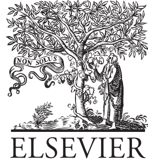


Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia
Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

Code Generation and Optimization of Distributed-Memory Dense Linear Algebra Kernels

Bryan Marker[a], Don Batory[a], Robert van de Geijn[a]

[a]*Department of Computer Science, The University of Texas at Austin*

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 (cluster) architectures. Getting good BLAS3 performance for such platforms requires deep domain knowledge, so their implementations are manually coded by experts. Unfortunately, there are few such experts and developing the full variety of BLAS3 implementations takes a lot of repetitive effort. A prototype tool, DxTer, automates this tedious task. We explain how we build on previous work to represent loops and multiple loop-based algorithms in DxTer. Performance results on a BlueGene/P 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.

© 2012 The Authors. Published by Elsevier B.V.
Selection and/or peer-review under responsibility of the [Organiser Name].

Keywords: program generation; dense linear algebra; high-performance software; distributed-memory computing

1. Introduction

Many scientific computing libraries and applications cast computations in terms of high-performing DLA interfaces such as the BLAS [1], LAPACK [2], and `libflame` [3]. By providing efficient implementations for such interfaces, portable high-performance can be achieved, enabling an application to be moved as-is to new hardware architectures.

A large portion of the knowledge needed to implement DLA libraries resides in the BLAS3 operations, listed in Figure 1 (left). DLA libraries must provide many variants of each operation (column 2 of Figure 1 (left)). To implement all variants, an engineer must be or become a domain expert. (S)he must consider many algorithms for each operation variation and many implementations of each algorithm (e.g. different parallelization schemes), so (s)he must have the knowledge to explore many options (column 3 of Figure 1 (left) quantifies some of these). Unfortunately, there are very few experts with such knowledge, and only they can write correct high-performance code. Further, applying this knowledge is both difficult and tedious. We believe the way forward is to automate the development of DLA libraries.

E-mail address: bamarker@cs.utexas.edu.

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

Type	Unique	Total
Algorithm refinement	19	30
Parallelization refinement	14	31
Redistribution optimization	32	758
Redistribution transposition	6	22

Fig. 1. (Left) DxTer code generation statistics for the BLAS3s. (Right) Rule count in DxTer's BLAS3 knowledge base.

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 subset DLA library functionality, the BLAS3. We introduced DxT in [4], and have made considerable progress since. We go beyond our initial work in three ways: 1) DxTer can now explore multiple abstract algorithmic variants for operations instead of just one, as in [4]. One variant is not always best, so exploring options is good when targeting various architectures. 2) DxTer generates the algorithm in a loop body of a DLA operation; to be able to select the best-performing algorithm, we must estimate the sum of all of its iterations; this was unnecessary in prior work and is necessary now. To this end, we now represent loops in DxTer. 3) We add a significant amount of knowledge to DxTer to enable followup work on more complicated DLA algorithms, seen if Figure 1 (right).

2. Design by Transformation (DxT)

Abstractions, Refinements, and Optimizations. We use *directed acyclic graphs (DAGs)* to encode DLA algorithms [5]. 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.

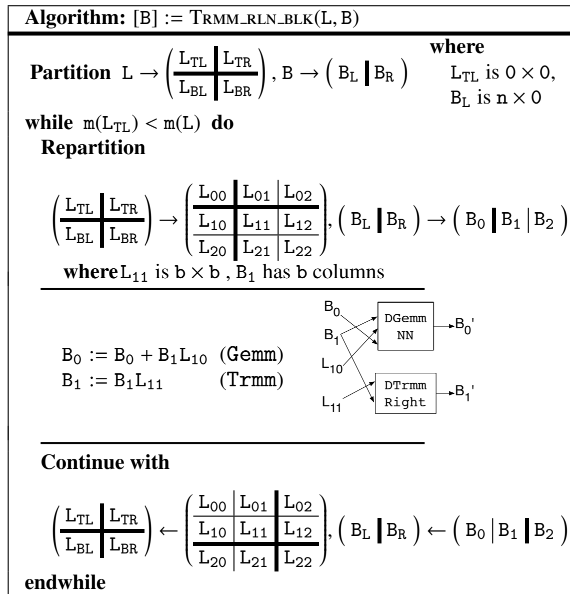
We start with a simple DAG that encodes a sequence of one or more BLAS3 operations. There are no implementation details for these operations other than preconditions and postconditions. Nodes without implementation details are called *abstractions*.

