First International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering 2013

Interfaces are Key

Bryan Marker
The University of Texas
Austin, Texas, USA
bamarker@cs.utexas.edu

Robert van de Geijn
The University of Texas
Austin, Texas, USA
rvdg@cs.utexas.edu

Don Batory
The University of Texas
Austin, Texas, USA
batory@cs.utexas.edu

Keywords

high-performance dense linear algebra, program generation, automatic programming, interfaces, abstraction

1. INTRODUCTION

Many *dense linear algebra (DLA)* operations are easy to understand at a high level and users get functional DLA code on new hardware relatively quickly. As a result, many people consider DLA to be a "solved domain." The truth is that DLA is not solved. DLA experts are rare because the "tricks" and variety of algorithms they need to get high performance take time to learn. DLA implementations are only available on a new architecture when an expert with enough experience goes through a rote process to implement many related DLA operations. While so much of the manual work is rote, this hardly suggests the domain is "solved." We have not proven that we understand the field until we have automated the expert. Automate the expert for the entire field, and the field is closed. We view that goal as the equivalent of going to Mars. In practice, we will get to the moon automatically, and experts will then be freed up to worry about how to get from there to Mars.

Given the focus of SEHPCCSE, we summarize progress we have made towards that goal over the last 10+ years of interface design and 3 years of automation research. We explain our experiments working towards a sustainable solution to this tedious, laborious, and largely unnecessary process with fundamental and reusable DLA interfaces. We talk about how those interfaces allow us to generate most code automatically as an expert would manually. With influences from the *software engineering (SE)* literature, we present *Design by Transformation (DxT)* to encode knowledge about domain interfaces. We talk about lessons that can be learned from DLA and applied to other *computation science and engineering (CSE)* domains with the hope that their code can also be generated automatically.

2. LAYERING AND INTERFACES

DLA operations are vitally important to many domains. CSE applications are built from layers of functionality, with DLA often at the bottom. With each successive layer, abstraction reduces the amount of detail a software engineer must consider. One does not need to think about the exact implementation of a function interface. One only concerns herself with the functionality of an interface (maybe in terms of preconditions and postconditions). Testing can be done on the interfaces by themselves (e.g. via unit testing) to gain trust without looking through and understanding an implementation of each. The domain expert builds a layer on those interfaces in terms of computational pieces that make sense to her, for example thinking of the math involved in a simulation without concern to target hardware specifics like the cache structure, which is only considered at lower levels.

Performance is a software requirement for CSE domains. For example a scientist cannot accept code that takes too long to run on an expensive distributed-memory (cluster) machine for which she is paying per core-hour. Therefore, it is essential to find the right application interfaces that enable **both** engineering productivity and performance. DLA experts have been refining and polishing DLA interfaces for decades. From the CSE perspective, the well-established DLA interfaces found in the BLAS [3] and LAPACK [1] layers are great for implementing higher-level functionality quickly, easily, and with good performance. When moving to a new architecture, different library implementations of those interfaces are assumed to exist and perform well, and all code built on top of them simply works.

At this point, the interfaces internal to DLA code (those that CSE engineers do not look at) have also become well-designed. We have reached a point where large portions of DLA libraries are ported to a new architecture by only re-implementing a relatively small set of routines [12]. Abstraction has enabled DLA library developers this great productivity enhancer, but for those routines that need to be implemented, an experts' work is often rote, tedious, and error-prone.

The interfaces are such that an expert must implement many families of related functionality with small tweaks. They are close enough that it is rote to do, but they are different enough that it is not simply copying and pasting code with minor changes. For example take the classic general matrix-matrix multiplication (`Gemm`) operation $C := AB + C$ where A, B, and C are general matrices (no special form). A and B can each be (conjugate-)transposed or not, which means an expert has four varieties of implementation to provide. On some architectures (e.g. distributed-memory), an expert specializes the implementation based on which of the three matrices is largest to avoid communicating it. That is a family of $4 \times 3 = 12$ implementations an expert needs to implement. With the internal interfaces developed for DLA, though, the code for each of the twelve is very similar.

