Mechanizing the Expert Dense Linear Algebra Developer

Bryan Marker,
Robert van de Geijn, Don Batory
Dept. of Computer Science
The Univ. of Texas at Austin
Austin, TX, USA
{bamarker, rvdg,
batory}@cs.utexas.edu

Andy Terrel
Texas Advanced Computing Center
Austin, TX, USA
aterrel@tacc.utexas.edu

Jack Poulson
Institute for Computational Engineering
and Sciences
The Univ. of Texas at Austin
Austin, TX, USA
poulson@cs.utexas.edu

Abstract

The efforts of an expert to parallelize and optimize a dense linear algebra algorithm for distributed-memory targets are largely mechanical and repetitive. We demonstrate that these efforts can be encoded and automatically applied to obviate the manual implementation of many algorithms in high-performance code.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]
; D.1.3 [*Concurrent Programming*]
; G.4 [*Mathematical Software*]: Efficiency

General Terms Design,Performance

Keywords high-performance numerical algorithms, software for distributed-memory computing, dense linear algebra, program generation, MDE, libraries of the future

1. Introduction

Parallelizing and optimizing *dense linear algebra (DLA)* algorithms for distributed-memory machines has historically been done by domain experts who are very familiar with both linear algebra and the oddities of a target class of machines. When a DLA expert has no experience with a new architecture and wants to implement an algorithm, (s)he must live with an existing library, learn a lot about that architecture, or find an experienced developer. This is inefficient and unnecessary because the work of an expert is mechanical and systematic, and therefore automatable.

Expert-tuned, high-performance parallel code for distributed-memory architectures can be automatically produced by a tool via an approach we call *Design by Transformations (DxT)* [3], pronounced "dext". We demonstrated DxT on a handful of prototypical examples, simple and complex, in a broad class of dense linear algebra operations (e.g., the commonly used matrix operations in the BLAS and operations supported by libraries like LAPACK and `libflame` [5]). As our examples were targeted to a distributed-memory architecture, we believe DxT can be extended to target other architectures (such as multi-core processors, GPGPUs, and many-core processors).

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

We expect the insights from this work to have a profound impact on the FLAME project [6], which encompasses a formalism for deriving DLA algorithms, notation for expressing these as algorithms, and APIs for implementation in code. Two library instantiations exist to support a variety of parallel architectures: the `libflame` library that targets sequential, multicore, and (multi-) GPU architectures, and Elemental [2], which targets distributed memory architectures. DxT would allow us instead to support a single encoding of algorithms and knowledge, with libraries like `libflame` and Elemental being the products (outputs) of applying DxT.

2. Mechanizing Expert Transformations

When an expert parallelizes a DLA algorithm for a distributed-memory target, (s)he typically focuses on the loop body. For each of the operations or functions in that loop body, (s)he chooses an implementation code. In distributed-memory programs, matrix data is distributed among processors. In Elemental, for example, the default distribution views processes as a 2-dimension grid and stores the data in a 2-dimensional, block-cyclic distribution with a blocksize of one. To parallelize each operation in the loop body, an expert redistributes the data from the default distribution in some way that enables the computation to proceed in parallel, and then redistributes the result back to the default distribution. There are often multiple implementation choices that perform each loop body operation correctly, but they get varying performance depending on the machine architecture, problem size, etc. An expert chooses implementation codes based on a rough idea of the runtime cost of redistributing data and the runtime cost of computation. Redistributing data in Elemental requires an expensive collective communication operation, so an expert optimizes programs by reducing the amount of data redistribution. Once parallel implementations are chosen for loop body operations, an expert can see how data is redistributed and can remove redundant communication.

Step by step, an expert implements a DLA algorithm in high-performance code by transforming the algorithm with implementation choices of sub-operations, using a rough estimate of runtime costs, and transforming with optimizations that decrease the estimated runtime cost of communication. DxT attempts to mechanize this strategy by encoding the transformations performed by an expert instead of the resulting code. Then, those transformations can be applied automatically using a tool, as described below, instead of being re-applied by rote across many algorithms in the domain.

We view software as a stack of layers. Each layer provides details about an operation's implementation, so the transformations we encode break through these layers to expose implementation details that can be optimized (we encode transformations for many implementation choices, not just one). The optimizing transforma-

