Understanding Performance Stairs: Elucidating Heuristics

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

ABSTRACT

How do experts navigate the huge space of implementations for a given specification to find an efficient choice with minimal searching? Answer: They use "heuristics" – rules of thumb that are more street wisdom than scientific fact. We provide a scientific justification for *Dense Linear Algebra (DLA)* heuristics by showing that only a few decisions (out of many possible) are critical to performance; once these decisions are made, the die is cast and only relatively minor performance improvements are possible. The (implementation $\times$ performance) space of DLA is stair-stepped. Each stair is a set of implementations with very similar performance and (surprisingly) share key design decision(s). High-performance stairs align with heuristics that prescribe certain decisions in a particular context. Stairs also tell us how to tailor the search engine of a DLA code generator to reduce the time it needs to find implementations that are as good or better than those crafted by experts.

Categories and Subject Descriptors

D.1.2 [**Automatic Programming**]; D.1.3 [**Concurrent Programming**]; G.4 [**Mathematical Software**]: Efficiency; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering*

General Terms

Design, Performance.

Keywords

program generation; dense linear algebra; high-performance software; distributed-memory computing; model driven engineering

1. INTRODUCTION

Writing high-performance software requires considerable skill. In immature domains, programmers not only have to know *what* to code, but *how* to code; they shoulder the burden of both application design and attaining high performance.

In mature domains, standard *application programming interfaces (APIs)* are used: programmers code *what* they want in terms of these APIs, while experts bear the burden of *how* to implement APIs so that user applications run efficiently. This separation of concerns has generally worked well: it is the basis of modern software engineering tools called middleware.

The task of an expert is non-trivial. There may be hundreds or thousands of possible ways to implement each API operation. An expert must (at least intuitively) be familiar with them all. Yet, experts explore only a few possible ways, using insights to navigate myriad prospects to find good, if not optimal, implementations quickly. Such insights are *heuristics*; they are intuitive guides that experts learn over time. Why heuristics work is more street wisdom than scientific fact.

We found ourselves in a unique position to relate heuristics to scientific insight. Our research automates the development of *Dense Linear Algebra (DLA)* libraries – libraries that are defined by standard APIs and that are coded by experts per hardware architecture. DLA experts shoulder the burden of writing high-performance library routines; they must know how to code efficient algorithms, how to customize algorithms for target architectures, and how to navigate huge implementation spaces quickly to find the highest-performing algorithms. We have automated the exploration of these spaces (by generating all implementations using a methodical process) and we evaluate the efficiency of each implementation via cost estimation.[1] This is how we find the best-performing algorithm that experts would intuitively select [16, 17, 18].

We discovered that the space of (implementation $\times$ performance) is not smooth but stair-stepped; each "stair" represents a set of implementations that perform approximately the same and – surprisingly – that share design decisions. That is, a few design decisions (out of many possible) are critical to performance. Once these decisions have been made, the die is cast and only relatively minor performance improvements are subsequently possible: the important decisions relate to algorithm selection and parallelization while the less important ones optimize data communication. Of course, some "stairs" overlap, where the performance difference between key decisions is minimal, and secondary decisions rise to significance.

Heuristics align with combinations of decisions that eliminate outright poor-performing designs. *To exploit this observation requires design knowledge to be modularized by design decisions, so that good decisions (or decision combinations) can be distinguished from bad decisions (decision combinations).* We modularize such decisions in DLA by *transformations*, not code modules.

In this paper, we explain how tools can be built based on these ideas and show how to exploit the staired structure of an (implementations $\times$ performance) space to reduce the number of implementa-

[1] Runtimes can be quite long and require thousands of cores on expensive machines, so empirical timing data is not viable to judge implementation efficiency. Estimates must be used.

tions to explore. For complicated algorithms with intractably large search spaces, this reduces search time to less than an hour and the final code that it produces is as good as or better than that produced by an expert.

The main contributions of this paper are:

(1) Our explanation of how an expert effectively codes a high-performance algorithm without spending months exploring options.

(2) We use two classes of transformations – refinements and optimizations – and now further classify refinements in two ways, "variant refinements" and "rephrasing refinements" (explained below) so we can identify which DLA design decisions are important. We identified 54 variant refinements and 53 rephrasing refinements.

(3) Our evidence of how stairs of an (implementation × performance) space can be exploited to effectively reduce the search for efficient implementations.

We conjecture that our findings are representative of other domains, where our analysis and approach could be applied. Admittedly this will take more time to prove (or disprove). This paper presents a valuable data point in this quest: the domain of DLA.

We begin with a brief overview of how we generate the space of implementations for a given operation in the domain. Our approach is called *Design by Transformation (DxT)* – more details are given in [8, 16, 17, 18, 25].

2. DESIGN BY TRANSFORMATION

Consider the universe of dataflow applications. Each application is represented by a *directed, acyclic graph (DAG)*. Every node or *box* denotes an *operation*. Operation inputs and outputs are incoming and outgoing edges, respectively. Operations can be either an interface or a primitive. An *interface* has no implementation details, other than a name, inputs, outputs, preconditions and postconditions [10]. A *primitive* additionally has a given code implementation (*eg* an API function call).

An expert starts with a DAG that specifies a desired algorithm independent of implementation or hardware-architecture specifics. Usually, this DAG contains only interfaces. We encode design knowledge of how to implement interfaces using refinements. A *refinement* is a transformation that replaces an interface with a graph (with lower-level interfaces or primitives) implementing that interface. It might use architecture-specific operations or algorithms. For example, a refinement can encode how to parallelize an operation.

Refinements are applied repeatedly until the graph contains only primitives. This represents an implementation of the starting graph, but not necessarily the most efficient implementation. Optimizations can yield higher performance. An *optimization* is a transformation that replaces a subgraph with another subgraph that implements the same functionality but in a different way. An expert generally applies a sequence of optimizations to improve performance.

DxTer is our tool that automatically applies refinements and optimizations to a user-specified graph. It relies on a knowledge base of transformations that we have mined from re-engineered DLA code. DxTer applies all transformations that it can to form a space of all possible implementations of the starting graph. It then rank orders these implementations using cost estimates. Each primitive (which represents a function call) has a cost estimate based on the data (matrix) sizes input to the node and the target hardware (*eg* an estimate of cost to move data between cores).

For distributed-memory DLA, first-order cost estimates are sufficient [16, 17, 18] to enable an expert to judge trade offs between the cost of communicating data over a network and increasing parallelism that is enabled by that communication. Just as an expert estimates efficiency when manually coding, DxTer does so automatically by summing the estimated runtime of all nodes on a graph [18]. Then, the graph with the lowest estimated cost is converted to code and output by DxTer.

3. BIG PICTURE

Classical development of an algorithm starts with a specification `S`, and through a series of design decisions – which we represent as refinements and optimizations – an algorithm a_1 is produced. The derivation of a_1 is represented as a path from `S` to a_1 (Figure 1(a)). In general, a large space of algorithms may exist for `S`. This space is encoded as a *decision* or *derivation tree* [2]; a tiny tree of three derivations is shown in Figure 1(a). All of the derived algorithms $\{a_1, a_{17}, a_{21}\}$ implement `S`, but do so differently and have different performance. It is the task of a domain expert to intuitively navigate this tree to find the best performing algorithm for a given situation.

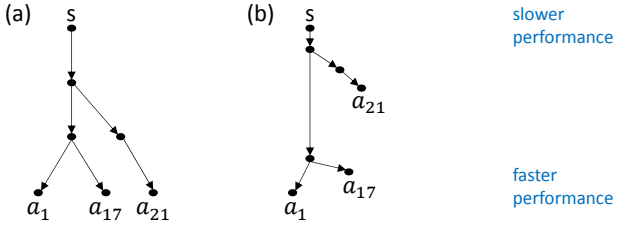

Figure 1: Tree of Algorithm Derivations from a Spec `S`.

