Working towards Sustainable Software for Science: Practice and Experiences, New Orleans, November 2014

Making Scientific Computing Libraries Forward Compatible

Bryan Marker Don Batory Field Van Zee
Robert van de Geijn
{bamarker,batory,field,rvdg}@cs.utexas.edu

July 14, 2014

Abstract

NSF's Software Infrastructure for Sustained Innovation funds the development of community software in support of scientific computing innovation. A requirement is that the developed software be sustainable. emphdDesign-by-Transformation (DxT) is an approach to software development that views libraries not as instantiated in code, but as expert knowledge that is combined with knowledge about a target architecture by a tool (DxTer) that synthesizes the library implementation. We argue that this approach makes libraries to some degree *forward* compatible in that a (disruptive) new architectural advance can be accommodated by encoding knowledge about that architecture. This is particularly important when bugs are not correctness bugs, but instead performance bugs that affect how fast an answer is obtained and/or how much energy is consumed to compute the answer. DxT allows a human expert to focus on developing the primitives from which libraries are constructed and new insights as opposed to the rote application of known ideas to entire libraries. We summarize our success in the domain of dense linear algebra as evidence of DxT's potential.

1 Introduction

A software engineer is called a domain expert when (s)he can implement domain-specific algorithms on some hardware and attain high performance. Performance is what is commonly important to end users, but to the expert developer there is another metric for success: sustainability of the software. Current expert-developed scientific software has a common flaw: it is not *forward compatible*, meaning significant future hardware and software changes are not easily incorporated.

Scientific software is generally redeveloped as a reaction to change. When new hardware is released, domain experts spend years reimplementing, redebugging, and/or reoptimizing a library of domain-specific algorithms implemented in high-performance code. In other words, they use their knowledge to essentially develop new code in response to a major architectural change. Often, it is the same domain knowledge (i.e., underlying math and algorithms) they employed for the last architecture, but they cannot effectively reuse large parts of that code. This has happened recently, for example, first when multicore CPUs and GPGPUs were introduced, and again now that many-core processors like the Intel Xeon Phi have arrived. The hardware and APIs are too different to just mutate code from one to the other. The expert has start over.

Similarly, when developing a library of functionality from scratch, an expert uses the same software-design knowledge to implement each algorithm in his domain. Often, as the expert gains new insights, those insights must be reapplied across much of the library.

Our view is that traditional libraries are combinations of different sets of expert knowledge. They are the result of applied domain knowledge and hardware and software-design knowledge (e.g., how to

implement a function on a particular architecture). This knowledge is intertwined in the code to attain high performance, so reactions to change are painful to incorporate.

Instead of encoding the result of applying that knowledge (i.e., the libraries of code), scientific software can become forward compatible by encoding knowledge sets explicitly in *knowledge bases*. How would a user then get the code he expects for his application? A code generator would take knowledge bases and an algorithm specification and would output optimized, executable code for the architecture *du jour*.

When a user needs to react to some change (e.g., targeting code to new hardware), only additional knowledge about new algorithms, hardware, or programming models needs to be added to the knowledge base – applicable existing knowledge would be reused automatically. Instead of reimplementing a library of code, the code generator is reexecuted to get new implementations. This is forward compatibility because there is extensive reuse of prior effort.

We present our approach to achieve future compatibility, *Design by Transformation (DxT)* [1, 3, 4, 5, 6, 7, 8], which enables experts to encode design knowledge. As evidence of DxT's utility, we summarize existing results for *dense linear algebra (DLA)*, a domain that is often found at the bottom of scientific software stacks. We expect DxT or, at least the idea of separating design knowledge and implementation code, to be similarly applicable to other scientific software domains. Indeed, a next target of our research is the domain of distributed-memory parallel libraries for tensor computations, which we will discuss in future papers.

We believe this is one essential way in which computer science can contribute to computational science: making domain knowledge explicit and systematic so that it can be applied mechanically, creating a software infrastructure that not only supports sustained innovation, but sustains itself.

2 Design by Transformation

In DxT, algorithms are represented by *directed, acyclic graphs (DAGs)*. Each node represents an *operation* while edges represent data flow. Operation inputs and outputs are represented by incoming and outgoing edges, respectively. Operations have two flavors. An *interface* has no implementation details and is defined by preconditions and postconditions on its inputs and outputs. A *primitive*, on the other hand, comes with a code implementation (e.g., an API function call).

One starts with a DAG of only interfaces representing a specification. Using design knowledge, one transforms this spec into an DAG with only primitives, which implements the spec and can be mapped to code and executed on specific hardware.

There are two types of graph transformations we use. A *refinement* replaces an interface with a graph implementing that interface. It might employ architecture-specific operations or architecture-agnostic domain algorithms. For example, a refinement can encode how to parallelize an operation. *Optimizations* are transformations that replace a subgraph with another that implements the same functionality in a different way. They are chained together to improve performance.

