middle iteration as well). For BLAS3 functions, though, the total cost of all iterations must be considered, so DxTer cost calculation was improved to do just that.

To calculate the cost of a loop, execution is simulated. Input matrix sizes are known, so the number of iterations is known in terms of blocksize. At each iteration, the size of the inputs' submatrices can be calculated, so the cost of a loop body graph can be calculated by summing the cost of all nodes. Then, the cost of the whole loop is the sum of the loop body's cost at each iteration.

Operation	Cost
LGemm NN $(m \times k,\ k \times n,\ m \times n)$	$\gamma 2mkn$
LTrmm Right $(n \times n,\ m \times n)$	γmnn
$[V_C, *] \rightarrow [M_C, *]$ $(m \times n)$	$\alpha \lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$ $+ \omega \frac{m}{p} n + 10 \omega c \frac{m}{p} n$
$[V_C, *] \rightarrow [*, M_C]^T$ $(m \times n)$	$\alpha \lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$ $+ 10 \omega \frac{m}{p} n + \omega c \frac{m}{p} n$

Figure 9: First-order approximations for the cost of operations. γ is time for a floating-point operation. α and β are network startup and transfer costs, respectively. ω is the cost of copying a piece of data at unit stride.

How Does This Work? The cost functions we use may seem too simple to be useful. They are sufficient in our tests, though, and this makes sense when we consider their role in mimicking manual design decisions. These cost estimates were not meant to predict the performance of an implementation on the target architecture accurately. Instead, they are meant to rank-order the generated implementations in terms of performance in the same way that an expert might, though (s)he usually cannot evaluate every implementation. The cost estimates we use are enough to point out bad parallelization choices, missed optimizations, etc.

6 Results

DxTer took as input Elemental primitives and transformations like those above. From this, DxTer automatically generated code for all BLAS3 variants that is the **same or better** than that of an expert. This is an important accomplishment because developing all of the code by hand is rote, tedious, and error prone (much of the development time goes into testing). Having a system to automatically generate code is a significant step and to our knowledge the first time it has been done.

The performance improvement offered over a human developer is noteworthy but limited in our tests cases because BLAS3 operations are simple and do not offer many opportunities for optimization (thus the human missed few of them). Keep in mind, though, that the transformations that enable these results can be used to generate code for more complicated algorithms built from BLAS3 operations. They would require complex combinations of the existing optimizations and would show much greater benefit from their application.

In this section, we describe the size of the knowledge base in DxTer, the size of the space encountered when using that knowledge, and the quality of DxTer-generated code. Performance results were taken from Argonne's BlueGene/P system Intrepid. We tested on 8192 cores (2 racks), which have a combined theoretical peak of over 27 TFLOPS. Two-thirds of peak performance is shown at the top of the graphs. Double precision arithmetic was used in all computations. For all runs, we tune the blocksize and choose the best-performing run. The algorithm and implementation selections of DxTer account for the vast majority of performance; tuning the blocksize provides a small performance boost.

Number of Transformations The BLAS3 are reused repeatedly when implementing code for a variety of targets. Further, refinements that implement suboperations are used repeatedly across libraries.

Redistribution optimizations are templatized to work for many communication patterns (recall Figure 7). Similarly, the transformations (algorithm and parallelization refinements) for Hermitian and symmetric BLAS3 operations are largely identical so the same knowledge can apply to both sets of operations.

To generate code for all BLAS3 operations, DxTer has a set of transformations that are reused repeatedly (i.e. its knowledge base). Figure 10 shows the

Type	Unique	Total
Algorithm refinement	19	30
Parallelization refinement	14	31
Redistribution refinement	30	30
Redistribution optimization	3	728
Redistribution transposition	6	22

Figure 10: Rule count in DxTer's BLAS3 knowledge base.

unique transformations encoded in DxTer for BLAS3 operations. It also shows the total number of transformations that are generated from those unique pieces of knowledge using templates (different distributions, symmetric and Hermitian, etc.).