As derivation trees go, Figure 1(a) is as typical as it is misleading. Its arrows (transformations) suggest that all are of equal importance. This is not the case – certainly not for DLA. Some arrows (design decisions) make a *big* difference in the ultimate runtime performance of an algorithm; most arrows merely tweak performance up or down. Figure 1(b) more realistically emphasizes the importance of decisions with respect to performance. The longer the arrow, the more significant is its impact on performance. The higher the algorithm's placement in Figure 1(b), the slower it executes (lower is better). It is the responsibility of a domain expert to know intuitively the arrows/decisions that are performance-significant from those that are not. This knowledge is rarely written down; it is an art mastered by domain experts. As we demonstrate throughout this paper, in DLA choices of how to parallelize operations (*ie* refinements) make a substantially greater impact on performance than subsequent optimizations.

> **Note:** The trees of Figure 1 are specific for a particular context such as matrix size or architecture. Change the context, algorithm a_{21} may become the fastest. By no means do the arrows of derivation trees have a fixed length or direction – this extra dimension of "context" (*eg* what are the inputs to the operations whose algorithms are to be optimized) significantly complicates the job of an expert and any generative tool.

Suppose we could plot algorithms *vs* runtime on a graph. Points along the `x`-axis are algorithms that are derived from spec `S`; the `y`-axis indicates algorithm runtime where lower (faster) is better. By sorting algorithms on the `x`-axis from poorest-performing to best-performing, one would imagine a graph like Figure 2(a), which has a smooth flowing curve; the best algorithms are at the far right.

But a plot of actual (algorithm,runtime) pairs yields something quite different: it is a discrete set of stairs as in Figure 2(b), where each stair represents variations of a set of algorithms that share a

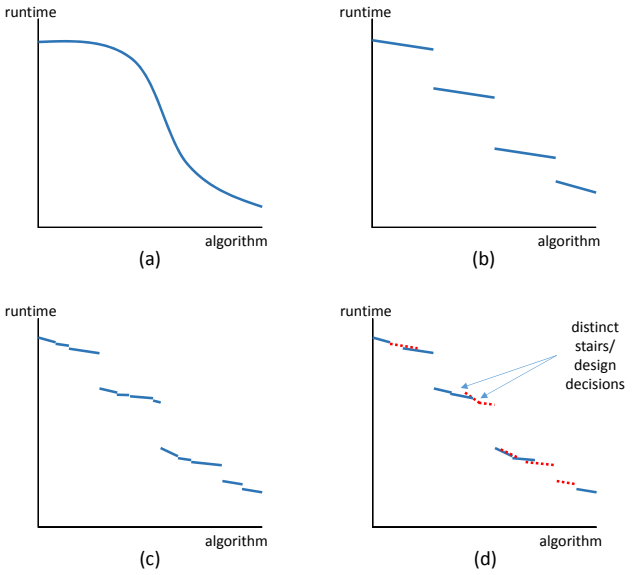

Figure 2: Shapes of Performance Space.

major design decision (*eg* a long arrow in Figure 1(b)). And like a fractal [15], a single stair internally is a set of closely performing stairs as in Figure 2(c), recursively, representing the progressively smaller effects of less-important design decisions. Distinct stairs can overlap as in Figure 2(d), where fundamentally similar refinements compete for the best performance. This is the shape of a performance space that experts intuitively navigate – they use their intuition about the relationship between design decisions and the stair(s) whose algorithms they explore, even though they may be completely unaware of the stair nature of their space.

As we are among the first to be able to generate a space of algorithms that implement a specification (at least in DLA), we can now "see" the entire performance space of which experts could only glimpse a few points at a time. This enables us to connect specific refinements to overall algorithm performance.

The following sections document concrete instances of these ideas in the domain of distributed-memory DLA.

4. DLA AND ELEMENTAL BACKGROUND

Elemental [22, 23] is a distributed-memory DLA library with functionality similar to ScaLAPACK [5]. It includes implementations of common high-level functionalities such as matrix factorization, eigensolvers, and solvers for systems of equations. Elemental also includes an API of primitives that are used to implement these high-level functionalities.

In prior work [16, 17, 18], we demonstrated how design knowledge for Elemental is encoded in DxT. We start with interfaces that represent sequential and architecture-agnostic DLA operations. We encode Elemental-specific transformations that parallelize and optimize algorithms. DxTer explores the implementation space for each input graph (a "spec" in the vernacular of Section 3) to generate a graph that references only primitives, which map to library calls.

Elemental views the $\mathtt{p}$ processes in a computer cluster as a 2D *process grid* ($\mathtt{p} = \mathtt{r} \times \mathtt{c}$) of $\mathtt{r}$ rows and $\mathtt{c}$ columns.[2] There is a default way to distribute matrix data on the process grid. This is a 2D cyclic distribution, which maps matrix element $(\mathtt{i}, \mathtt{j})$ to process $(\mathtt{i}\%\mathtt{r}, \mathtt{j}\%\mathtt{c})$. There are ten other distributions of interest; each has its own notation (*eg* the default denoted by $[\mathtt{M_C}, \mathtt{M_R}]$) [22]. Elemental uses a *single-program, multiple-data (SPMD)* programming model. All processes run the same program, but have different portions of the data on which they compute. The various data distributions enable this.

DLA algorithms are largely loop-based. In each iteration, input and output matrices are partitioned into submatrices. Loop-body operations, or *update statements*, perform computation using the submatrices and overwrite submatrices.

Loop updates are parallelized by *redistributing* data[3] from the default $[\mathtt{M_C}, \mathtt{M_R}]$ to other distributions in a way that enables computation to be performed in parallel across the $\mathtt{p}$ processes. The result is then redistributed back to the default distribution as needed, possibly with a reduction such as a summation of partial results.

Communication or data redistribution is expensive, but it enables parallelism. An expert balances the overhead of communicating data with improved parallelism. With larger matrices, the amount of computation, generally $\mathtt{O}(\mathtt{M}^3)$ where $\mathtt{M}$ is the matrix size, is much greater than with smaller matrices. Communication, an $\mathtt{O}(\mathtt{M}^2)$ cost, is less of a concern with large matrices, but it must still be chosen carefully. Depending on matrix sizes, an expert chooses among different parallelization schemes which incur different communication costs and provide different amounts of parallelism. For example, an expert might choose redundant computation with small matrices because the communication cost is reduced and more parallelism is not needed for a small amount of computation. The heuristics we study in this paper balance these details for particular operations.

Let's take a look at some of the deep knowledge that experts rely on when manually developing distributed-memory DLA algorithms.[4] We start with Cholesky factorization, a comparatively simple algorithm, and later examine the more complicated *symmetric positive definite (SPD)* inversion and even more complicated operations *two-sided Trsm* and *two-sided Trmm*.

> **Note:** We realize that few readers are familiar with DLA. We try to balance our discussions so that both DLA experts and DLA novices can appreciate our results. Novices should *not* focus on DLA details but rather on named rewrites and the performance that using them brings.

> **Note:** Although we (and our tools) derive DLA algorithms by transformation, it is unclear whether experts think in terms of transformations. The disconnect between algorithm derivation by machine and algorithm invention by humans is an interesting subject we are exploring and whose results are beyond the scope of this paper.

5. CHOLESKY FACTORIZATION

5.1 Parallelizing Cholesky

Cholesky factorization is an algorithm to compute the Cholesky factor $\mathtt{L}$ for a *symmetric, positive-definite (SPD)* matrix $\mathtt{A}$ such that $\mathtt{A} = \mathtt{L}\mathtt{L}^T$. Here, $\mathtt{A}$ is overwritten by $\mathtt{L}$ to reduce memory use. There are five refinements of Cholesky used in Elemental and in this paper. Three map a Cholesky operation on a large matrix $\mathtt{A}$ to a series of

[2] $\mathtt{r}$ and $\mathtt{c}$ are parameters tuned at runtime based on how many processes are available for execution. Their choice does not affect which final implementation code is used.

