DSLs, DLA, DxT, and MDE in CSE

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

***Abstract*—We narrate insights from a collaboration between researchers in *Software Engineering (SE)* and in the domain of *Dense Linear Algebra (DLA)* libraries. We highlight our impressions of how software development for computational science has traditionally been different from the development of software in other domains. We observe that scientific software (at least DLA libraries) is often developed by domain experts rather than legions of programmers. For this reason, researchers in SE need to impact the productivity of experts rather than the productivity of the masses. We document this and other lessons learned.**

I. Introduction

Our experience with the FLAME project, which has been funded by NSF for more than a decade, motivates this paper. Initially, FLAME pursued fundamental research related to the derivation and implementation of high-performance *Dense Linear Algebra (DLA)* libraries. More recently, as part of an NSF Software Infrastructure for Sustained Innovation grant, its mission has become to vertically integrate the DLA software stack from low level kernels to sequential, multi-threaded, and distributed-memory libraries [1]. While this vertical integration helps the DLA expert, we discuss how it also facilitates the application of a rather different kind of *Software Engineering (SE)*, which we call *Design by Transformation (DxT)*, that can become a template of how SE can codify expert knowledge and automate the optimizations performed by experts. Since DLA libraries are at the bottom of the *Computational Science and Engineering (CSE)* food chain, we postulate that DxT has applicability to CSE software development and beyond.

II. The FLAME Approach to DLA Library Development

Traditionally when developing DLA libraries, there has been a tension between the natural layering and abstraction that exists in its mathematics on one hand and the demand that libraries attain near-peak performance on the other. The street wisdom says that layering impedes performance and the need for speed overrules abstraction in the implementation. The FLAME project, especially when combined with DxT, contradicts this belief. Many of these lessons can be extrapolated for other CSE domains.

> ***Lesson**: Highly-layered abstractions can have highly-efficient implementations.*

A. The FLAME Notation

The FLAME project has always favored abstraction and elegance over performance. A notation for expressing dense linear algebra algorithms was invented [7], [12]. This notation hides details of indexing, allowing one to reason about the algorithm at a high level of abstraction. We illustrate it for a prototypical algorithm that computes the Cholesky factorization, in Figure 1 (left). The notation hides details of indexing by exposing regions of the matrix being updated, thereby allowing one to reason about the algorithm at a high-level of abstraction.

> ***Lesson**: Express algorithms at the same level of abstraction as one reasons minimizes the opportunity for mistakes and confusion while writing an algorithm.*

B. The FLAME methodology for deriving algorithms

We observed that our notation enabled algorithms to be systematically derived correct [4], [15]. That is, given the loop invariants, DLA algorithms are derived hand-in-hand with their proof of correctness.

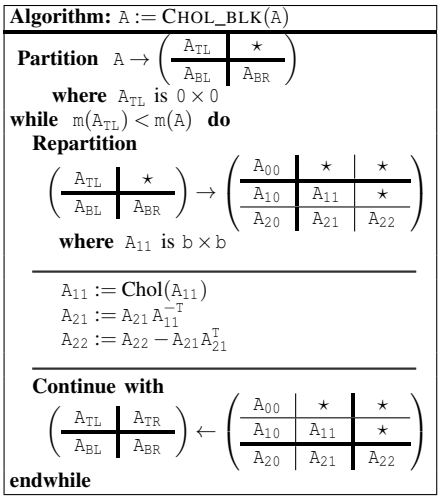
For a given matrix operation, a family of algorithms are so derived, since one operation gives rise to several or even dozens of loop invariants [13]. This is important, because different high performance architectures may require different algorithms to be chosen in order to achieve the best performance [3]. So although FLAME favors abstraction and elegance, it enables high performance because its methodology yields a multitude of algorithms from which the best can be chosen, thus offsetting any overhead incurred by abstractions [1].