Search Space and DxTer Results BLAS3 implementations for distributed memory must be tailored to the problem size and parameter combination. Consider, for example, `Gemm`: $\mathtt{C} := \mathtt{AB} + \mathtt{C}$. `Gemm` is best provided in a library with different implementations for when each of the 3 input matrices is the largest (to minimize communication of it) and for each of the 4 combinations of "`A`" and "`B`" being transposed. As a result, Elemental offers $\mathtt{12} = \mathtt{3} \times \mathtt{4}$ `Gemm` implementations. Implementations of `Trmm` could minimize communication of each of its 2 input matrices (whichever is biggest) and there are 3 parameters that lead to 8 different algorithms and parallelization schemes, yielding a total of 16 implementations. The second column of Figure 1 shows the number of implementation variants for each BLAS3 operation.

For each variant of each operation, we tested DxTer's ability to generate code. The third column of Figure 1 shows the total number of implementations generated by DxTer. Different parameters lead to different implementations (because different starting algorithms are used). For variants with the same parameter combination (but different matrix sizes), the same implementations are generated, but the cost estimates rank-order them differently. This count includes the repeated implementations that are re-generated for each of the variants. All implementations are generated within 30 minutes; the majority take less than a minute.

Many of the differences between implementations are due to the variety of ways in which data can be redistributed and transposed. Consider the number of transformations dealing with redistributions, shown in Figure 10. There are 5 algorithmic variants for `Her2k`, but only one parallelizing refinement for `DHer2k` in their loop bodies. This does not lead to many implementations options. The large space is the result of the many ways to redistribute and transpose operands to the local computation.

When the Elemental expert (Jack Poulson) first implemented the BLAS3, he explored a portion of these search spaces. At that point, he did not apply transposition optimizations. Later, he revisited the BLAS3 implementations and transposed redistributions to improve performance. The expert explored large implementation spaces using his intuition and experience. Because of