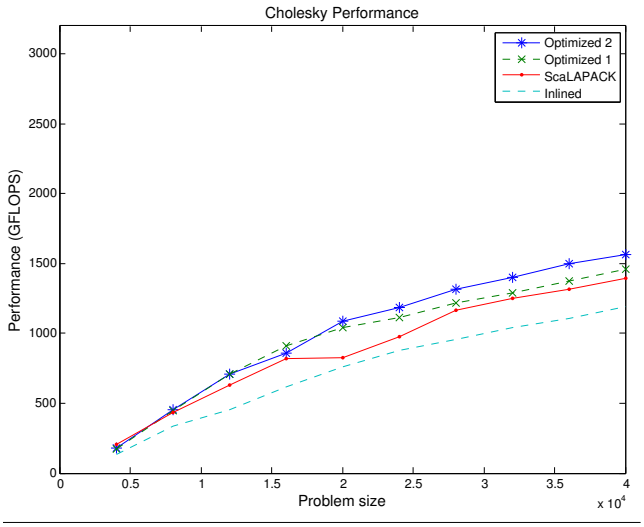

Figure 1. Automatically generated Cholesky Variant 3 implementations tested on 240 cores (and ScaLAPACK results for comparison). Peak performance, 3200 GFLOPS, is at the top of the graph.

tions are viewed as encoding equivalent sets of functionality. An expert's optimizing transformation replaces one piece of code with a functionally-equivalent piece of code that is better-performing. For DxT we encode many such equivalences.

When an expert applies transformations to generate parallel, optimized code, (s)he uses a rough estimate of operation costs to make choices on which transformation is best to use. We use rough estimates of operation costs in terms of the number of floating-point operations performed and the speed of the machine, and of the cost of collective communication operations in terms of the amount of data being moved and the network latency and bandwidth. For DxT we encode these cost estimates for each operation and they guide the choice of implementing and optimizing transformations to use [4]. The next section demonstrates the efficacy of this approach.

3. A Prototype for Automatic Program Generation

We developed a prototype to test DxT. Algorithms are represented as data-flow, directed, acyclic graphs (DAGs). Transformations are encoded as graph rewrites similar to the those performed by compilers on DAGs. The prototype has multiple implementing transformations for BLAS and LAPACK operations, and has dozens of optimizing transformations that replace patterns of collective communication with equivalent, but faster, implementations.

Our prototype takes as input the algorithm being implemented, encoded as a DAG, and iteratively applies transformations. At each step, the prototype applies any transformation that can be applied to any graph it has developed. Thus, all possible implementation codes from the input algorithm are generated. A cost estimate is attached to each graph, as described above, and is used to select the parallelized, optimized code that is least costly.

We tested the prototype, its transformations, and its cost estimates on two Cholesky factorizations variants, three matrix-multiplication (Gemm) variants, a triangular solve with multiple right-hand sides (TRSM) variant, and a significantly more complicated variant of a two-sided triangular solve [1]. The Cholesky factorization algorithm is prototypical in DLA codes. Its implementation and optimizations require an expert to perform many transformations that are common throughout the domain. For both variants of Cholesky that were tested, the system generated hundreds of implementations. The "best" versions were chosen using the above-described cost estimates, and these versions were the same as those hand-generated and optimized by the expert developer of Elemental. In Figure 1, we show performance of one of those variants on 240 cores. Here, the "Inlined" results come from simply implementing that algorithm directly in code, without applying any optimizations. The "Optimized 1" line shows performance with some simple optimizations applied by the system, removing redundant communication. The "Optimized 2" line shows performance for the expert-generated code, which the prototype also generated and calculated as "best" using its cost estimates. This version results from some additional optimizations being performed on the "Optimized 1" version; they have to do with memory access patterns and cache-reuse. ScaLAPCK results are shown for comparison.

Similar results can be seen for the other operations tested. The prototype generated and chose as "best" the same code developed by an expert. For the two-sided triangular side, the generated implementation was slightly better than the version written by the expert as the prototype applied an optimization the expert forgot (which has since been incorporated into Elemental).

4. Conclusion

We demonstrated for a handful of prototypical algorithms how the implementing and optimizing transformations an expert performs can be mechanized. Instead of requiring an expert to re-apply the transformations by hand for many algorithms in a domain, a tool can do this task automatically. We see DxT as a sustainable approach to library development as architectures continue to change, requiring code to be re-developed.

Acknowledgments. Marker was sponsored by a fellowship from Sandia National Laboratories and an NSF Graduate Research Fellowship under grant DGE-1110007. Poulson was sponsored by a fellowship from the Institute of Computational Engineering and Sciences. Batory is supported by the NSF's Science of Design Project CCF 0724979. This research was also partially sponsored by NSF grants OCI-0850750 and CCF-0917167 as well as by a grant from Microsoft.

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] J. Poulson et al. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. *ACM Transactions on Mathematical Software*. submitted.

[2] J. Poulson et al. Elemental: A new framework for distributed memory dense matrix computations. FLAME Working Note #44 TR-2010-20, The University of Texas at Austin, Department of Computer Sciences, 2010. Submitted to ACM TOMS.

[3] T. Riche et al. Software architecture design by transformation. Computer Science report TR-11-19, Univ. of Texas at Austin, 2011.

[4] P. G. Selinger et al. Access Path Selection in a Relational Database Management Syst em. In *ACM SIGMOD*, 1979.

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

[6] F. G. Van Zee et al. Introducing: The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6): 56–62, 2009.