MATFORCE: USING AUTOMATED SOURCE-TO-SOURCE TRANSLATION IN DEVELOPING C++ APPLICATIONS BASED ON MATLAB CODE

Levente Hunyadi — *hunyadi@aut.bme.hu* — PhD student

Miklós Nagy — *mikee85@gmail.com* — MSc student

Budapest University of Technology and Economics
Department of Automation and Applied Informatics

Abstract

MatLab is an essential tool in high-productivity development of applications that involve much scientific computation. However, for the sake of efficiency and homogeneity with other parts of the code, it is often necessary to convert MatLab code into C++, which is a tedious and error-prone task if performed manually. The authors present a tool named MatForce that automatically converts MatLab functions into C++ code, producing human-readable, extensible C++ sources that can in turn be fitted to the needs of the encapsulating application.

1 Motivation

MatLab [3] is the de-facto language of technical computing, it allows programmers to write algorithms in familiar mathematical notation as well as visualize their results in a straightforward manner. On one hand, MatLab has built-in support for matrix operations and a myriad of mathematical functions ranging from trigonometric functions to fast Fourier transformation. On the other hand, data can be displayed graphically and interactive tools allow manipulating graphs to achieve results that reveal the most information. Both of these aspects contribute to ease of use and quick initial discovery of potential pitfalls. However, beyond the initial prototyping phase, one is often faced with the need to integrate the developed algorithms into an existing application, often with subtle changes.

Our work has been motivated by developing a link capacity dimensioning algorithm as part of a mobile telecommunication network designer tool. Traffic in a radio network is often transported over an ATM network in which each unit of traffic is carried in a so-called Virtual Channel Connection (VCC). VCCs are classified into real-time (e.g. for voice), non-real-time (e.g. for packet-based traffic) and other types (e.g. for high-speed download packet access (HSDPA) traffic), each having different Quality of Service (QoS) requirements. However, different types of traffic affect one another due to the limited capacity of the physical link. The goal of the algorithm was to find the minimal capacity for each VCC for which all QoS requirements are satisfied given a general traffic mix defined by the user.

As traffic is frequently modeled as a Markov chain where each state is associated with a given traffic rate, matrix operations were inherent in the dimensioning algorithm. Particular examples included matrix inversion, matrix factorization as part of solving linear systems of equations, submatrix extraction and Kronecker operations for composing sets of states. While having a sufficient level of abstraction in order to provide easy access to data stored in an object-oriented fashion, the algorithm was expected to run efficiently, avoiding temporary copies and array reordering (especially for the sake of transposing the matrix) whenever possible. Consequently, our goals were threefold: (1) avoiding laborious manual conversion of code given as MatLab functions; (2) seamless integration with existing C++ code and (3) generating highly efficient code. Direct use of linear algebra libraries, such as LAPACK [7] or ATLAS [16], was inadequate due to their low level of abstraction, while other tools, such as the compiler shipped with MatLab, were not suitable as they would produce coarsely-grained execution units (DLLs wrapping MatLab code) that could not utilize C++ objects specific to our domain.

In order to meet the outlined requirements, we propose MatForce [10, 4], an open-source tool comprising of a compiler and a utility library, which translates MatLab code into human-readable C++ source code. The *compiler* features a simple yet powerful type inference algorithm that uses primitive as well as compound types in compiled code. The resultant C++ code is dependent only on the *utility library* that contains definitions for various matrix types, shapes and operations (such as product of two double-precision matrices or Fourier-transform of a vector), which makes compiled code succinct and sufficiently abstract. The utility library links against freely available BLAS and LAPACK [7] linear algebra routines written in Fortran but no other third-party libraries. As a result, the compilation process yields intuitive, type-safe, high-performance C++ source code, which the programmer can modify at will without relying on external heavy-weight libraries.