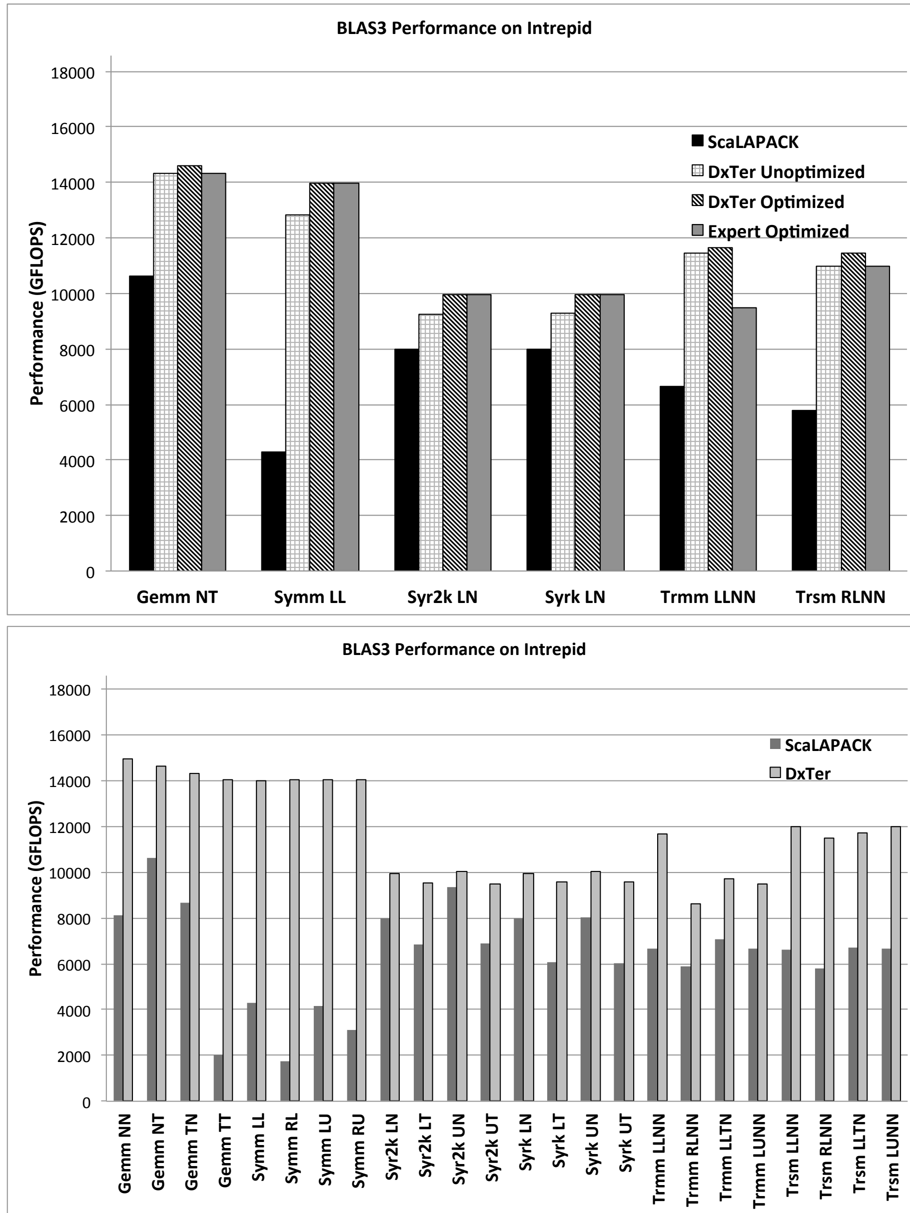

Figure 11: Performance of BLAS3 functions. Problem size is 50,000 along all dimensions.

the number of possibilities and the difficulty with reoptimizing existing code, though, he did choose sub-optimal implementations in some cases. The last column of Figure 1 compares DxTer's implementations to the code in Elemental.

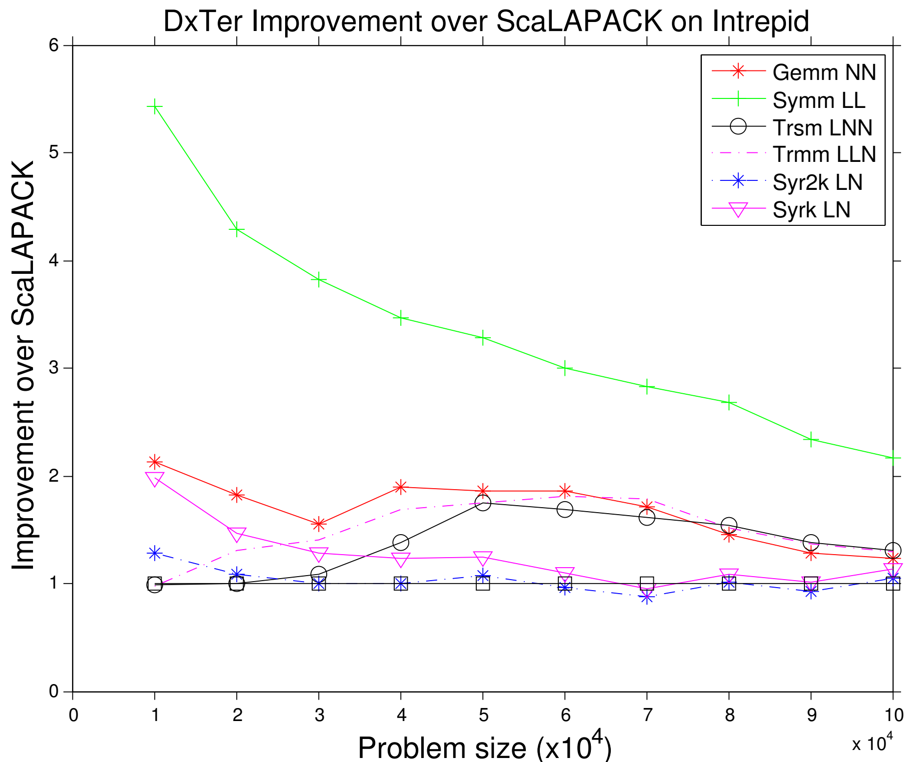

Figure 12: Performance of a collection of real BLAS3 functions.

Figure 11 (top) compares representative variants of each of the double-precision, real BLAS3 functions with problem sizes along each dimension of 50,000. We show performance from ScaLAPACK, Elemental, DxTer without optimization (only parallelization), and DxTer with optimization. In many cases, the expert and DxTer produced the same implementations, but there were some notable improvements. **In all cases, DxTer generated implementations that were the same or better than the expert.**

For `Gemm`, the expert missed a number of transposition opportunities that improved performance. DxTer determined when those transpositions were worthwhile (the cost functions predicted that runtime decreased) and generated code that incorporated the optimization.