> ***Lesson**: For DLA, the expert activity of identifying algorithms given an operation can be made systematic. Indeed, the methodology is systematic to the point that it has been automated [3].*

C. From correct algorithms to correct implementations

A correct algorithm must still be translated into code, which could introduce of coding errors (bugs). To avoid this, the FLAME project defined *domain-specific*

[1]DLA may currently be unique in that multiple algorithms can be systematically derived, but it is common for experts to choose from multiple algorithms for the same operation in CSE domains.




```
PartitionDownDiagonal( A, ATL, ATR,
                       ABL, ABR, 0 );
while( ABR.Height() > 0 ){
    RepartitionDownDiagonal( ATL, /**/ ATR,  A00, /**/ A01, A02,
                            /*************/ /******************/
                                  /**/       A10, /**/ A11, A12,
                             ABL, /**/ ABR,  A20, /**/ A21, A22 );
    //--------------------------------------------------------//
        A11_Star_Star = A11;
        LocalChol( Lower, A11_Star_Star );
        A11 = A11_Star_Star;

        A21_VC_Star = A21;
        LocalTrsm
           ( Right, Lower, Transpose, NonUnit,
             (T)1, A11_Star_Star, A21_VC_Star );
        A21_MC_Star = A21_VC_Star;
        A21 = A21_MC_Star;

        A21_MR_Star = A21_VC_Star;
        LocalTriangularRankK( Lower, Transpose,
            (T)-1, A21_MC_Star, A21_MR_Star,
            (T)1, A22 );
    //--------------------------------------------------------//
    SlidePartitionDownDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
                                     /**/       A10, A11, /**/ A12,
                               /*************/ /******************/
                                ABL, /**/ ABR,  A20, A21, /**/ A22 );
}
```

Fig. 1. **Left:** Blocked algorithms for computing 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:** Parallel Elemental code developed from sequential algorithm.

languages (DSLs), implemented as library calls, for C, C++, and scripting languages like M-script (Matlab's scripting language) so that the code closely resembles the algorithms expressed in FLAME notation. Mapping from a FLAME algorithm to code is straight forward.

> ***Lesson**: Express algorithms in code at the same level of abstraction as one reason using DSLs or good APIs. This minimizes translation effort and reduces the opportunity for error.*

D. From sequential code to parallel code

Parallelization of DLA algorithms to distributed-memory architectures requires the distribution of matrices to (MPI) processes, redistribution of data (in the form of collective communication), and local computation that can be performed in parallel [14]. The Elemental library, based on FLAME notation and APIs, was developed for distributed-memory DLA computations. It incorporates a C++ API that is used as a DSL for expressing algorithms with "=" operator overloading to hide communication details in order to achieve high performance on large distributed-memory architectures [10]. Code for the Elemental Cholesky factorization is given in Figure 1 (right) which we describe in more detail in Section III-B.

> ***Lesson**: If sequential algorithms and code are expressed in the right way, parallel implementations can closely resemble their sequential counterparts, without sacrificing performance.*

The SE challenge is to take the sequential algorithm in Figure 1 (left) to the highly optimized parallel code in Figure 1 (right). To do so requires considerable expert knowledge. Such experts are few and far between. But once an expert understands how to apply this knowledge, largely gained by parallelizing a few operations, parallelizing the remaining operations in a DLA library is more repetitive and time consuming than difficult. This raises the interesting question: can this repetitive exercise be mechanized? Can we enable the rare expert to be more productive in producing the codes of many operations?

> ***Lesson**: Imposing a standardized, conceptual structure to both understand and organize design activities often leads to the ability to the mechanization of these activities.*

III. DxT: Enter the Software Engineer

A. DxT Introduction

DxT is an approach to automate software design by mechanizing expert knowledge of program construction. Algorithms are represented in *Pipe-and-Filter (PnF)* graphs where computations (or communications) are nodes and I/O relationships are edges. A node can be an *interface* — a specification of a computation that has no implementation details — or a node can be a *primitive*, which maps directly to an API call.

DxT encodes expert knowledge as graph rewrites. *Refinements* replace an interface with a graph implementing it (using lower-level interfaces and/or primitives). *Optimizations* replace a subgraph with another subgraph implementing the same functionality in a different way.

One or more optimizations are applied (as the name suggests) to improve performance.

The goal of DxT is to derive correct implementations of input PnF graphs (called *Platform-Independent Models (PIMs)* in MDE-speak [6]) to platform-specific PnF graphs (called *Platform Specific Models (PSMs)*) using rewrite rules of the knowledge base.