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 clusters), 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 goal of program optimization. In effect, experts optimize a DAG by repeatedly 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 an executable 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



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

Fig. 2. (left) Variant of Trmm: right, lower, non-transposed and the DxT representation of the loop body. (right) Examples of distributions on a $p = r \times c$ process grid to be used to parallelize the algorithm.

for DLA on clusters 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 [6]. Example cost functions were presented in [4]. We explain below how loop costs are now estimated by extending our previous work.

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 clusters. 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.

A prototypical BLAS3 algorithm. Figure 2 (left) shows a prototypical BLAS3 algorithm in FLAME notation [7], which, given a lower triangular matrix L and general matrix B, overwrites B with the product 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 (left)).

This is a prototypical example of how all BLAS3 can be implemented by casting most computation in terms of Gemm [8]. 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 (left)) 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, the p MPI processes on a cluster are viewed as a two-dimensional grid, $p = r \times c$. For the default distribution, Elemental uses a 2D element-wise cyclic distribution, labeled $[M_C, M_R]$ where M_C and 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. There are a handful of other one and two-dimensional distributions of matrices, examples listed in Figure 2 (right), 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: $B = LB, B = L^T B, B = UB, = U^T B, B = BL, B = BL^T, B = BU$, and $B = BU^T$, where L and U are lower and upper triangular matrices, respectively. Each of these eight possibilities is implemented separately with different algorithms. We focus on $B = BL$ for which Figure 2 (left) gives one of several algorithms an expert considers. The inputs L and B have the default $[M_C, M_R]$ distribution. The updates `Trmm` and `Gemm` in Figure 2 (left) are parallelized by redistributing submatrices, performing local computation (via calls to sequential BLAS3 routines) on each process, and (if necessary) reducing and/or communicating the result.

An expert would need to consider the various ways to parallelize the suboperations. The three 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, B_0 (defined in Figure 2 (left)) 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 2 (right), 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 in Figure 3.

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 $[M_C, M_R]$ to $[M_C, *]$ (denoted $[M_C, M_R] \rightarrow [M_C, *]$) and then $[M_C, M_R] \rightarrow [V_C, *]$. The $[M_C, M_R] \rightarrow [V_C, *]$ redistribution can be implemented with an `AllToAll` or it can be implemented in terms of the two redistributions, $[M_C, M_R] \rightarrow [M_C, *] \rightarrow [V_C, *]$, 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 $[M_C, *]$, which results in the best performance.

An expert explores this option in code by replacing 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 duplicated line `B1_MC_STAR = B1;` The resulting optimized code, which *is* in the Elemental library, is shown in Figure 4.

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

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

Fig. 3. Parallelized code for Figure 2.

```
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;
```

Fig. 4. Optimized version of that code.

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 mentioned in Figure 1 (left). 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 ones to be considered in well-layered code thanks to decades of software engineering in this field [2, 3, 9, 10].

General rules for attaining high performance are that communication and redundant computation should be reduced 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 implementations 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 [11]. Using formal derivation, a person or a mechanical system [12] can derive multiple correct algorithmic variants, expressed similarly to Figure 2 (left), 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. In [4], we implemented only one such variant for each operation. For this work, we encoded multiple algorithmic variants, which allows DxTer to explore the options, so a human does not need to choose one as best manually. BLAS3 operations and their FLAME-derived algorithms are mathematical in nature (e.g., Figure 2 (left)) 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., to optimize the `Trmm` operation, the starting graph to be implemented consists of a single node labeled `Trmm`). These are purely mathematical abstractions with 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 (i.e. as input to DxTer).

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 (left), the update statements are BLAS3 operations themselves, operating on smaller submatrices. 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.

