First steps to compiling Matlab to X10

Sable Technical Report No. sable-2013-2
(Revised, previous version 2013-01, March 15, 2013)

Vineet Kumar and Laurie Hendren

May 15, 2013

Contents

List of Figures

List of Tables

Abstract

Matlab is a popular dynamic array-based language commonly used by students, scientists and engineers, who appreciate the interactive development style, the rich set of array operators, the extensive builtin library, and the fact that they do not have to declare static types. Even though these users like to program in Matlab, their computations are often very compute-intensive and are potentially very good applications for high-performance languages such as X10.

To provide a bridge between Matlab and X10, we are developing MiX10, a source-to-source compiler that translates Matlab to X10. This paper provides an overview of the initial design of the MiX10 compiler, presents a template-based specialization approach to compiling the builtin Matlab operators, and provides translation rules for the key sequential Matlab constructs with a focus on those which are challenging to convert to semantically-equivalent X10. An initial core compiler has been implemented, and preliminary results are provided.

1 Introduction

Matlab is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide[12]. Matlab programmers appreciate the high-level matrix operators, the fact that variables and types do not need to be declared, the large number of library and builtin functions available, and the interactive style of program development available through the IDE and the interpreter-style read-eval-print loop. However, even though Matlab programmers appreciate all of the features that enable rapid prototyping, their computations are often quite compute intensive and could benefit from a system more suited to high performance computing.

On the other hand, X10 is an object-oriented and statically-typed language which uses cilk-style arrays indexed by *Point* objects, and has been designed with well-defined semantics and high performance computing in mind.

We have been working on MiX10, a source-to-source compiler that helps to bridge the gap between Matlab, a language familiar to scientists, and X10, a language designed for high performance. In particular, this paper identifies the key challenges and our approach to compiling Matlab to X10, focusing on the sequential core of X10.

The ultimate goal of the MiX10 compiler is two-fold. First, it can be used as a back-end for a Matlab system, producing high-performance code via X10. Second, it can be used to help programmers port their Matlab code to X10 source code. The techniques presented in this paper provide the core upon which these two ultimate goals can be achieved.

The major contributions of this paper are as follows:

Identifying key challenges: We have identified the key challenges in performing a semantics-preserving translation of Matlab to X10.

Overall design of MiX10**:** We provide the design of a source-to-source translator, building upon the McLab front-end and analysis toolkits.

MiX10 **IR design:** In order to provide a convenient target for the first level of translation, we have defined a high-level MiX10 IR. This IR is currently used for code generation, but in the future will also be used for code simplifications and transformations.

Template-based builtin framework: Matlab supports many builtin operations that can operate over a wide variety of run-time types. We have designed and implemented a template-based system that allows us to generate specialized X10 code for a collection of important builtin operations.

Code generation strategies for key language constructs: There are some very significant differences between the semantics of Matlab and X10. A key difference is that Matlab is dynamically-typed, whereas X10 is statically-typed. Furthermore, the type rules are quite different, which means that the generated X10 code must include the appropriate explicit type conversion rules, so as to match the Matlab semantics. Other Matlab features, such as multiple returns from functions, a non-standard semantics for `for` loops, and a very general range operator, must also be handled correctly.

Working core implementation: We have implemented the core functionality for the MiX10 compiler, concentrating on the sequential part of X10, and we provide some initial results.

The remainder of this paper is structured as follows. In Section 2 we describe the overall structure of MiX10, and how we build upon the McLab framework. Section 3 provides the high-level design of MiX10 backend with details about the MiX10 IR. In Section 4 we discuss the need to have several overloaded methods corresponding to a Matlab builtin method and describe the specialization technique to select the correct method in the generated X10 code. In Section 5 we explain how various Matlab features are mapped to X10. Section 6 Provides a performance comparison of generated X10 code with the original Matlab code. Section 7 talks about previous work related to static Matlab compilation. Finally, in Section 8 we conclude and discuss some planned future work.

2 Background

MiX10 is implemented on top of several existing Matlab compiler tools. The overall structure is given in Figure 1, where the new parts are indicated by the shaded boxes, and future work is indicated by dashed boxes.

Matlab is actually quite a complicated language to compile, starting with its rather unusual syntax, which cannot be parsed with standard LALR techniques. There are several issues that must be dealt with including distinguishing places where white space and new line characters have syntactic meaning, and filling in optional **end** keywords, which are sometimes optional. The McLab front-end handles the parsing of Matlab through a two step process. There is a pre-processing step which translates Matlab programs to a cleaner subset, called *Natlab*, which has a grammar that can be expressed cleanly for a LALR parser. The McLab front-end delivers a high-level AST based on this cleaner grammar.

After parsing, the next major phase of MiX10 uses the McSaf framework [4, 3] to disambiguate identifiers using *kind analysis* [5], which determines if an identifier refers to a *variable* or a *named function*. This is required because the syntax of Matlab does not distinguish between variables and functions. For example, the expression a(i) could refer to four different computations, a could be an array or a function, and i could refer to the builtin function for the imaginary value *i*, or it could refer to a variable i. The McSaf framework also simplifies the AST, producing a lower-level AST which is more amenable to subsequent analysis.

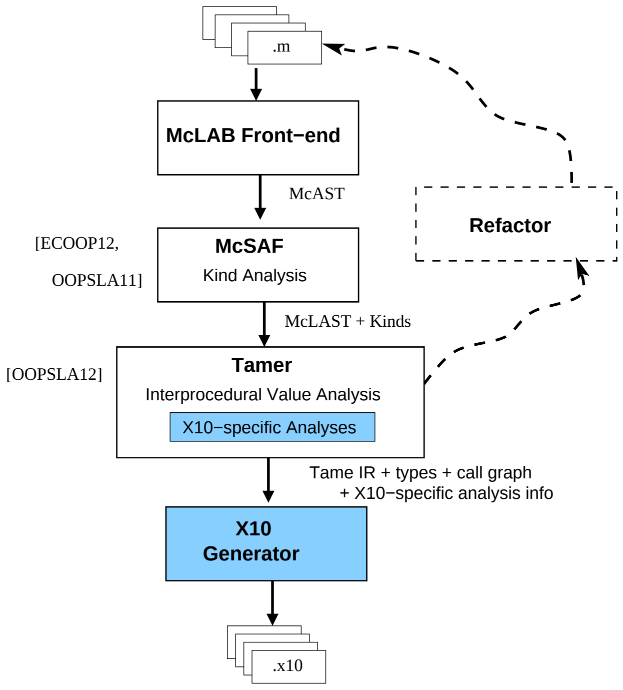

Figure 1: Overview of MiX10 structure

The next major phase is the Tamer [6], which is a key component for any tool which statically compiles Matlab. The Tamer generates an even more defined AST called *Tamer IR*, as well as performing key interprocedural analyses to determine both the call graph and an estimate of the base type and shape of each variable, at each program point. The call graph is needed to determine which files (functions) need to be compiled, and the type and shape information is very important for generating reasonable code when the target language is statically typed, as is the case for X10.

