2016

Fastnumerics: Compiling Matlab To C++

Sujan Khadka
University of Alabama at Birmingham

FASTNUMERICS: COMPILING MATLAB TO C++

by

SUJAN KHADKA

PETER M. PIRKELBAUER, COMMITTEE CHAIR
MARJAN MERNIK
PURUSHOTHAM BANGALORE

A THESIS

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Master of Science
BIRMINGHAM, ALABAMA

2016

FASTNUMERICS: COMPILING MATLAB TO C++

SUJAN KHADKA

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

Matlab is a popular language among researchers and scientists. It allows mathematical and scientific calculations to be formulated in a way that is close to mathematical notation which makes developing prototypes easier and faster. Our preliminary results show that computation intensive programs written in Matlab tend to be slower than equivalent programs written in C++. C++ being a compiled language exposes advanced optimization opportunities that will help speed up sequential code as compared to Matlab. Hence, there is a need to translate Matlab code to languages like C/C++ for maximum performance. C++ also provides libraries to run a program in heterogeneous architectures like co-processors, GPUs and in distributed environments. Although Matlab makes it possible to run Matlab codes in parallel and in GPUs using its parallel toolbox, we would like to utilize C++ to squeeze out maximum performance and gain more portability. With C++ we gain more flexibility and portability to run in any architectures and environments due to the ubiquity of C++ compilers.

Manual translation to C++ is one option but it is tedious, cost inefficient and can introduce errors. In this thesis we introduce a tool to automatically compile/translate Matlab programs to C++. The code generated will rely on existing numerical libraries. We would like researchers to continue writing codes in Matlab but also have the added benefit of being able to run their translated code much faster and possibly on heterogeneous architectures.

DEDICATION

This thesis is dedicated to my advisor Dr. Peter Pirkelbauer for providing me with all the support and encouragement.

ACKNOWLEDGEMENTS

I cannot express in words how grateful I am to my advisor and my mentor Dr. Peter Pirkelbauer. I am fortunate to have an advisor who provided me the freedom to explore and also guided me during my difficult times. His suggestions, ideas and encouragements made my journey wonderful.

I would like to thank Dr. Purushotham Bangalore and Dr. Marjan Mernik for being on my thesis committee and providing feedback on my thesis. Dr. Bangalore has provided me with a lot of feedback during our weekly meetings and I want to thank him for that too.

I am extremely thankful to my friends Sagar Thapaliya, Amin Hassani and Suraj Poudel for their wonderful suggestions and advice. I am very grateful to my team mates Reed M Milewicz, Hadia Ahmed, Juan Felipe Gonzalez and Nick Dzugan for helping me with the ROSE compiler and sharing fun times at our weekly meetings where we discussed research papers. I also want to thank my friends Pradip Chitrakar, Prakash Shrestha, Ligaj Pradhan, Farig Yusuf Sadeque and Abhishek Anand for our good times, laughs and coffee when I was working on my thesis. I used a machine called iprogress-phi for my thesis and I would like to thank Walker Haddock for setting up this beautiful machine. This list would be incomplete without thanking Nathan Day and Amalee Wilson for helping me thrive on my project with their questions and feedback.

I got an opportunity to spend a summer with the ROSE compiler team at Lawrence Livermore National Laboratory. I would like to thank my mentor Dr. Dan Quinlan for

his ideas on how Matlab should be integrated to ROSE. I would like to acknowledge Tristan Vanderbruggen and Dr. Pei-Hung Lin for teaching me ROSE compiler's internals and helping me when I faced problems with ROSE.

This the project was supported by the CAS interdisciplinary team award (March 2014) and NSF grants CNS-0821497 and CNS-1229282.

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1

DESCRIPTION

1.1 Introduction

Matlab[1][2] is easy to use and provides a high level of abstraction to solve scientific problems. People who code in Matlab would find it harder and time consuming to use general purpose languages like C++ [3] and Java[4] to solve the same problem. Matlab offers a platform to easily express mathematical and scientific ideas with high level of abstraction.

Our results obtained by manual translation of Matlab code to C++ show that solutions written in Matlab that are computation intensive tend to take a long time to run as compared to equivalent programs written in C++. This can be attributed to Matlab being an interpreted and dynamically typed language. There are ways in Matlab to leverage some parallelizations to make code faster but we would like to generate C++ code so that we could use the generated code for further researches like scheduling the computation on heterogeneous processors.

Matlab can be seen as a prototyping language which allows people to quickly test their ideas without investing too much time looking for libraries and writing code. After the prototype is created, it would be better to port the solution to a language like C++ which has aggressive optimization options and can be configured to produce native machine code. The other benefit of writing a program in a language like C++ is that it is more portable and there are many libraries in C++ that allows computation

to be performed on parallel architectures. This will allow programmers to leverage existing power of the machines to write programs that execute much faster. It will also provide an opportunity to explore porting C++ codes to utilize different processors, architectures and GPUs so as to squeeze the maximum available performance; all of these relying on ubiquity of C++ compilers. Manual translation to C++ could be done but it has some additional costs. It is time consuming; there could be errors during translation; and skilled programmers are needed who have a good understanding of both Matlab and C++.

To address this problem, we propose a tool to automatically translate Matlab code to C++. The translated C++ code will be such that it can utilize any numerical libraries. The tool will provide a framework to perform analysis on Matlab code.

The rest of the thesis is organized as follows: Section §1.2 explains the motivation behind our work and the Section §1.3 briefly states the gist of our thesis. Section §1.4 will discuss about the approach we use to compile Matlab to C++. Chapter §2 introduces some terminologies used in the thesis. Chapter §3 describes the different components of FastNumerics. Chapter §4 discusses FastNumerics Support which provides implementation of Matrix data structure and Matlab built-ins. The section §4.1 describes the matrix wrapper used by the generated code. Chapter §5 discusses our approaches to type inference in more detail as well as the challenges involved. Transformations are discussed in more details in chapter §6. Then we will present our experiments and results in chapter §7. Chapter §8 will discuss related works on Matlab transformation. In chapter §9, we discuss how FastNumerics can be used as a framework for future researches. Finally we end the thesis with a conclusion.

1.2 Background and Motivation

A team in UAB's Bio-medical Engineering department is studying electrical activation waves and mechanical deformation over the entire surface of the heart [5][6].

The Matlab program written by the team will track images of heart from multiple high-speed cameras, align them and generate a model of heart motion. The whole simulation takes around 30 minutes - 1 hour to execute. The component that is computation intensive is the calculation that computes the centroid of markers on a heart in multiple video frames and tracks them.

The issue with this program is that the work must be redone if the markers cannot be properly aligned during the simulation resulting in delays to produce the model. Thus there is a need to reduce the computation time to get early feedback and make changes to the setup. The researchers are comfortable writing programs in Matlab and it is difficult for them to re-write the program again in C++ to speed it up. Our framework will automatically translate the code to C++ which could then be run in a way to increase the performance.

This thesis is also motivated by a work that manually translates Matlab to C++ which is also going on at the Department of Computer and Information Sciences in UAB to check how efficient the code produced can be. As a parallel work, this thesis is looking into automatic translation of Matlab to C++.

1.3 Thesis Statement

This thesis will study various phases and challenges associated with the translation of a dynamically typed programming language(Matlab) into a statically typed language(C++). Here are the contributions of this thesis:

1. This work will develop a light-weight framework that translates general Matlab code to type inferred C++ code with adequate level of abstraction.

2. We will also add Matlab to the list of languages supported by ROSE which is an open source compiler infrastructure.

3. The framework will generate code which depends on numerical libraries that

can be swapped without changing the translated code.

4. A comparison between flow dependent and flow independent type inference algorithms will also be done in this work.

1.4 Approach

Compiling a programming language to another language involves generating an Abstract Syntax Tree(AST) [7] by parsing the input language source code, performing analyses and transformations on the AST to convert it to the AST of the target language, and then unparsing the AST to produce the target source code [8]. We follow a similar principle in FastNumerics which has been inspired by previous works on translating Matlab to other languages[9] [10]. Figure 1.1 illustrates our approach to transform Matlab to C++.

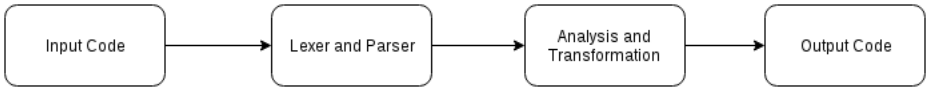

Figure 1.1: Approach for programming language translation

CHAPTER 2

TERMINOLOGIES

Here are a few terminologies we will use throughout this thesis:

2.1 ROSE Compiler Infrastructure

ROSE [11] is an open source compiler infrastructure developed at Lawrence Livermore National Laboratory(LLNL) to build source-to-source program transformation and analysis tools [12][13] for large-scale C(C89 and C98) [14], C++(C++98 and C++11) [15], UPC [16], Fortran (77/95/2003) [17] [18], OpenMP [19], Java [20], Python [21] and PHP [22] applications. ROSE defines a rich API to build the AST, perform transformations on the AST and perform data flow and control flow analysis [23]. ROSE has a good set of APIs for traversing the AST. ROSE can be used to write source to source transformation [24], code analysis and optimization tools. FastNumerics uses ROSE Compiler Infrastructure to represent the AST, to perform analyses and transformation and to unparse the AST. All the analyses written in FastNumerics heavily use ROSE API for AST traversal and manipulations.

ROSE is a good choice for writing source to source translators because many AST nodes are common between different languages and therefore an AST built for one language can be un-parsed using the unparser for another language. For example a Matlab function can be represented in ROSE AST as a SgFunctionDeclaration which can be directly unparsed to C++ without doing any transformations. Hence we can re-use most of the existing AST nodes. ROSE did not have support for Matlab based

nodes like matrix, ranges, etc. FastNumerics contributes to ROSE by adding those nodes. These nodes have already been added to the ROSE compiler Github repository [1].

2.2 Lexer and Parser

A lexer scans the source code text, tokenizes it, and passes it to the parser. The parser will then execute actions according to the rules specified in the grammar to create an Abstract Syntax Tree. We modified GNU Octave's lexer and parser to use in FastNumerics. We chose GNU Octave's [25] grammar because its grammar is similar to Matlab grammar[26].

2.3 Abstract Syntax Tree(AST)

Abstract Syntax Tree is the representation of source code as a tree of nodes[27]. Each node represents different language constructs like expression, statement, function, scopes, etc.

An AST is the typical output from the parser of a compiler. It represents the lexical/syntactical structure of the program text. We use ROSE compiler's SageBuilder API to create an AST from Matlab code. AST nodes are hierarchical in nature. In ROSE, every AST node is a SgNode. Each AST node derives from SgNode to represent different constructs like operators, statements, values, etc. Figure 2.1 shows a general hierarchy of ROSE compiler's AST nodes. All expressions like Matrix expressions (SgMatrixExp), binary expressions, Range expressions, etc. inherit from SgExpression, all statements like for loops (SgMatlabForStatement), if statements, etc. inherit from SgStatement and so on.