Other *level-3 BLAS (BLAS3)* operations similarly have families of implementations for each operation. They range from needing

SEHPCCSE'13, November 17 - 21 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2532352.2532359

4 implementations (e.g. for `Herk`) to 16 (e.g. for `Trsm`) for good performance of each of the exposed interfaces. To implement all of these, a lot of knowledge is repeatedly applied (rote application of that knowledge) in their code development. For example all of the BLAS3 are implemented in terms a `Gemm` suboperation, so knowledge of how to implement `Gemm` in different ways is reapplied for each implementation. Further, many of the same optimizations are applied repeatedly throughout BLAS3 code. Good interfaces enable an expert to implement varied functionality with repeated knowledge application, which improves productivity.

3. EXPERIMENTS WITH INTERFACES

There are two key benefits of abstracting to interfaces that lead to automating an expert's work. In this section, we explain those benefits from the perspective of how the interfaces were developed naturally by experts (to ease their burden). In the next section, we explain how these apply to automatic program generation.

First, code is easier to develop because a DLA expert does not think of all implementation details. Instead, layers of internal interfaces enable an expert to consider only part of the details at a time. Software layers often coincide with architecture layers. For example in sequential code there are layered interfaces that target layers of the cache [4]. At each layer of code, an expert assumes the interfaces keep the correct pieces of data in certain layers of the cache for reuse. Therefore, an expert only considers the next layer of cache. For shared memory, interfaces build on the sequential interfaces to parallelize for multiple cores. For distributed memory, interfaces hide sequential processor details, so experts consider collective communication and computation as interfaces.

Second, interfaces leads to fewer operation types. One can think of the set of interfaces as primitives for a DLA *domain-specific language (DSL)*. Experts then code in that DSL, only considering valid programs in that DSL, which is a small subset of all programs that can be expressed in a general language like C or Fortran. The DSLs we talk about here are implemented as libraries within the general language. As one can code using only the libraries' interfaces, we can think of them as DSLs. Experts only need implementation knowledge and optimization tricks for those interfaces to get high-performance code. This is part of the reason why an expert's task looks rote for so much of the library development effort. Limited (but deep domain) knowledge is repeatedly reused.

The FLAME [5] project explores ways to abstract DLA algorithms and their implementations. A hallmark of the project is the ability to derive correct families of algorithms from a specification of the computation to be performed. A prototypical algorithm is shown in Figure 1 (left), which computes Cholesky factorization. This uses FLAME interfaces to hide details of indexing, allowing one to reason about the algorithm at a high level of abstraction. Each of the loop body operations (between the horizontal lines) is one of FLAME's computation interfaces[1] (which include the standard BLAS- and LAPACK-like routines). There are a collection DSLs following the FLAME approach to abstraction/interfaces, which we now describe.

Elemental [9] is a library of distributed-memory DLA functionality (BLAS and LAPACK-level, i.e. functionality similar to ScaLAPACK [2]) as well the basis for that functionality (which makes up the Elemental DSL). In Elemental there is a small number of ways (around 10) to distribute data on the processes in a cluster. The DSL consists of operations to perform collective communication to redistribute data between the different distributions and there are interfaces to perform computation (the latter are largely the same as the standard BLAS and LAPACK interfaces). The FLAME interfaces for indexing are also included to omit indexing in favor of reasoning about matrix partitions. The benefit of these interfaces is that parallelizing most sequential DLA algorithms in high-performance Elemental code is rote (this is described and automated in [6, 7]). An expert needs to decide which distributions are efficient and how to redistribute between them. Implementing these decisions in the DSL is straight-forward from there.

For BLAS-level routines on sequential and shared-memory architectures, BLIS [12] can be used as a DSL. BLIS is a framework for quickly porting all functionality of the BLAS to new architectures. It achieves this goal with good interfaces, where only a limited number of functions need to be implemented in hardware-specific code to provide all BLAS functionality. The BLIS framework is built around those interfaces. The interfaces of BLIS, then, form a DSL for BLIS developers to implement BLAS and BLAS-like algorithms. The core interfaces are similar to those used in the popular GotoBLAS library [4]. To perform an operation like `Gemm` efficiently, some data is "packed" into special data buffers, where the data is rearranged in a way that allows computation kernels to proceed at high performance. The DSL is a language with those few packing and computation interfaces and FLAME loop-related interfaces. BLAS implementations consist of combinations of those few interfaces.