The Tamer may find dynamic Matlab features which cannot be statically compiled, in which case it flags that feature as not tame, and the ultimate goal is to support a refactoring tool which would aid the programmer to restructure their input Matlab program in order to eliminate the wild feature.

The Tamer also provides an extensible *interprocedural value analysis* and an interprocedural analysis framework that extends the intraprocedural framework provided by McSaf. Any static backend will use the standard results of the Tamer, but is also likely to implement some target-language-specific analyses which estimate properties useful for generating code in a specific target language. We have currently added an analysis for determining if a Matlab variable is *real* or *complex*.

For the purposes of MiX10, the output of the Tamer is a low-level, well-structured AST, which along with key analysis information about the call graph, the types and shapes of variables, and X10-specific information. These Tamer outputs are provided to the code generator, which generates X10 code, as discussed in the next section.

3 Design of X10 Generator and MiX10 IR

The MiX10 code generator is the key component which makes the translation from the Tamer IR, which is based on Matlab programming constructs and semantics, to X10. The overall structure of the MiX10 code generator is given in Figure 2.

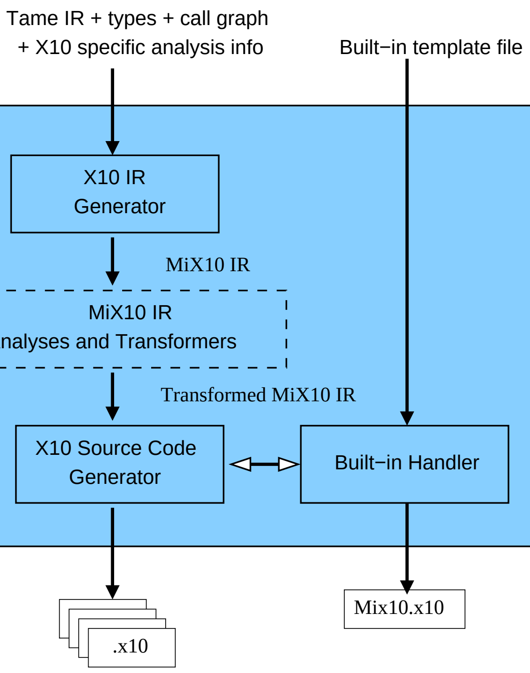
Tame IR + types + call graph
+ X10 specific analysis info
Built−in template file
X10 IR Generator
MiX10 IR
MiX10 IR Analyses and Transformers
Transformed MiX10 IR
X10 Source Code Generator
Built−in Handler
.x10
Mix10.x10

Figure 2: Structure of the MiX10 code generator

Rather than do a direct code generation to X10 source code, we have defined a general and extensible IR to represent X10. We have implemented the IR using JastAdd [7, 2], which allows us to easily add new AST nodes by simply extending the JastAdd specification grammar.

Although we currently do not transform the MiX10 IR very much, the ultimate goal is to support a variety of analyses and transformations that can be used to: (1) produce more efficient X10 code, and (2) produce more readable X10 code.

Note that there are potentially two places that optimizations and transformations may happen: either at the Tamer IR level or at the MiX10 IR level. It is our intent to put any analysis or transformation that is not X10-specific into the Tamer IR, so that other back-ends can benefit from those improvements. However, optimizations and transformations that are specific to X10 programming constructs (such as points and regions) and semantics will need to be done on the MiX10 IR. We may also use the MiX10 IR as a convenient place to insert instrumentation code.

As shown in Figure 2, the X10 source code generator actually gets inputs from two places. It uses the MiX10 IR to drive the code generation, but for expressions referring to built-in Matlab functions it interacts with the *Built-in Handler*. In the next section, Section 4, we discuss this process in in more detail, and in the subsequent section, Section 5, we address source code generation for the key X10 constructs.

4 Matlab Builtins

Matlab builtin methods are the core of the language and one of the features that make it popular among scientists. They provide a huge set of commonly used numerical functions. All the operators, including the standard binary operators (`+, -, *,/`), comparison operators (`<, >, <=, >=, ==`) and logical operators (`&, &&, |, ||`) are merely syntactic sugar for corresponding builtin methods that take the operands as arguments. For example an expression like `a+b` is actually implemented as `plus(a,b)`. An important thing to note is that unlike most programming languages, all the Matlab builtin methods by default operate on matrix values as a whole. For example `a*b` or `mtimes(a,b)` actually performs matrix multiplication on matrix values `a` and `b`. However, most of the builtin methods also accept one or more scalar, or more accurately, 1×1 matrix arguments. Builtin methods are overloaded to accept almost all possible shapes of arguments. Thus `mtimes(a,b)` can have both `a` and `b` as matrix arguments (including 1×1 matrices) with number of columns in `a` equal to number of rows in `b`, in which case the result is a matrix multiplication of `a` and `b` or one of them can be a 1×1 matrix and other can be a matrix of any size and the result is a matrix containing each element of the non-scalar argument times the scalar argument. Wherever possible, Matlab builtins also support complex numerical values. X10 on the other hand, like most of the programming languages operates on scalar values by default.

Due to the fact that X10 is still new and evolving, it has a very limited set of libraries, specially to support a large subset of available Matlab builtin methods. The X10 Global Matrix Library (GML) supports double-precision matrix operations however it is still not as extensive as Matlab's set of operations and it poses some restrictions:

1. It works on values of type `Matrix` instead of X10 type `Array` which means it needs explicit conversion of `Array` values to `Matrix` values before performing a matrix operation and and then a conversion of the results back to `Array` type. This conversion may be a large overhead, especially for small data sizes.
2. GML is limited to Matrix values of two dimensions and containing elements of type `Double`, whereas many Matlab builtin methods support values of greater number of dimensions.
3. GML currently does not support complex numerical values whereas Matlab naturally supports them.
4. Currently GML requires a separate installation and configuration which is non-trivial specially for scientists who need something that works out of the box.