[3] Redistribution is implemented with collective communication found in the *Message Passing Interface (MPI)* [27]. The cost of collective communication is estimated by the lower-bounds given in [6].

[4] The knowledge we present largely came from backwards engineering existing code in terms of refinements and optimizations [18, 17, 16].

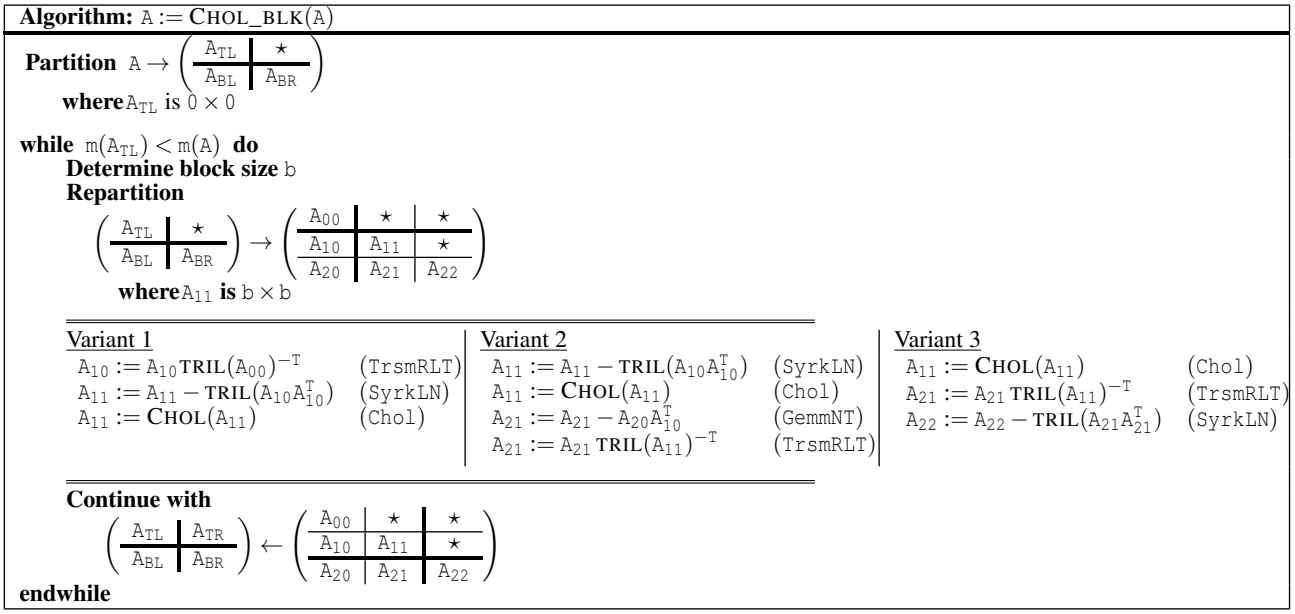

Figure 3: Three algorithms for computing the Cholesky factorization. `m(A)` stands for the number of rows of `A` and `TRIL(A)` indicates the lower triangular part of `A`. The '⋆' symbol denotes entries that are not referenced. The *Partition*, *Repartition*, and *Continue with* operations effectively partitioning the matrix `A` into submatrices, the boundaries of which change with each iteration of the loop. These details are unimportant as the loop body is our focus.

Cholesky operations on smaller submatrices. These refinements are called (for historical reasons) *variants* [28]. The fourth and fifth refinements are different: one is a primitive that directly implements Cholesky; the other rephrases a Cholesky operation in terms of another (Cholesky) operation that uses a different (matrix) data distribution. We call this latter a *rephrasing* refinement. Figure 4 sketches the basic Cholesky rewrite rules that we use in this paper.

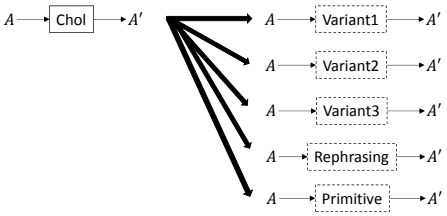

Figure 4: Cholesky Rewrites Used in Elemental.

Now, let's look closer. Figure 3 shows the three variant refinements for Cholesky. Each algorithm computes the same factor, but does so with different computation (updates) in the loop. In parentheses, we show the names given to each individual update. A DLA expert will recognize them. For everyone else, they are simply DxT interface nodes.

Each update is a commonly-encountered DLA operation, so it is represented by an interface in DxT. The updates in Variant 3, for example, are `Chol`, `TrsmRLT`, and `SyrkLN`, from top to bottom. `SyrkLN` can have multiple parallelized implementations, but only one is considered in this case because of the structure of the operands (we will not detail this low-level domain knowledge). `Chol` has only one parallelization scheme in Elemental (one refinement to parallelize the loop body).

Notice there is recursion in the Cholesky variant refinements of Figure 3 – all have a `Chol` operation in the loop body. This is expected, as mentioned above, since variants map a Cholesky operation on a large matrix to a Cholesky operation on smaller (sub)-matrices. These "smaller" Cholesky operations can be implemented as a primitive or by a refinement that rephrases the operation using another Cholesky operation that expects a different data distribution (*ie* parallelizes it for distributed memory). Figure 5 shows one such refinement that rephrases `Chol` into `LChol`, an operation that is later transformed with an implementation that calls a `Chol` function provided by an external library. In a Elemental, one does not use a variant implementation for this "smaller" Cholesky operation because the result would be an even-smaller Cholesky with matrix sizes that are too small – communication overhead would be too great.[5] One uses a variant implementation and then parallelizes the loop-body operations with rephrasing refinements.

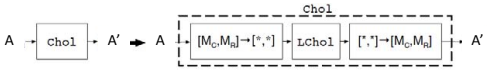

Figure 5: A Rephrasing of `Chol` Operation in Terms of `LChol`.

> **Note:** Each iteration of a variant's loop completes before the next iteration starts, so there is no parallelism between iterations. Within each iteration, parallelism is attained by distributing the loop-body operations' computation across the process grid. The abstractions used in Elemental (shown in our rephrasing transformations) hide this parallelism somewhat. Each box is executed on all processes in parallel and different data distribution choices determine which processes work on which portion of the overall computation.

[5] Recursive applications of variant refinements *do* arise in other DLA libraries (e.g., BLIS [29]), but *not* in Elemental.

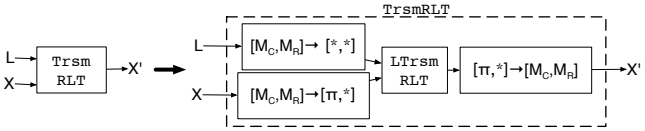

Figure 6: `TrsmRLT` refinements with $\Pi \in \{V_C, V_R, M_C, M_R, *\}$.

In contrast to the single rephrasing options for `SyrkLN` and `Chol`, `TrsmRLT` has five refinements. Figure 6 shows them in a templatized form. The templatization parameter Π is instantiated such that $[\Pi, *]$ represents an Elemental distribution. Boxes labeled $A \rightarrow B$ transform data in distribution `A` to distribution `B`. `LTrsmRLT` is a primitive that performs a local computation in parallel across processes (since that box is executed on all processes).

The choice of Π leads to different amounts of communication and parallelism. An expert explores these options based partly on the problem size and knows (from experience looking at cost estimates repeatedly) that $\Pi = V_C$ or V_R is best for large problem sizes[6] since they lead to parallelism where each process performs a different portion of the computation. $\Pi = M_C$ or M_R results in a process row or column performing redundant computation, with different process rows or columns computing different portions in parallel. Due to the amount of computation, these are good choices for a moderate but not large amount of data. A choice of $\Pi = *$ results in all processes computing the same result, so it is best when there is a small amount of data.