For LAPACK-level routines on sequential and shared-memory architectures, `libflame` [11] uses BLAS and LAPACK-like computation and FLAME interfaces. Thanks to the interfaces, sequential `libflame` implementations look very much like FLAME algorithms, as shown in Figure 1 (right), which implements the Cholesky algorithm.

Each of these libraries can be viewed as DSLs with FLAME and architecture-specific interfaces. Performance in each case is comparable to or exceeds similar products that have less-refined interface. Abstraction was embraced for each as an experiment to see what could be done with DLA software to improve an expert engineer's productive, code readability, and software reliability.

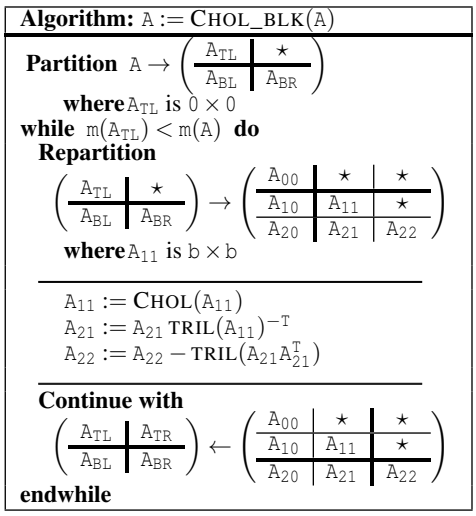
4. ENCODING EXPERT KNOWLEDGE FOR AUTOMATIC CODE GENERATION

So over time DLA experts have experimented with incorporating layered interfaces, and the result is better code. Now we can further improve DLA software and ease an experts' burden. Anytime a person's task becomes rote, automation should be investigated. In the case of DLA libraries, much of an experts' development work is rote thanks to good abstraction, and we can indeed automate it. In this section, we present the basics of DxT [6, 7], which is used to encode expert knowledge about DLA interfaces. A system can then utilize that knowledge to generate high performance code.

In DxT sequential algorithms and their implementations are represented in *directed acyclic graphs (DAGs)*. Edges represent data flow. Nodes represent some piece of functionality. A node can either be 1) an interface with no implementation details, only a description of the computation to be performed, or 2) a primitive, which maps to given code in the DSL. The set of node types used to construct graphs includes the interfaces found in the DLA domain.

Here, the goal with DxT is to start with a DAG representing a sequential DLA algorithm (e.g. that of Figure 1), encoding knowledge about the operation to be implemented. Then, we want to transform it with implementation details into high performance sequential or parallel code (e.g. distributed-memory Elemental code). To do that, we encode how to implement DLA interfaces in parallel code (using

[1]"Chol" is Cholesky factorization and "Tril" takes the lower-triangular portion of the input matrix.




```
FLA_Part_2x2( A,    &ATL, &ATR,
                    &ABL, &ABR,     0, 0, FLA_TL );
while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){
  b = FLA_Determine_blocksize( ABR, FLA_BR, FLA_Cntl_blocksize( cntl ) );
  FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                         /* ************* */   /* ******************** */
                                                &A10, /**/ &A11, &A12,
                            ABL, /**/ ABR,      &A20, /**/ &A21, &A22,
                            b, b, FLA_BR );
  /*-----------------------------------------------------------*/
  FLA_Chol( FLA_LOWER_TRIANGULAR, A11 );

  FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                       FLA_CONJ_TRANSPOSE, FLA_NONUNIT_DIAG,
                       FLA_ONE, A11, A21 );

  FLA_Herk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                       FLA_MINUS_ONE, A21, FLA_ONE, A22 );
  /*-----------------------------------------------------------*/
  FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                   A10, A11, /**/ A12,
                            /* ************** */  /* ****************** */
                              &ABL, /**/ &ABR,      A20, A21, /**/ A22,
                              FLA_TL );
}
```

Figure 1: Left: Algorithmx to compute the Cholesky factorization. `m(B)` is the number of rows of `B` and '⋆' denotes matrix entries that are not referenced. Only the lower triangular part of the matrix is updated. Right: Sequential implementation of algorithm in sequential code using the `libflame` DSL.

lower-level interfaces).