Due to above restrictions, X10 Global Matrix Library is useful in some situations, for example when there is a matrix multiplication of a very large data size, but cannot be used or is not a good choice for a large number of operations.

For a language with open-sourced libraries, it would be possible to actually compile the library methods to X10. However, many Matlab libraries are closed source and thus it is not possible to translate them to X10.

4.1 MIX10 builtin support framework

We decided to write our own X10 implementations of the commonly used MATLAB builtin methods. Currently we have implemented only those methods that are used in our benchmarks. In this paper, we concentrate on how these methods are included in the generated X10 code with minimal loss of readability and performance rather than the actual implementation.

The code below shows the X10 code for the MATLAB builtin method `plus(a,b)`.

```
public static def
  plus(a: Array[Double], b:Array[Double])
    {a.rank == b.rank}{
        val x = new Array[Double](a.region);
        for (p in a.region){
            x(p) = a(p)+ b(p);
        }
    return x;
}

public static def plus(a:Double, b:Array[Double]){
    val x = new Array[Double](b.region);
    for (p in b.region){
        x(p) = a+ b(p);
    }
    return x;
}

public static def plus(a:Array[Double], b:Double){
    val x = new Array[Double](a.region);
    for (p in a.region){
        x(p) = a(p)+ b;
    }
    return x;
}

public static def plus(a:Double, b:Double){
    val x: Double;
    x = a+b;
    return x;
}
```

This X10 code contains four overloaded versions (and it still does not contain methods to support complex values and of types other than Double) based on whether the arguments are scalar or array and their relative position in the list of arguments.

Including all the overloaded versions in the generated X10 code would result in lot of lookup overhead, would require producing redundant code (versions of methods with arguments of similar shape but different types will have the same algorithm) and would generate large code with less readability. Instead we designed a specialization technique that selects the appropriate versions of only the methods used in the source Matlab program.

After studying numerous builtin methods we categorized most of them into following five common types:

Type 1: All the parameters are scalar values or no parameters.

Type 2: All the parameters are arrays.

Type 3: First parameter is scalar, rest of the parameters are arrays.

Type 4: Last parameter is scalar, rest of the parameters are arrays.

Type 5: Variable number of parameters.

Each of these categories use the same code template for different types of values.

We build an XML file that contains the method bodies for each category for every builtin method (that we support). We implement the following strategy to select and generate the correct and required methods. First, we make a pass through the AST to make a list of all the builtin methods used in the source Matlab program. Next, we parse the XML file once and read in the X10 code templates for all the categories of the builtin methods collected in the first step. Next, whenever a call to a builtin method is made, based on the results of the value analysis we generate the correct method header and select the corresponding builtin template for that method. The generated methods are finally written to a X10 class file named `Mix10.x10`. In the code generated for actual Matlab program the call to a builtin method is simply replaced by a call to the corresponding method in the Mix10 class. For example, Matlab expression `plus(a,b)` is translated to X10 expression `Mix10.plus(a,b)`.

Using the above approach not only improves the readability of the generated code, but it also allows for future extensibility, better maintenance and more specialization. One of the specialization that we are currently working on is the ability to use the Global Matrix Library for the available methods in it and whenever the data size is large enough.

5 Mapping wild Matlab features to X10

Matlab is a programming language designed specifically for numerical computations. Every value is a *Matrix* and has an associated array shape. Even scalar values are 1×1 matrices. Vectors are $1 \times n$ or $n \times 1$ matrices. All the values are by default of type `double`. Matlab naturally supports imaginary components for all numerical values and almost all operators and library functions support complex inputs. In the rest of this section we describe some of the key features of Matlab that demonstrate what makes Matlab different and challenging to compile statically and techniques used by MiX10 to translate these "wild" features to X10.

5.1 Methods

A function definition in Matlab takes one or more input arguments and returns one or more values. A typical Matlab function looks as follows:

```
function[x,y] = foo(a,b)
    x = a+3;
    y = b−3;
end
```

This function has two input arguments `a` and `b` that can be of any type and any shape and returns two values `x` and `y` of the same shape as `a` and `b` respectively and of types determined by MATLAB's type conversion rules. The Tamer IR provides a list of input arguments and a list of return values for a function. The interprocedural value analysis identifies the types, shapes and whether they are complex numerical values for all the arguments and the return values.

MATLAB functions are mapped to X10 methods. If it is the entry function, the type of the input argument is specified by the user (Tame IR requires to have an entry function or a driver function with one argument. This function may call other functions with any number of input arguments). For other functions the parameter types are computed by the value analysis performed by the Tamer on the Tame IR. The type information computed includes the type of the value, its shape and whether it is a complex value. Other statements in the function block are processed recursively and corresponding nodes are created in the X10 IR. Finally, if there are any return values, as determined by the Tame IR, a return statement is inserted in the X10 IR at the end of the method. If the function returns only one value, say `x` then the inserted statement is simply `return x;` but if the function returns more than one values (which is quite common in MATLAB) then we return a one-dimensional array of type `Any` whose elements are the values that are returned. So, for the above example the return statement is `return [x as Any, y as Any];`. Note that the use of short syntactic form for one-dimensional array construction improves the readability of the generated code. Below is the generated code for the simple example above.

```
static def foo(a: Double, b: Double){
    var mc_t0: Double = 3;
    var x: Double = Mix10.plus(a, mc_t0);
    var mc_t1: Double = 3;
    var y: Double = Mix10.minus(b, mc_t1);
    return [x as Any, y as Any];
}
```

Also note that the variables `mc_t0` and `mc_t1` are introduced by Tamer in the Tame IR. Note that their type is `Double` because in MATLAB values are `double` by default, to specify an integer in MATLAB one must use an explicit conversion, such as `int32(3)`.