The refinement of node `Trmm` for the algorithm of Figure 2 (left) is a loop with abstract update statements `Trmm` and `Gemm` (the update statements are shown in that figure). 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* [1]). 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 refinements for clusters, described below.

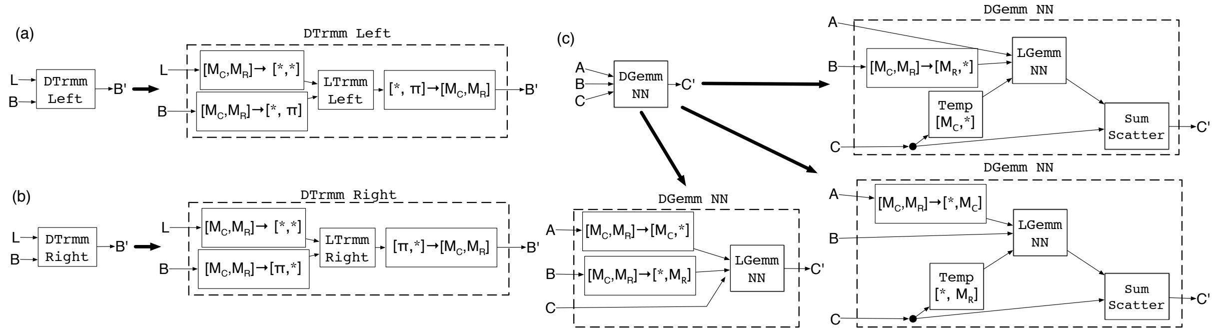


Fig. 5. Templatized refinements for DTrmm: triangular matrix on the left (a) or right (b) with $\pi \in \{*, M_C, M_R, V_C, V_R\}$. (c) Three refinements for DGemm NN (DGemm without transposition), stationary A, B, and C from the top to bottom.

BLAS3 Cluster Refinements. With knowledge of algorithmic variants encoded, we now need transformations to refine and parallelize D* boxes. Examples are shown in Figure 5. When an expert implements abstract suboperations, (s)he chooses from the ways to redistribute the operands in order 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 code).

Consider DTrmm. In Figure 5, we show a templatized form of the refinements for DTrmm with the triangular matrix on the left (a) or the right (b) with $\pi \in \{*, M_C, M_R, V_C, V_R\}$. These options parallelize the computation over the process grid's rows or columns or over the entire grid. An expert considers these options based on other operations in the loop body, the problem size, etc. Each possible refinement is included in the DxT knowledge base. The refinement of Figure 5 (b) with $\pi = V_C$ was used for the code of Section 3.

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 (c), 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 [13]. 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 [13, 14], 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 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.

Redistribution boxes map one-to-one to a single "=" operation in Elemental. This operation is implemented with default MPI collective communication, but there are other 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 $[\mathtt{M_C},\mathtt{M_R}] \rightarrow [\mathtt{V_C},*] \rightarrow [\mathtt{V_R},*]$). Optimizations like Figure 6 (c) break through a layer of code to expose this detail. In Section 3, we demonstrated why this is necessary to remove inefficient communication. The template optimizations of Figure 6 (a) and (b) can remove inverse or redundant redistributions, respectively that were hiding behind the "=" interface. These optimizations are applied often by experts.

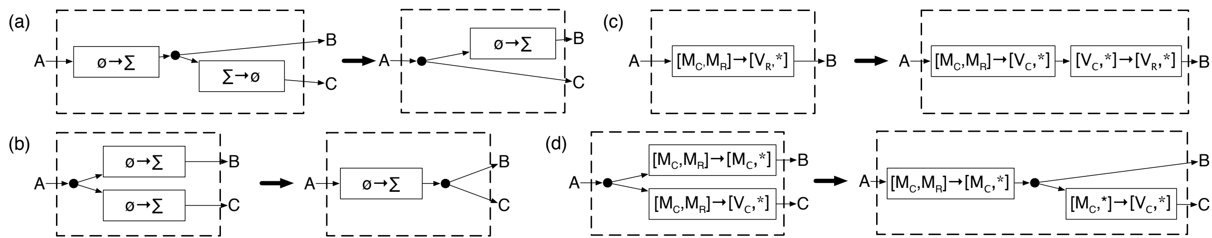


Fig. 6. Templatized optimizations to remove inverse (a) and redundant (b) redistribution operations. Σ and ϕ can be any Elemental distribution. (c) An optimization to expose a hidden intermediate redistribution. (d) Reusing an intermediate redistribution.