We call this the *Trsm heuristic*: choose $\Pi = V_C$ or V_R for large problem sizes, $\Pi = M_C$ or M_R for medium problem sizes, and $\Pi = *$ for small problem sizes. We will see the correlation of this heuristic to performance stairs in the next section.

> **Note:** We name and summarize two heuristics in this paper. There are similar heuristics that apply to each interface of the widely-used *level-3 Basic Linear Algebra Subprograms (BLAS3)* API [7, 17]. All of these heuristics are similar to the two we name here, all form stairs like those we describe below, and all work with the generalized heuristic we present in Section 6.2. `Trsm`, for example, has another heuristic very similar to the one we show that is used for a different flavor of the operation ($B := \text{TRIL}(L)^{-T}B$ or `TrsmLLT`).

Besides parallelizing computations, different values of Π enable different optimizations to be subsequently applied. We now illustrate a simple (and rather typical) optimization that could be applied depending on the Π value chosen for the `Trsm` refinement. An expert explores alternate ways to implement one redistribution as a series of others or to remove redundant redistributions as in the templatized optimization of Figure 7. More optimizations and refinements are given in [16, 17, 18].

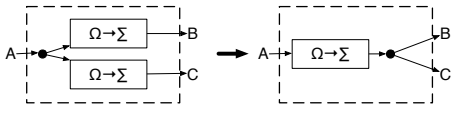

Figure 7: Templatized optimization to remove a redundant $\Omega \rightarrow \Sigma$ redistribution.

For large problem sizes, the Trsm heuristic tells us to choose V_C or V_R to refine `TrsmRLT`. The choice between these two depends on what optimization can be subsequently applied given how data is already distributed.

[6]The idea of what is "large" or "small" is rough and depends on the number of processes, so we do not provide specific numbers here.

> **Note:** We do not detail optimizations further because they are unimportant to understand the structure of the DLA (implementation $\times$ performance) space. Optimizations can incrementally improve performance, but refinements make more of an impact.

5.2 Cholesky Stairs

We use `k`-means clustering, explained below, on DxTer's explicitly enumerated search space to reveal the performance stairs of Cholesky algorithms. We identify stairs with known heuristics that limit the implementations that an expert would explore.

5.2.1 Variant 3

Figure 8 shows the performance of all Variant 3 Cholesky implementations ordered by performance[7] and numbered along the horizontal axis. The vertical axis predicts performance in cycles, so lower is better.[8] Notice that all graphs use a logarithmic scale on the vertical (runtime) axis: there is a factor of $33\times$ difference from the worst algorithm (far left) to the most efficient algorithm (far right).

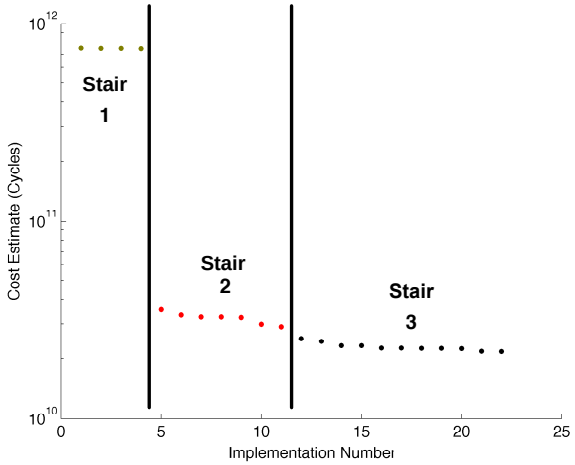

Figure 8: Predicted performance of the Cholesky Variant 3 implementations, clustered with $k = 3$.

k-means clustering partitions data (implementation costs here) into `k` clusters to minimize $\sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|$ where C_i is the i^{th} cluster, x_j is the j^{th} piece of data in C_i, and μ_i is the mean of the data in C_i. We used 3-means clustering in Figure 8. The implementation clusters (henceforth "stairs") that arise are linked to the three option groups for refining `TrsmRLT`.

DxTer keeps track of the transformations used to generate a given implementation, so we can look at which transformations are common to all implementations within a stair. For Stair 1, all implementations use the `TrsmRLT` refinement with $\Pi = *$. The Trsm heuristic tells us this refinement is the worst for large problem sizes – which Figure 8 clearly indicates. There are four implementations in this stair with varying performance due to optimizations that were applied. This is an important feature of distributed-memory DLA: *refinement choices impact performance greatly while optimizations simply tweak performance after refinement decisions have been made*.

Similarly, Stair 2 contains implementations that use $\Pi = M_C$ or M_R, and Stair 3 contains implementations with $\Pi = V_C$ or V_R. Figure 9

[7]DxTer implements and optimizes an algorithm given a specific problem; we use 80,000 here, which is large.

[8] Predicting the number of cycles does not necessarily reflect the actual number of cycles at runtime, but rather is an estimate to enable ordering.

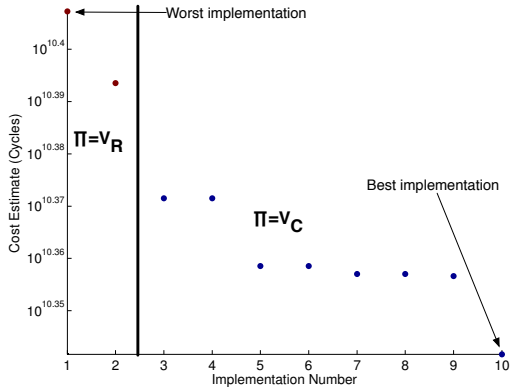

Figure 9: Close-up of Stair 3 from Figure 8.

shows these "micro-stairs" within Stair 3, where $\Pi = \mathtt{V_R}$ leads to two implementations and $\Pi = \mathtt{V_C}$ leads to eight. Again, there is lower-order cost variation within stairs due to optimizations that are subsequently applied.

5.2.2 *Variant 2*

Cholesky Variant 2 also has a `TrsmRLT` interface that is again best refined (for large problem sizes) using $\Pi = \mathtt{V_C}$ or $\mathtt{V_R}$, as prescribed by the Trsm heuristic. The third update ($\mathtt{A_{21}} := \mathtt{A_{21}} - \mathtt{A_{20}}\mathtt{A_{10}}^T$) is a `GemmNT` operation that can be refined (parallelized) in one of three ways, shown in Figure 10.

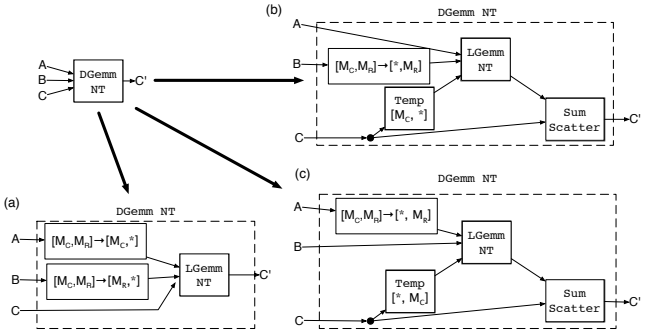

Figure 10: Refinements of `GemmNT`.

Experts use the *Gemm heuristic*: Whichever input matrix is largest (`A`, `B`, or `C` in Figure 10) should be kept *stationary*, meaning it stays in the default distribution. This limits their options to only one refinement.

For the third update of Cholesky Variant 2, $\mathtt{A_{20}}$ is much larger than the other operands, so an expert employs the Gemm heuristic to keep $\mathtt{A_{20}}$ stationary. The refinement of Figure 10 (c) does this; the others redistribute $\mathtt{A_{20}}$.