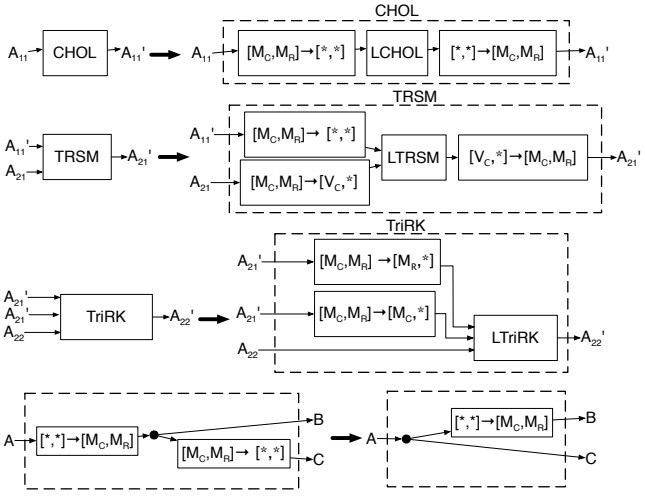


Figure 2: The top three transformations are refinement options experts explore to parallelize the loop-body operations of Figure 1 (left) for the Elemental DSL. The bottom transformation is an optimization to remove an unnecessary redistribution.

A *refinement* is a transformation that replaces an interface with a graph that implements the interface's functionality (possibly using other interfaces for layering of interfaces). The top three transformations of Figure 2 show refinements replacing each of the loop body operations for Cholesky factorization with one implementation in parallel Elemental code. The boxes with a → symbol are primitives that map to the Elemental DSL primitve for redistributing data via collective communication (from the distribution on the left of the arrow to the distribution on the right). The other boxes are computation primitives. Each transformation encodes knowledge about one way to parallelize an interface; we encode the various options an expert considers.

The bottom transformation of Figure 2 is an *optimization*, which replaces one subgraph with another. It encodes knowledge about equivalent ways to implement functionality. Optimizations are chained together to improve performance. In this example, the optimization removes an unnecessary collective communication operation, which is important for an expert to do to decrease runtime.

With these two types of transformations, we can encode the design knowledge that experts use to develop most DLA functionality. The knowledge can now be passed on to the next generation of experts (optimization knowledge will not be lost when an expert retires or when ten years pass without exercising the knowledge). Expert knowledge is made concrete in terms of the transformations. Further, by encoding the knowledge, we can enable a system to generate code automatically. This eases an experts' burden to implement libraries of related functionality via the rote reapplication of knowledge. Further, it gives higher confidence in code when the transformations are correct by proof or by reasoning and the derived implementation is then correct by construction — start with a correct algorithm, apply transformations that maintain correctness, end with a correct implementation.

To explore the idea of automated program generation for DLA, we have a prototype system called DxTer [6]. DxTer takes an input graph and transformations and outputs a high performance implementation of the input graph. It does this by applying all of the transformation it can to the input graph, forming a combinatorial search space of implementations, and choosing the "best" implementation.

DLA experts make implementation decisions based on an estimate of "cost" like runtime. They aim for the shortest runtime possible. With Elemental experts predict runtime to choose which parallelization schemes to use or optimizations to apply. Estimates are first-order approximations in terms of the amount of computation performed and the amount of data communicated between processes [6, 7]. Thanks to the interfaces in Elemental, BLIS, and `libflame`, relatively rough cost estimates are good enough to guide experts without having to implement, compile, run, and time code. Further, in DxTer we can encode these estimates to automate the expert's analysis.

Each primitive has a runtime estimate in terms of the problem size. The cost of each implementation in the search space is the sum of all primitives' costs. The lowest cost implementation is output from DxTer. The cost estimates are good enough to rank-order the search space well [6] just as they are good enough to guide an experts' choices when manually developing code.