Optimizations like Figure 6 (c) 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.

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 6 (d), to represent both steps. This reuses the intermediate distribution $[\mathtt{M_C},*]$. There are eight versions of this transformation that are implemented using a templatized version of Figure 6 (c). Template parameters are limited to distributions that make sense for this optimization.

For redistributions, 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 redistributions. 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 optimizations that transpose data during redistribution and implicitly untranspose it in BLAS3 function calls.

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

Estimating Loop Costs. 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 (left), 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 [4], 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.

5. Results

We now 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. For all runs, double precision arithmetic was used and we tune the blocksize, choosing the best-performing run. DxTer's algorithm and implementation selections 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 for use by many communication patterns (Figure 6 (a) and (b)). 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 1 (right) shows the unique (i.e. counting each template once) 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 clusters must be tailored to the problem size and parameter combination. Consider, for example, `Gemm`: $C := AB + C$. `Gemm` is best provided in a library with different implementations for when each of the three input matrices is the largest (to minimize communication of it) and for each of the four combinations of "A" and "B" being transposed. As a result, Elemental offers $12 = 3 \times 4$ `Gemm` implementations. Implementations of `Trmm` could minimize communication of each of its two input matrices (whichever is biggest) and there are three parameters that lead to eight different algorithms and parallelization schemes, yielding a total of 16 implementations. The second column of Figure 1 (left) 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 (left) 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. Each implementation is 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 1 (right). There are five 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