We start with a PIM that represents a high-level definition of our DLA algorithm. It contains only interconnected interfaces. We then derive, using graph rewrites, a PSM that contains only primitives. This PSM an implementation of the PIM.

There can be many PSMs that implement a PIM. Each implementing PSM has a cost (time-to-completion is a common cost measure for DLA); domain knowledge can be used to estimate this cost. By searching the space of derivable PSMs, we can automatically find the most efficient PSM for a target architecture and problem size.

> ***Lesson**: Well-known ideas in the history of automated SE, coupled with more recent ideas on MDE, provide a simple and elegant framework for generating high-performance codes.*
>
> ***Lesson**: The above is possible only if a standard conceptual structure (e.g., FLAME) is imposed on a domain; this structure reduces the number of rewrite rules that an expert uses/needs and promotes the reuse of such rules. Without such structure—which tells us how the big pieces fit together—all of what we describe above becomes intellectual chaos.*

B. Using DxT for Cholesky

Let's now see the expert knowledge that is needed to generate the high-performance implementation of Figure 1 (left). To fully understand this parallelization requires deep domain knowledge, presented in [8]. For here, just understand that matrices are stored in objects, which distribute matrix data across the distributed-memory process grid in several ways.[2]

An expert *must* know the various legal ways to parallelize common DLA interfaces. These options are encoded as DxT refinements. The top three transformations in Figure 2 show refinements used for the Cholesky example (boxes that are internally labeled $[\mathtt{x},\mathtt{y}] \rightarrow [\mathtt{z},\mathtt{w}]$ represent redistribution from distribution $[\mathtt{x},\mathtt{y}]$ to $[\mathtt{z},\mathtt{w}]$). These are only some of the options that must be encoded—experts have multiple parallelization schemes.

Experts must also know how to improve the cost of an implementation by, for example, removing unnecessary communication. This knowledge is encoded by DxT

[2]For readability, the variables in Figure 1 (right) are named by the submatrix they represent and the way they are distributed, though we will not explain those distributions here.

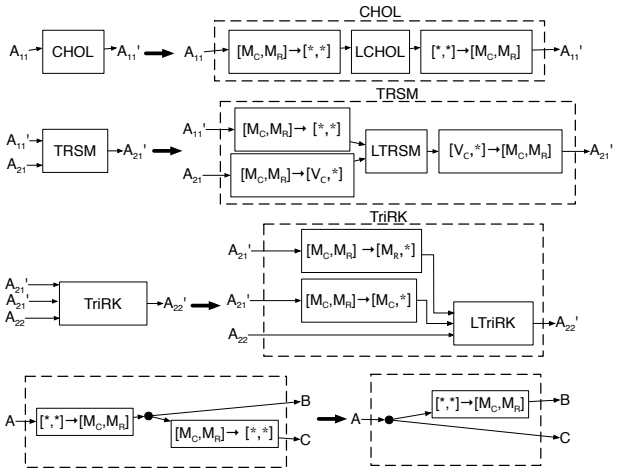


Fig. 2. The top three transformations are refinement options experts explore to parallelize the loop-body operations of Figure 1 (left). The bottom transformation is an example optimization to remove an unnecessary and expensive redistribution.

optimizations, one of which is shown at the bottom of Figure 2.[3] The transformations shown here are not specific to this Cholesky algorithm; they are reusable pieces of expert knowledge that are applied repeatedly to develop a distributed-memory DLA library like Elemental.

> ***Lesson**: Expert program design knowledge can be expressed by graph rewrites that capture incremental steps in domain-specific program development.*
>
> ***Lesson**: Tools that automatically apply graph rewrites and that can estimate the execution cost of a produced graph can generate code that is as good or better than human experts.*

IV. Automated Software Generation

A. Sustainable development

DxTer is a prototype that automatically explores the space of derivable PSMs from an input PIM, and selects the most efficient PSM. DxTer produces the same or better implementations than an expert for distributed-memory systems [5], [8], [9] and we are seeing promising results for sequential implementations as well.