5. LESSONS LEARNED

DLA software has been studied longer than many CSE domains. We have described how the results are interfaces that help an expert engineer. Now, those interfaces can also be used to generate automatically a lot of the code experts manually develop (relieving them of rote work). Granted, interfaces can result in some lost performance by hiding opportunities for optimization. With good interface design, these missed opportunities are minimized. The gains in engineer productivity and software reliability are deemed worth small losses.

Further, we have found that in many cases the increase in productivity allows engineers to explore new algorithms/ideas that lead to better performance. With FLAME and its interfaces, one can derive a family of algorithmic variants for a particular operation. Generally, one algorithmic variant is not best in all cases. An engineer can choose from this family for a particular hardware architecture and problem size, specializing code as needed. Well designed interfaces enable a variety of algorithms to be implemented quickly for prototyping and deployment. Automation takes this a step forward by automating the implementation process and even the exploration of algorithmic options [8]. Such automation is not possible without good interface design. Therefore, **DLA interfaces may increase overhead in low-order terms, but the benefit is increased programmer productivity and automation, which enables faster development of better code**.

A lot of the push towards more interfaces came from the need to port software to very different architectures that were changing quickly. Only a few experts in the world could do that well, so they had to increase their productivity. With FLAME **a few people decided on interfaces based on their expertise and experience**. Work on PLAPACK [10] led to interfaces specifically for distributed memory, and that experience was applied to the entire domain. This experimentation with interfaces has paid off for a variety of FLAME's projects (i.e. derived libraries and DSLs) and has led to a lot of automatic code generation for those libraries.

The lesson for other CSE domains is that experimentation in designing good interfaces for software and algorithms is worthwhile. Fear of increased overhead is reduced by the promise of greater productivity. Automatic code generation would be an extreme, where good interfaces take the human developer out of the loop almost entirely. For DLA it is a reasonable extreme given our results, so why isn't the same possible for other domains? **Experiments with interfaces have significantly improved DLA software, so similar experiments should be undertaken for CSE domains.**

A key is that such a push to experiment with, develop, and introduce interfaces must come from domain experts willing to abandon existing software in favor of new software coded in terms of the better interfaces. **Just as we throw away old hardware that is obsolete, so too must we throw away old software that is obsolete**. Learn from experience and invest in software that is more manageable for the future.

6. CONCLUSION

We have explained how interfaces have been developed for DLA through years of experience. The people with that experience are considered experts, who are rare. Their expertise enables them to balance the overhead of interfaces with the benefits that come from more understandable, manageable, and maintainable software. This is a worthwhile end point, but we have also explained how good interfaces have enabled us to take the human out of the engineering process entirely for much of the DLA library development effort. That is to say that an experts' work is largely a rote reapplication of knowledge to manipulate the right combination of interfaces. That knowledge can be encoded using DxT, and the experts' tedious task of reapplying knowledge repeatedly for many related functions is automated.

The ability to automate DLA software generation is the result of years of effort and experimentation. It is the result of an investment in interfaces throughout the domain. We believe a similar experimentation should be undertaken throughout CSE domains. For many we believe the right interfaces are not immediately clear because people have not attempted to abstract the domain in a wholehearted effort. Instead, experts have simply responded to changing architectures or new demands of functionality without finding the domain's software patterns. It is time to take a step forward and out of the dark ages of software engineering where graduate students slave away to produce the code we need for our science.

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).

7. REFERENCES

[1] E. Anderson et al. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[2] J. Choi et al. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Comput. Soc. Press, 1992.

[3] J. J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990.

[4] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[5] J. A. Gunnels et al. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, Dec. 2001.

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

[7] B. Marker et al. Code generation and optimization of distributed-memory dense linear algebra kernels. In *ICCS*, 2013.

[8] B. Marker et al. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, To Appear.

[9] J. Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, Feb. 2013.

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

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

[12] F. G. Van Zee and R. van de Geijn. BLIS: A framework for rapidly instantiating blas functionality. *ACM Trans. Math. Softw.*, Submitted.