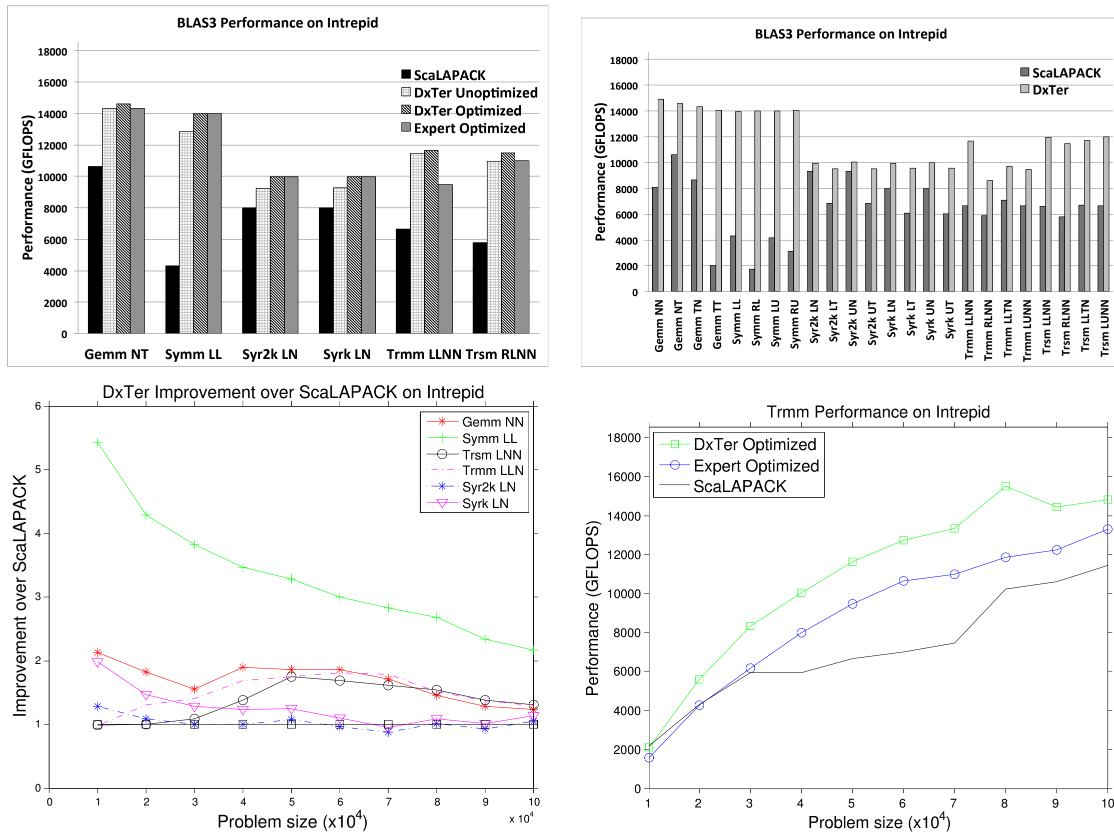


Fig. 7. Performance of real BLAS3 functions. Problem size is 50,000 along all dimensions for top graphs.

and transposed redistributions to improve performance. The expert explored large implementation spaces using his intuition and experience. Because of the number of possibilities and the difficulty with reoptimizing existing code, though, he chose sub-optimal implementations in some cases. The last column of Figure 1 (left) compares DxTer's implementations to the code in Elemental.

Figure 7 (top left) 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 7 (top left) 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 three algorithms encoded for the "left-side" and "right-side" versions of `Trmm`, each. 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 (left). Figure 7 (bottom right) shows the performance of DxTer's implementation over the expert-optimized version.

Figure 7 (top right) 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 7 (bottom left) 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. Figure 7 (bottom right) shows the performance improvement DxTer gained when exploring many algorithms for `Trmm`, choosing one that is better than what the expert developer of Elemental used, highlighting the utility of automatic code generation.[1]

6. Conclusion

We showed how the knowledge an expert uses to develop BLAS3 code for clusters 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 showing that the burden of coding sequential algorithms in code for clusters 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 [4, 15], 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 and add support for loops. We expect to apply the knowledge to more algorithms and demonstrate more utility from automatically generating DLA code for clusters. Further, we intend to use DxT to generate sequential and shared-memory parallel code.

DxT is applicable beyond the DLA domain [16], 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 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 NSF.*

References

[1] J. J. Dongarra, et al., A set of level 3 basic linear algebra subprograms, ACM TOMS 16 (1).
[2] E. Anderson, et al., LAPACK Users' guide (third ed.), SIAM, Philadelphia, PA, USA, 1999.
[3] F. G. Van Zee, `libflame`: The Complete Reference, `www.lulu.com`, 2009.
[4] B. Marker, et al., Designing linear algebra algorithms by transformation: Mechanizing the expert developer, in: iWAPT 2012.
[5] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[6] E. Chan, et al., Collective communication: theory, practice, and experience, Concurrency and Computation: Practice and Experience 19 (13) (2007) 1749–1783.
[7] R. A. van de Geijn, et al., The Science of Programming Matrix Computations, `lulu.com`, 2008.
[8] B. Kågström, et al., GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark, ACM Trans. Math. Soft. 24 (3) (1998) 268–302.
[9] J. Poulson, et al., Elemental: A new framework for distributed memory dense matrix computations, ACM TOMS 39 (2).
[10] E. Chan, et al., SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks, in: PPoPP 2008.
[11] J. A. Gunnels, et al., FLAME: Formal linear algebra methods environment, ACM TOMS 27 (4).
[12] P. Bientinesi, Mechanical derivation and systematic analysis of correct linear algebra algorithms, Ph.D. thesis, UTCS, The University of Texas at Austin (2006).
[13] J. Poulson, `code.google.com/p/elemental` (2010).
[14] M. Schatz, et al., Parallel matrix multiplication: 2d and 3d, CS TR-12-13, Univ. of Texas at Austin (June 2012).
[15] 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).
[16] T. Riche, et al., Architecture design by transformation, Comp. Sci. TR-10-39, Univ. of Texas at Austin (2010).

[1] We omit related works here as they are the same as in [4].