Figure 11 shows the predicted performance of all Variant 2 implementations, clustered with $\mathtt{k} = 6$. Stairs 4-6 all use the `GemmNT` refinement that keeps the largest matrix ($\mathtt{A_{20}}$) stationary (as prescribed by the Gemm heuristic). Stairs 1-3 have implementations that use one of the other two refinements that violate the Gemm heuristic. Further, within Stairs 1-3 and Stairs 4-6, the difference among the three stairs in each is due to which `TrsmRLT` refinement is used (where an expert would employ the Trsm heuristic).

This is a great example of how two independent heuristics "compose". The `GemmNT` refinement tells an expert to limit his consideration to only Stairs 4-6 and then the `TrsmRLT` refinement tells him to limit consideration to Stair 6. These two orthogonal decisions reduce the search of an implementation space considerably – to one-tenth of the entire space.

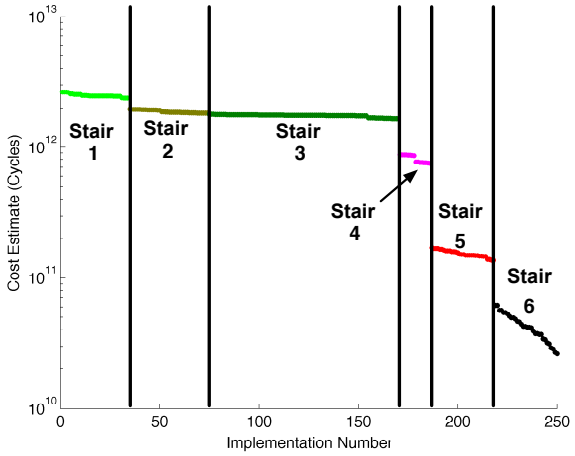

Figure 11: Predicted performance of the Cholesky Variant 2 implementations, clustered with $\mathtt{k} = 6$.

The Cholesky update (the second update of Variant 2) has only one refinement. The first update ($\mathtt{A_{11}} := \mathtt{A_{11}} - \mathrm{TRIL}(\mathtt{A_{10}}\mathtt{A_{10}}^T)$, called `SyrkLN`, has four refinements. Because this operation does not account for much computation (*ie* smaller orders of magnitude than the `GemmNT` and `TrsmRLT` updates), the choice between these four makes little impact on performance. This means the four choices do not form four clearly visible stairs within the six stairs of Figure 11, but rather look as if they all belong to a single stair.

5.3 Bringing All Variants Together

Figure 12 shows predicted (implementation × performance) space for all three Cholesky variants (294 implementations total). Notice that the worst performing implementation runs for an estimated 13,842-times as many cycles as the best.

All Variant 1 implementations are in Stairs 1-3 (the worst performing). Variant 1 is known to perform badly because none of the five `TrsmRLT` refinement parallelize well with a large `L` input, which is what Variant 1 uses.

The remaining stairs (4-10) come from the union of Variant 2 and 3 implementations (*ie* the overlaying of Figures 8 and 11).

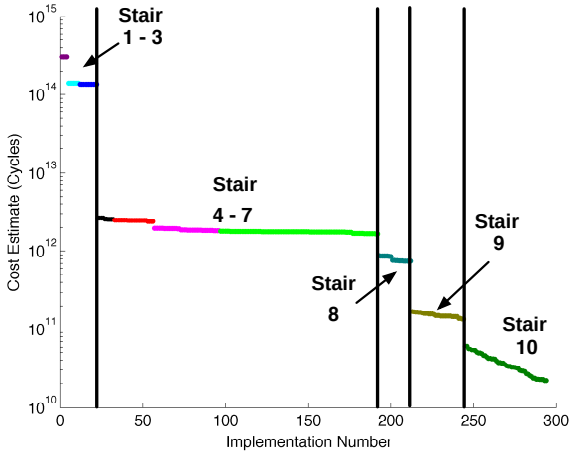

Figure 12: Predicted performance of all Cholesky implementations, clustered with $\mathtt{k} = 10$.

Stair 10 contains 50 implementations which are a union of Variant 3 implementations of Stair 3 in Figure 8 and Variant 2 implementations of Stair 6 in Figure 11. In other words, a decision between implementing Variant 2 or Variant 3 is not immediately obvious.There

is no heuristic to tell an expert quickly that Variant 2 or 3 is better; he must explore both. This graph shows why: performance of the best implementations of each variant are similar.

6. SPD INVERSION

6.1 Combinatorial Explosion

While Cholesky factorization is a prototypical DLA operation, it is relatively simple compared to most operations supported by Elemental. For example, SPD Inversion [4] takes an SPD matrix A, computes 1) its Cholesky factor L, 2) L^{-1}, and then 3) L^TL (A is overwritten with the result of each of these three operations). Each operation can be implemented independently. For example, one can use any implementation of Cholesky from the previous section for the first operation of SPD Inversion. However, an expert can do better. By choosing the right algorithmic variant for each operation, an expert can fuse loops [4, 19]. This enables further optimization on the fused loop body, shown in Figure 13 with interface names in parenthesis. DxTer fuses loops automatically as an optimization transformation when it is legal to do so [19].

$$
\begin{array}{ll}
A_{11} := \text{Chol}(A_{11}) & (\text{Chol}) \\
A_{01} := A_{01}A_{11}^{-1} & (\text{TrsmRLT}) \\
A_{00} := A_{00} + A_{01}A_{01}^{T} & (\text{SyrkLN}) \\
A_{12} := A_{11}^{-T}A_{12} & (\text{TrsmLLT}) \\
A_{02} := A_{02} - A_{01}A_{12} & (\text{GemmNN}) \\
A_{22} := A_{22} - A_{12}^{T}A_{12} & (\text{SyrkLT}) \\
A_{01} := A_{01}A_{11}^{-T} & (\text{TrsmRLT}) \\
A_{12} := -A_{11}^{-1}A_{12} & (\text{TrsmLLN}) \\
A_{11} := A_{11}^{-1} & (\text{TriInv}) \\
A_{11} := A_{11}A_{11}^{T} & (\text{Trtrmm})
\end{array}
$$

Figure 13: SPD inversion loop body.

There are two TrsmRLT updates and two TrsmLLT updates, which have similar refinement options and stairs; the Trsm heuristic applies to TrsmLLT, too. For these operations alone, there are $5^4 = 625$ refinement combinations from the Π instantiation options. Then, there are refinements for the other loop-body operations and a combinatorial increase in the search space when optimizations are applied. Further, there are multiple algorithmic variants explored for each of the three operations and combinations of partially- and non-merged implementations.

In short, the result is a implementation space that is far too large for an expert developer to explore completely. Similarly, it is too large for DxTer to form explicitly, so we had to develop a way to limit it as an expert might do. For example, we know from the Trsm heuristic that only two options ($\Pi = V_C$ and V_R) are reasonable for each of the four Trsm operations, so an expert would only consider $2^4 = 16$ of the 625 refinement combinations for Trsm alone. We want DxTer to exploit the stairs of the performance space to limit the space similarly.

6.2 A Search Heuristic of Our Own

As we saw above, refinements lead to a particular stair of implementations while optimizations only tweak performance within the stair. We want DxTer to limit consideration to a few stairs with high performance and then explore optimized versions within those stairs.

There are two basic and disjoint flavors of distributed-memory DLA refinements of interest here. The first chooses an algorithmic variant (*eg* one of the three for Cholesky). We call those *variant refinements*. These are the initial decisions on how to partition the input matrices and how to structure the loop-body computation. We have 54 of these refinements encoded in DxTer.[9]

After a variant refinement is chosen, the loop-body updates must be implemented for distributed-memory hardware. The second type of refinement encodes these choices of how to implement loop-body updates. We call those *rephrasing refinements*. These are architecture-specific (distributed-memory) ways to implement a loop-body computation by distributing data and computing in parallel. We have 53 of these refinements encoded in DxTer. We now explain how we characterize refinements as one of these two types and use a heuristic to significantly limit the number of rephrasing refinement options that need to be explored.