[1] https://github.com/rose-compiler/rose-develop

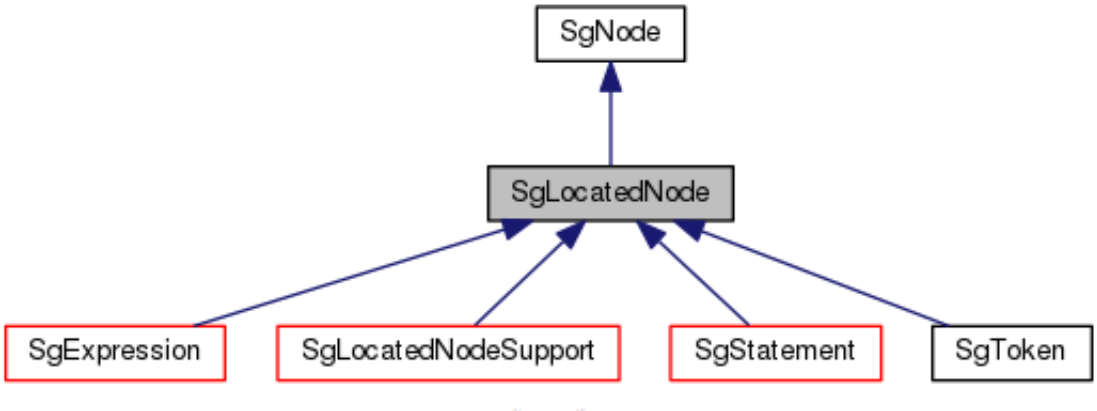

Figure 2.1: A sample of ROSE compiler's AST hierarchy

2.4 AST Transformation

AST Transformation is the process of modifying the AST by inserting, removing or replacing nodes. The purpose of transformation is to produce a different code that produces the same result. For example, an AST node to represent a function call could be modified to add an extra argument at the end of the call or a Matlab for loop node could be replaced by an equivalent C++ for loop.

2.5 AST Traversal

In order to query an AST, or to make changes to the AST, the nodes of an AST need to be visited. This process is called AST Traversal. The two main ways of traversing the AST in ROSE are:

1. Bottom-up traversal: where the children nodes are visited first and they can pass information up to the parent node.

2. Top-down traversal: where the parent node is visited first and the parent node can send information to its children.

ROSE compiler infrastructure provides APIs to easily traverse the AST.

CHAPTER 3

FASTNUMERICS COMPONENTS

A block diagram of FastNumerics can be seen in Figure 3.1. FastNumerics can be divided into three components- the frontend, the midend and the backend. The frontend deals with parsing the Matlab source file and producing an AST. The midend deals with all the semantic analysis and structural transformations done to convert Matlab based AST to C++ based AST. Midend is also responsible for performing type inference in the AST. The backend is responsible for generating C++ code from the transformed AST. Backend also provides an option to unparse the AST back to Matlab.

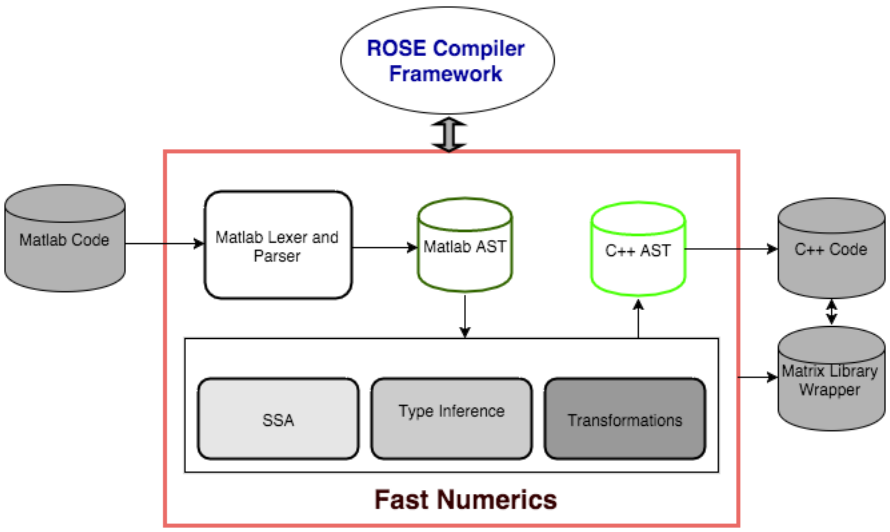

Figure 3.1: Overview of FastNumerics

3.1 Frontend

FastNumerics frontend takes a Matlab .m file as an input and produces an AST. The root of the AST is a SgProject node. Some nodes specific to Matlab were also added to ROSE for a better representation of the AST. The following sub-sections describe the frontend.

3.1.1 Lexer and Parser

We modified GNU Octave's[28] grammar and built the AST using ROSE. GNU Octave uses GNU Bison [29] which creates a LR grammar [7]. GNU Octave is a free and open source alternative to MATLAB, so its grammar resembles Matlab's grammar [26].

Grammar in GNU Bison is represented by rules and actions associated with each rule. We replaced the actions in each rules to use ROSE's SageBuilder [30] API to build the AST.

3.1.2 Matlab Constructs Supported

Matlab has many constructs and supporting all of them is a tedious job. For our thesis, we have chosen to support a subset of the Matlab constructs. We chose constructs in such a way that it is sufficient to write a Matlab program in a flexible way. We did not support constructs like lambda functions, structs, classes, switch cases, global and specific commands like nargin, tic, etc. Currently it supports all assignment expressions, matrix representation, range expressions, operators, if-else, function declarations, function calls and for loop which can be used to write fairly decent Matlab programs.

3.1.3 Nodes Added to ROSE

Many nodes existing in ROSE were utilized to create Matlab AST but some new nodes had to be introduced to ROSE to properly represent a Matlab program. In particular, matrix operations and nodes are missing in standard C++. For example it is cumbersome to represent a Matlab matrix using an Array node, or to represent a range expression in Matlab by mixing some primitive nodes. Here are the nodes added to ROSE to support the parsing of Matlab:

1. Types: The nodes added to ROSE to represent types are TypeMatrix which represents Matrix type and TypeTuple which represents a list of types. A normal matrix expression is of type TypeMatrix. Also a range expression (eg. 1:2:100) is of type TypeMatrix. Matlab can return multiple values of different types from a function and TypeTuple can be used to represent the return type of such functions.

2. ElementWiseOperators: The languages that ROSE supported did not have element-wise operators. Element-wise operators are the operators which operate on each element in a Matrix. For example in Matlab, A .* B will create a new Matrix C (where Cij = Aij * Bij) by multiplying each element in A to corresponding element in B instead of doing a matrix multiplication. To represent such operators like A .* B, A .+ B, etc. element-wise operators for addition, subtraction, division, multiplication were added.

3. Other operators: ROSE did not have operator nodes that represent left division (eg. $A\backslash x$), transpose and power operator. Left division operator is used to solve a system of linear equations; transpose operator is a unary operator that represents transpose operation on a Matrix; and power operator as the name implies is used to raise a matrix or a number to a certain power. FastNumerics extends ROSE by adding these operators.

4. Expressions: A Matlab AST would not be complete without expressions that represent Matrix, Range and MagicColon. These nodes are new to ROSE as the existing languages like C++, Java do not have a direct representation. These AST nodes were also added to ROSE compiler infrastructure.

5. For loop: None of the languages that ROSE supports have a for loop that has the semantics of Matlab. A MatlabForStatement node was added to ROSE to represent a Matlab for loop.

   A MATLAB for loop is of the form:

```
1 for i = 1:5:100
2    [operations]
3 end
```

Now any one can create a Matlab AST using ROSE's SageBuilder using these nodes which is one of the contributions of this thesis.

3.2 Midend

Midend takes the AST from the frontend as input and performs various analysis and transformations on it. The output from midend will be a C++ AST that is ready to be unparsed to C++ source code.

3.2.1 Analyses Performed

Type inference. Since C++ is a statically typed language and Matlab is a dynamically typed language, type information needs to be added in the AST to generate a well typed C++ code. This process of finding out types from an untyped representation is called type inference. There are many approaches to perform type inference. In our work we tried two different approaches, one is a flow based type inference and

the other is a flow insensitive inference. The type information generated from type inference is used to construct type declarations of variables which is needed in a static language like C++. More details can be found in Chapter 5.

Transformations. Transformation is the heart of source to source translators. Matlab based AST generated from the frontend has the semantics of Matlab and contains many nodes that are only present in Matlab. For example, a SgMatrixExp node that represents a matrix in Matlab has to be transformed into something equivalent which C++ can recognize and compile without any syntax errors. For example, a matrix X in Matlab is represented as:

```
1 X = [ 1 2; 3 4]
```

Figure 3.2 shows the AST that represents this Matrix in Matlab:

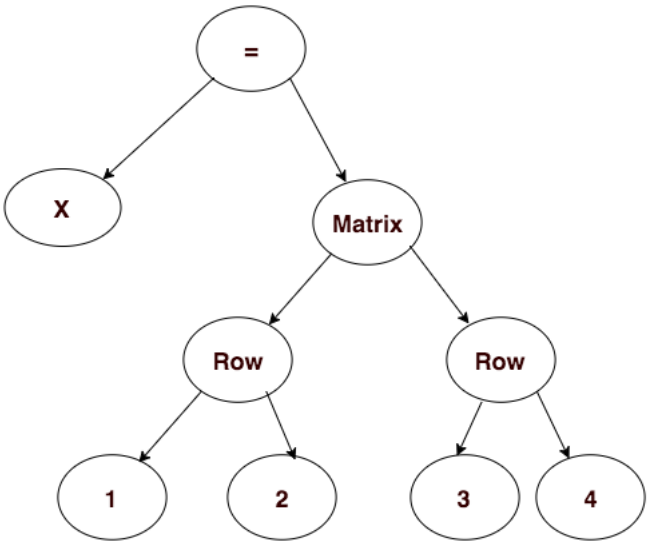

Figure 3.2: Small section of Matlab AST to represent a matrix

The generated code shown in the listing below contains a Matrix data structure whose details can be found in Section 4.5. It can also be noticed that there is an enumerated constant **endr** to represent the end of a row.