Architectures change very quickly. More algorithms need to be implemented. But experts are still rare and their time is valuable. Automating their job as much as possible will enable them to be more productive. We see this as a more sustainable approach to code development

[3]This particular optimization is easy to understand. On the bottom-left of Figure 2, matrix A is redistributed from the $[*,*]$ distribution to $[\mathtt{M_C},\mathtt{M_R}]$. This result is then output as matrix B and then redistributed back to $[*,*]$ to produce output C. An expert knows a more efficient implementation is to output matrix A as C, and redistribute once to produce B; this implementation is at the bottom-right.

for DLA and other CSE libraries: encode the expert's knowledge and automatically apply it to generate code for various operations and hardware architectures.

There are a number of projects that have a similar view and goal: Spiral [11] and Tensor Contraction Engine (TCE) [2] are well-known examples. In each, expert knowledge is encoded to implement domain computations and to optimize code. How this is done for each project varies, but the key is that domain knowledge is encoded in a reusable way as rewrites and these rewrites are automatically applied to find the "best" implementation.

> ***Lesson**: Experts are rare. There is far more demand on their time than they can provide. Mechanizing software development for well-understood domains should be a primary goal of SE. Doing so will achieve a sustainable way for domain-specific program development.*

B. Abstractions and layering are essential

The lessons described above have resulted in well-abstracted algorithms, reasoning, and software for DLA. We are finding that these abstractions enable us to encode expert knowledge. There is a simple theory of how DLA libraries can be constructed (see the simplicity of Figure 2) and it is enabling automatic code generation.

We find the same story about abstraction to be true for other code generators. Spiral, for example, has abstractions to represent hardware-specific features and domain-specific mathematical constructs [11]. The right abstractions and layering are essential to success.

Finding the right balance with abstraction and layering is not easy. Too many abstractions complicates software and reasoning. Too few abstractions and performance suffers. It takes a deep understanding to find the "sweet-spot" structure and organization for a domain and to fit the domain's algorithms into that structure. Gaining this knowledge takes time and effort, and even more time to make the domain's software explicitly use the structure. This results from years of prolonged efforts, guided in the belief that software can be elegant and simple.

> ***Lesson**: We are geniuses at complicating the simplest of things; the challenge is to discover its underlying simplicity.*

V. Let's Move Forward

It seems obvious that using good software abstractions and layering aid the expert. With some extra consideration (possibly a lot), we believe the abstractions can be tuned and leveraged to encode domain knowledge and automatically generate code. While we have focused on DLA here, we believe the same ideas can be applied to CSE software in general.

It is clear that this requires deep domain knowledge, so we cannot wait for external forces to reengineer our software to abstract it better. Further, we cannot wait for external forces to obviate the rote work domain experts go through. For sustainability of our software as hardware continues to change in the future, we need to work towards encoding expert knowledge and automating program construction. With code generation, we do not need to suffer (as much) with the next hardware change as we have with distributed memory, shared memory, and co-processors. This will not happen through democracy (e.g. standards committees) but by a dictatorship of domain experts who know how to get things done "best".

Acknowledgements. Marker was sponsored by fellowships from Sandia National Laboratories and NSF (grant DGE-1110007). This work was also partially sponsored by NSF grants CCF-0917167, and OCI-1148125. *Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.*

References

[1] A linear algebra software infrastructure for sustained innovation in computational chemistry and other sciences. `http://nsf.gov/awardsearch/showAward?AWD_ID=1148125`.

[2] Alexander A. Auer et al. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. *Molecular Physics*, 2005.

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

[4] Paolo Bientinesi et al. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[5] Jack J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16(1), March 1990.

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

[7] John A. Gunnels et al. FLAME: Formal linear algebra methods environment. *ACM TOMS*, 27(4), December 2001.

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

[9] Tze Meng Low et al. Theory and practice of fusing loops when optimizing parallel dense linear algebra operations. Technical Report TR-12-18, The University of Texas, Dept. of Comp Sci.

[10] Jack Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. *ACM TOMS*, 39(2), 2012.

[11] Markus Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[12] Enrique S. Quintana et al. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[13] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, June 2003.

[14] Martin Schatz et al. Scalable universal matrix multiplication algorithms: 2d and 3d variations on a theme. *ACM Transactions on Mathematical Software*, 2012. submitted.

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