As we saw with Cholesky factorization in Section 5.3, we cannot always determine a priori which variant refinements are bad or good because one has to apply some rephrasing refinements to judge an implementations' performance. For example, an expert cannot quickly glance at Cholesky Variants 2 and 3 and say one will be better; one has to explore implementation details for each. There are exceptions, with one mentioned in the next section, where a variant requires a large amount of extra computation. Such always bad refinements are simply not encoded in DxTer.

For rephrasing refinements on the other hand, experienced developers limit their consideration of parallelization schemes. The Gemm and Trsm heuristics, for example, do so based on what we quantified when looking at the implementation space with Cholesky: cost estimates point to good and bad sets of rephrasing refinements. We can leverage this to create a heuristic to limit the number of rephrasing refinements DxTer explores.

When DxTer finds multiple rephrasing refinements that apply to a particular interface, it estimates the cost of each *right-hand side (RHS)* graph that would be used as an implementation.[10] It then ranks all RHS refinements and applies (explores) only the n best options. This is a local decision of which n are best, not considering optimizations that can be applied subsequently to the RHS when part of the surrounding implementation graph. We henceforth call this the *locally n-best (L_nB)* heuristic. It is a generalization of the Gemm and Trsm heuristics, where one rank-orders rephrasing refinements, respectively, and explores only the best n.[11] Doing so limits the search to particular stairs in the implementation space. This is how we exploit the stair structure of the domain in DxTer to make the search space tractable for complex operations like SPD Inversion.

6.3 L_nB Heuristic Results

We evaluated the L_nB heuristic in two ways:

1. Does the implementation space become tractable for complicated operations like SPD Inversion and does it get smaller and faster to search for simpler operations like Cholesky?
2. Is the output generated by DxTer still the same or better than what an expert would code by hand?

For all test operations we have studied [16, 17, 18], the L_nB heuristic succeeds with respect to both metrics (we re-ran previous studies using L_nB).

For Cholesky, with a heuristic value of $n = 2$ the search time goes from over two seconds to under one second. The search space is

[9] For perspective, we have 703 optimizations, most of which come from instantiating a template optimization on various data distributions.

[10] This requires input matrix sizes. As interfaces can be in loops, this analysis is done for each time the code is run across all iterations of the loop. All of the costs are summed.

[11] Setting $n = 1$, n-best reduces to a Greedy heuristic, discussed later in Section 7.3.

reduced from 294 implementations to 62. The output code is exactly the same as when the heuristic is not used.

The L_2B refinements obey the Trsm heuristic, so only $\Pi = V_C$ and V_R are explored. For Gemm, this heuristic limits DxTer to explore what is prescribed by the Gemm heuristic and one additional refinement (since the Gemm heuristic only prescribes the single locally best refinement). Therefore, DxTer does not generate implementations that come from bad Trsm refinements, for example, and only some of the implementations in the stairs with bad Gemm refinements (from the three options, it omits the worst Gemm refinement and includes the second-worst).

Figure 14 shows the implementations generated. The L_2B heuristic limits the search space to three stairs that are a subset of the stairs in Figure 12 (Stairs 3, 7, and 10) and with the expected fewer implementations.

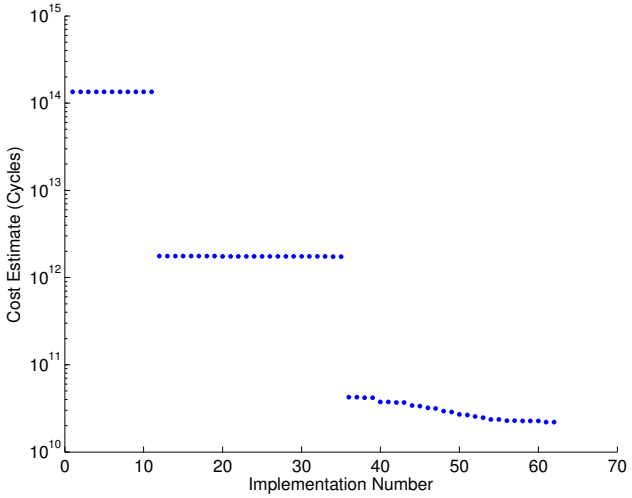

Figure 14: Cholesky implementations with only locally 2-best refinements explored.

Without this heuristic, the implementation space for SPD Inversion is simply too large to generate. After one day of running, DxTer generated over 50 million implementations and was still forming more. With $n = 3$ and the L_nB search heuristic activated, 3,257,506 implementations are generated in 496 seconds, and the output is the same as developed by hand (*ie* the same Elemental users run in their applications).

With $n = 2$, only 107,986 implementations are generated in 181 seconds, and, again, the same code is output as best. The heuristic limited the implementation search space to make it tractable and quickly formed. For example, this limits the options for the four Trsm interfaces to 2^4 combinations one would consider in manual development instead of the full 5^4.

Figure 15 shows the implementations generated with $n = 2$. Even with fewer rephrasing refinement options, stairs are still formed, but they are smaller and there are many fewer of them. Almost all variety of these implementations comes from different combinations of optimizations. The L_2B heuristic limits consideration to a few stairs, where many similarly-performing implementations are generated with similar parallelization and different optimizations.

Indeed, this heuristic exploits DLA performance stairs to limit the number of generated implementations considerably. This enables DxTer to generate the same implementations as a person does manually, but much faster (for SPD Inversion, DxTer code generation took minutes instead of hours of manual development and testing).

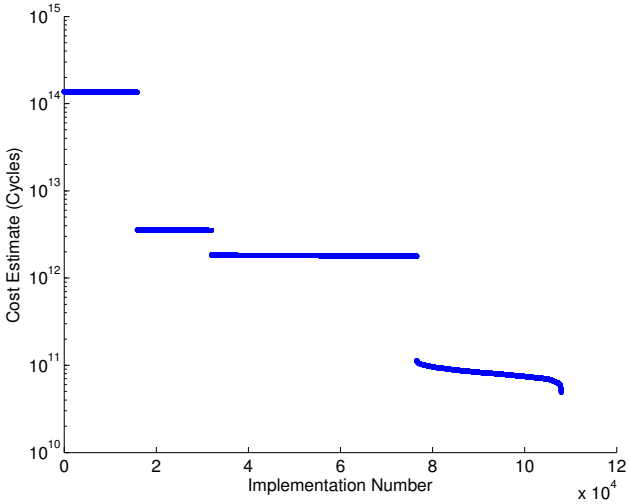

Figure 15: Cost of SPD Inversion implementations generated with locally 2-best heuristic.

7. TWO-SIDED PROBLEMS

Other DLA operations also lead to intractably large search spaces. We now look at two more complex operations to see that the the L_2B heuristic is again necessary and effective. Two-sided Trsm and two-sided Trmm are similar operations used as preprocessors for the generalized eigenvalue problem [23]. Each has four or five variants to explore.[12] The loop body of one variant is shown in Figure 16 to convey its complexity. Our L_nB heuristic is again vital to making the search space tractable.

$$
\begin{array}{ll}
A_{10} := L_{11}^{-1} A_{10} & (\texttt{TrsmLLN}) \\
A_{20} := A_{20} - L_{21} A_{10} & (\texttt{GemmNN}) \\
A_{11} := L_{11}^{-1} A_{11} L_{11}^{-T} & (\texttt{Two-sided Trsm}) \\
Y_{21} := L_{21} A_{11} & (\texttt{SymmRLN}) \\
A_{21} := A_{21} L_{11}^{-T} & (\texttt{TrsmRLT}) \\
A_{21} := W_{21} = A_{21} - \frac{1}{2} Y_{21} & (\texttt{Axpy}) \\
A_{22} := A_{22} - (L_{21} A_{21}^T + A_{21} L_{21}^T) & (\texttt{Syr2kLN}) \\
A_{21} := A_{21} - \frac{1}{2} Y_{21} & (\texttt{Axpy})
\end{array}
$$

Figure 16: Variant 4 of two-sided Trsm.