For `Trsm`, DxTer again found a missed transposition opportunity in one variant. Figure 11 (top) shows this is a modest improvement, but it is worthwhile and it came without human effort. The improvement is greater for smaller problem sizes. Additionally, the expert had not implemented some of the `Trsm` variants. DxTer had sufficient knowledge to generate code for all variants.

The greatest DxTer successes came when studying `Trmm`. DxTer has 3 algo-

rithms encoded for the left-side and right-side versions of `Trmm`. DxTer explored all implementations of these algorithms and chose as best a different algorithm than that chosen by the Elemental expert. He did not explore the algorithm in Figure 2. Figure 11 (top) shows the performance of DxTer's implementation over the expert-optimized version for one problem size, but DxTer's version is better across problem sizes.

Figure 11 (bottom) shows many parameter combinations for the real BLAS3 functions. We compare DxTer's predicted-best implementations against ScaLAPACK's implementations. The majority of these are the same as Elemental, so we omit its performance. Figure 12 shows a sample of these functions across a range of problem sizes, demonstrating DxTer-generated Elemental code performs better than or roughly equal to that of ScaLAPACK.

7 Conclusion

We demonstrated how the knowledge an expert uses to develop BLAS3 code for distributed memory can be encoded as reusable transformations in the Design by Transformation (DxT) style. Using this knowledge, our tool DxTer automatically generates code for the many BLAS3 variants. This shows how the burden of coding sequential algorithms in distributed-memory code can be taken from a human and given to a machine. Instead of requiring an expert to apply knowledge repeatedly — a tedious and error-prone process — a system like DxTer can be trusted to do it automatically. BLAS3 operations do not allow many opportunities for optimization, but even an expert developer missed some. DxTer missed none. DxTer even explored a different algorithmic variant than that chosen by the expert and generated substantially better-performing code. This is the power of automatic code generation.

In [8, 10], some of the knowledge used in this paper was applied to much more complicated algorithms (with many BLAS3 operations in their loop bodies). This paper extends that knowledge base to support all BLAS3 operations. We expect to apply it to more algorithms and demonstrate more utility from automatically generating distributed-memory DLA code. Further, we intend to use DxT to generate sequential and shared-memory parallel code.

DxT is applicable far beyond the DLA domain [15], but DLA is a prime candidate for initial evaluation. DLA code can be cast in terms of a relatively small number of operations whose refinements and optimizations well-known. The results in this paper are major step to automating code development for DLA and many other domains.

Acknowledgments. Marker was sponsored by a fellowship from Sandia National Laboratories and an NSF Graduate Research Fellowship under grant DGE-1110007. This work was also partially sponsored by NSF grants OCI-0850750, CCF-0917167, and OCI-1148125 and 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. 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] E. Anderson et al. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

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

[3] E. Chan et al. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP 2008*.

[4] E. Chan et al. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[5] J. J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, March 1990.

[6] J. A. Gunnels et al. Flame: Formal linear algebra methods environment. *ACM TOMS*, December 2001.

[7] B. Kågström et al. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[8] B. Marker et al. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *iWAPT 2012*.

[9] B. Marker et al. DxTer: A program synthesizer for dense linear algebra. CS TR-12-17, Univ. of Texas at Austin, 2012.

[10] T. Meng Low et al. Theory and practice of fusing loops when optimizing parallel dense linear algebra operations. CS TR-12-18, The Univ. of Texas at Austin, 2012.

[11] J. Poulson. `code.google.com/p/elemental`, 2010.

[12] J. Poulson et al. (parallel) algorithms for two-sided triangular solves and matrix multiplication. *submitted*, 2012.

[13] J. Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. *ACM TOMS*, to appear.

[14] M. Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005.

[15] T. Riche et al. Architecture design by transformation. Comp. Sci. TR-10-39, Univ. of Texas at Austin, 2010.

[16] M. Schatz et al. Parallel matrix multiplication: 2d and 3d. CS TR-12-13, Univ. of Texas at Austin, June 2012.

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

[18] R. A. van de Geijn et al. *The Science of Programming Matrix Computations*. `lulu.com`, 2008.

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