5.2 Types, Assignments and Declarations

MATLAB provides following basic types:

- `double, single`: floating point values
- `uint8, uint16, uint32, uint64`: unsigned integer values
- `int8, int16, in32, int64`: integer values
- `logical`: boolean values
- `char`: character values (strings are vectors of `char`)

These basic types are naturally mapped to X10 base types as follows. Floating point values are mapped to `Double` and `Float` respectively, unsigned integers are mapped to `UByte, UShort, UInt` and `ULong`, integer values are mapped to `Byte, Short, Int` and `Long`, `logical` is mapped to `Boolean` and `char` is mapped to `Char` (vector of chars is mapped to `String` type). If the shape of an identifier of type `T` is greater than 1×1 it is mapped to `Array[T]`. The type conversion rules are quite different from standard languages. For example, an operation involving a `double` and an `int32` results in a value of type `int32`.[1] MiX10 inserts an explicit typecast wherever required.

All the Matlab operators are designed to work on matrix values and are provided as syntactic sugar to the corresponding builtin methods that take operands as arguments. Operators are overloaded to support different semantics for 1×1 matrices (scalar values). Matlab provides two types of operators - *matrix operators* and *array operators*. Matrix operators work on whole matrix values. These include matrix multiplication (`*`) and matrix division (`\`, `/`). Array operators always operate in an element-wise manner. For example array multiply operator `.*` performs element-wise multiplication. MiX10 implements all operators as builtins as described in Section 4.

Matlab is a dynamically typed language which means that variables need not be declared and take up any value that they are assigned to. X10 however is statically typed and requires variables to be declared before being assigned to. MiX10 maintains a list of all the declared variables. It starts with an empty list. Whenever an identifier appears in an assignment statement on LHS, if it is not already present in the list, a declaration statement is added to the X10 IR and the variable (with its associated type and value information) is added to the list, else if it is already present in the list, the assignment statement is added to the X10 IR and the associated type and value information is updated. In case the Matlab assignment statement is inside a loop and needs a declaration, the declaration statement (without any assignment) is added to the method block outside any loop or conditional scope and the assignment statement is added in the scope where it is present in Matlab code. If the identifier on LHS is an array, then the declaration creates a new array with the region corresponding to the shape of the array. For example a Matlab statement like `a=b;` where shape of `a` is, say, 3×3 and type is `double` will be translated to `a:Array[Double]=new Array[Double](1..3*1..3,b);` (outside the scope of any loops or conditionals). Note that the indexing starts from `1` and not `0`, the way it is done in Matlab.

5.3 Loops

Loops in Matlab are fairly intuitive except for one semantic difference from most of the languages. In a `for` loop if the loop index variable is redefined inside the body of the loop then its new value is persistent only in a particular iteration and does not affect the number of loop iterations. For example, consider the following listing.

```
function [x] = forTest1(a)
  for i = (1:10)
    i=3;
    a=a+i;
  end
  x=a;
end
```

[1]The type rules are explained in detail in the Tamer documents, `www.sable.mcgill.ca/mclab/tamer.html`.

Note that inside every iteration, the value of loop index variable `i` is 3 but the loop still terminates after ten iterations. The above code would be translated to the following X10 code:

```
static def forTest1 (a: Double)
{
  var mc_t0: Double = 1;
  var mc_t1: Double = 10;
  var i_x10: Double;
  var b: Double;
  var i: Double;
  for (i_x10 = mc_t0; (i_x10 <= mc_t1); i_x10 = (i_x10 + 1))
      {
          i = i_x10;
          i = 3 ;
          b = Mix10.plus(a, i) ;
      }
  var x: Double = a;
  return x;
}
```

To handle this somewhat different semantics we introduce a new loop index variable and assign it to the original loop index variable at the beginning of the loop body. The rest of the loop body is translated by standard rules. Note that the new loop index variable is introduced only if the actual loop index variable is redefined inside the loop body.

5.4 Conditionals

In Matlab conditionals are expressed using the if-elseif-else construct and do not have any wild semantics. Matlab also allows switch statements which are converted to equivalent if-else statements by the Tamer. It also recursively converts a statement like `if (B1) S1 elseif (B2) S2 else S3` to a series of if-else clauses like `if (B1) S1 else{ if(B2) S2 else S3}`. This if-else construct is intuitively mapped to the if-else construct in X10.

5.5 Array access and Colon operator

Arrays (or matrices) are the core of Matlab and most of the data read and write operations involve accessing one or a set of elements of an array. There are two basic ways of accessing elements of an array, as described below.

Accessing individual elements: This type of access is similar to that in C or Java where an array element is accessed given its location index along each dimension of the array. Matlab naturally supports linear indexing[2] More precisely, if the number of subscripts in an array access is less than the number of dimensions of the array, the last subscript is linearly indexed over the remaining number of dimensions in a column-major fashion. (Support for linear indexing in MiX10 is currently a work in progress). Note that array indexing in Matlab starts from 1. Matlab allows

[2]`http://www.mathworks.com/help/matlab/math/matrix-indexing.html`

the use of keyword `end` or an expression involving `end` (like `end-1`) as a subscript. `end` denotes the highest index in that dimension.

This subscripting operation to access individual elements is mapped to X10 array subscripting operation. If the rank of array is 4 or less, it is subscripted directly by integers corresponding to subscripts in MATLAB otherwise we create a `point` object from these integer values and use it to subscript the array. In case `end` is used, if we have complete shape information we easily know the highest index for a particular dimension, otherwise if shape information cannot be determined at compile time we use the `max(Int i)` method provided by the `Region` class of X10. Thus an array access such as `a(i,end)` is translated to `a(i as Int, a.region.max(1))`. Whenever an identifier of type `Double` (default in MATLAB) is used as a subscript, we need to explicitly cast it to `Int`.

Accessing a set of elements: MATLAB supports accessed and operations on a set of elements as a whole. To achieve this MATLAB allows the use of an expression involving `colon` operator in place of an integer subscript. An expression such as `a:b` (or `colon(a,b)`) creates a vector of integers `[a, a+1, a+2, ...b]`.[3] In a second form, an interval size can also be provided. For example `a:i:b` with interval size `i` creates a vector `[a, a+i, a+2i, ...k]` where `k` is the greatest integer such that `b-k<i`. Use of a `colon` expression for array subscripting takes all the elements of the array for which the subscript in a particular dimension is in the vector created by the `colon` expression in that dimension. For array subscripting we can also use ":" without specifying the lower and the upper limit. In this case elements for all the indices in that particular dimension are accessed.