With these, one can transform a spec DAG into a high-performance implementation. Each transformation encodes software-design knowledge about the domain or hardware. Ending graphs represent the code that has traditionally been found in libraries (i.e., the result of applying design knowledge). It is these transformations and starting graphs that we want to encode instead. We want an automated system to generate these implementations. Then, transformations are reusable across algorithms and/or architectures unlike a piece of code. Further, we can better trust a system not to miss optimizations or make coding errors.

We have developed a generator called *DxTer*. DxTer takes as input hardware-specific and hardware-agnostic transformations (knowledge bases). A user writes a specification. DxTer searches the space of implementations that can be generated from applying all combinations of input transformations. DxTer

estimates the *cost* (e.g., run time) of each implementation or compiles and times each implementation, depending on which is sufficient for the domain. It outputs the best-performing implementation just as an expert would do when manually developing code.

We see what DxTer does as what an expert would do given infinite time and patience to explore and evaluate all design options. Transformations encode pieces of knowledge that the expert would consider and DxTer explores all combinations of transformations.

Using this approach, one can add to the knowledge base or replace a subset of it to target new hardware. DxTer is then rerun for each algorithm and new implementations are generated.

3 DLA Results

We demonstrated [4, 5, 6] how to encode DLA, hardware-agnostic knowledge (algorithms) as transformations and how to encode implementation knowledge of domain interfaces (e.g., matrix-matrix multiplication) in parallel code for distributed-memory machines. DxTer-generated code used the Elemental [9] API. In all tests, DxTer-generated code that was the same as or better than what an expert developed by hand or it generated new code.[1]

In [7], we demonstrated how the architecture-agnostic transformations used to generate Elemental code were reused and augmented to target two different architectures using a new API, thus demonstrating compatibility of transformations to a completely different target architecture. This supports our claim that one can similarly target future architectures. BLIS [11] is a new framework for instantiating the *Basic Linear Algebra Subroutines (BLAS)* [2], a widely-used set of standard DLA functionality. In BLIS, BLAS routines are built from a small set of BLIS-specific computation and data movement operations and an API for blocking matrices.

We demonstrated how to encode knowledge of BLIS and the target hardware. Further, we added knowledge of the code formats for BLIS loops and matrix blocking. This additional knowledge represents what is special to the new hardware and the new BLIS API. With these additions and existing (architecture-agnostic) domain algorithms, DxTer generated sequential level-3 BLAS operations, the BLAS subset that deals with matrix multiplication.

Furthermore, we demonstrated how to encode knowledge about multicore systems to parallelize the level-3 BLAS (i.e., we targeted new hardware using the same API). We augmented the BLIS knowledge base to target the new architecture by reusing all BLIS-related transformations and just adding to them. With DxTer, we were very productive. We did not know how to best parallelize the BLAS but could rely on DxTer to regenerate all code each time we had a new idea instead of manually reimplementing all code ourselves. Figure 1 summarizes, for a representative subset of the level-3 BLAS, the improvement of generated code over Intel's highly optimized MKL library on two Intel Xeon E5 (Sandybridge) processors, which have a total of 16 cores.

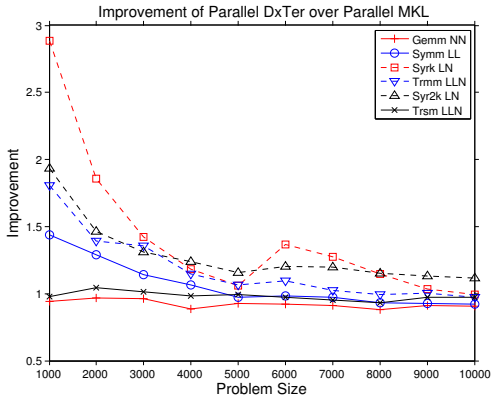

Figure 1: Improvement over Intel MKL.

[1] In one case, DxTer output code was correct while the expert-developed code had a bug. When transformations are correct, DxTer's output is correct-by-construction and can be better trusted than hand-developed code.

4 Not Just DxT

While we have had success with DxT across a variety of architectures, we are not arguing that it is the only key for forward compatibility [10]. Instead, we suggest the scientific computing field investigate ways to explicitly encode domain, hardware, and software-design knowledge instead of just the software that results from it. We want to stop reacting to hardware changes by completely reimplementing software. Instead, we aim to encode the core pieces of what we know about our software and automatically reuse as much as possible each time a change is required.

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).

References

[1] Don Batory et al. Dark knowledge and graph grammars in automated software design. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2013.

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

[3] Rui C. Gonçalves et al. ReFlO: An interactive tool for pipe-and-filter domain specification and program generation. *submitted*, 2013.

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

[5] Bryan Marker et al. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, 27(4):439–452, 2013.

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

[7] Bryan Marker et al. Code generation to aid parallel code development. Technical Report TR-14-08, The University of Texas at Austin, Department of Computer Sciences, 2014.

[8] Bryan Marker et al. Understanding performance stairs: Elucidating heuristics. In *ASE*, 2014. accepted.

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

[10] Markus Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 2005.

[11] Field Van Zee and Robert van de Geijn. BLIS: A framework for rapid instantiation of BLAS functionality. *ACM Trans. Math. Softw.* accepted.