7.1 Limiting the Locally n-Best Heuristic

In [16], we demonstrated how searching just two of the 4-5 variants of each two-sided operation leads to a massive search space. We presented ways to reduce the search space by restructuring some transformations. We also showed how limiting the ways one interface (called Axpy) is refined reduces the search space by 100-fold or more[13]. We now quickly review this *Axpy heuristic* and explain how it compliments our L_nB heuristic. The Axpy heuristic leads us to a guideline for the interfaces to which the L_nB heuristic does not apply well (one could think of this as a meta-heuristic to tell when the L_nB heuristic applies and does not).

Axpy is a $O(M^2)$ operation on $O(M^2)$ data. This means the amount of computation performed is roughly the same as the amount of

[12] One variant of two-sided Trmm's five is omitted because it has an extra $O(M^3)$ computation, which means it is always bad no matter the target hardware architecture

[13] The previously presented heuristics are important, but insufficient for exploring all variants of two-sided operations. The L_nB heuristic presented here in conjunction with previous heuristics is sufficient.

communication required (*ie* $O(M^2)$ data is redistributed and used for computation). In such cases, optimizations are as important as the rephrasing refinement choice for interfaces. Choosing a locally-suboptimal parallelization of `Axpy` can be best when a subsequent optimization removes the $O(M^2)$ redistribution cost. The heuristic presented in [16] caters to this case, so `Axpy` refinements are limited to those that can be followed by an optimization to remove redistribution. We expect a generalization of this for other interfaces with similar refinement versus optimization cost impact.

For such interfaces, we do not use the L_nB heuristic to limit refinement options because locally-bad choices can be globally best. Therefore, the L_nB heuristic's local view of the graph is ineffective when the best choice depends on available optimizations. Interface refinements are marked in DxTer to specify if the L_nB heuristic is used to limit refinement options that are explored.

Optimizations affect the cost of communication by removing redistribution operations (*eg* as in Figure 7). The RHS of rephrasing refinements consists of a computation component (from computation boxes) and a communication component (and the total cost is the sum of these components). When the computation component's cost is the same order of magnitude as the communication component's cost, then optimizations can make locally-suboptimal refinements globally best – the L_nB heuristic does not work. When the computation component is larger by an order of magnitude, lower-order optimizations are less important in the overall design than achieving good parallelism, local choices are sufficient. This is demonstrated by the stairs in previous sections, where steps are defined by rephrasing choices and optimizations affect where an implementation is in the step. This provides an understanding, using cost estimates, of expert heuristics when exploring `Axpy` and other interfaces with similar cost structures.

There are many similar cases to this in distributed-memory DLA (where computation and communication cost have the same order of magnitude). The L_nB heuristic does not apply in such cases, but it is still applicable to most interfaces we have studied in the past [16, 17, 18], where generally computation is $O(M^3)$ and communication is $O(M^2)$.

7.2 L_nB Heuristic Results

Without L_nB, DxTer runs out of 96GB of memory after a day of execution in generating two-sided Trmm code. Once again, we need to limit the search space to only the best implementations. We do so with the L_nB heuristic to omit the worst stairs from consideration. With the L_3B heuristic, DxTer generates 37,200 implementations after 715 seconds. With $n = 2$, DxTer generates 1,982 implementations after 47 seconds. See Figure 17.

In both cases, the output DxTer generates for each operation is the same (about 40 lines of code). In fact, as reported in [16, 18], the output is slightly better than what was manually developed, but the time to generate code is significantly reduced with the heuristic.

The Elemental expert implemented these operations before creating an optimization to improve the way data is redistributed. He forgot to re-optimize these operations. Since the optimization was added to DxTer, it is automatically applied to any generated code, including the two operations.

In such cases, it is even more important to speedup implementation space generation. When an expert thinks of a new optimization, he can add it to and task DxTer with reimplementing large amounts of library code automatically. It would consider where the new optimization applies to existing code. This is much easier and less error prone than doing so manually.

If implementation space generation takes a long time for each operation, regenerating a library of code would be painfully slow.

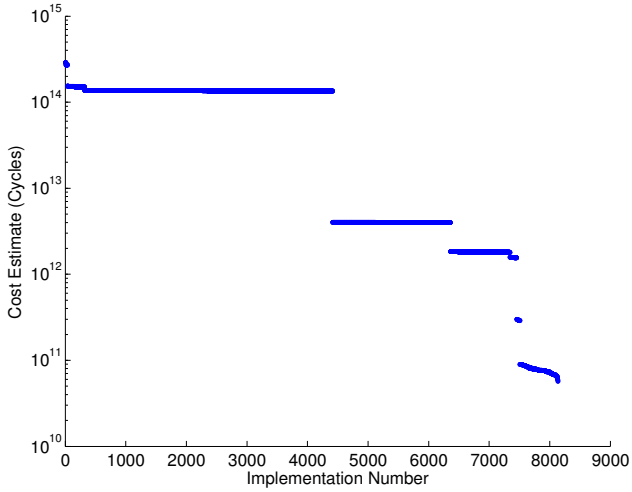

Figure 17: Cost of two-sided Trsm implementations generated with locally 2-best heuristic.

As our heuristic reduces search time for each operation, library generation time is also reduced.

7.3 Why Not Fully Greedy?

One might ask what happens when L_nB has $n = 1$. This would be a purely *greedy* search, where only the locally-best option is explored. The answer is that DxTer generates worse final implementations. With $n = 2$, the best stair for each of the operations we have studied includes implementations with each of the L_2B rephrasing refinements. Sometimes the locally-best option is globally suboptimal because optimizations make a difference between which of the two options is best within a stair – their computation is roughly the same so performance difference comes from communication, which is optimized differently. In other words: *when a stair is formed by two (or more) different refinements, optimizations make the difference within the stair, so more than one refinement option must be included*.

This is similar to `Axpy`, discussed in Section 7.1. In both cases, optimizations make the difference in which of the similarly-performing refinements is best. For all operations we have studied, $n = 2$ is the lowest value that results in the same implementations. For distributed-memory DLA refinements, this value is the size of the smallest selection of refinements for each interface such that optimizations make the different of which choice is best. For this group (*ie* the locally 2-best), the RHS costs are the same or similar because the $O(M^3)$ computation is parallelized to the same degrees. Then, like with `Axpy`, optimizations on communication are the differentiating design decisions.

When using DxTer with another domain, we expect the L_nB heuristic to be useful when there are stairs in the implementation space. We expect the value of `n` to be determined by the size of the smallest set of locally-chosen refinements for which optimizations differentiate the globally-best solution, but demonstrating this is future work.

8. RELATED WORK

The grandfather (origin) of DxT and DxTer can be traced to *rule-based query optimization (RBQO)* [14] in the late 1980s. DxT and DxTer generalize RBQO to the domain of DLA. Not surprisingly, DxT is closely related to many projects, as detailed in [16, 18]. We discuss below those that are related with respect to DxTer's search and the locally `n`-best heuristic.