Consider the MATLAB code below:

```
function [x] = crazyArray(a)
    y = ones(3,4,5);
    x = y(1,2:3,:);
end
```

In this code `y` is a 3-dimensional array of shape $3 \times 4 \times 5$. `x` is an array created by copying the elements of `y` at `(1,2,1), (1,2,2),`
`... (1,2,5), (1,3,1), (1,3,2), ... and (1,3,5)`. However `y` itself is of shape $1 \times 2 \times 5$ and is indexed normally. This code is translated into the following X10 code.[4]

```
public static def crazyArray(a: Double){
    var mc_t1: Double = 3;
    var mc_t2: Double = 4;
    var mc_t3: Double = 5;
    val y: Array[Double] =
            new Array[Double]( Mix10.ones(mc_t1, mc_t2, mc_t3));
    var mc_t4: Double = 2;
    var mc_t5: Double = 3;
    val mc_t0: Array[Double] =
            new Array[Double]( Mix10.colon(mc_t4, mc_t5));
    var mc_t6: Double = 1;
    val x: Array[Double];
    val mix10_pt_y: Point;
```

[3]See `http://www.mathworks.com/help/matlab/ref/colon.html.`

[4]Note that we are currently implementing aggregation transformations which will aggregate expressions, including folding constants into expressions.

```
    mix10_pt_y = Point.make(1−(mc_t6 as Int),
                1−(mc_t0(mc_t0.region.min(0)) as Int), 0);
    x = new Array[Double]((1..1)∗
        ((mc_t0.region.min(0)) as Int..
         (mc_t0.region.max(0)) as Int)∗
         ((y.region.min(2))..y.region.max(2)),
          (p:Point(3))=>y(p.operator−(mix10_pt_y)));
}
```

Our current shape analysis engine does not compute the shape of arrays involving `colon` operator but we can use the
`Region.min(Int i)` and `Region.max(Int i)` methods to compute the correct values at run time. In the above example, we first create a new `Point` object that serves as an offset to get the elements at the correct position of the array accessed. Then we create the new array with region derived from the resultant vector from the `colon` operator for second dimension and from the third dimension of the source array `y`. Thus the resultant array `x` has the region `1..1*1..2*1..5`. Note that MiX10 creates arrays with starting index 1 to maintain readability of the generated code for Matlab users. This is easy due to region-based arrays in X10. Providing support for `colon` operator in array access on LHS of an assignment statement and support for `colon` operator with specified interval value is currently a work in progress.

5.6 Function calls

Function calls in Matlab are similar to other programming languages if the called function returns nothing or returns only one value. However, Matlab allows a function to return multiple values. Whenever a call is made to such a function, returned values are received in a list in the order specified by function definition. For example in the statement `[x,n] = bubble(a);` a call is made to the function `bubble` which returns two values that are read into `x` and `n` respectively. This statement is compiled to following code in X10.

```
var x: Double;
var n: Double;
val _x_n: Array[Any];
_x_n = bubble(A) ;
x = _x_n(0 as Int)as Double ;
n = _x_n(1 as Int)as Double ;
```

The key idea here is to create an array of type `Any` and read the returned value. Remember that MiX10 packs the multiple return values of a method in an array of type `Any` and returns it. Individual elements of the list simply read the values from this array. If the function call is inside a loop, all the declarations are moved out of the loop and only assignments are inside the loop.

5.7 Cell Arrays

Cell arrays in Matlab are arrays of data containers called cells and each cell can contain data of any type. For example `fooCell = {'x',10,'I like',ones(3,3)};` creates a cell array containing values of type char, double, char array and a double array. To convert to X10, the elements of the cell array are packed into an X10 array of type `Any`. While accessing an element it is type cast into its original type. Consider the following MATLAB listing:

```
function [x] = cellTest(a)
  m = ones(2,3);
  n = [4,5];
  myCell = {m, n*100};
  x = myCell{1,2};
end
```

It creates a cell array containing two arrays. It is translated to the below X10 code:

```
static def cellTest (a: Double)
    {
        var mc_t2: Double = 2;
        var mc_t3: Double = 3;
        var m: Array[Double] = new Array[Double](Mix10.ones(mc_t2, mc_t3));
        var mc_t5: Double = 4;
        var mc_t6: Double = 5;
        var n: Array[Double] = new Array[Double](Mix10.horzcat(mc_t5, mc_t6));
        var mc_t0: Array[Double] = new Array[Double](m);
        var mc_t7: Double = 100;
        var mc_t1: Array[Double] = new Array[Double](Mix10.mtimes(n, mc_t7));
        var myCell: Array[Any] = [mc_t0 as Any ,mc_t1 as Any];
        var mc_t9: Double = 1;
        var mc_t10: Double = 2;
        var x: Array[Double];
        x = myCell(mc_t9 as Int, mc_t10 as Int) as Array[Double];
        return x;
    }
```

6 Evaluation

In this section we present the preliminary results of testing our current implementation. To test our compiler we used some of the benchmarks used in the previous McFor project [10] plus some standard programs like bubble sort, sieve of Eratosthenes, Fibonacci sequence generation, etc. For this paper we present the results for the following seven benchmarks.

- *bubble* is the standard bubble sort. We chose this because it involves nested loops and consists of many array read and copy operations.
- *capr* computes the capacitance per unit length of a coaxial pair of rectangles. It involves four methods and operations on 2-dimensional matrices. It is also dominated by a large number of 2-dimensional array accesses. *capr_rank* is a specialized version of *capr* for which we statically declare the rank of all the arrays in the generated X10 code.
- *dich* finds the Dirichlet solution to Laplace's equation. It involves mathematical operations on a 2-dimensional matrix and is dominated by a large number of 2-dimensional array accesses

inside nested loops. *dich_rank*, similar to *capr_rank*, is the specialized version of *dich* with statically declared ranks for the arrays.

- *fiff* is a finite difference solution to a wave equation. It is dominated by a number of trigonometric operations and involves 2-dimensional matrix data.
- *mbrt* computes a mandelbrot set. The main features of this benchmark are computations involving complex numerical values and loops.
- *nb1d* simulates the gravitational movement of a set of objects. It involves computations on column vectors inside nested loops. *nb1d_arr* uses a specialized version of the MiX10 library which has specialized methods for column vectors.