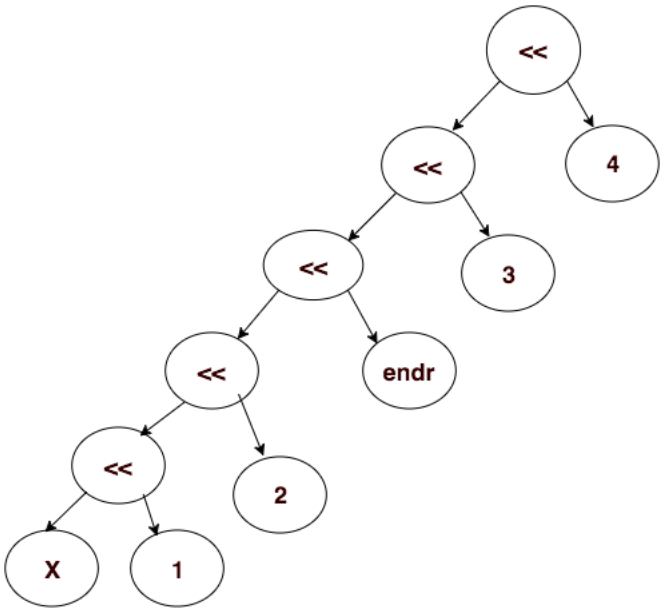

Figure 3.3: Transformed C++ AST that represents the matrix

```
1 Matrix<int> X;
2 X << 1 << 2 << endr
3          << 3 << 4;
```

Figure 3.3 shows the transformed C++ AST. Each transformation component in FastNumerics does one unit of transformation. We have implemented a transformation framework where each module represents a specific set of transformations to be performed on the AST. For example, a ForLoopTransformer transforms a for loop from Matlab to C++ and a RangeExpressionTransformer transforms a Matlab range to a matrix representation in C++. More transformations can be easily plugged to the system using the framework.

AST transformation depends upon the interface of the Matrix wrapper in C++. For example, FastNumerics' Matrix wrapper expects a matrix to be initialized using left shift operators, so the Matlab matrix node (SgMatrixExp node) is converted to C++ AST by appending left shift operators on each elements of the Matrix.

Similarly, a range expression in Matlab can be specified as follows:

```
x = 1:5:100
```

In order to transform this to valid C++, we generate a code something like this:

```
Range<int> range0;
range0.setBounds(1,5,100);
x = range0.getMatrix();
```

Each transformation has certain rules specified to modify the AST. It works by traversing the AST and then performs structural transformation on the nodes that match the rules. Transformations make extensive use of ROSE SageInterface[31] and SageBuilder APIs to remove, lookup, create and insert AST nodes.

3.3 Backend

Backend is responsible for generating source code from the AST. Generating code from the AST is also referred to as unparsing. FastNumerics has two backends which are as follows:

3.3.1 C++ Backend

C++ backend is the existing backend in ROSE compiler framework that takes an AST which is valid C++ and produces C++ source code. It takes the output from the midend which is a valid C++ AST and then produces C++ code. The generated C++ code depends on a Matrix library wrapper. The Matrix library wrapper can be implemented using any linear algebra library. The Matrix wrapper can be used to implement an interface to highly optimized libraries like Eigen, Armadillo, etc.

3.3.2 Matlab Backend

Matlab backend accepts a Matlab AST (that has not been transformed to C++ AST) and it produces back the same Matlab code from which it was generated. This backend is used to verify if the AST has been constructed properly or not by comparing the input Matlab code and the unparsed Matlab code. If the files match, the AST has been constructed properly.

The Matlab backend is implemented by doing a bottom-up traversal on the AST. Each node visited will pass its string representation up to its parent node. The parent node assembles the strings from its child to form a valid Matlab code and then passes it up to its parent. This process continues until the root node is reached which writes all the strings collected from the traversal to a file in an ordered way so that the output is a valid Matlab program.

CHAPTER 4

FASTNUMERICS SUPPORT

The translated C++ code is not self contained. It needs implementation of the Matrix data type, the implementation of built-in functions and implementation of different symbols. FastNumerics Support is the component responsible for providing these implementations. This chapter will describe the design of FastNumerics Support and its components.

4.1 Matrix Wrapper

FastNumerics generates C++ code which is independent of the library that implements a Matrix or linear algebra functions. That is FastNumerics targets a Matrix wrapper. The design of how the translated C++ code interacts with the wrapper and its supporting components are shown in figure 4.1.

4.2 Advantages of Generating Code for Wrapper

We want to preserve the information in Matlab code as much as possible and also want to make the translated C++ code look similar to the input Matlab code. It would be much better if a matrix multiplication operation represented in Matlab as A * B would also be translated to A * B in C++. This is where the role of Matrix wrapper comes in place. The Matrix wrapper would overload operators so as to make the translated code similar to Matlab.

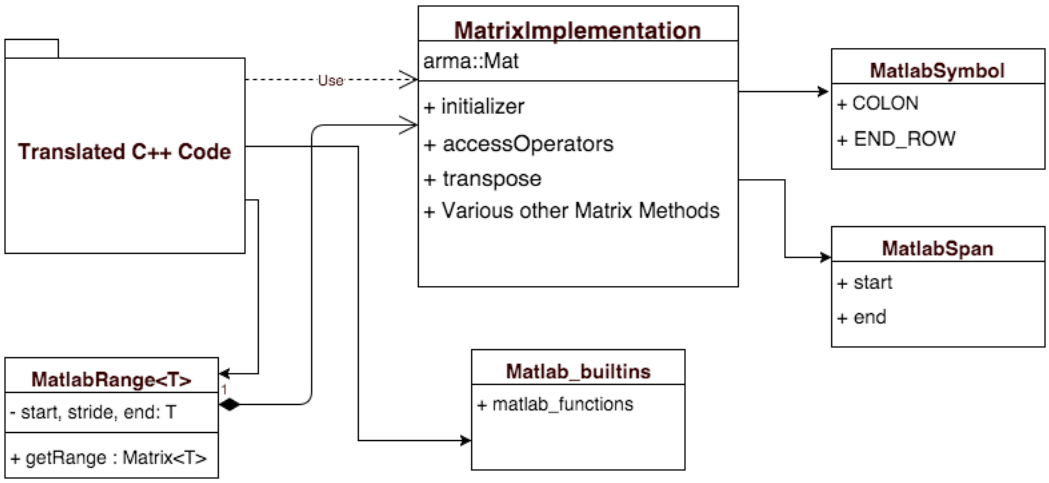

Figure 4.1: How translated code interacts with the Matrix Wrapper

In general we can say that the translated C++ code will encapsulate Matlab specific constructs in a class. For example, a Matlab range is represented by a Range class, any Matlab symbols like end of row, colon, etc. will be represented by an enumeration MatlabSymbol.

4.3 Translated C++ Code

The translated C++ code component in 4.1 represents the C++ code generated by FastNumerics. Listing 4.1 shows a range expression 1:2:100 transformed to a C++ code. The generated C++ code depends upon the class Range<T>. When compiling the generated code, the compiler needs a definition of Range<T>. Similarly, the generated code may produce code that uses MatlabSymbol and MatlabSpan.

```
1 Range<int> range0;
2 range0.setBounds(1,2,100);
3 x = range0.getMatrix();
```

Listing 4.1: Generated C++ code to represent a range

4.4 Armadillo

Armadillo[32] is a high quality C++ linear algebra library. Its API resembles Matlab function calls and is easy to use. Armadillo encapsulates complex function calls to underlying numerical libraries by providing an elegant API that looks like Matlab function calls. It is possible to dynamically link Armadillo with different libraries like LAPACK [33], Intel MKL [34], OpenBLAS [35], CudaBLAS [36], etc. Another feature of Armadillo is that it supports C++11 constructs and allows Matrices to be initialized using initializer lists. This feature has been used in our transformations (discussed in 6) to make writing transformation easier.

FastNumerics transforms the matrix representation and operations into C++ code that targets an abstract Matrix wrapper. The Matrix<T> class has a number of overloaded operations to represent operations like element access, multiplication, division, sub-matrix access, etc. Concrete wrappers can be written which can be implemented using linear algebra libraries like Eigen [37], BLAS, Intel MKL, etc. For our testing purposes, we have a concrete implementation of the wrapper based upon Armadillo.

When FastNumerics transforms the Matlab code,

```
1 A = [1 2 3]
2 B = [2; 3; 4]
3 C = A * B
```

it gets transformed into identical looking C++ code as follows:

```
1 Matrix<int> A, B, C;
2 A = 1 << 2 << 3;
3 B = 2 << MatlabSymbol::endr <<
4         3 << MatlabSymbol::endr << 4;
5 C = A * B
```

Here the matrices A, B and C are of type Matrix<T>. The matrix declarations are inserted at the top of the function in which the initialization is done. The underlying implementation of Matrix<T> can be any numerical library that allows a programmer to represent a matrix and follows the interface that FastNumerics expects it to follow.

4.5 MatrixImplementation

The generated C++ code represents a matrix as an object of Matrix<T> interface. However it does not generate the definition of Matrix. The MatrixImplementation component serves to define the Matrix interface. Currently for our experimental purpose, we have implemented Matrix which uses Armadillo library. The implementation of Matrix provides the following:

- All the operators on matrices like *, /, .*, .+ etc
- Element access functions
- Iterator for elements of a matrix

MatrixImplementation does not provide support for all the functions in Matlab. It does provide support for operators that can be overloaded in Matrix, but functions like sin, rand, etc. will be implemented in a different component called Matlab built-ins.

4.6 MatlabSymbol

The Matlab language uses some symbols which may not have an equivalent representation in C++. For example, one of the uses of colon symbol in Matlab is to access elements in a Matrix. When accessing elements in a Matrix (as shown in Listing 4.2), the presence of : in either row or column index is to instruct Matlab to select all the rows or all the columns. To represent this in C++, we use an enum COLON of enum class MatlabSymbol. So the Matlab code in Listing 4.2 gets converted to the equivalent C++ code in Listing 4.3.

```
1 A = [1 2 3; 4 5 6]
2 A(:, 2)   % Access the second column of A
3 A(2, :)   % Access the second row of A
```

Listing 4.2: Matlab matrix element access

```
1 Matrix<int> A;
2 //A << initialize A
3 A(MatlabSymbol::COLON, 2)   % Access the second column of A
4 A(2, MatlabSymbol::COLON)   % Access the second row of A
```

Listing 4.3: Equivalent C++ matrix element access

We also use another enum endr to represent the end of the row while initializing a matrix.

4.7 Range

In Matlab, range expressions are used in a lots of places like loops, matrix accesses, creating vectors, etc. It is represented as a two or three element expression start:stride:end where stride is optional. They do not have an equivalent representation

in C++. So FastNumerics creates a Range object whenever it sees a range expression (except in For loops). Section 6.4 discusses how transformations are done on Range expressions in detail.

4.8 Matlab Builtins

Matlab builtins contains the implementation of Matlab built-in functions. The built-in functions may be implemented using any API. In FastNumerics, we rely heavily on Armadillo to implement linear algebra operations and on C++ standard libraries like math and iostream for other implementations.

Currently, we only provide limited support for these built-ins. It is easy to extend or modify this component to support more Matlab built-in functions. All the built-ins are defined in a namespace called **fastnumbultins**.

CHAPTER 5

TYPE INFERENCE

Matlab programs do not need types to be specified whereas C++ programs need to have types annotated for every variables. So the translation process should include a type inference phase that computes the types of variables. Type inference is an interesting problem to solve and there are many literature on type inference [38] [39]. Two different methods for type inference were studied in this thesis to compare their applicability in a dynamic language like Matlab. The first method is a self developed flow-based [40] inference algorithm and the second method is a modified version of Hindley-Milner [41] type inference. In our framework, we used type inference by propagation (flow-based approach) due to the reasons discussed in section 5.4. FALCON Matlab compiler [9] uses an advanced type inference algorithm.

5.1 Source of Type Information

Type inference needs to have some source of information from which it has to start determining the types. Those source of information are constants, operators and built-in functions in the program. Constants like integers and matrices by default have a type. Operations on two terms (in a binary expression) that have known types will yield a typed result value. Similarly, built-in functions will have a known return type depending upon the type of input. For example, transpose function (also can be represented in Matlab by ' operator) is a built-in function that acts like an identity function when looked from a type perspective. It means if it accepts a term with type

T, it will always return a term with type T. Whereas, det is a function (that calculates determinant of a matrix) which always returns a term of type double. Constants and operators are the information which we can find by querying the structure of the AST. For built-in functions, we need to have some database that specifies the types returned by those functions.

5.2 Type Inference by Propagation

Type inference by propagation method works by getting the type of a terminal value(eg. integer, matrix) from the bottom of the AST and propagates it up the AST. Whenever the algorithm encounters an assignment node, it checks if the LHS expression already has already been assigned a type. If the new type is larger than the existing type, the new type is assigned to the LHS. The LHS node then propagates the type down to its child nodes till the leaf nodes are reached. Whenever a variable gets assigned a type, the symbol table for the current scope is updated.

Algorithm 1 Type Inference by Propagation

```
1: procedure INFERTYPES(function) ▷ A function whose types should be inferred
2:     Collect all the subtrees that represent Assignment expressions
3:     for all subtrees do
4:         expressionType ← BottomUpTraversal(RHS)
5:         lhsType ← get_dominant_type(typeof(lhsType), expressionType)
6:     functionReturnType ← typeOf(returnVariable)
7:     return functionReturnType
```

Algorithm 5.2 shows an algorithm for type inference by propagation and Figure 5.1 shows x being assigned a type integer. In this figure, 5 passes its type i.e. integer to its parent node =. The equals node then transfers this type to its children on the left branch. x then gets type integer. The equals node then sends integer type up to its parent. This can also be addressed as a data-flow problem [42] where at each assignment node we compute the types of a variable and determine which types flow in and which types flow out after the assignment operation [43].

Algorithm 2 Bottom Up traversal on RHS node

```
1: procedure BottomUpTraversal(subtree)
2:     nodeType ← typeof(subtree)          ▷ subtree is the RHS of the assignment expression
3:     if nodeType is ValueExpression then
4:         return ValueExpresion− >innerType
5:     if nodeTypeisFunctionCall then
6:         Get argument types of function call
7:         Update the called function's symbol table with the argument types
8:         return inferTypes(calledFunction)
9:     if nodeTypeisExpressionList then
10:        Get the dominantType from the expression list
11:        return dominantType
```

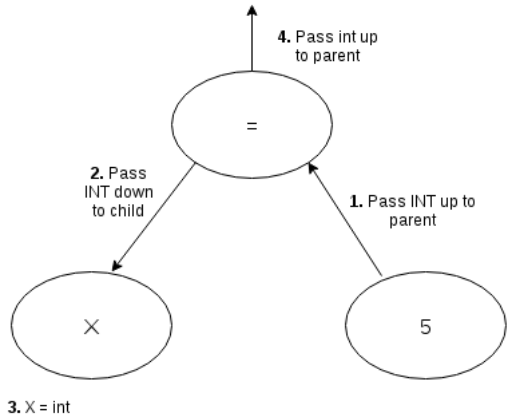

Figure 5.1: Type inference by propagation of terminal types

5.2.1 Dominating Types

Whenever a node in the AST has n children ($n >= 2$), then it will get types $T1, T2, ...Tn$ from all of its children. It has to decide which type Ti to choose to propagate up/down the tree. In Matlab, a matrix row is a list of numbers which could be of multiple types. Also a Matrix is a list of rows which could be of multiple types. So these nodes should pick a dominating type from the types propagated upwards by its children. In our thesis, dominating type represents a type $DT \in Ti$ such that $DT >= Ti$. For example, DominatingType(Matrix, Scalar)→ Matrix because Matrix dominates the binary operation and will be known as the **dominating**

type. Also DominatingType(double + int) → double where double will be known as the dominating type. Figure 5.2 describes the process. In order to calculate the dominating type from a list of types, we rank each type. Matrix has the highest rank whereas Integer has the lowest. So dominating type is the type in the list that has the maximum rank.

This approach can also be described using a lattice where the Matrix<double> type is at the top of the lattice [44] followed by other Matrix types, then the double type and integer type. In the work of Joisha and Banarjee, the algorithm starts with all the variables having the type at the top of the lattice and the types get more constrained at each assignment nodes. Whereas in our approach we start from the bottom of the lattice and the types get more general as we go through each assignment nodes.

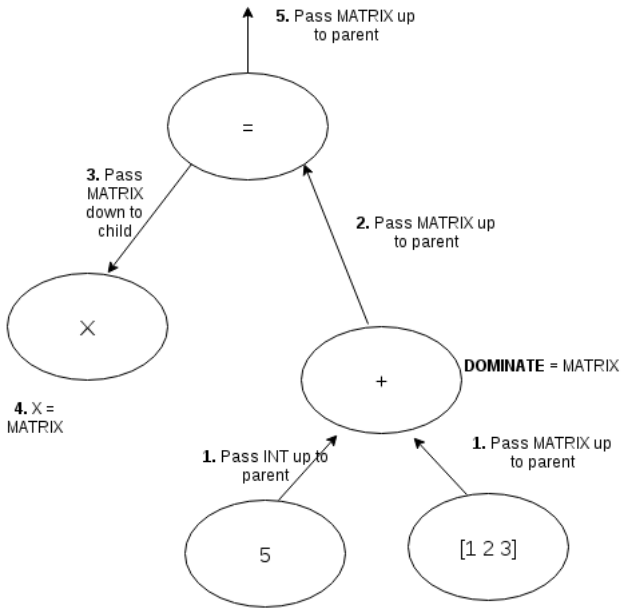

Figure 5.2: Propagating the dominating type

5.2.2 Variable Assigned Values of Multiple Types

In Matlab, a variable can be assigned values of multiple types. Our type inference algorithm takes in account the assignments whose values are compatible with each other. In the code below, x is being assigned an integer first, this will update the type of x to be an integer. After that, it gets assigned a double type. Since double can represent an integer as well, so in the given scope, x will be of type double. However, if at first x was a double and then it was assigned an integer, the type of x would still be a double as an integer cannot dominate a double type. Currently FastNumerics does not handle cases where the values assigned are non-numeric types.

```
1 x = 1; // x is an int
2 x = 2.5; // x is a double
```

```
1 x = 2.5; // x is a double
2 x = 1; // x still a double
```

5.2.3 Types from Function Calls

Whenever the algorithm sees a function call, it recursively performs type inference on that function. The algorithm then propagates the type returned by the function for that function call.

For example:

```
1  function foo()
2  x = 5
3  y = 6
4  sum = add(x, y)
5
6  s2= add(1.5, 2.4)
7  s3 = add([1 2 3], [4 5 6])
8  end
9
10 function s = add(n1, n2)
11          s = n1 + n2
12 end
```

Listing 5.1: Function Call to add

When the algorithm reaches a function-call node **add**, it figures out that the types of input variables x and y should already have been calculated. Then it updates the symbol table of function **add** by making x an integer and y an integer. Now it symbolically executes the function **add** and calculates all the types that can be inferred due to x and y being integers. After it finds out the type of the return variable, it returns that type. In this case, the algorithm sets n1 = typeof(x) = int, n2 = typeof(y) = int. Then we can find out that s = dominatingtype(n1, n2) = int. Since s is also the return variable, int is returned and sum gets the type int.

Similarly, when the algorithm sees another function call to add with matrix as arguments, it again symbolically executes the function add with n1 and n2 set to Matrix type. Then it gets Matrix as the return type which is assigned to s3.

5.2.4 Built-in Functions versus User Defined Functions

When the type inference algorithm sees a function call, it should decide whether the function call is a user defined function or a built-in Matlab function. First, the

algorithm checks if the function resides in a table that contains all the function declarations of the user file. If the function is not found there, then it should be a built-in function. If the function is not found in the built-in list, it is an error.

5.2.5 Type Inference From Built-in Functions

It is difficult if not impossible to symbolically execute a built-in Matlab function found in Matlab installation. To make it easy, we built a database of the built-in functions. The database can be used to find the return type of a Matlab built-in function. The way we did it is quite interesting. Instead of having a map from the name of function and its input type to the return type, we created a separate Matlab file that contains a very small implementation of each built-in function. This is an extendable file, so for now we have just added a few built-in functions to it.

In Listing 5.2, we have added a dummy implementation of the built-in function **rand** which takes two integer dimensions and returns a random matrix of doubles (Matrix<double>) having dimension m x n. Now when our type inference algorithm symbolically executes this function, it finds out that this function returns a Matrix of doubles. Similarly, the type returned by transpose function is the type of the input Matrix. Hence in our implementation of transpose, we just return the input matrix m. This way our type inference algorithm when called with t = transpose([1 2 3]) will return the type of [1 2 3] which is Matrix<int>. Hence t gets its type assigned as Matrix<int>.

```
1 function z = rand(x, y)
2   z = [1.2 3.4];
3 end
4
5 function t = transpose(m)
6  t = m
7 end
```

Listing 5.2: Builtin function file

The limitation with this approach is that a built-in function can only return one type for one input. For example, a sin function can take a double or an int and return a double. The sin function can also take a $Matrix < double >$ and return a Matrix<double>. For double and Matrix, the sin function could just return the type of input argument. But in case of integer, it cannot be instructed to return a double. So for now we have built the sin function to always return a Matrix<double> as it is the safe approximation for the type returned by sin.

This limitation could have been solved by defining those dummy built-in functions in C++ instead of defining them as a Matlab file. In C++ we could take advantage of the facilities provided by templates. We could have a template function definition for sin and also a specialized sin function that accepts int and returns a double. But for our experimental purposes, this is a fairly good approach. Also it is difficult in ROSE compiler to instantiate a template function declaration with a specific type, so we chose the simple approach instead.

We could also extend our type inference algorithm based on the fact that some built-in methods only accept specific argument types. For example, a transpose function always accepts a Matrix type, so any variable passed to the function transpose will have a type Matrix. However, this feature has not been implemented for now, as we assume that all the variables should be initialized before they are used.

5.2.6 Creating Overloaded Functions

Matlab can have one definition of a function that can accept input arguments of any types. For example in Listing 5.1, we can call sum with arguments x and y of types integer, double, matrix. We could even have called sum as sum(1.2, [1 2 3]). This is possible because Matlab checks the types of variables dynamically when an operation is applied on them. However in C++, we need to specify types for the input parameters in a function declaration/definition. Having multiple function calls to sum with different input types implies calling overloaded version of the function sum. Therefore, FastNumerics creates an overloaded copy of a function definition when it sees a function call.

For example, in Listing 5.1, we will create three overloaded versions of add. The first one will accept (int, int), the second one will accept (double, double) and the third one will accept (Matrix<int>, Matrix<int>).

Instead of creating overloaded versions by copying functions, we could have created a template version of the original function. Although this is not difficult to implement, it is difficult in ROSE compiler to instantiate a template function declaration with a type T = concreteType (whenever we see a function call with some concrete type). Therefore, we chose to create overloaded functions instead of having a single template.

5.2.7 Type Attributes

Attributes are objects that can be attached to an AST node in ROSE. Whenever a variable gets assigned to a type or a constant returns a type, FastNumerics attaches a custom attribute to the node called TypeAttribute which stores the type information that the node represents. Although this information is also stored in the symbol table of the scope in which the variable resides, some implementation issues (especially for some new Matlab nodes added) have prevented the algorithm to store exact type information in some specific cases. Therefore as a workaround, we attach this attribute

to each variable which can later be retrieved by other components of FastNumerics.

TypeAttributes also came out to be helpful while implementing the algorithm. It could be used to graphically display the type assigned to a node. Figure 5.3 displays the type attribute information associated with each node.

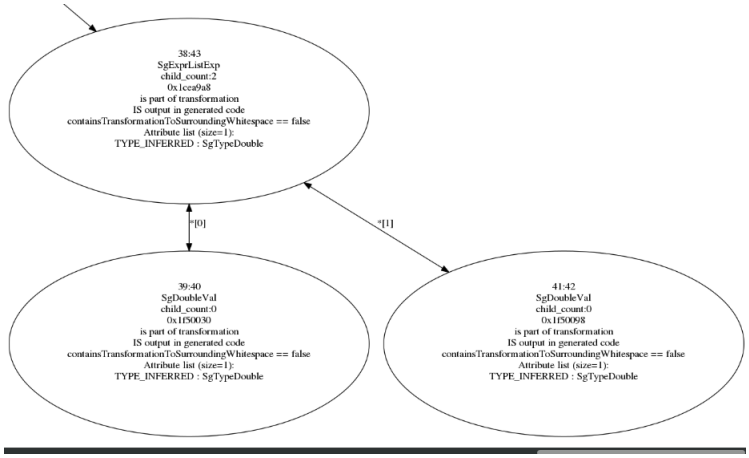

Figure 5.3: Type Attributes in the AST

5.2.8 Type Soundness

The algorithm we use produces sound types. This is because if it finds some node(eg. a matrix access) to return a type that it cannot guess perfectly, then it can fallback to a type Matrix<double> which is valid for any numeric type. For our work, we are only considering values to have numeric types. Hence the program cannot produce types which will cause run-time errors.

The algorithm we developed still has some limitations, especially in cases when a variable is defined inside a control-flow based on a condition. These issues can be resolved in the future by applying a proper data-flow analysis.

5.2.9 Integer Usage

By default, Matlab assumes that every variable has a base type double. A declaration x = 5 will set x to be a floating point number internally. In Matlab, a

programmer should explicitly state x = int16(5) to make x an integer. FastNumerics automatically checks if a number is an integer and creates an Integer AST node to store its value. The type inference algorithm then declares those variables to have an int type which will be computationally efficient as compared to using doubles everywhere.

5.2.10 Inserting Variable Declarations

For all the variables in a scope, we insert variable declarations with their inferred types. This is done after the end of type inference analysis. FastNumerics goes through all the function declarations and extracts the set of variables in the scope. Since each variable will have a TypeAttribute attached to it, this attribute is queried to get the type of the variable. A variable declaration is created using ROSE's api and prepended to the scope.

5.3 Type Inference By Solving Constraints

We use Hindley-Milner system to infer types by solving constraints and the algorithm has been described in a very clear way in [45]. Type Inference by solving constraints involves three phases which are as follows:

5.3.1 Generating Constraints

This phase traverses the Abstract Syntax Tree and from each relational operator or a function definition generates a constraint. For example, from an assignment

```
1 x = [1 2 3 4; 4 5 6 7]
```

, the generated constraint is

```
1 x = Matrix
```

Similarly, the constraints generated from the following function definition

```
function s = add(n1, n2)
s = n1 + n2
end
```

will be

```
add = (n1, n2) -> s
s = Operation(n1, n2)
```

5.3.2 Solving Constraints

We can then solve these constraints by using Unification algorithm.

Here is how a basic unification algorithm works. Suppose we have the following constraints:

```
x = int
y = x
z = x + y
```

Then we can infer that

```
x = int
y = int and
x = int + int = int
```

5.4 Comparing Flow-based vs Flow Insensitive Approach

Matlab language has many overloaded operators and functions that make type inference difficult. A function may have different types based on the type of the

input and the control flow. Flow-based and flow-independent solutions have their own advantages and disadvantages. For a dynamic language like Matlab, flow-based approach to type inference seems to be the most appropriate. For example, the operator A * B may return different types depending upon the type and shape of A and B. If both A and B are Matrices, the type returned is a Matrix. If A is a row vector and B is a column vector, the type returned will be either an integer or a double. if A is a column vector and B is a row vector, the type returned will be a Matrix.

With flow insensitive approach, we may not immediately know the types of A and B when applying the operator *. Since the constraints generated are solved at once, it is not guaranteed that A and B have been assigned a value when the algorithm is at C = A * B. So the flow insensitive approach cannot make decisions when it is solving a set of constraints within a scope. Only after the constraints are solved, we can know the types of the variables in the scope. This will particularly hinder type inference in Matlab when we need to know the types of A and B at some node. In a flow-based approach, when the algorithm reaches C = A * B, it can query the type of A and the type of B and make decisions accordingly. Also implementing a flow-based algorithm will provide more control over the way types can be returned by each expressions.

In Matlab, accessing elements of a Matrix look exactly like function calls. A(1, 2) could mean call a function A or access the value of matrix A at the specified position. In a flow insensitive approach, the algorithm is not guaranteed to know that A is matrix when it reaches A(1,2). But in a flow-based approach, it is already known that A will be a matrix because elements are always initialized before they are used.

But it does not mean that a flow-based approach does not have its limitations. A flow insensitive algorithm can properly infer the type of a variable x at statement, x = y, defined at the top of the loop whereas y has been initialized at the bottom of the loop as shown in Listing 5.3. Since the flow insensitive approach does not depend where the variable was initialized; it will set x to be the type of y which is integer.

But the flow-based approach will not have seen y = 5 when it reaches x = y, so cannot assign a valid type to x. This problem can be solved by repeatedly analyzing the function until all the types have been found. Flow insensitive approach really make it easy to determine the function type of a recursive function as well.

```
1 for loop
2 if (condition) x = y
3 y = 5
4 end
```

Listing 5.3: Limitation of flow-based approach

5.5 Challenges With Type Inference in Matlab

5.5.1 Same Variable assigned to Values of Incompatible Types

FastNumerics for now assumes that a single variable can be assigned values whose types are compatible with each other. For example (as shown in the listing below), a variable x can be assigned to an integer as well as a Matrix. However, FastNumerics does not deal with a variable x that is assigned to a string and a number.

```
1 x = 1; %OK
2 x = [1 2];  %OK
3 x = "Hello"; %Not OK. Incompatible assignment
```

Different return types from a function. In Matlab, a function can have different return types depending upon a condition. Right now FastNumerics will always generate a function that has a return type which dominates all the types in a return statement. For example (as shown in the listing below), if a function has two return statements where the first returns a Matrix and the second returns an integer; the function will have a signature whose return type is a Matrix. We cannot have a function which will

have two different return types based on a condition. So foo's signature will always state that it will return a Matrix.

```
function y = foo(x)
if ..
   y = [1 2 3];
else
   y = 12;
end
```

A solution in which a variable y can be either a string or an integer can be represented using a type called Boost.Variant [46] provided by the Boost[47] library. However we could not implement this feature due to time constraints.

CHAPTER 6

TRANSFORMATIONS

FastNumerics transformations are a set of rules applied to modify the Matlab AST so that we get a valid C++ AST. ROSE provides a very good API to query and modify the AST using SageBuilder and SageInterface. A short introduction to transformations has already been provided at Section 3.2.1.0.2. In this section we will discuss the approaches we applied to some important transformations which will also point out some peculiarities of Matlab language.

6.1 Generating C++11 Code

FastNumerics' transformation rewrites the AST so that we can get a C++11 code. The reason we do this is to utilize some of the new features provided by C++11. These features help simplify writing transformations. For example it would have been difficult to convert a Matlab function that returns multiple variables to C++98 code. However, C++11 provides constructs std::tuple and std::tie to allow a function to return a list of variables easily. Here are a few C++11 constructs that we use in our transformed C++ code:

6.1.1 Auto

C++11 introduces auto so that we can declare a variable to be of auto type if it is initialized during declaration instead of explicitly stating its type. FastNumerics declares the type of the for loop iterator to be of auto type.

6.1.2 std::type, std::tuple And std::make_tuple

FastNumerics generates these C++11 constructs when dealing with functions that return multiple variables. If these constructs were not available then we may need to generate code to return a vector and then explicitly copy the values from the vector to the assigned variables. Therefore generating C++11 code makes transformation easier.

6.1.3 Initializer Lists

C++11 allows a function to accept initializer lists. We use this feature while passing vectors to a function by representing the vector as an initializer list.

6.2 Inserting Include Statement

The generated C++ code depends upon a lots of external components like the Matrix implementation, the built-ins implementation etc. Instead of adding many include statements, to include those dependencies, FastNumerics adds an include statement to include a file called fastnumerics.h which includes all the dependencies needed by the translated code.

6.3 Transformation: Matrix Assigned To a Variable

Whenever FastNumerics sees a variable being initialized with a Matrix, as shown in the left side on Figure 6.1, it transforms the matrix to a form as shown in right on the same figure. This is because the Matrix interface has an overloaded operator << that helps initialize the matrix. The overloaded operator can be implemented in any way as long as it initializes the matrix object with the supplied values.

The transformation works by going through all the assignment operators and checking if the RHS of the assignment is a Matrix or not. The matrix expression (represented in ROSE by the class SgMatrixExp) is then replaced by a new expression

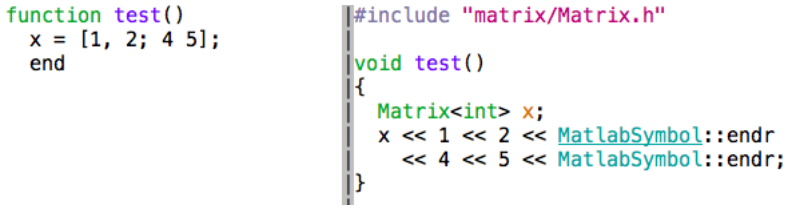

Figure 6.1: Matrix initialization transformation

composed of left-shift operators and the elements of matrix.

6.4 Transformation: Range Expression

Ranges in Matlab have the form start:stride:end where stride is optional and defaults to 1. Range expressions produce a vector of elements from start:end. FastNumerics transforms these range expressions into a range.getMatrix() function call where range is an object of class Range< T >. The range object is initialized with a call to its setBounds function that has a signature setBounds(start, stride, end).

The listing 6.2 shows the transformed range expression shown in Listing 6.1. Here the range variable x will have a type Matrix<T> where T is the type of the elements in the range. If there are multiple ranges in a scope, each range variable will have a unique name. The call getMatrix() will return a Matrix. In our implementation of Range class, we return an Armadillo matrix when getMatrix() is called.

```
1 x = 1:2:100;
```

Listing 6.1: Matlab Range

```
1 Range<int> range0;
2 range0.setBounds(1,2,100);
3
4 Matrix<int> x;
5 x = range0.getMatrix();
```

Listing 6.2: Range transformation

Range expressions that represent the loop range of a for loop are skipped when doing this transformation. This is because a for loop will use the Range information to create a loop whose index in each iteration represents one of the values of the range.

6.5 Transformation: For Loop

A Matlab for loop can be represented in two different ways and they are show in Listing 6.3. Although the two representations can be unified as one since start:stride:end is also an expression, but start:stride:end can be treated in a different way to generate better code. We know that start:stride:end is looping from start to end with each loop jumping by stride. In case stride in not provided, the default stride is 1. So this expression can be converted to a normal for loop.

In the case where the RHS of the index in a for loop is an expression which is not a Range, it should be iterated using iterators. FastNumerics expects that expression has begin() and end() implemented and they return iterators. expression could be a variable that has been assigned to a range expression, or a function call that returns a range expression or a matrix.

```
for i = start:stride:end
%statements
end

for j = expression
%statements
end
```

Listing 6.3: For loops in Matlab

For the first for loop where RHS of index has an explicit range expression, we generate an equivalent C++ for loop as shown in Listing 6.4. The index i has been assigned an "auto" type.

```
for (auto i = start; i <= end; i += stride) {
  //statements
}
```

Listing 6.4: Normal C++ for loop

The second loop with an expression on RHS (which is not a Range expression) of the loop index gets transformed to a for loop that uses iterators as shown in Listing 6.5.

```
for(auto j = expression.begin(); j != expression.end(); ++j)
{
//statements
}
```

Listing 6.5: For loop that uses iterators in C++

6.6 Matrix Argument In Function-Call

In Matlab we can pass a matrix constant as function argument. Listing 6.6 shows an example of how a matrix constant can be passed to a function. In the listing, a vector [1, 2, 3] is passed to the function foo. Similarly, we can see that [1 2;] is passed to the matrix A. Passing a matrix to access the elements of A is done to access a sub-matrix of A. In the listing, c will be assigned a sub-matrix of A (Rows 1, 2 and 2^{nd} column).

Since C++ does not support passing terms in the form [1 2;], FastNumerics transforms those expressions to an initializer list expression. Initializer list is a list of terms enclosed in { } braces separated by commas. C++11 supports functions that accept initializer lists, so the function foo can accept initializer lists. Also a Matrix should have an access function that takes an initializer list. Initializer lists are represented in ROSE by SgAggregateInitializer.

```
1  y = foo([1 2 3]);
2
3  A = [7 8 9; 4 5 6;]
4  c = A([1 2;], 2);
```

Listing 6.6: Passing matrix as argument in a function call

Listing 6.7 shows how Listing 6.6 is transformed to C++.

```
1  y = foo({1,2,3});
2
3  A = ....
4  c = A({1,2}, 2);
```

Listing 6.7: Matrix argument transformed to C++

The transformation works by going through each arguments of all the function call expressions. If it finds that the argument is a Matrix, it replaces that Matrix by an equivalent initializer list.

Here is how FastNumerics' Matrix implementation accepts initializer lists to access elements:

```
1  //element access A([1 2 3])
2  Matrix<T> operator()(initializer_list<uword> indicesList)
3  {
4     ............
5  }
```

6.7 Functions That Return Multiple Values

Matlab functions can return multiple values. It is a very nice feature in Matlab so that we do not have to pass references or return a vector if we want to return multiple values. Here is a very simple Matlab code that calls a function which returns two values a and b:

```
1  function main()
2    [x , y] = foo();
3  end
4
5  function [a, b] = foo()
6    x = 5;
7    y = 6;
8    a = x + y;
9    b = x - y;
10 end
```

Function foo() can return two values which can be assigned to x and y respectively.

In C++11 we could use std::tuple to return a tuple of values and std::tie to bind those return values to the variables. FastNumerics transforms the above Matlab code to the following C++ code by utilizing tie and tuple constructs:

```
1  std::tuple<int,int> foo()
2  {
3    //var declarations
4    x = 5;
5    y = 6;
6    a = x + y;
7    b = x - y;
8    return std::make_tuple(a,b);
9  }
10
11 void main()
12 {
13   int y;
14   int x;
15   std::tie(x,y) = foo();
16 }
```

The function foo returns an object of type std::tuple and the function main ties the returned object to individual variables. FastNumerics is able to do this on the assumption that the generated code will be compiled as a C++11 code.

6.7.1 Representing Multiple Return Values

FastNumerics stores a list of return variables in a function declaration object associated with the function. Whenever it finds out that there are more than one return variables in the list, it creates a **std::make_tuple** function call as an expression in the return statement as shown in the listing above. For a single return variable in

the function, a normal return var; statement is generated.

6.7.2 TypeTuple

FastNumerics stores the type returned by a function returning multiple variables type as a TypeTuple (in ROSE this is represented as SgTypeTuple). This is a new Type that FastNumerics added to ROSE compiler. A TypeTuple can store a list of types. In FastNumerics, foo has a return type of SgTypeTuple<int, int> which gets unparsed to a std::tuple representation as seen in the listing above.

6.8 Colon Operator in Matrix Access Operations

In Matlab we can use a colon operator (:) to select either all the rows or all the columns when accessing the elements of a matrix. For example, if A is a matrix then we can call A(:, 5) to access all the rows of the 5^{th} column. Similarly, the colon operator can be used to select all the columns for a specified row. Since C++ does not allow passing such symbol/operator to a function, FastNumerics replaces occurrences of the colon operator by an enumerator MatlabSymbol::COLON in a function call.

```
1 A = [1 2 3; 4 5 6;]
2 y = A(:, 2);
```

The Matlab code in the above listing gets transformed to

```
1 A = ....
2 y = A(MatlabSymbol::COLON, 2);
```

This transformation expects the Matrix wrapper to have an implementation of the overloaded operator () that accepts a MatlabSymbol enumerator. Our implementation is shown in the listing below:

```
1   //A(:, 2)
2   Matrix<T> operator() (MatlabSymbol symbol, int col)
3   {
4     assert(symbol == MatlabSymbol::COLON);
5
6     Matrix<T> column(matrix.col(col - 1));
7
8     return column;
9   }
10
11  //A(2, :)
12  Matrix<T> operator() (int row, MatlabSymbol symbol)
13  {
14    assert(symbol == MatlabSymbol::COLON);
15
16    Matrix<T> rowMatrix(matrix.row(row - 1));
17
18    return rowMatrix;
19  }
```

6.9 Calls to Matlab Built-in Functions

Matlab has a rich set of built-in functions which is one of the reasons for its popularity. Some of the built-in functions in Matlab are sin, randn, disp, etc. Whenever FastNumerics transformation sees a call to a function, it does not transform it. So all the calls to built-ins will remain untranslated. The translated code requires those built-in functions to be implemented for it to compile.

For example the Matlab code below contains a function call to sin which gets translated to C++ exactly as is. In FastNumerics sin is implemented in namespace fastnumbuiltins using standard C++ math library.

```
x = sin(y);
```

The above code gets translated to

```
double x;
x = sin(y);
```

6.10 Transformation: Binary Expressions to Built-in Functions

Some binary expressions in Matlab can be implemented using an operator overloaded function in Matrix implementation. For example a + operation between two matrices A and B does not need to be transformed at all because there is an overloaded operator + in the Matrix implementation. However not all operators in Matlab have an equivalent operator in C++.

Operator \which is a binary operator known as left divide operator solves the system of linear equations $Ax = b$ when executed as x = A \b. C++ does not have a \operator, hence we need to transform those operators to some function calls with the operands as the arguments.

Similarly operator ^represents a power operator in Matlab and C++ does not have a power operator. Hence this also needs to be translated to some function call like power.

All of these functions have their implementation in Matlab built-ins component. Since all of these transformations are similar, FastNumerics has an implementation of an abstract module that transforms a binary operator to a function call. This is an easily extendable module.

6.10.1 \ Operator to solveLinear Call

All the occurrences of A \ b are transformed to a function call solveLinear(A, b).

6.10.2 ^ Operator to Power Call

All the occurrences of x ^ y are transformed to a function call power(x, y).

6.11 Matrix Operations to Matrix Method Calls

There are some operations on a Matrix that could be more natural to translate to a member function call on the matrix instead of a standalone function. For example, A' in Matlab represents the transpose operator ' applied to a matrix A. Again since C++ does not have such operator, so FastNumerics translates that function call to A.t() member function call.

CHAPTER 7

EXPERIMENTS

We compared the running times of Matlab code, its corresponding hand translated C++ code and automatically translated C++ code. We took six Matlab benchmarks to perform our experiments.

7.1 Benchmarks

The benchmarks that we took are as follows:

7.1.1 Belief Propagation

The main operations involved in this benchmark are matrix vector multiplication and sum of vector inside a loop. For our experiment we have a 500 x 500 dense matrix, a 500 x 1 vector and number of loop 100000. The time is averaged over 50 runs.

7.1.2 Determinant

In this benchmark, the costly operation involved is just the calculation of a determinant that is repeated 1000 times for a 1500 x 1500 dense matrix.

7.1.3 Linear Regression

This benchmark solves a system of linear equations Ax = b. A is a 3000 x 3000 dense matrix and b is a 3000 element row vector. The operations that are timed are the calculation of x and transpose of b. The times are averaged over 100 runs.

7.1.4 Numerical Quadrature

This benchmarks has a loop that performs some trigonometric calculations and a division and addition. The number of loops is set to 10^7. The time is averaged for a run of 10 rounds.

7.1.5 Matrix Access

This benchmarks tests how well Matlab code can work when accessing elements of a Matrix. A square matrix of size 2^{14} is constructed and all the elements of the matrix are accumulated. The time is averaged over 5 runs.

7.2 Experimental Settings

We performed all of our experiments on an Intel(R) Xeon machine with 20 x86_64 cores each having a speed of 2.6GHz. The machine also has a Phi-co-processor with 60 cores however these cores were not utilized by the benchmarks. The RAM in the machine is 16 GB. The translated C++ codes were compiled using using GCC 4.8.3 with -O3 optimization and with the option -march=native. We used Armadillo for linear algebra operations which was compiled with Intel MKL [34] as the underlying high performance library. For Matlab benchmarks, Matlab R2015b was used to execute Matlab programs.

7.3 Results

We compared the execution times of Matlab versus hand translated C++ code and automatically translated C++ codes. We also compared the number of lines of code in Matlab, hand translated C++ and automatically translated C++.

7.3.1 Comparison of Execution Times

Figure 7.1 shows the execution times of the six benchmarks we discussed in the section above. All the times are normalized with Matlab = 1. Any benchmark having time less than 1 is faster than Matlab and benchmark having time greater than 1 is slower.

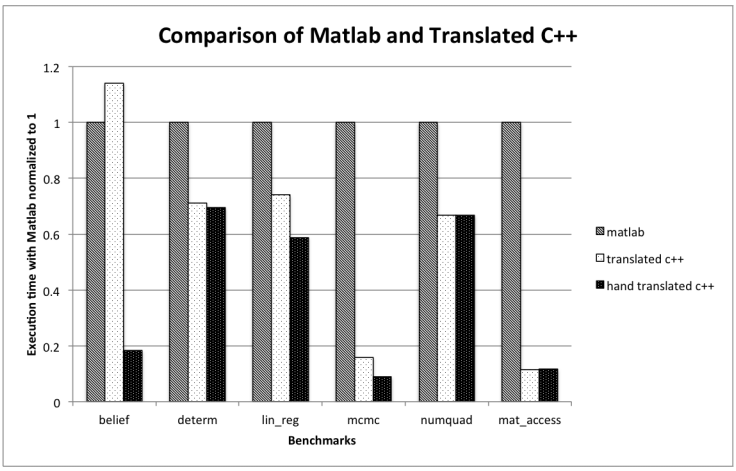

Figure 7.1: Benchmarks to show the execution time of translated C++ codes relative to the corresponding MATLAB code

Discussion of results. As we can see in Figure 7.1, automatically translated code runs as fast as the hand translated code except in one benchmark called belief propagation. It even runs slower than Matlab. The reason is that the automatically translated code does not use vectors to represent vectors but uses matrix. FastNumerics represents vectors as a Range which internally is a Matrix. The reason why belief propagation turned out to be slow is because of a division operation **x / sum(x)** where x is a vector. In FastNumerics, sum has been implemented to return a Matrix (actually sum should return a scalar if a vector is passed); so the overloaded division operator has to extract a scalar (first element) from the Matrix before performing the division. This means that the supporting components for the translated code should be written in an efficient way to gain a proper speedup.

Other five translated benchmarks perform as well as the hand translated C++ code. Benchmarks mcmc and mat_access have almost 10 times speedup. The reason is that these benchmarks have control flow and matrix accesses involved. Indexed accesses are one of the places where Matlab spends time performing run-time checks [48]. Thus C++ code obtained by translating Matlab code that has lots of control flow and matrix access turns out to perform very well. Type inference also has an impact on the performance [48].

To summarize, we can say that hand translated code (if translated optimally) will usually be faster than automatically translated code. This shows that there are lots of places to improve upon when writing a translator like FastNumerics. We could have improved the results for the benchmarks belief and lin_reg by doing a shape analysis and representing the vectors as vectors instead of a matrix. We could also have improved the sum function to return a scalar if it is operated on a vector.

7.3.2 A Sample Translation

Figure 7.2 shows how the translated version of benchmark belief propagation looks like. The generated code is readable and is similar to a hand-written code. It has a structure similar to the Matlab code.

7.3.3 Comparison of Lines of Code

Figure 7.1 shows the comparison of number of lines of code in the original Matlab benchmark, the number of lines of code in the hand translated C++ code and in automatically translated C++ code. We did not make any changes to the source code when generating the number of lines of code and we did not ignore blank lines and some comments when counting the lines of code. Therefore, the number of lines of code in the figure is just an approximation of the actual total lines of code without blank lines.

```
function belief()
  rounds = 10;
  duration = 0;
  N = 1000000;
  output = zeros(25, 1);

  for i = 1:rounds
    A = randn(25, 25);
    x = randn(25, 1);

   output = output + beliefprop(A, x, N);
  end

  disp(output);
end

function x = beliefprop(A, x, N)
  for i = 1:N
    x = A * x;
    x = x / sum(x);
  end
end
```

```
#include "matrix_armadillo/fastnumerics.h"

Matrix<double> beliefprop
( Matrix<double> A,Matrix<double> x,int N)
{
  for (auto i = 1; i <= N; i += 1) {
    x = A * x;
    x = x / sum(x);
  }
  return x;
}

void belief()
{
  Matrix<double> x;
  int rounds;
  Matrix<double> output;
  int duration;
  int N;
  Matrix<double> A;

  rounds = 10;
  duration = 0;
  N = 1000000;
  output = zeros(25,1);
  for (auto i = 1; i <= rounds; i += 1) {
    A = randn(25,25);
    x = randn(25,1);
    output = output + beliefprop(A,x,N);
  }
  disp(output);
  duration = duration / rounds * 1000;
}

int main()
{
  belief_sample();
```

Figure 7.2: Translation of belief benchmark

Discussion. We can see that the number of lines of code in hand translated C++ and automatically translated C++ are almost equal. The number of lines of code in C++ is always larger than the number of lines of code in Matlab. Matlab program is usually concise than equivalent C++ programs and that could be the reason why people choose to implement programs using Matlab. One place where the generated C++ could have much more number of lines of code than Matlab is when there are overloaded calls to functions. In that case FastNumerics generates overloaded versions of the original function. Normally the difference between the number of lines in the automatically translated C++ code and Matlab is not very high.

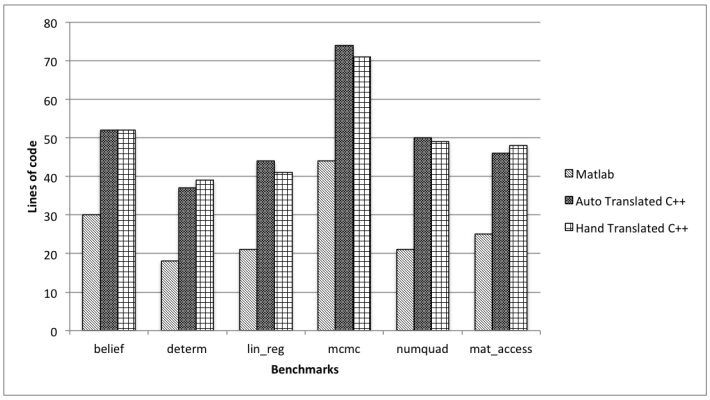

Figure 7.3: Benchmarks to show the number of lines of code in Matlab and its translated C++ code

CHAPTER 8

RELATED WORKS

There are many projects implemented previously that deal with converting Matlab to other general purpose languages. These compilers usually translate Matlab to some static language like C/C++/Fortran or target a specific numerical library. FALCON[9] is a very popular Matlab to FORTRAN translator developed in the nineties and has inspired other translators to utilize its translation principles and analysis. MEGHA [49] is a Matlab to C++ compiler that automatically maps the control flow dominated regions to the CPU and the data parallel regions to the GPU. The limitation of MEGHA is its very limited support of Matlab language. It can only operate on single dimensional arrays which limits its general usability. Also it does not support any calls to user defined functions. Our implementation supports multi-dimensional arrays and also supports user defined functions which are present in almost any Matlab program. MEGHA also requires extensive type annotation from the user whereas our implementation performs complete type inference. There is also a work on generating C and OpenCL code from MATLAB [50] which relies on programmer directives and targets embedded systems.

Instead of compiling Matlab to other languages, a work on optimizing Matlab expressions by following some guidelines (like pre-allocation of matrices) and also using vectorization also exists [48]. Our compiler is also able to work on Matlab code and re-generate transformed Matlab code, but this thesis only focuses on Matlab to C++ and more work has to be done to fully utilize this feature. Another approach taken is

to compile Matlab in order to target high performance libraries like ScaLAPACK [51]. The notable part of our work is that by using a wrapper we can target the Matlab code to use any libraries, so the translation does not need to be tailored to focus a specific library.

MiX10[52] [53] is a source-to-source compiler that automatically translates Matlab programs to X10 aiming for better use of high performance computing systems and also deals with handling concurrent Matlab code. The generated X10 code can further be compiled to either C++ or Java. MAGICA [54] is a Mathematica application that performs extensive type inference on Matlab code but just annotates existing Matlab code with inferred types. It does not perform any translation. There are translators based upon MAGICA.

There are tools like MATLAB Coder from MathWorks [55], which produce C/C++ code for a subset of Matlab. Matlab coder also supports generation of OpenMP pragmas from Matlab parfor loops. Our work can also be easily extended to support these kind of pragma insertion. The code generated is based on arrays and also requires a separate purchase. There is also an implementation [56] that produces parallel Matlab code from sequential Matlab to run on heterogeneous processors. It automates mapping, scheduling, and parallel code generation.

Also The Hiphop Compiler for PHP [57] discusses the general ideas to convert PHP to C++. PHP being a dynamic language (somewhat similar to MATLAB), this paper is relevant to us. Our implementation has been inspired by different approaches in the works listed above.

CHAPTER 9

FASTNUMERICS: FRAMEWORK FOR FURTHER RESEARCH

FastNumerics opens an area to write transformations that help enhance the performance of Matlab code. One of the areas where FastNumerics could be used is to schedule Matlab computations on heterogeneous processors. FastNumerics could be extended to insert compiler directives or some statements in the generated C++ code to send a computation to different processors.

9.1 Preliminary Experiments

Currently a framework has been created that could be instructed to send a computation to either GPU or an Intel Xeon Phi co-processor. This framework is very primitive and is just created as a proof of concept. The framework intercepts BLAS routine calls generated from the Armadillo matrix library and based upon a flag set by the user, calls an equivalent function to execute the computation in either GPU or Phi. To execute the computation on GPU, it calls the routine in a NVIDIA library called nvblas. To execute the computation on the Phi, it calls the routine in Intel MKL's BLAS library.

Figure 9.1 shows a basic flow of how our framework intercepts BLAS calls. Computations in Phi and GPU can be run in parallel as long as they do not have any data dependency.

In the Matlab code given below, the computations do not have any dependency on each other and hence can be run in parallel. The matrix multiplication A * B could

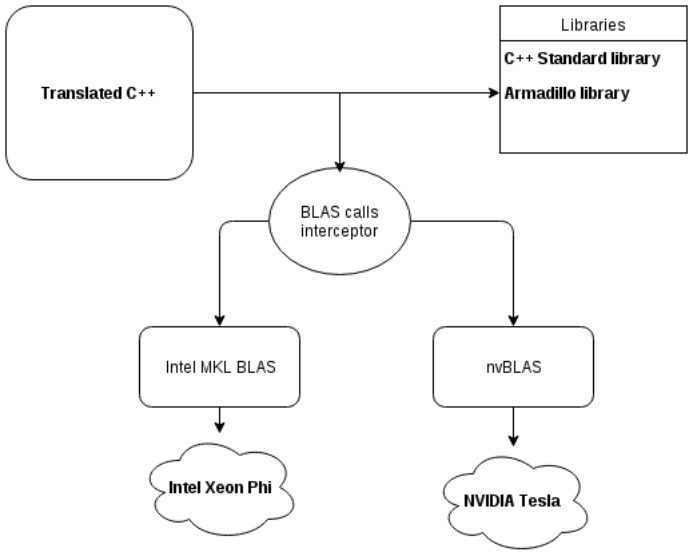

Figure 9.1: A framework to intercept BLAS calls and send them to different processors

be executed on the GPU and lu factorization lu(A) could be executed on the Intel Xeon Phi.

```
1 C = A * B
2 [L, U] = lu(A)
```

9.2 Initial Results

We tried running a matrix multiplication on the GPU and LU factorization on the Phi in parallel and in series. The machine we used has 60 Intel Xeon Phi co-processors and an Nvidia GPU, Tesla K40m. For LU, we factorized a dense square matrix of dimension 30000 and for matrix multiplication we multiplied two square matrices of dimension 30000. Figure 9.2 shows the times when running the computations in series and in parallel.

As we can see, running two independent operations in GPU and Phi in parallel will have a good improvement on speed.

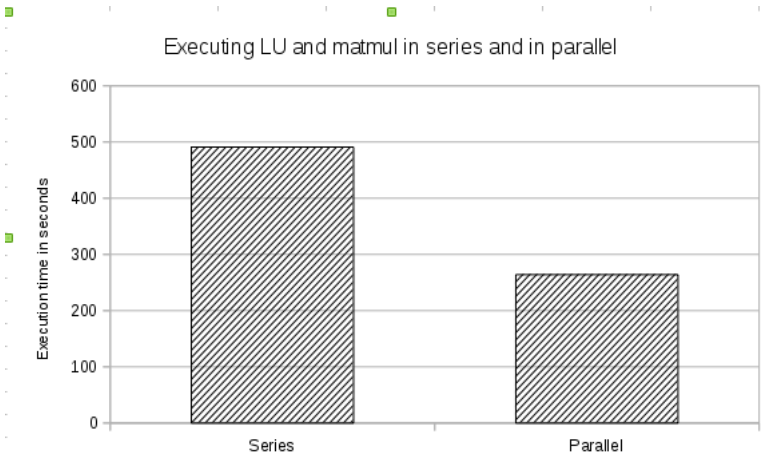

Figure 9.2: Comparison of execution time when computations are run in parallel and in series on GPU and Phi

CHAPTER 10

CONCLUSION AND FUTURE WORK

The work in this thesis focused on different aspects of transforming general Matlab code to equivalent C++ code. This thesis may also help someone to learn how general language translation can be done. The main components in this thesis are the way Matlab code is represented in an AST, how it is analyzed to find out types and how nodes are transformed to represent a C++ AST. We were able to develop a framework (FastNumerics) to perform the transformation which is extendable to add new Matlab constructs in the future. We were also able to contribute to open source software by adding a minimal Matlab support to ROSE compiler infrastructure.

From this thesis we can say that Matlab code can be transformed to C++ to achieve good performance, but it is not simply guaranteed that the C++ code produced will be faster than Matlab. The supporting libraries that is needed by the translated code need to be implemented in an efficient way. Type inference on Matlab code is best done using a flow based analysis but we should still work on improving the type inference algorithm to handle all the different ways Matlab deals with types.

FastNumerics also helps to serve as a framework to conduct different researches on transforming Matlab code. For now FastNumerics can transform Matlab codes written by following some restrictive guidelines (like assigning a variable to compatible types only, using limited constructs, etc.). Therefore, FastNumerics is still not a complete solution to handle all popular Matlab constructs. However, future additions to FastNumerics by adding new Matlab nodes, writing more transformations, adding

more built-in functions will certainly make it general purpose tool.

Bibliography

[1] Mathworks. *Matlab, version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[2] The MathWorks Inc. Matlab documentation. `http://www.mathworks.com/help/matlab/`, 2015.

[3] ISO/IEC 14882 International Standard. *Programming Language C++*. JTC1/SC22/WG21 - The C++ Standards Committee, 2011.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Prentice Hall, PTR, 4th edition, August 2005.

[5] Jack M Rogers and Andrew D McCulloch. A collocation-Galerkin finite element model of cardiac action potential propagation. *IEEE Transactions on Biomedical Engineering*, 41:743–757, 1994.

[6] Elliot B. Bourgeois, Vladimir G. Fast, Rueben L. Collins, James D. Gladden, and Jack M. Rogers. Change in Conduction Velocity due to Fiber Curvature in Cultured Neonatal Rat Ventricular Myocytes. *IEEE Trans. Biomed. Engineering*, 56(3):855–861, 2009.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] David B Loveman. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*, 24(1):121–145, 1977.

[9] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, B Marsolf, and D Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer, 1996.

[10] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):286–323, 1999.

[11] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1, 2011.

[12] Hongyi Ma, Qichang Chen, , Liqiang Wang, Chunhua Liao, and Daniel Quinlan. OpenMP-Checker: Detecting Concurrency Errors of OpenMP Programs Using Hybrid Program Analysis. In *Poster paper ICPP'12, The 41st International Conference on Parallel Processing*. September 2012.

[13] Jacob Lidman, Daniel J Quinlan, Chunhua Liao, and Sally A McKee. ROSE:: FTTransform-A source-to-source translation framework for exascale fault-tolerance research. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[14] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[15] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[16] Peter Pirkelbauer, Chunhua Liao, Thomas Panas, and Dan Quinlan. Runtime detection of c-style errors in upc code. In *Proceedings of fifth conference on partitioned global address space programming models, PGAS*, volume 11, 2011.

[17] Matthew J Sottile, Craig E Rasmussen, Wayne N Weseloh, Robert W Robey, Daniel Quinlan, and Jeffrey Overbey. ForOpenCL: transformations exploiting array syntax in Fortran for accelerator programming. *International Journal of Computational Science and Engineering*, 8(1):47–57, 2013.

[18] Digital Equipment Corporation. *FORTRAN Language Reference Manual*. Digital Equipment Corporation, 1979.

[19] Pei-Hung Lin, Chunhua Liao, Daniel J. Quinlan, and Stephen Guzik. Experiences of Using the OpenMP Accelerator Model to Port DOE stencil applications. In *OpenMP: Heterogenous Execution and Data Movements - 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings*, pages 45–59, 2015.

[20] Ken Arnold, James Gosling, David Holmes, Bill Joy, Guy Steele, Gilad Bracha, Tim Lindholm, Frank Yellin, et al. Java Language Specification. 2000.

[21] Guido Van Rossum and Fred L Drake. The python language reference manual. *Network Theory Ltd*, 15, 2011.

[22] Stig Saether Bakken, Zeev Suraski, and Egon Schmid. *PHP Manual: Volume 2*. iUniverse, Incorporated, 2000.

[23] Matthew J Sottile, Craig E Rasmussen, Wayne N Weseloh, Robert W Robey, Daniel Quinlan, and Jeffrey Overbey. ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming. In *2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011)*, page 23, 2011.

[24] Rose Compiler Wiki. How to create a translator. `https://en.wikibooks.org/wiki/ROSE_Compiler_Framework/How_to_create_a_translator`, 2016.

[25] John W Eaton, David Bateman, Søren Hauberg, and Octave GNU. A high-level interactive language for numerical computations. *Bristol, United Kingdom*, 2005.

[26] Wikibooks. Matlab programming/differences between octave and matlab — wikibooks, the free textbook project, 2015. [Online; accessed 23-November-2015].

[27] c2.com. Abstract syntax tree. `http://c2.com/cgi/wiki?AbstractSyntaxTree`, 2015.

[28] Soren Hauberg John W. Eaton, David Bateman and Rik Wehbring. *GNU Octave version 3.8.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2014. ISBN 1441413006.

[29] Charles Donnelly and Richard Stallman. Bison. the yacc-compatible parser generator. 2000.

[30] ROSE. Rose:sagebuilder namespace reference. `http://rosecompiler.org/ROSE_HTML_Reference/namespaceSageBuilder.html`, 2015.

[31] ROSE. Rose:sageinterface namespace reference. `http://rosecompiler.org/ROSE_HTML_Reference/namespaceSageInterface.html`, 2015.

[32] Conrad Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

[33] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A McKenney, and D Sorensen. *LAPACK Users' guide*, volume 9. Siam, 1999.

[34] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[35] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

[36] Srinidhi Kestur, John D Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE computer society annual symposium on*, pages 288–293. IEEE, 2010.

[37] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[38] Luca Cardelli. Basic polymorphic typechecking. *Science of computer programming*, 8(2):147–172, 1987.

[39] Daan Leijen. HMF: Simple type inference for first-class polymorphism. In *ACM Sigplan Notices*, volume 43, pages 283–294. ACM, 2008.

[40] Jens Palsberg and Christina Pavlopoulou. From Polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11:263–317, 5 2001.

[41] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[42] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[43] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(4):576–599, 1995.

[44] Pramod Joisha and Prithviraj Banerjee. Lattice-based type determination in matlab, with an emphasis on handling type incorrect programs. Technical report, Tech Report CPDC-TR-2001-03-001, Northwestern University, 2001.

[45] Shriram Krishnamurthi. Programming languages: Application and interpretation. 2007.

[46] IM Eric Friedman and Itay Maman. Boost variant, 2002.

[47] Björn Karlsson. *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.

[48] A case for source-level transformations in MATLAB, journal=Proceedings of the 2nd conference on Domain-specific languages - PLAN '99, author=Menon, Vijay and Pingali, Keshav, year=1999.

[49] Ashwin Prasad, Jayvant Anantpur, and R Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *ACM Sigplan Notices*, volume 46, pages 152–163. ACM, 2011.

[50] João Bispo, Luís Reis, and João MP Cardoso. C and OpenCL Generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1315–1320. ACM, 2015.

[51] Shankair Ramaswamy, Eugene W Hodges IV, and Prithviraj Banerjee. Compiling Matlab programs to ScaLAPACK: Exploiting task and data parallelism. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 613–619. IEEE, 1996.

[52] Vineet Kumar and Laurie Hendren. MiX10: Compiling MATLAB to X10 for high performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 617–636. ACM, 2014.

[53] First steps to compiling Matlab to X10, author=Kumar, Vineet and Hendren, Laurie, booktitle=Proceedings of the third ACM SIGPLAN X10 Workshop, pages=2–11, year=2013, organization=ACM.

[54] Pramod G Joisha and Prithviraj Banerjee. The MAGICA type inference engine for MATLAB®. In *Compiler Construction*, pages 121–125. Springer, 2003.

[55] Coder MATLAB. Generate C and C++ code from MATLAB code,© 2012 The MathWorks.

[56] Sam Skalicky, Sonia Lopez, Marcin Lukowiak, and Andrew G Schmidt. A Parallelizing Matlab Compiler Framework and Run time for Heterogeneous Systems.

[57] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, et al. The HipHop compiler for PHP. In *ACM SIGPLAN Notices*, volume 47, pages 575–586. ACM, 2012.