The rest of the paper is structured as follows. Section 2 gives the motivation for using type inferring code translation as opposed to other approaches and surveys related work. Sections 4, 5 and 6 deal with the details of the MatForce compiler, the utility library and the (external) linear algebra library, respectively. The paper ends with Section 7, which summarizes results and drafts possible future work. The reader is assumed to have a basic knowledge of the MatLab language.

2 Related work

There have been attempts at increasing MatLab performance by means of partial evaluation [9], just-in-time compilation [6], parallelization [12], direct compilation [11] and type estimation [13, 2].

Partial evaluation is a technique for program optimization by specialization, constraining code to a particular set of possible inputs, producing the so-called residual program. The MatLab partial evaluator described in [9] transforms a MatLab abstract syntax tree into a simpler but equivalent form by deducing type, shape and value range information from expressions, evaluating static subexpressions and eliminating dead code whenever possible as well as unrolling loops. This partial evaluation can substantially increase performance but produces MatLab code and hence can only serve as a possible preprocessing step in our scenario.

MaJIC [6] is a just-in-time compiler targeted at speeding up MatLab execution in an interactive environment. The compiler comprises three constituents: an analyzer that annotates code, a very fast code generator and a code repository that caches compiled code. Despite its effectiveness due to preallocating temporary arrays, eliminating unnecessary temporaries and unrolling loops, the just-in-time scheme is heavily dependent on run-time information and is therefore not suited to our needs.

Direct compilation speeds execution by translating to a compiled language such as C or C++ but by using a dynamic typing scheme. As such, it is a simple approach that may significantly speed execution by compiling control structures but it does not address type safety, nor can it exploit the speed gain from type specialization. Nevertheless, compiled code can bear close resemblance to the original, facilitating future extension.

In the case of type estimation (or type inference), variables are assigned well-defined types, allowing code to be translated into a strongly-typed language. As a result, type inference combined with code translation meets our demands both in terms of speed and type safety. Not only does type inference enable aggressive optimization of generated C++ code (by allowing cross-optimization between hosting application and MatLab algorithm) but also decreases the heterogeneity of a complex system by making it possible to use a single programming language.

In [13] a type estimator written for the Octave [8] language is described, which is very similar to MatLab in syntax and semantics. The type estimation scheme is based on flow graphs to guess the intrinsic type, size and value range of Octave matrices in an iterative process. This information is stored as a triplet and attached to each variable in an Octave program. Once the Octave program is annotated, a corresponding C++ program can be generated.

3 Overview

The MatForce system comprises of three easily identifiable parts. The *compiler* performs the MatLab to C++ translation with type inference, yielding C++ code. The *utility library* contains the implementation of matrix operations. The C++ compiler translates both the utility library and the C++ source code produced by the MatForce compiler into object files. The linker binds the object files to the high-performance *linear algebra library*, which carries out computation-intensive operations. (Figure 1)

4 The compiler

The MatLab code *compiler*, written in SWI-Prolog [15, 5], is the major constituent of the system. Compilation is done in four stages: tokenization, parsing, type inference and code generation.

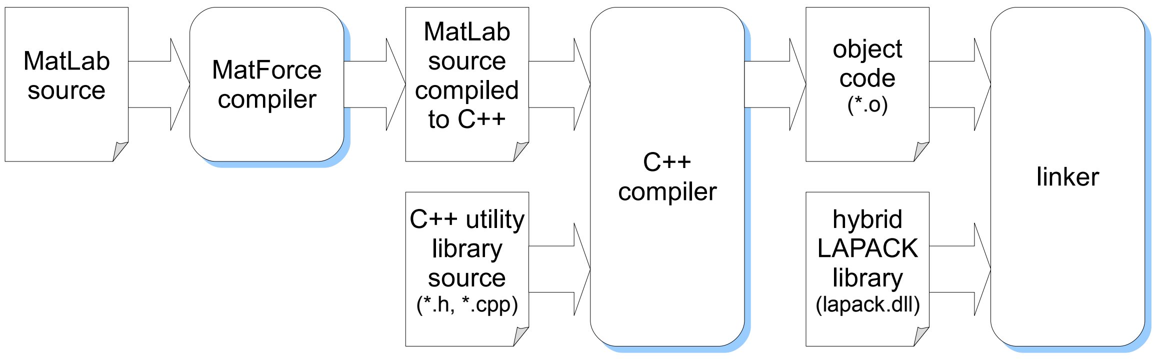
MatLab source
MatForce compiler
MatLab source compiled to C++
C++ utility library source (*.h, *.cpp)
C++ compiler
object code (*.o)
hybrid LAPACK library (lapack.dll)
linker

Figure 1: The MatLab to C++ compilation and link process

Tokenization At this stage, m-files are read and converted into a stream of tokens. The term *token* may refer to an integer, a real number, a keyword, a string, a literal, a symbol (which corresponds to a MatLab operator), a comment or a newline. Line folding is also performed at this stage.

Parsing During the parsing process, tokens are interpreted in their context and an abstract syntax tree (AST) is built. Each m-file contains one or more *functions*, each of which comprises multiple *statements*. A statement may be a control statement (*return*, *break*, *global* etc.), an assignment, an *if*, a *while* or a *for* statement, or a function call. The exact semantics of each statement is defined by means of the DCG [14] formalism, which is similar to BNF and is common to Prolog-flavor languages.

Type inference As MatLab is a dynamically-typed language, variables are not explicitly associated with types. In contrast, their type is determined run-time using the implicit type of the initializer or assignment expression, and that type is changed as necessary whenever another expression demands so. In our case, a simplifying (but scarcely restrictive) "same type" assumption is made, that is, a variable cannot have incompatible types in the same MatLab function. For instance, if a variable *i* has been concluded to have the type *integer*, it cannot be part of an expression where it should be interpreted as a *string*, which is not more general than *integer*.

Having made the "same type" assumption, the compiler uses a domain narrowing technique, common to constraint logic programming (CLP), to infer the type of a variable. Once a variable has been introduced, it is associated with a type domain. Initially, the domain contains all possible MatLab types. Types are classified into

- intrinsic types; *boolean*, *integer*, *real* and *complex*
- primitive types; all intrinsic types and *string*
- matrices and vectors, which are made up of elements of the same intrinsic type

Narrowing occurs in any of the following two situations:

1. *Direct assignment*. The variable is assigned an expression. For instance, the assignment *a* = *1* + *2* allows the compiler to infer that *a* is an integer, a real or a

complex number; *a = b + 4* leads the compiler to conclude *a* is a real or complex matrix if *b* is a real matrix.

2. *Implicit from context.* The variable is used in an expression in a context that allows deductions to be made as to the type of the variable. For instance, the assignment *a(6,b) = 11* implies *a* is a matrix because only matrices can be accessed with two indices. Similarly, the compiler can conclude that *b* is an integer or an integer vector: other types are not allowed as indexers.

In each case, the domain of the variable is the intersection of its current domain and the possible domain deduced from the expression. If the domain of a variable becomes the empty domain, the compilation halts with an error. Ideally the domain reduces to a single type by the end of the compilation unit. If not, the hierarchy of types (e.g. the type *integer* is a special kind of 1-by-1 matrix) can be exploited to choose the simplest type that does not harm code semantics.

Code generation In the last compilation stage, the type-annotated AST is converted into a series of C++ expressions. The compiler takes operator precedence into account in order to use the least number of parentheses for improved legibility. Some of the simplifications and optimizations at this stage include using increment and decrement operators, simultaneous arithmetic and assignment (e.g. add-and-assign), and iterators in loops.

5 The utility library

The *utility library,* written in C++, exposes a strongly-typed class hierarchy of different types of vectors, matrices and 3-dimensional arrays. Matrix types are differentiated based on the intrinsic type they store, which can be unsigned integer, signed integer, double-precision real or complex numbers. Generic functionality, common to more types and shapes, is implemented in corresponding storage and matrix classes. Operators are overloaded to provide a convenient way to express arithmetic operations. While simple operations (such as computing the element-wise minimum of two matrices) are implemented directly in C++, more complex operations (such as matrix multiplication or inverse) are performed by the underlying linear algebra library.