We compiled the X10 code generated by our current implementation of MiX10 using both the managed backend (generates Java) and the native backend (generates C++). For the managed backend we compiled the generated X10 code with following optimization switches: no optimizations, `-O`, `-NO_CHECKS` and `-O -NO_CHECKS`. Only `-NO_CHECKS` and `-O -NO_CHECKS` flags were used for the native backend since [8] states that `-NO_CHECKS` option is required for acceptable multi-dimensional array performance. We also used the X10 compiler with libraries compiled with `-DNO_CHECKS=true` when we used `-NO_CHECKS` for compiling generated X10 code. We first verified the correctness of the generated code by using small data. All the benchmarks produced accurate results compared to the results produced by Matlab for original Matlab implementations.

All the programs were executed on a machine with Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz processor and 16 GB memory running GNU/Linux(3.2.0-26-generic #41-Ubuntu). The Matlab version used was R2011a and X10 programs were built and executed using x10dt-2.3.1, Oracle Java version 1.6.0-01 and gcc version 4.6.3.

Table I and Table II show execution results for original benchmarks executed in Matlab compared with execution results for managed backend. Table III shows execution results for native backend. The scales 1x, 5x and 25x corresponds to problem sizes that take approximately 20 seconds, 100 seconds and 500 seconds respectively to execute the original Matlab code. The columns labeled 'speedup' show the speedup factor or relative execution time ($Matlab\ execution\ time/X10\ execution\ time$) compared to the original Matlab code. Numbers greater than one indicate that our generated X10 code is faster than the original Matlab version.

6.1 Managed backend (Java)

Let us first evaluate the performance of our generated code using the managed X10 backend which generates Java code.

For *bubble*, the MiX10 generated code is about 30% slower than the original Matlab code. Inlining of methods for array accesses in `x10.lang.array.Array` was enabled by addition of the X10 `-O` flag, giving it a speedup of about 22% over the Matlab code. Adding the X10 `NO_CHECKS` flag does not give a significant speedup because currently *bubble* involves 1-dimensional arrays which do not seem to incur a lot of array bounds check overhead.

capr and *dich* show very surprising results. The unoptimized versions give acceptable performance, and enabling the X10 `NO_CHECKS` flag gives further improvements. However, very surprisingly, adding the X10 `-O` flag caused enormous slowdowns, up to two orders of magnitude in some cases.

		X10 library compiled with CHECKS			
	MATLAB	No opt.	speedup	-O	speedup
bubble 1x	22.36	32.07	0.70	18.39	1.22
bubble 5x	139.88	201.2	0.70	114.4	1.22
capr 1x	23.53	31.41	0.75	1875.63	0.01
capr 5x	117.89	160.49	0.73	11395.21	0.01
capr_rank 1x	23.53	29.41	0.80	18.91	1.24
capr_rank 5x	117.89	146.18	0.81	90.77	1.30
dich 1x	19.9	44.59	0.45	2447.73	0.01
dich 5x	99.62	220.46	0.45	11735.06	0.01
dich_rank 1x	19.9	28.61	0.70	1762.39	0.01
dich_rank 5x	99.62	143.18	0.70	8757.56	0.01
fiff 1x	21.25	44.36	0.48	37.38	0.57
fiff 5x	107.38	221.72	0.48	195.05	0.55
mbrt 1x	18.6	117.45	0.16	117.27	0.16
mbrt 5x	93.55	579.09	0.16	581.73	0.16
nb1d 1x	24	77.95	0.31	65.71	0.37
nb1d 5x	116.28	591.65	0.20	531.24	0.22
nb1d_arr 1x	24	20.56	1.17	17	1.41
nb1d_arr 5x	116.28	163.3	0.71	124.02	0.94

Table I: Execution results (time in seconds) for managed (Java) backend - with CHECKS

Both of these benchmarks are dominated by a large number of two-dimensional array accesses inside nested loops where most of the computation is done. It turns out that the X10 optimizer inlines the code for these array accesses, and the inlined code is fairly long and complex for each access, including a dynamic check that the rank of the array is 2. With all of the array reads and writes inlined, the core computation methods become too large/complex for the Java JIT compiler to handle and thus the core computation can no longer be JIT compiled and is instead interpreted, leading to huge slowdowns (this was observed both for the Java Hotspot and J9 JITs).

To reduce the amount of code that the X10 compiler generates for the two-dimensional array accesses, we defined *capr_rank* and *dich_rank* which are the specialized versions of *capr* and *dich* where ranks of the arrays are declared statically in the generated X10 code (currently this specialization is done by hand to test its effects on performance and will be implemented automatically in MiX10 in future). The results of this specialization were quite surprising. It eliminated the dynamic rank checking code from the generated Java code which was quite a large overhead given that it was inserted for every array access and included exception handling. For *capr_rank* the generated code Java code was now short enough for the JIT compiler to compile the core computation method and thus gave speedups of over 120 times compared to unspecialized version (with optimization switched on). However, for *dich_rank* even though this specialization did provide a speedup of about 1.4 times over the optimized *dich*, it was still over 90 times slow compared to the unoptimized version (with `-NO_CHECKS` enabled). This specialization did shorten the core method, but apparently not enough to be acceptable by the JIT compiler used in our experiments.

fiff contains a large number of library calls to trigonometric functions in a loop. These calls are

		X10 library compiled with NO_CHECKS			
	MATLAB	-NO_CHECKS	speedup	-O -NO_CHECKS	speedup
bubble 1x	22.36	28.28	0.79	12.19	1.83
bubble 5x	139.88	176.96	0.79	75.86	1.84
capr 1x	23.53	22.82	1.03	1458.8	0.02
capr 5x	117.89	121.59	0.97	8578.2	0.01
capr_rank 1	23.53	22.75	1.03	4.85	4.85
capr_rank 5	117.89	104.12	1.13	23.64	4.99
dich 1x	19.9	27.54	0.72	1942.04	0.01
dich 5x	99.62	138.37	0.72	10072.32	0.01
dich_rank 1x	19.9	21.27	0.94	1584.48	0.01
dich_rank 5x	99.62	100.97	0.99	7448.48	0.01
fiff 1x	21.25	38	0.56	19.97	1.06
fiff 5x	107.38	186	0.58	85.24	1.26
mbrt 1x	18.6	115.42	0.16	116.48	0.16
mbrt 5x	93.55	571.41	0.16	577.64	0.16
nb1d 1x	24	56.82	0.42	52.69	0.46
nb1d 5x	116.28	472.35	0.25	409.27	0.28
nb1d_arr 1x	24	16.26	1.48	9.48	2.53
nb1d_arr 5x	116.28	129.22	0.90	80.08	1.45