Model Driven Engineering (MDE) is a basis for our work. *Platform Independent Models (PIMs)*, like our interface-only graphs, represent functionality that is transformed (refined) into architecture-specific (primitive-only) graphs or *Platform Specific Models (PSMs)* [9, 12]. Optimizing transformations, not particularly prominent in MDE, are essential in DxTer to squeeze out the last bit of performance. Optimizations also complicate the implementation search process. Without optimizations, a purely greedy search of refinements would be sufficient because locally suboptimal choices are always globally suboptimal as well.

It is common for software generation projects [1, 3, 24, 30] in scientific computing to apply rewrite rules similar to our transformations to search a space of high-performance implementations. Because of the nature of their application domain, cost estimates are not accurate enough to judge an implementation's expected performance; the code must be compiled and run on sample data.[14] As a result, these projects often use machine learning techniques to limit the search space based on empirical data. Such techniques could be used in conjunction with our L_nB heuristic to limit the search space more if it every becomes too large with future algorithms. DxT and these other software generation projects borrow many search ideas from artificial intelligence techniques [13] and will likely incorporate or apply more existing algorithms as needed with new domains in the future.

The DxTer search process is closely related to *path planning*, where one plans a path through some space for which there is a cost associated with portions of the path. With DxT, we search design choices – path segments – that have positive and negative weights. Our goal is to find a path that ends in a primitive-only implementation that performs well (*ie* the path is low-cost). The work of [20, 21] shows how subspaces of decisions / paths can be ignored, or pruned, to limit consideration. In DxTer, *simplifiers* are optimizations that are always applied because they reduce an implementation's cost and it is never worth exploring implementations that do not have the transformation applied because they always perform worse [16]. This is an example of a *dominance relationship* [20]. Loosely, this means the subspace of implementations that result from applying the simplifiers *dominate* – or always have lower cost than – the subspace of implementations without the simplifier applied. Optimizations are manually identified as simplifiers in DxTer. The L_nB heuristic is meant to identify dominating rephrasing refinements in DLA automatically and only explore those.

Lastly, recent work on feature-based program generation [11] may reveal a counterpart to our work. Namely, how a few decisions of features (instead of rephrasing and optimization choices) predict the performance of a program configuration accurately. In [26], a heuristic is presented to limit the space of configurations that must be sampled for performance, limiting to the most important decisions. They assume limited interactions between choices and mainly pairwise, lower-order interactions. For DLA this can sometimes apply, as we have developed the L_nB heuristic to apply to such cases, but the Axpy heuristic shows that it is not always the case. In both our work and theirs, the unit of modularity is a transformation. DxT transformations manipulate algorithms, while feature-based program generation uses larger transformations that manipulate layers of systems.

[14] For our code, empirical test are not feasible because runtime is too large and the machines are financially expensive to use. Further, distributed-memory DLA cost estimates are accurate enough to judge implementations well.

9. CONCLUSION

Explicitly enumerating the (implementation $\times$ performance) space for DLA operations allows us to give scientific meaning and justification to the heuristics that DLA experts have used for years. These heuristics have made experts effective in navigating a large number of design decisions to produce highly-efficient code without taking days to analyze options.

The Trsm and Gemm heuristics (and others that we do not detail here but are similar) now make sense when analyzing the structure of the (implementation $\times$ performance) space by looking at the stairs and the design choices that yield them. This view helps us understand heuristics that experts have used and explain or motivate them with cost estimates. They are no longer rough design "rules of thumb" passed down by generations of expert developers.

Further, we have shown how performance stairs enabled us to develop the locally `n`-best (L_nB) heuristic that reduces the time it takes to generate a space of implementations or makes it tractable for complicated algorithms without sacrificing the performance of generated code. Basically, we have abstracted many of the heuristics DLA developers use to exploit the implementation space's stair structure (though we only detail two here). Instead of taking days to generate complicated code (or not completing given system resources), it takes minutes. This is an important step to make automated code generation a useful design tool for expert developers. In fact, automatically generated code improved on hand-developed code for two-sided Trmm (and other operations), so some generated code is now used in the Elemental library.

We are in a rare position to enumerate the implementation space of a specification and analyze the performance of its members. As automatic code generation becomes more popular (and necessary with more complicated hardware and software), such a benefit must be further exploited. We expect other domains to have similar structure to that of DLA and to similarly benefit from heuristics based on the stair stepping structure. We believe this is especially true in scientific computing domains, which have often taken design cues from DLA in the past. We will study other domains as part of our future work.

Finally, while we have not studied the pedagogical value of identifying stair steps, we see potential. When teaching how to engineer distributed-memory DLA software to a novice, stairs demonstrate that refinements must be well understood to make the best choices and then optimizations are a less important lesson. This gives us a partial order of lessons based on their importance on performance. Also, when examining a student's implementation, we can explain why his/her choices lead to a lower performing solution than the best choices. We can identify to which stair the implementation belongs and, therefore, which set of refinements were chosen incorrectly, leading him/her to a suboptimal stair. We can explain "you should have chosen refinement `ref1` instead of `ref2` because it leads to better performance in this algorithm." Exploiting the pedagogical benefits to identifying stairs and being able to enumerate the search space (or important parts of it with the L_nB heuristic) is future work.

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).

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).

10. REFERENCES

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

[2] I. D. Baxter. Design Maintenance Systems. *CACM*, April 1992.

[3] G. Belter, E. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *SC*, 2009.

[4] P. Bientinesi et al. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[6] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience: Research articles. *Concurrency and Computation: Practice & Experience*, 19(13):1749–1783, Sept. 2007.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990.

[8] J. Feigenspan, D. S. Batory, and T. L. Riché. Is the derivation of a model easier to understand than the model itself? In *ICPC*, pages 47–52, 2012.

[9] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2003.

[10] R. C. Gonçalves, D. Batory, and J. Sobral. ReFlO: An interactive tool for pipe-and-filter domain specification and program generation. *submitted*, 2013.

[11] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013.

[12] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model-Driven Architecture*. Addison-Wesley, Boston, MA, 2003.

[13] R. E. Korf. Artificial intelligence search algorithms. In *In Algorithms and Theory of Computation Handbook*. CRC Press, 1996.

[14] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *ACM SIGMOD*, 1988.

[15] B. R. Mandelbrot. *The Fractal Geometry of Nature*. Macmillian, Philadelphia, PA, USA, 1983.

[16] B. Marker, D. Batory, and R. van de Geijn. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, 27(4):439–452, 2013.

[17] B. Marker, D. Batory, and R. A. van de Geijn. Code generation and optimization of distributed-memory dense linear algebra kernels. In *ICCS*, 2013.

[18] B. Marker, J. Poulson, D. Batory, and R. A. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *VECPAR*, 2012.

[19] T. Meng Low, B. Marker, and R. van de Geijn. FLAME Working Note #64. Theory and practice of fusing loops when optimizing parallel dense linear algebra operations. Technical Report TR-12-18, The University of Texas at Austin, Department of Computer Sciences, 2012.

[20] S. Nedunuri, D. R. Smith, and W. R. Cook. Synthesis of greedy algorithms using dominance relations. In *NASA Formal Methods*, pages 97–108, 2010.

[21] S. Nedunuri, D. R. Smith, and W. R. Cook. Theory and techniques for synthesizing efficient breadth-first search algorithms. In *FM*, pages 308–325, 2012.

[22] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, 2013.

[23] J. Poulson, R. van de Geijn, and J. Bennighof. (Parallel) algorithms for reducing the generalized hermitian-definite eigenvalue problem. *ACM Trans. on Math. Softw.*, 2012. submitted.

[24] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[25] T. Riché, R. Goncalves, B. Marker, and D. Batory. Pushouts in Software Architecture Design. In *GPCE*, 2012.

[26] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 167–177, Piscataway, NJ, USA, 2012. IEEE Press.

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

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

[29] F. Van Zee and R. van de Geijn. BLIS: A framework for rapid instantiation of blas functionality. *ACM Trans. Math. Softw.* accepted.

[30] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.