6 The linear algebra library

The *linear algebra library,* written in Fortran, serves as the fundamental layer for high-performance computation. It contains efficient implementation of such operations as matrix multiplication, matrix decomposition or eigenvalue computation. The linear algebra library itself comprises two layers: the lower BLAS (Basic Linear Algebra Subroutines) layer and the higher LAPACK (Linear Algebra PACKage) [7] layer. The most critical part of the library in terms of efficiency is the BLAS layer, which implements basic vector, matrix-vector and matrix-matrix operations, all of which are used both by the LAPACK layer and the C++ utility library. ATLAS [1, 16], which is an automatically tuned BLAS (and partial LAPACK) implementation, is used to exploit the specific features of the target computer's hardware.

7 Conclusions and future work

In this paper, a type-inferring compiler that converts MatLab scripts to C++ source code as well as an accompanying utility library has been presented. Albeit it uses optimized open-source BLAS and LAPACK routines written in Fortran for the sake of efficiency, the compiled code does not have any external dependencies on third party proprietary libraries. The generated C++ code is properly annotated with types, using primitive types whenever possible. The code is fully legible from a human perspective using matrix and vector types as well as related operators, which allows further additions or modifications to the translated code.

Future work includes a user interface to control the compilation process, support for a greater proportion of MatLab functions in the supplemental C++ utility library, and a study of comparative performance w.r.t. other approaches.

References

[1] Automatically tuned linear algebra software. `http://math-atlas.sourceforge.net/`, 2008.

[2] Catalytic MCS family: MATLAB to C synthesis. `http://www.catalyticinc.com/`, 2008.

[3] MATLAB: The language of technical computing. `http://www.mathworks.com/products/matlab/`, 2008.

[4] SourceForge.net project site of MatForce. `http://matforce.sourceforge.net/`, 2008.

[5] SWI-Prolog. `http://www.swi-prolog.org/`, 2008.

[6] George Almasi and David A. Padua. MaJIC: A Matlab just-in-time compiler. In S. P. Midkiff et al., editors, *LCPC2000 (LNCS 2017)*, pages 68–81, 2001.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition edition, 1999. ISBN 0-89871-447-8 (paperback).

[8] John W. Eaton. Octave: Interactive language for numerical computations. `http://www.gnu.org/software/octave/doc/interpreter/`, 2007.

[9] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In F. Pfenning and Y. Smaragdakis, editors, *GPCE 2003 (LNCS 2830)*, pages 344–363, 2003.

[10] Levente Hunyadi. MatForce: Supporting rapid algorithm development by automated translation of MatLab prototypes into C++. In *IASTED International Conference on Software Engineering*, 2008. (in print).

[11] Y. Keren. MATCOM: A MATLAB to c++ translator and support libraries. Technical report, Israel Institute of Technology, 1995.

[12] S. Pawletta, T. Pawletta, W. Drewelow, P. Duenow, and M. Suesse. A MATLAB toolbox for distributed and parallel processing. In C. Moler and S. Little, editors, *Proceedings of the Matlab Conference*, Cambridge, MA, October 1995. MathWorks Inc.

[13] Jens Rücknagel. An octave type estimator – essential part of an octave to c++ compiler. Master's thesis, Technical University Ilmenau, Faculty of Computer Science and Automation, Department of System and Control Theory, April 12 2005.

[14] C. M. Sperberg-McQueen. A brief introduction to definite clause grammars and definite clause translation grammars (a working paper prepared for the W3C XML Schema Working Group). `http://www.w3.org/People/cmsmcq/2004/lgintro.html`, July 18 2004.

[15] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 1994.

[16] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.