Table II: Execution results (time in seconds) for managed (Java) backend - with NO_CHECKS

made to methods in the generated MiX10 library which in turn call the methods in the X10 Math library thus explaining the slowdown of more than 50% for unoptimized code compared to Matlab code. Switching on the `-O` flag gives a speedup of about 20% over unoptimized code. `-NO_CHECKS` also provides speedup by about 20% over unoptimized code due to 2-dimensional array accesses inside nested loops. However `-O -NO_CHECKS` gives significant speedup of above 100% compared to unoptimized code because all the code is now inlined and has "no checks" applied to it.

mbrt shows a large slowdown of 84% compared to original Matlab code. It mainly consists of mathematical operations on scalar values of type `Complex`. Matlab naturally supports complex numerical values and operations on them efficiently, whereas X10 implements constructors for complex numerical values and operations on them via the standard library. Our generated X10 code does not directly call methods in the X10 standard library but via calls to methods in the MiX10 library. We believe that the slowdown in managed backend is due to the overhead involved in dealing with all the `Complex` objects, including an allocation of a new `Complex` object for every scalar operation. Because this is mostly a scalar benchmark, there is no effect of `NO_CHECKS`.

nb1d involves operations on column vectors as a whole inside a doubly-nested loop unlike other benchmarks which operate on individual elements inside an array. Also, the size of column vectors increase proportional to $\sqrt{scale}$. The operations on the column vectors are implemented in the MiX10 library and involve iterating over every point in the column vector, represented as a 2-dimensional array. Since there are few array accesses and more method calls to the MiX10 library both `-O` and `-NO_CHECKS` have insignificant effect on runtime. For the 1x problem size the unoptimized version is about 70% slower than Matlab code and with optimizations and "no checks"

		X10 library compiled with NO_CHECKS			
	MATLAB	-NO_CHECKS	speedup	-O -NO_CHECKS	speedup
bubble 1x	22.36	121.1	0.18	17.11	1.31
bubble 5x	139.88	753.33	0.19	104.69	1.34
bubble 25x	557.63	3020.53	0.18	429.06	1.30
capr 1x	23.53	129.55	0.18	22.23	1.06
capr 5x	117.89	654.98	0.18	135.83	0.87
capr 25x	617.63	3240.32	0.19	566.1	1.09
capr_rank 1x	23.53	76.31	0.31	5.17	4.55
capr_rank 5x	117.89	390.87	0.30	28.74	4.10
capr_rank 25x	617.63	1959.83	0.32	137.91	4.48
dich 1x	19.9	127.63	0.16	25.83	0.77
dich 5x	99.62	664.62	0.15	129.49	0.77
dich 25x	508.22	3157.01	0.16	515.92	0.99
dich_rank 1x	19.9	94.93	0.21	20.23	0.98
dich_rank 5x	99.62	474.60	0.21	101.29	0.98
dich_rank 25x	508.22	2397.54	0.21	507.15	1.00
fiff 1x	21.25	137.58	0.15	24.28	0.88
fiff 5x	107.38	684.33	0.16	117.7	0.91
fiff 25x	537.22	3485.94	0.15	600.38	0.89
mbrt 1x	18.6	43.08	0.43	12.83	1.45
mbrt 5x	93.55	218.51	0.43	63.34	1.48
mbrt 25x	389.63	1083.79	0.36	317.54	1.23
nb1d 1x	24.00	157.93	0.15	94.12	0.25
nb1d 5x	116.28	1196.36	0.10	716.52	0.16
nb1d 25x	566.15	10591.97	0.05	5875.75	0.10
nb1d_arr 1x	24.00	62.34	0.38	11.46	2.09
nb1d_arr 5x	116.28	498.71	0.23	104.04	1.12
nb1d_arr 25x	566.15	4184.31	0.14	811.66	0.70

Table III: Execution results (time in seconds) for native (C++) backend

turned on it is still 54% slower. You may note that on a problem size of 5x, the MiX10-generated code is proportionally slower than that for 1x. This may be due to the fact that the Matlab array/vector-based library routines have been highly tuned and may be better optimized and/or use multithreaded libraries for larger data sizes.

Since our generated *nb1d* performed poorly due to the slowness of our general-purpose MiX10 library code for array/vector operations, we experimented with generating more efficient specialized library operations. We created a specialized version of *nb1d* called *nb1d_arr* that uses a version MiX10 library containing specialized methods for column vectors.[5] In these specialized methods, instead of iterating over the points, we use the traditional for loop to iterate over $n * 1$ elements of the array. Comparing the performance *n1bd nb1d_arr* shows that the specialized library approach gives up to 5 times faster execution times over unspecialized version.

[5]This specialization for MiX10 library is currently done by hand and will be implemented in MiX10 in the future.

6.2 Native backend (C++)

Results for compilation to native backend are reasonably good, and contained no big surprises. Overall, the `-NO_CHECKS` flag by itself does not produce performant code, but combined with `-O`, performance nearly equals original MATLAB code, with *bubble* and *mbrt* surpassing it.

capr_rank gives up to 4 times faster results as compared to *capr* with optimization and "no checks" switched on. Without the optimizations also it is over 1.5 times faster than the unspecialized version. *dich_rank* is about 1.3 times faster than the unspecialized version for both optimized and unoptimized cases. This difference in improvement for these two benchmarks is probably due to the fact that *capr* has nearly 3 times more array access operations than *dich* thus this specialization has greater impact on *capr*.

The results for *mbrt* compiled to native backend are about 9 times faster than those for managed backend. The reason is that `x10.lang.Complex` can be implemented much more efficiently in C++ using structs which can be allocated more efficiently by either being allocated on the stack or are embedded in a containing object.

Results for *nb1d* do not show any improvement with `-O` because it uses for loops with points which are not optimized [8]. *nb1d_arr* however gives a huge speedup of up to 8 times compared to *nb1d*. This shows that generating specialized MIX10 library methods is beneficial for both X10 back-ends.

6.3 Summary

Our initial experiments have given some reasonable performance results, and pointed out some places where both our code generation, and the X10 managed code generation could be improved.

