Code Generation to Aid Parallel Code Development

Bryan Marker, Tyler Smith, Don Batory, Field Van Zee, and Robert van de Geijn
Department of Computer Science
The University of Texas at Austin
{bamarker,tms,batory,field,rvdg}@cs.utexas.edu

Abstract

Design by Transformation (DxT) encodes domain-specific software design knowledge as graph rewrites; a tool applies these rewrites to a specification (an initial graph) to automatically derive the most efficient implementation of that graph (code). Previously, we used DxT to generate high-performance, distributed-memory dense linear algebra (DLA) code for the level-3 Basic Linear Algebra Subprograms (BLAS3) and other operations. In this paper, we use DxT to target BLAS3 operations for other architectures: first a sequential machine, and then a shared-memory, multithreaded machine. As new heuristics were invented to parallelize code, we added them as DxT rewrites, and DxT automatically applied them to 32 BLAS3 operations, sparing us tedious, rote, and error-prone work to write their code. We explain how DxT eases development and automates much of the task of developing architecture-customized DLA libraries. As of this writing, the experts behind the BLIS framework for implementing BLAS, upon which we build, still have not hand coded most of the functionality for multithreaded architectures that we have mechanically generated.

1 Introduction

Scientific computing developers often go through a tedious and rote process when parallelizing a library of code for multithreaded, distributed memory, GPUs, and other architectures. Design by Transformation (DxT) automates software design by expressing expert design knowledge as graph rewrites. Specifications are initial graphs. A tool (called DxTer) uses graph rewrites to explore all implementations of the initial graph and selects the best, and translates the selected graph into executable code. Doing so, it explores options more thoroughly than a human expert, who is limited by patience and time and burdened by human error.
The level-3 Basic Linear Algebra Subprograms (BLAS3) are a fundamental set of operations for dense linear algebra (DLA) libraries; they express matrix-matrix operations from which more complex operations can be built. Each BLAS3 operation comes in multiple flavors, for example, by implicitly transposing some of its arguments. Figure 1 lists BLAS3 operations, the number of flavors (listed as parameter combinations), and an equation that it represents. An expert must implement all of these flavors, reusing a lot of the same design knowledge repeatedly. High performance from each operation is critical to the applications built using the BLAS3, so a library developer’s task is rote but must be done well — only an expert can do it (or possibly an automated approach, explained in Section 5). Normally, one must implement additional BLAS3 operations for complex datatypes, but for sequential and multithreaded architectures we use datatype templatization so the same code can be used for complex and real datatypes.

<table>
<thead>
<tr>
<th>BLAS3</th>
<th>Parameter Combinations</th>
<th>Sample Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gemm</td>
<td>4</td>
<td>C := αAB + βC</td>
</tr>
<tr>
<td>Symm</td>
<td>4</td>
<td>C := αAB + βC</td>
</tr>
<tr>
<td>Syr2k</td>
<td>4</td>
<td>C := α(AB^T + BA^T)/βC</td>
</tr>
<tr>
<td>Syrk</td>
<td>4</td>
<td>C := αAA^T + βC</td>
</tr>
<tr>
<td>Trmm</td>
<td>8</td>
<td>B := αBL</td>
</tr>
<tr>
<td>Trsm</td>
<td>8</td>
<td>B := αA^{-1}B</td>
</tr>
</tbody>
</table>

Figure 1: The BLAS3 operations. We omit the complex-datatype flavors. They are implemented using the same code as the real-datatype versions using templatization.

In prior work [14, 15, 16], we created a knowledgebase of rewrites that DxTer used to generate algorithms for BLAS3 operations for distributed-memory architectures. The implementations DxTer produced were as good or better than their hand-coded counterparts. These improved implementations are now being shipped with Elemental [19], a DLA library for distributed memory.

In this paper, we show how we modified this knowledgebase to generate high-performance BLAS3 algorithms for a sequential machine and then a shared-memory, multithreaded machine. We did so by adding knowledge about sequential implementations and by adding parallelizing heuristics. DxTer provided a productivity enhancer for us to develop new ways to parallelize code. Implementation ideas were added to DxTer as we developed them. DxTer, in turn, generated algorithms for all 32 BLAS3 operations in less than a minute, effectively searching a superset of the same options as we would have, but it did so automatically, much faster, and more reliably. This allowed us to spend time on developing new parallelization options instead of applying them to code.

The main contribution of this paper is an explanation of the changes needed to retarget our distributed-memory knowledgebase to generate sequential and multithreaded code. This includes an assessment of both how much distributed-memory knowledge could be reused (a lot) and how much knowledge had to be added (not a lot), demonstrating the power of automation in moving to a new
hardware architecture and the potential for changing architectures in the future. Further, we summarize our experience using DxTer as a productivity enhancer in developing new code.

In the next sections, we explain the basics of DxT and how implementation and parallelization knowledge is represented as graph rewrites for automatic exploration and code generation. We present the heuristics employed in DxT and performance results obtained from generated BLAS3 code, for a sequential and a multithreaded machine. In both cases, the generated code shows good speedup for corresponding hand-crafted code.

2 Design by Transformation

This section provides a minimal summary of DxT from previous work [7, 14, 15, 16, 21].

2.1 Graphs and Transformations

DxT represents algorithms and implementations as dataflow graphs — directed, acyclic graphs (DAGs). Each node, also called a box or operation, denotes a function call. Function inputs are represented by incoming edges and outputs by outgoing edges.

Each node is a fundamental operation in a domain. There are two kinds of nodes, interfaces and primitives. An interface is a node without implementation details, defined only by preconditions and postconditions. A primitive is a box with a given code implementation – a function call.

A developer starts with a DAG that specifies the desired computation and references only interfaces. The challenge is to determine how to implement this DAG efficiently for a particular architecture. With DxT, we encode how to implement interfaces with specific algorithms using refinements. A refinement replaces an interface with a graph — referencing lower-level interfaces or primitives — that implements its preconditions and postconditions, possibly in an architecture-specific way.

After repeated refinements, a graph, containing only primitives, can be mapped directly to code. That code implements the starting graph’s functionality, but it might not perform well. Optimization rewrites are subsequently applied. An optimization replaces a subgraph with another subgraph that implement the same functionality, but in a different way. Optimizations can be strung together to improve performance. They could, for example, remove unnecessary data communication or parallelize a loop.

2.2 DxTer and Code Generation

Design knowledge is encoded as refinements and optimizations, so that DxTer (our tool) can automatically generate a high-performance, architecture-specific implementation/code for a given input graph. For each BLAS3 operation, a
A graph containing a single interface (an BLAS3 operation) is input to DxTer. DxTer applies all transformations it can to form a combinatorial search space of implementations of this graph. DxTer rank orders these implementations using cost estimates. The cost of an implementation (graph) is the sum of the costs for all of its nodes. (This mimics the same way experts use to estimate efficiency [16].) The graph with the lowest estimated code (most efficient) is chosen, converted to code, and output.

2.3 Cost Estimates

Execution time is the cost metric for this paper; lower is better. Other metrics might be power consumption or memory usage. Cost estimates need not be perfectly accurate. First-order approximations are sufficient as that is what an expert would use when searching the space manually. For computation operations, we predict performance in terms of the number of floating point operations (FLOPS) multiplied by some constant $\gamma$ representing the number of processor cycles required to compute a FLOP. We use $\gamma = 1$ for convenience.

Multithreaded and single-core code is slowed each time data is brought from main memory to CPU registers (as opposed to using data in cache). This is penalized in the cost model. We chose the cost $\omega n$, where $n$ is the amount of data and $\omega$ characterizes the cost of bringing in that data. $\omega$ is set to $10\gamma$ to make a memory operation more expensive than FLOPS. This rough estimate is good enough to penalize unnecessary data movement even though it does not represent the exact cost of such movement on a real system.

In Section 3, we explain how loops are an essential structure in DLA code. In each iteration of a loop, submatrices are partitioned from input matrices, a computation is performed, and some submatrices are updated. Loops are represented in DxTer by grouping a subgraph of nodes, which represents the loop body [15]. Loops can be nested, where a loop body subgraph contains another loop. Loops are tagged with information such as blocksize (explained below), and nodes represent how submatrices are partitioned from loop inputs. The cost of loops is calculated by summing the loop body nodes’ cost for each iteration, adjusting input submatrix sizes for each iteration.

We demonstrate in Section 4 how DxTer parallelizes loops and nodes. Cost estimates must reflect how runtime decreases when splitting computation across threads. Loops or nodes can be tagged with the way they are parallelized. This tells DxTer how many threads are used, so a node’s computation cost is divided by that number of threads[4]. When loops are parallelized, the input matrices are partitioned to split computation evenly among threads. Then, the loop cost is modeled by summing the cost of one thread’s iterations over the portion of the input for which it is “responsible” to compute.

1We assume for simplicity that work can be evenly divided. Even if this is not the case, the cost model is not so inaccurate that DxTer would choose a “bad” implementation. The cost model mainly ranks an implementation that has more parallelization over one that has less.
3 Sequential BLIS

BLIS [22, 24, 26] represents a new approach to implement the BLAS [4, 5, 13]. As opposed to existing open-source BLAS libraries that require a lot of effort to port, BLIS allows one to implement a small and easy-to-understand set of hardware-specific kernels. With this relatively small effort, one can attain good performance from all BLAS operations for a particular architecture [26]. BLIS’s software and algorithm layering simplifies understandability and portability, a boon to developers. This layering enables us to encode design knowledge to generate code.

BLIS’s sequential BLAS3 implementation follows the performance lessons learned in the GotoBLAS [8, 9]: particular pieces of matrix data are targeted to stay in specific layers of the cache to reduce the cost of reading data. In this section, we describe how we reuse (knowledge of) BLAS3 algorithms encoded for distributed memory to generate high-performance sequential BLIS code following the lessons of GotoBLAS. In Section 4, we explain how that code is parallelized.

3.1 Layering

All BLAS3 operations are implemented in sequential BLIS by layering (nesting) loop-based algorithms that partition input matrices in one or more dimensions, reducing the problem size to some length (called a blocksize). The loop body contains at least one operation or update statement, including a recursive operation applied to smaller matrices. The most common BLAS3 operation is General matrix-matrix multiply (Gemm) that computes $C := AB + C$, where $A$, $B$, and $C$ are matrices. It is implemented in terms of itself on a smaller problem size. Algorithms (loops) are layered until the problem size is small enough to call a given function, as explained below.

We encode the outer three layers of BLIS BLAS3 operations where the remaining, innermost operations are primitives. In BLIS, a primitive is called a macrokernel [24] and generally operates on a processor-specific problem size of around 128 to 1024. It is implemented with more layers of algorithms around an architecture-specific microkernel, which is often coded at a low level, with vector intrinsics or assembly code [24, 26]. A microkernel generally operates on a problem where two dimensions ($m$ and $n$) are small — around four or eight. All layers above the microkernel are reusable across processors, which makes porting BLIS easier: only microkernels must be specialized and loop blocksizes modified per architecture.

3.2 High Performance Gemm

As an example, consider Gemm with the following sizes: $A$ is $m \times k$, $B$ is $k \times n$, and $C$ is $m \times n$. The outer most loop, or layer, decomposes Gemm along the $n$ dimension to blocksize $n_c$. This is depicted in Figure 2 where the top row is the Gemm operation and the next row shows (in red) the loop body Gemm of
the outer most loop, which has size $m \times k \times n_c$. Similarly, an expert layers more algorithms, partitioning in the $k$ and $m$ dimensions with blocksizes $k_c$ and $m_c$, respectively. The result is a Gemm with problem size $m_c \times k_c \times n_c$, which a macrokernel implements. All BLAS3 operations are implemented similarly with algorithms partitioning in the $n$, $k$, and then $m$ dimensions.

Figure 2: Derivation of Gemm. Each row shows the input matrices of the Gemm layer in the outline and the input matrices to the recursive call within the loop in red. The bottom line is sized so the macrokernel implements it.

To attain high performance, the panel of $B$ and block of $A$ shown in the bottom row of Figure 2 are packed. This means the data is reorganized and stored in a physically contiguous, temporary packed buffer that uses a minimal number of translation lookaside buffer (TLB) pages to address [9, 24]. The packed panel of $B$ and block of $A$ are sized to stay in the L3 and L2 caches, respectively, throughout the macrokernel computation. Further, a portion of $B$ is bought into the L1 cache for microkernel computation within the macrokernel. This enables high performance by reducing time spent moving data from main memory and the levels of cache [9, 24].
3.3 Other BLAS3 Operations

Triangular solve with multiple right-hand sides (Trsm) is another BLAS3 operation with eight versions, one of which is Trsm LLN, where there is a lower-triangular matrix that is on the left-hand side and not transposed: \( \mathbf{B} := L^{-1} \mathbf{B} \). It is again implemented with an \( n \), \( k \), and then \( m \) dimension loop built around macrokernel and packing primitives. The \( n \)-dimension loop has a Trsm LLN update, which is implemented with a \( k \)-dimension loop has both a Trsm LLN and Gemm suboperation. Figure 3 shows this loop partitioning, with partitions labeled.

![Figure 3: k-dimension partitioning for Trsm LLN. In each iteration of the loop, the red portions are used in one update and the result of that is used with the blue portions by another update.](image)

The loop body updates are \( \mathbf{B}_1 := L^{-1}_{11} \mathbf{B}_1 \) (Trsm LLN), operating on the red portions, followed by \( \mathbf{B}_2 := \mathbf{B}_2 - L_{21} \mathbf{B}_1 \) (Gemm), operating on the Trsm LLN result (\( \mathbf{B}_1 \)) and the blue portions. The Trsm suboperation is implemented with an algorithm partitioning in the \( m \)-dimension. Portions of \( L_{11} \) are packed for the \( \mathbf{L}_2 \) in each iteration of that \( m \)-dimension loop. \( \mathbf{B}_1 \) is packed and kept in the \( \mathbf{L}_3 \) throughout that computation. The Gemm suboperation is implemented with the same \( m \)-dimension algorithm as used for the Gemm operation, shown in Figure 2. The packed \( \mathbf{B}_1 \) buffer is reused from the Trsm update and portions of \( L_{21} \) are packed for the \( \mathbf{L}_2 \) cache in each iteration of the \( m \)-dimension loop.

All other BLAS3 operations are similarly implemented, partitioning in the \( n \), \( k \), and then \( m \) dimensions. Each loop has just a recursive operation and optionally an additional Gemm operation [11]. This underscores how rote a developer’s job is: Gemm and other loop-body operations are implemented in the same way for all BLAS3 operations, so it is a matter reapplying the same design knowledge. This similarity throughout the BLAS3 enables us to reuse existing design knowledge in DxTer by adding BLIS-specific rewrites.

3.4 Encoding in DxTer

In prior work, we showed how DxTer explored many implementation options, including various ways to partition data in loops to decompose computations into smaller pieces [14, 15, 16]. For BLIS’s sequential BLAS3, algorithms (from a subset of those used in previous work) are layered in a fixed way and suboperations are implemented in only one way. Thus, the search space contains only
Each implementation is derived by DxTer in terms of using a specific series of refinements, progressively exposing n, k, and m-dimension BLAS3 algorithms. Most were encoded for our previous work on distributed-memory systems \[15\]. These refinements replace a BLAS3 operation (e.g., a Gemm node) with a loop-based algorithm that partitions the inputs in one or two dimensions.

Figure 4 shows the Gemm refinements for partitioning matrices in the m, k, and n dimensions, from top to bottom. The outer boxes in the right-hand side represent loops, first introduced to DxT to represent BLAS3 algorithms in \[15\]. Operations within the box are loop body updates. The boxes or ports on the left-hand side of the loop boundaries represent inputs to the loop. A port can either pass the whole input to the loop body for each iteration (e.g., as with B for the top refinement) or it can pass a different partition of the input for each iteration (e.g., a partition of A in the m dimension for for the top refinement).

When ports partition matrices, we label the outgoing edges with the partition index (0, 1, or 2 in these cases). Otherwise, the outgoing edge is labeled with the name as the input data. In DxTer, these are encoded such that ports know the partition direction, the blocksize, and information used for loop fusion \[18\]. DxTer can determine the number of iterations a loop has and the size of inputs for each iteration, which is used for cost analysis.

3.4.1 Layered Architectures and Layered Rewrites

BLAS3 refinements can be reused, as is, in different layers in a DLA layer stack. A layer here means a level of abstraction in hardware or software. At the top of a DLA (hardware) stack is distributed-memory. A layer underneath that might be a layer for shared memory, and below that a layer for sequential memory. To designate when a refinement (or any DxT rewrite for that matter) can be applied within a specific layer, we templatize refinements to stamp out instances that are restricted for use within a particular layer.

Stated another way, DLA stacks require DxT to rewrite operations that are invoked at the top layer of a stack into primitives defined in the bottom-layer of the stack. Templatization allows us to reuse rewrites by customizing them as rewrites from operations on layer n to operations on layer n – 1, thereby enabling us to control how (layered) algorithms are derived. If a different layer ordering is ever desired for a new architecture, one would just change the layers used for instantiating the refinements.

For example, Figure 4b replaces Gemm with a k-dimension loop. To derive the implementation of Figure 2, we apply this refinement to replace a Gemm operation in the n-dimension loop (this Gemm has inputs of size n_x in the n dimension) with an implementation that loops over the k dimension, reducing it in size to k_c. To express this, we instantitate the refinement of Figure 4b such that it replaces Gemm with a LayerN tag (on the left-hand side) with an implementation using

\[\text{LayerN}\]

\[2\]When there are multiple BLAS3 operations in an input graph, DxTer fuses loops, combines redundant packing, and searches reversed loops, so the search space contains more than one implementation.
a LayerK Gemm on the right-hand side. This refinement would then specifically look for Gemm of the correct size and in the correct loop simply by looking at the layer tag and would replace it with the implementation expected for a BLIS software design. These layer tags are not used for distributed memory (other tags are used), so this instantiation is used only for sequential and multithreaded derivations.

3.4.2 Encoding for BLIS

BLIS-specific implementation knowledge is eventually needed to pack data and call a macrokernel. Figure 5 shows a refinement for a LayerK Gemm. Like other BLIS-specific BLAS3 refinements, it is a modified version of an existing refinement (Figure 4a in this case).

Figure 5: BLIS-specific refinement of Gemm.
The implementation in Figure 2 is thus encoded as a series of three refinements. The first adds a loop that partitions in the \( n \) dimension, the second adds a \( k \)-dimension loop, and the third adds the BLIS-specific partitioning in the \( m \)-dimension with packing and macrokernel calls. The first two rewrites were defined earlier in our distributed-memory work, but are now assigned to distinct layers via template instantiation. The third came from copying a previously encoded refinement and adding BLIS-specific implementation details, an easy task. The resulting refinement is used repeatedly to generate the BLAS3 code as \texttt{Gemm} arises in all of the other BLAS3 algorithms.

Similarly for each BLAS3 operation, two existing refinements were reused and one BLIS-specific refinement was added by modifying an existing refinement. Complex and real algorithms are encoded together and specialized on data type (e.g., \texttt{Syrk} and \texttt{Herk} use the same refinements), so no additional work was needed for these as DxTer was already keeping track of datatypes. Further, \texttt{Trsm} and \texttt{Trmm} operations use many of the same algorithms with different suboperations, so they reuse rewrites too.

In total, this amounts to \textbf{nine reused} refinements (two for most operations and three for the combination of \texttt{Trsm} and \texttt{Trmm}) and \textbf{four new refinements}.\footnote{BLIS often reuses macrokernels. For example, \texttt{Syrk} uses the \texttt{Gemm} macrokernel. Further, macrokernel refinements are often the same except for options like transposition that require minimal changes, so the same refinement is used with small case structures for specialization.} This is quantitative characterization of what new knowledge required to target a different architecture and underscores how much knowledge is reused. Encoding knowledge once and reapplying it across architectures is indeed practical.

We also added architecture-specific code generation knowledge. Nodes representing loop structures (e.g., those that partition data) were changed to output BLIS-style variable declaration and loop code instead of Elemental-style. As they are similar with respect to partitioning matrices, this change was easy.

### 3.4.3 Benefits and Perspective

It might seem odd to encode knowledge just to generate the same code that already exists. The payoff is in the next section, where we explain how previously encoded knowledge benefits the exploration of new parallelization options. Even for sequential code, though, DxTer can optimize combinations of BLAS3 operations. The results of these optimizations is modest so far. Having said this, we are in the position to affect greater changes should more significant optimizations be discovered.

One can think of DxTer as a “build-your-own” domain-specific compiler. Here, we have encoded design knowledge to enable DxTer to optimize sequential DLA code. It can then be applied to LAPACK-level algorithms\footnote{BLIS often reuses macrokernels. For example, \texttt{Syrk} uses the \texttt{Gemm} macrokernel. Further, macrokernel refinements are often the same except for options like transposition that require minimal changes, so the same refinement is used with small case structures for specialization.}. Algorithms that use a combination of BLAS3 operations can often be optimized by fusing loops (which DxTer does automatically) and by removing redundant data packing. This is a common manual optimization for code requiring the highest possible performance. It is not typically applied throughout a library of code, though, because it is tedious and makes maintenance harder as the
resulting code is more complicated. With automation, we found these issues are less burdensome as a system is performing this tedious and error-prone work for us.

When we applied DxTer to optimize the BLAS3 operations in LU and QR factorization, it fused loops and removed unnecessary packing. The result was a 1-3% improvement over non-optimized code on an Intel Xeon 7400 processor. This is not a massive improvement, but it was an automatic benefit of encoding design knowledge. We will explore similar optimizations and benefits of DxT for sequential (and parallel) optimizations in the future.

4 Parallel BLIS

We now explain how we used DxT to explore and develop BLAS3 parallelizing heuristics. We used DxT as a productivity enhancer. DxTer’s generated code performs well on some test architectures, but it is not guaranteed optimal — the heuristics need to be augmented for some systems. In the future, new systems might require different parallelization schemes, which would require new heuristics. Again, a developer would add new rules to DxTer (possibly disable unneeded or not-applicable rules) and allow it to use old and new ideas to implement the BLAS3 quickly. This allows an expert to concentrate on ideas for parallelization without being burdened with manually coding all parallel BLAS3 operations each time a new idea is introduced (a tedious task). This will be a boon in the future when developing new parallelization ideas (or even moving to new architectures). Further, if rewrites are correct, they generate correct code. We have found less time is spent debugging our code base.

4.1 The Gemm Heuristic

On multithreaded architectures, threads often share layers of cache. For example, there may be a multisocket system where cores on the same socket share an L3 cache, there are multiple L2 caches per L3, and there are multiple cores sharing each L2. Then, there could be multiple threads per core. Sequential BLAS3 operations are designed to keep a panel of B in L3, block of A in L2, and a sliver of B in L1. With this in mind, a domain expert developed a simple heuristic to parallelize Gemm, the results of which are presented in [22]. The heuristic prescribes: 1) Only one packed block of A or B should reside in the cache to which it is targeted. 2) The threads sharing a layer of cache should collaborate to pack the block targeted to each cache and collaborate to compute with that data.

This heuristic enables data reuse and promotes locality: the threads sharing an L3 cache also share the work on a panel of B. They collaborate to pack it and they split up the computation on it so the multiple iterations of the m-dimension partitioning of Figure 2 are split between the threads in some way. The goal is for the threads to keep their panel of B in their shared L3 throughout computation.
Since only one packed block is kept in each cache, it should remain there in the multithreaded case just as in the single threaded case. With multiple blocks, data would be evicted. With only one block for each cache, the macrokernel attains high performance just as before, but with the benefit of parallelization of the \(m\) and \(n\) loops around it. This heuristic can also be applied to parallelize the \(m\) and \(n\) loops within the macrokernel (loops around the microkernel). Then, a portion of \(B\) is kept in the L1 and a portion of \(A\) in the registers and they are used by the threads sharing those resources.

The lesson here is that DxT rewrites capture not only fundamental implementation relationships between operations and algorithms, but also can encode low-level heuristics that are essential for high-performance parallelization.

4.2 Thread Communicators

With the Gemm heuristic, groups of threads split computation over shared data. We employ thread communicators to control groups of threads in a way inspired by controlling processes with the message-passing interface (MPI) [23].

In MPI, all threads execute the same code and use communicators to determine which portion of each loop they are “responsible” for computing, further explained below. A communicator is created for each L1, L2, and L3, and each thread has a pointer to its caches’ specific communicators.

With MPI Broadcast [23], one process (the root) sends data to all other processes in a communicator. With multithreading, data is accessible by all threads, so Broadcast allows the root thread to send to all other threads a pointer to some data. For example, if all threads are sharing the same packed buffer, one thread allocates the buffer and broadcasts it. Then, all threads work together to pack their buffer. Each thread uses the communicator to get a unique number (similar to a rank in MPI) and the total number of threads in the group (communicator size). This is sufficient for a thread to determine which portion of the data it is responsible for packing; blocks of data are assigned to each. After threads pack, they use a Barrier operation, similar to that in MPI. This ensures that all threads in the communicator have finished packing before any proceed to computation using the packed data.

We model the cost of a barrier by \(10^\gamma p\), where \(p\) is the number of threads participating. This is meant to penalize large barriers over small barriers. It does not need to accurately represent the time spent in barriers when executing code on a given architecture. After all, an expert manually developing the same level of code generally does not need to time barriers. If this is necessary to make the right decision, though, such information could be added to DxTer so it still implements code just as a person would.

All threads in a communicator split the packing of data evenly. To parallelize loops, a hierarchical structure assigns data to groups of threads. For example, a top-level loop might split data to hold in the L3. Multiple threads might

---

\(^4\) The \(k\)-dimension loop is generally not parallelized because each iteration updates all of the \(C\) matrix, so locking and temporary buffers would be required. This limits speedup.
share that L3, so they should share that data and its associated computation. Therefore, the top-level communicator cannot split data among all threads, but rather among the L3s. Figure 6 shows a hierarchical communicator with 2-way splits at each level (e.g., two L2’s per L3).

GlobalComm (in Figure 6) is the communicator that contains all threads. The above Gemm heuristic specifies that it is used to parallelize the n-dimension loop. The computation of this loop is split up so that each L3Comm group (immediate subgroups of GlobalComm in Figure 6, one per L3) gets a block of the B matrix.

4.2.1 Parallel BLIS Code

The hierarchical structure of Figure 6 allows a DLA library programmer to use the function DeterminePart to get the portion of data on which to compute, based on a communicator. DeterminePart uses the number of subcommunicators as the number of partitions for parallelism and the subcommunicator to which the thread belongs as the partition number it is assigned. With DeterminePart, a programmer “shifts” the portion of matrices on which computation is performed in a parallelized loop, where each group of threads gets a different portion of the data.

In addition to changing loops to add parallelism, packing and macrokernel functions were augmented to take a communicator for parallelism. Parallel code calls these variants. The code around a packing call had to be changed, too, so only the root of the parallelizing communicator would allocate a buffer, which is then broadcast. Adding knowledge to parallelize sequential code automatically was much easier than manually modifying all of the packing calls, loops, etc. throughout the BLAS3.

In some cases, packing, loops, or macrokernels are not parallelized to respect loop dependencies, explained below. When this happens, only the root thread of the relevant communicator should perform the operation and then it should broadcast the result (in the case of packing) or all threads should Barrier (in the case of computation). DxTer takes care of such code considerations automatically.
4.3 Encoding in DxTer

The changes required to parallelize BLIS loops were encoded in DxTer as optimizations on sequential code — or rather DxT graph rewrites of DAGs that represent sequential code. Loops, macrokernels, and packing primitives in DxTer graphs can now be tagged with a parallelizing communicator. A parallelizing optimization looks for a loop or node with certain characteristics (i.e., legal to parallelize, described below, and partitioning in the right dimension as the heuristic dictates) and tags it with a specific communicator.

The heuristics to choose which loops to parallelize for Gemm were encoded as rewrite rules so DxTer would do exactly what the expert developer did for Gemm, but automatically, across all BLAS3 operations. Basically, the heuristic is encoded as a combination of parallelizing optimizations chosen specifically to parallelize the n dimension with GlobalComm, the m dimension with L3Comm, etc.

By adding these to DxTer, we enabled it to explore implementations with these parallelization options. Below, we talk about augmented heuristics that add new (parallelizing) options. DxTer explores these in conjunction with existing heuristics. The cost functions penalize idle threads, when extra data is packed, etc., so DxTer can find the “best” combination of parallelized loops for each of the 32 BLAS3 operations.

4.3.1 Valid Parallelism

Loops can be parallelized only if they have independent iterations, meaning iterations do not carry dependencies between them. For example, Gemm m and n loops are independent, but the k loop is not because each iteration changes the same elements of C. DxTer could be augmented to analyze independence of iterations [17]. As DxTer is a research prototype and manual analysis goes very quickly, we manually augmented refinements with tags specifying if loops have independent iterations or not. Tagging all refinements took less than an hour and doing so represented encoding expert knowledge that a person would have needed for manual coding anyway. Now that it is encoded, it can be used for future target architectures such as GPUs. To maintain correctness, parallelizing optimizations do not apply to non-independent-iteration loops.

Parallelizing optimizations must ensure only valid nesting of parallelism. For example, the same communicator cannot be used to parallelize two nested loops, so optimizations can not be applied to create this situation. When an optimization can otherwise be applied to a loop or node, the graph is searched going outward and inward to make sure there is not already parallelism using the same communicator.

Further, a parallel loop must use a communicator higher in the hierarchy than the communicators of all loops within. Again, the optimization searches outward and inward to ensure this requirement.

Other checks, that an expert would use manually, have also been encoded. The benefit here is that they were encoded once, but can be applied automatically any number of times with no further effort, unlike manual development.
4.3.2 Code Generation

Sequential loops iterate over all data. Parallel loops only iterate over the portion they have been assigned, so DxTer’s output code must shift the loop bounds using \texttt{DeterminePart} (to determine each thread’s data portion, as described above). Therefore, DxTer’s loop code generation was augmented to do this. For parallel flavors of primitives, DxTer’s output code had to call a slightly different function that takes the communicator as input. These were quick changes in DxTer (unlike if we had done this manually for all loops and primitives throughout all code).

Further, code was added as needed to call \texttt{Barrier} and \texttt{Broadcast} and to limit a portion of the code to be executed only by a root thread or group: \texttt{if (IAMRoot(GlobalComm))}. Again, we believe these changes took much less time than would have been required to make them manually in code.

These are changes we consider code generation alterations. For example, changing the way loops print code or the primitive that is called do not require new transformations, just changing nodes’ output behavior in DxTer. They are generally easy to make and contain little domain knowledge, just DSL knowledge. Other changes, like those to parallelize loops, actually affect the graph represented in DxTer or how interfaces are refined. They represent domain knowledge about how to implement the starting algorithm on a parallel architecture.

With these modifications, DxTer generated essentially the same code as desired for all four \texttt{Gemm} versions. Next, we started generating code for functionality that did not exist: parallel versions of all other BLAS3 operations.

4.4 Fixing Errors

We generated algorithms for 32 BLAS3 operations (the complex data type flavors come for free from type templatization). DxTer applied the heuristic to all of them in less than one minute. The code was then tested for correctness on random matrices (as is standard for any DLA code). Two errors were quickly found in DxTer-generated code that would likely have been made in manually parallelized code, too. With DxTer, the bugs were fixed in one place and code was re-generated correct instead of requiring all code to be manually analyzed and changed.

First, an implicit assumption was that all levels of the communicator hierarchy would be used. For each parallel loop, data is split among subcommunicators. If one loop is not parallelized, all subcommunicators perform the same computation redundantly. As they all modify the same data, a race condition is created.

\footnote{Consider what happens when \texttt{GlobalComm} is not applied to the outer \texttt{n}-dimension loop as with the \texttt{n}-dimension loops of right-hand side \texttt{Trsm} operations (they do not have independent iterations). The code would have the threads of each of the \texttt{L3Comm} groups all performing the same computation.}
In such a case, only the root subcommunicator should perform computation in the non-parallelized loop. Lower levels of the hierarchy would then provide some parallelism among the threads in that subcommunicator. This bug was fixed by changing DxTer code generation for loops to only have threads in the root subcommunicator (e.g., the first L3Comm) perform computation. This leads to idle threads, which is remedied by an augmented heuristic described below.

After fixing this bug, another remained in Trsm code. In Figure 3, there is a data dependency between the updates Trsm LLN and Gemm. Each is implemented in separate m-dimension loops that cannot be fused. The Trsm LLN m loop does not have independent iterations, so it is not parallelized while the Gemm loop is.

Some of the threads would skip the Trsm LLN since they were not part of the root subcommunicator. They would then start computing Gemm with incomplete data since the Trsm LLN computation was not done. DxTer now checks for such a dependency with different parallelization and adds a Barrier.

After fixing these bugs in DxTer and generating 32 correct BLAS3 function, we tested performance. In some cases, the heuristic described above works well. For others, it was insufficient. Our next task was to augment the heuristics to improve performance.

![Figure 7: Alternate communicator hierarchy.](image)

### 4.5 Adding BLAS3 Heuristics

When a particular loop cannot be parallelized, inner loops should be parallelized to a greater degree instead of having idle threads. For example, when the n loop cannot be parallelized to split data between different L3s, it is better to split the m-dimension across all L2’s, not just those sharing the root L3.

To implement this heuristic, the alternate communicators shown in Figure 7 are used. When the n loop cannot be parallelized (by GlobalComm), the m loop is parallelized by AltGlobalComm so DeterminePart splits computation among all L2s. This heuristic was encoded in DxTer by adding an m-dimension parallelizing optimization with this hierarchy’s GlobalComm.

Once encoded, DxTer explored this new parallelization option for all BLAS3 operations. DxTer deemed it worthwhile to use for all cases with unparallelized n loops. For cases like Gemm, it did not find the resulting implementations better than the previous best because of the additional communication penalized by the cost estimates and more expensive Barrier operations. This added heuristic was

---

*Outer loops cannot be parallelized to a higher degree because this would cause more than one packed block to target the same cache and thrashing would occur.*
automatically applied and evaluated by DxTer without human intervention and improved performance for many BLAS3 implementations. All implementation codes were tested for correctness (they were correct) and manually evaluated to make sure DxTer made the “right” choices in our opinion (it did).

In addition to this alternate hierarchy, we added versions that skip the L2Comm so L1Comm’s are subcommunicators of L3Comm. Further we added a version that skips the L1Comm. DxTer examines all of these options, generates correct code, and generates code using parallelization options that we deem (by subsequent manual inspection) to be the best.

The final improvement fixed load imbalance in Syrk and Syr2k (which also implement the complex flavors Herk and Her2k). Both operations update a triangular matrix C. If the n-dimension loop is parallelized, the subgroup with all rows of C will have much more work than the subgroup with only some of the rows (as the rest of the rows are not updated in a triangular matrix). The solution was to use a new function DeterminePartWeighted that returns unequally-sized partitions. This is always best because one never wants load imbalance, so DxTer was changed to use this function for the appropriate loops for these operations.

4.6 Limited Concerns

We do not allow DxTer to search all possible ways to parallelize all loops. We use expert-designed heuristics to limit choices. DxTer is meant to search just like a person, possibly more exhaustively, but not blindly. Further, it is meant to help an expert develop new heuristics easily, not requiring manual analysis and modification of all code.

One might question why macrokernels are considered primitives instead of microkernels. First, the optimizations applied in microkernels are much more complicated. We expect they could be encoded, but they would be much more difficult and they would not be as reusable across code as the parallelizing optimizations. Further, macrokernel code does not use the software abstractions that allows us to represent higher level code in terms of loops and a handful of primitives. We would expect this if DxTer is to be used. For now, Spiral [20], described below, already demonstrated success in generating code for microkernel-sized computation. Here, we demonstrate success for much larger input matrix problem sizes by taking implementations for small problem sizes as given.

Additionally, encoding design knowledge about the macrokernels is not as useful as knowledge about higher levels. DxTer fuses higher-level loops around the macrokernel when combinations of BLAS3 operations are used together. This creates opportunities for removing redundant packing. It is rare that macrokernels can be fused, and when it is possible, performance often degrades because of cache evictions. Thus, we consider macrokernels primitives for now, but a DxT encoding their design might be future work.
Figure 8: (Left) Speedup of DxTer-generated multithreaded code over sequential BLIS code. (Right) Improvement of DxTer-generated multithreaded code over MKL multithreaded code. The results are from 16 cores of Xeon E5.

4.7 Results

Our most important results come from how quickly we generated BLAS3 code and made far-reaching changes in that code by adding transformations to DxTer. As of this writing, BLIS still only has hand-coded, multithreaded implementations of the four Gemm versions. With DxTer, we were able to generate all other BLAS3 implementations quickly. Here, we present performance results of some of the real BLAS3 operations just to confirm the quality of DxTer’s output code.

Figure 8 shows results with two Intel Xeon E5 (Sandy Bridge) octo-core processors. Each processor has one L3 cache, eight L2 caches, one L1 per L2, and one thread running on the single core attached to the L1. Each core has a peak performance of 21.6 GFLOPS, so the peak of all 16 is 345.6 GFLOPS.

Figure 9 shows results from a system with four Intel Xeon 7400 hexa-core processors. This contains four L3 caches, three L2 per L3, two L1 per L2, and one thread per L1. Each core has a peak performance of 10.6, so the peak of all 24 is 254.4. We compare against Intel MKL [10] version 11.1, which is
Figure 9: (Left) Speedup of DxTer-generated multithreaded code over sequential BLIS code. (Right) Improvement of DxTer-generated multithreaded code over MKL multithreaded code. The results are from 24 cores of Xeon 7400.

the trusted, high-performance, vendor-optimized BLAS library for these architectures. The two architectures, due to different memory hierarchies, require DxTer to generate different amounts of parallelism at various levels.

In the left-side graphs, we show the speedup running on all cores versus running on one for a sample of the real BLAS3 operations across a range of problem sizes. In the right-side graphs shows how that performance outperforms MKL (1 is the same performance and higher is better for DxTer). We only show two variants for each of the real-datatype BLAS3 operations because we had similar results for the others.

MKL performance of Syr2k LN in Figure 9 (top right) is very slow for small problem sizes, so DxTer’s code’s performance looks especially good by comparison. DxTer’s Syrk and Syr2k code does not speed up well in Figure 9 (left). This is likely a load-balancing issue that requires additional heuristics, which we will add to DxTer in the future. Still, these DxT implementations perform well compared to MKL. These results demonstrate that multithreaded code generated by DxTer attains good speedup and roughly matches or outperforms MKL.
5 Related Work

Code generation projects such as SPIRAL [20] and the BTO BLAS [2] similarly start with high level algorithm specifications and transform to lower-level code. Their transformations search a much larger space of options and code performance is less predictable, so they require empirical timing feedback to search for the best implementations. They benefit from optimizing much smaller input problem sizes than we target. Our work is different in that we aid an expert engineer who knows what he is doing. He has intuition and ideas of what will be best, but does not have the patience or time or self-trust to do so for all the necessary functions well and consistently. DxT provides a mechanism to encode that design knowledge and have it automatically applied across operations, alleviating his burden. By focusing on the BLAS3, timing code has not been necessary to judge the quality of implementations. These approaches could be used to generate the microkernel (or macrokernel) and then DxTer would generate code using their microkernel.

The linear algebra compiler by Fabregat-Traver and Bientinesi [6] takes a specification of a mathematical operation and derives a family of algorithmic variants. It optimizes the algorithms by reusing variables and mapping to BLAS function calls. The output of this system produces the type of algorithms we use as input to DxTer. DxTer then optimizes the implementation by parallelizing for multithreading or distributed memory.

Autotuning is often viewed as a way to improve performance automatically. DxT generates its search space from a high-level specification of algorithms and how they can be implemented. We use cost estimates that guide us to the best implementation(s). ATLAS [25], for example, searches a relatively limited number of algorithmic options and largely tunes factors such as algorithmic blocksize and unrolling factors that are best chosen after the options given to DxTer (or a person) are explored. We see utility in adding autotuning to this approach in the future to choose the best parameters such as, for example, the algorithmic blocksize, after code is generated.

SuperMatrix [3] and PLASMA [12] are DLA runtime schedulers that use sequential code to form a task graph that is scheduled on the cores of a CPU (and/or co-processor). They attempt to optimize an execution schedule by applying heuristics to reduce communication between cores and to maximize reuse of data in cache. These heuristics are less aware of the specific operation than those we use, and instead apply to patterns in the task graph. By applying more specific heuristics, we avoid the need for runtime schedulers. If a scheduler is desired, DxTer code generation could be changed to output SuperMatrix scheduler code from the starting graph. Then, the algorithm knowledge already in DxTer would not be wasted when targeting new software.
Retargeting BLAS3 algorithms to new architectures is a non-trivial task. We explained how we retargeted a set of rewrite rules that generated BLAS3 algorithms for distributed memory to now generate efficient code for sequential and shared-memory machines.

One BLAS3 operation (Gemm) was hand-written and optimized by experts for the target machines. We then re-engineered this code to derive it as a sequence of DxT rewrites, starting first with sequential code and then parallelizing it. Once we got the ‘general idea’, we were able to retarget our rewrites for distributed-machine to rewrites for a sequential machine and then rewrites for a shared memory machine. This was not difficult to do when cast in a sufficiently general way, which we explained.

We did not know how to parallelize all 32 BLAS functions for these target architectures, so we used DxTer as a productivity enhancer. Each time we had a new idea, it was encoded in DxTer and all 32 functions were re-generated within a minute. This is significantly faster (and easier) than what would have been required for manual development. The code performed well against MKL, but more importantly it was generated in much less time (and with fewer bugs) than by a person developing the code manually. In effect, MKL was developed by a team of experts over many years; the automation of DxT with good DSLs allows an individual expert to compete with teams in far less time.

We do not expect the heuristics we presented to perform well across all architectures. However, the speed with which a developer can add new heuristics to DxTer gives us hope that heuristic development will be easier in the future.

We believe DxT will be especially useful for other scientific computing libraries, where lots of related code has to be parallelized in different ways. Using a system to explore options quickly and generate a library of code offers a vast improvement over conventional approaches involving manual, rote development. As DxT is new, the work we have described is meant to be a preliminary proof of concept, a beginning to many more case studies across hardware architectures and domains.

Acknowledgements We gratefully acknowledge support for this work by NSF grants CCF-0724979, CCF-0917167, and ACI-1148125. Marker held fellowships from Sandia National Laboratories and the NSF (grant DGE-1110007). This research used resources of the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. We thank Francisco Igual Peña for his microkernels.

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