In its current form, the code MIX10 generates is directly translated from low-level Tame IR and keeps all the extra variables introduced in it. Also, since default data type for numbers used in MATLAB is `double` and in X10 is `Int`, there is a large amount of type conversion overhead, specially for array accesses (MATLAB used subscripts of type `double` but X10 uses subscripts of type `Int`). Another factor is that MATLAB builtins are mapped to method calls (some of which make further library calls) which introduces further overhead. Currently it is a trade-off between readability and performance but it can be reduced once we implement method inlining optimization (which can be invoked optionally).

Our experiments also show that generating more specialized X10 code can lead to much faster executions. In particular, it appears that declaring the ranks of arrays in the generated X10 code is very important. Our MIX10 compiler can use the shape information provided by the Tamer framework to automatically infer the ranks of many arrays, so future versions of MIX10 will specialize array declarations whenever possible. We also observed that generating specialized MIX10 library routines can also provide excellent performance improvements, and we also plan to add this to MIX10.

Our results also point out some places where the code generation for the managed backend for X10 could be improved. The aggressive inlining done by the X10 `-O` option seems to be counter-productive in situations where this creates code that is too large/complex for the JIT compiler. Furthermore, the simplifications performed by the X10 compiler to enable the inlining appears to introduce many spurious temporary variables and type casts, which may be putting further pressure on the JIT compiler. We also observed significant overheads for the benchmark using `Complex` scalars. For these cases it may be worth implementing some object inlining in the generated X10 code.

In this first phase of MiX10 development we achieved our goal to generate correct and robust code for many of the commonly used Matlab features. In the next phase of MiX10 development we plan to focus on optimizations for performance of generated sequential code and to identify parallelism in Matlab code and map them to X10's concurrency controls. Matlab vector instructions and `parfor` loops are a good starting point.

7 Related Work

As discussed in Section 2, this work builds upon the previous work from the McLab group, including the front-end, the McSaf analysis framework [3, 4] analysis framework, and the Matlab Tamer [6].

There have been previous research projects on static compilation of Matlab which focused particularly on the array-based subset of Matlab and developed advanced static analyses for determining shapes and sizes of arrays. For example, FALCON [14] is a Matlab to Fortran90 translator with sophisticated type inference algorithms. The McLab group has previously implemented a prototype Fortran 95 generator [10], and is developing the next generation Fortran generator in parallel with the MiX10 project. Some of the solutions can be shared between the projects, especially the parts which extend the Tamer. The MEGHA project[13] provides an interesting approach to map Matlab array operations to CPUs and GPUs, but only supports a very small subset of Matlab.

There are also commercial compilers. One such product is the *MATLAB Coder* recently released by MathWorks[11], which produces C code for a subset of Matlab.

There are other projects providing open source implementations of Matlab-like languages, such as Octave[1] and Scilab[9]. These add valuable contributions to the open source community, however their focus is on providing interpreters and open library support and they have not tackled the problems of static compilation. We are investigating if there is any way of sharing some of their library support with MiX10.

In terms of source-to-source compilers for X10, we are aware of two other projects. StreamX10 is a stream programming framework based on X10 [16]. StreamX10 includes a compiler which translates programs in COStream to parallel X10 code. Tetsu discusses the design of a Ruby-based DSL for parallel programming that is compiled to X10 [15].

8 Conclusions and Future Work

In this paper we have outlined the important first steps in building MiX10, a source-to-source compiler from Matlab to X10. We have provided an overview of the design, including our use of existing tools and the new components we have defined for the X10 code generation.

We presented our approach to handling many Matlab features including our treatment of Matlab built-in operators, Matlab style `for` loops, and colon-style indexing operations on arrays.

We demonstrated that we could generate working X10 code for a collection of Matlab benchmarks, showing that the core translation is in place. In many cases the execution speed of the generated X10 code was comparable, and sometimes slightly faster than the original MATLAB code. In particular, the results with the native backend were quite encouraging. This initial experimental evaluation also pointed out several areas where we can improve our code generation further including: reducing extra function call overhead, minimizing type conversions, and specializing code based on the results of shape analysis.

Based on the foundations in this paper we plan to continue the project in several directions. On the performance side we intend to further optimize the generated code, and to expose the parallelism inherent in MATLAB vector instructions and MATLAB `parfor` loops. We also intend to add some X10-specific optimizations, including one that will identify immutable variables, which can thus be declared as such in the generated X10 code.

For peak performance of the generated X10 code we use the `-NO_CHECKS` switch which disables the array bounds checks and type conversion checks. However, the semantics of MATLAB require that array bounds checks be made. We are currently working on array bound analyses which will allow us to generate array bounds checks where necessary in our generated code.

One of our goals is to generate readable X10 code, so that programmers could use MIX10 to port MATLAB code to X10. The structure of the generated code is already quite clear, but the individual statements are too low-level because they are following the simplified form of the Tamer IR. We are currently developing a collection of aggregating transformations which will rebuild the expressions to a level more like what a programmer would specify.

Acknowledgments

This research has been supported, in part, by NSERC. We would also like to thank Anton Dubrau for his help in using the Tamer, and Xu Li for his collaboration as he works on a related project for generating Fortran code. Finally, we thank David Grove for helping us understand and validate some of our experimental results.

References

[1] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[2] JastAdd. `http://jastadd.org/`.

[3] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[4] Jesse Doherty and Laurie Hendren. McSAF: A static analysis framework for MATLAB. In *Proceedings of ECOOP 2012*, pages 132–155, 2012.

[5] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind analysis for MATLAB. In *In Proceedings of OOPSLA 2011*, pages 99–118, 2011.

[6] Anton Dubrau and Laurie Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, pages 503–522, 2012.

[7] Torbjörn Ekman and Görel Hedin. Reusable language specifications in JastAdd II. In Thomas Cleenewerck, editor, *Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004. Available from: `http://prog.vub.ac.be/~thomas/ERLS/Ekman.pdf`.

[8] IBM. Performance tuning. `http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html`, February 2012.

[9] INRIA. Scilab, 2009. `http://www.scilab.org/platform/`.

[10] Jun Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

[11] MathWorks. MATLAB Coder. `http://www.mathworks.com/products/matlab-coder/`.

[12] Cleve Moler. The Growth of MATLAB and The MathWorks over Two Decades. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.

[13] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[14] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[15] Tetsu Soh. Design and implementation of a DSL based on Ruby for parallel programming. Technical report, The University of Tokyo, January 2011.

[16] Haitao Wei, Hong Tan, Xiaoxian Liu, and Junqing Yu. StreamX10: a stream programming framework on X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM.