A MATLAB Subset to C Compiler Targeting Embedded Systems

João Bispo, and João M. P. Cardoso

Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP),
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto Portugal
{jbispo, jmpc}@fe.up.pt

ABSTRACT

This paper describes MATISSE, a compiler able to translate a MATLAB subset to C targeting embedded systems. MATISSE uses LARA, an aspect-oriented programming language, to specify additional information and transformations to the input MATLAB code, e.g., insertion of code for initialization of variables, and specification of types and shapes of variables. The compiler is being developed bearing in mind flexibility, multitarget and multitoolchain support, allowing for the generation of several implementations in C from the same reference code in MATLAB. In this paper we also present a number of techniques being employed in MATLAB to C compilation, such as element-wise mapping operations, matrix views, weak types, and intrinsics. We validate these techniques using MATISSE and a set of representative benchmarks. More specifically, we evaluate the compiler with a set of 31 benchmarks using an embedded system board and a desktop computer. The results show speedups up to 1.8× by employing information provided by LARA aspects, when compared with C code generated without additional user information. When compared with the execution time of the original code running on MATLAB, the execution time of the generated C code achieved a geometric mean speedup of 13×.

KEYWORDS: MATLAB; C Code Generation, Source-to-Source Compilers; Aspect-Oriented Programming; LARA; Embedded Systems.

1. INTRODUCTION

MATLAB [1] is a matrix-oriented, high-level, dynamic programming language and interactive numerical computing environment. It is a *de facto* standard in many domains of engineering and science, including embedded systems. It is ubiquitously used by engineers and scientists to quickly develop and evaluate their models and solutions.

In most embedded system settings, however, the use of a MATLAB runtime is not feasible, either because it is not available, or due to performance and/or resource constraints. To address this potential shortcoming, a typical solution relies on the development of an implementation in programming languages such as C/C++, and thus statically compiled to the target, once the base solution has been validated using MATLAB. This implementation must then in turn be validated against the output of the MATLAB code resulting in a lengthy and error prone process that further complicates the overall application development cycle and cost. The existence of two codes - the original MATLAB code and the C/C++ code - exacerbates maintenance costs, and hampers programmer's productivity.

Another solution is to automatically translate the MATLAB code to the target programming language as provided, e.g., by the MATLAB Coder [2], which translates MATLAB to C code. However, such a solution lacks low-level support to control and guide the code translation process. The code generation is typically based on directives (GUI based in the case of the MATLAB Coder) addressing types, shapes, and target. When dealing with the myriad of target architectures and toolchains in embedded systems, this approach presents a low level of flexibility. For instance, the style of the C code generator might need to be tuned to the toolchain as is the case when targeting hardware via C to hardware compilers (e.g., Xilinx Vivado HLS, Calypto Catapult-C). A MATLAB compiler framework aware of both the target computing platform and the toolchain when generating C code can be very important during the development process as it contributes to increase productivity and to significantly reduce maintenance costs.

Our approach relies on a compiler, MATISSE [3][4], which generates C code from MATLAB code. MATISSE also supports generation of highly customized C code by employing user-provided hints. To achieve this, the compiler explores the use of Aspect-Oriented Programming (AOP) [5] concepts, using the LARA language [6][7][8] as a vehicle to convey information to the compiler, such as types and shapes, and to specify compiler and monitoring actions. The compiler uses user-provided information to complement its own analysis and to derive more specialized implementations. MATISSE is being developed as a modular and flexible compiler framework, which includes custom Intermediate Representations (IRs) for MATLAB and C, keeping in mind the generation of C code from a higher-level programming language. In particular, the IR representing the output C code (C-IR) supports matrix types natively, and can be easily extended to support additional types and language constructs. The end result is a synergy between compiler analysis and the user allowing the compiler to generate very high quality code from MATLAB. Furthermore, it allows the generation of multiple C code versions from the same MATLAB source, a key capability when targeting different embedded systems, platforms, and/or toolchains. MATISSE also addresses other embedded systems concerns, e.g. generation of C code that only uses static memory allocation, when targeting C-to-hardware tools.

This paper mainly focuses on the following aspects of the MATISSE [9] compiler:

– It describes the overall architecture of MATISSE. The compiler is developed in Java exhibiting an extremely modular software architecture that can be easily augmented with specific transformations and code generation steps, thus facilitating compiler development.
– It describes the aggressive application of several techniques, such as weak-types, intrinsics, and views with pointers.
– It describes the use of LARA to complement the compiler analyses for types and shapes of array variables in MATLAB programs.
– It presents experimental results of C code generation from MATLAB examples, guided by LARA specifications with different arithmetic precision contexts and specific array shapes.
– It evaluates the compiler with 31 MATLAB benchmarks and performance execution measurements using an embedded system board and a desktop computer. The performance results are very promising as the generated C codes are competitive with existing MATLAB-to-C solutions.

The remainder of this paper is organized as follows. Section 2 briefly describes the MATLAB programming language and the main features that make it very useful and possibly also make the MATLAB to C translation difficult. Section 3 presents the MATISSE compiler and its architecture, with particular emphasis on the techniques used internally. Section 4 describes the main transformations and optimizations applied during the MATLAB-to-C translation. Section 5 shows the experimental results, Section 6 presents the related work and finally, Section 7 concludes the paper.

2. ABOUT MATLAB

MATLAB [1] is an interpreted, dynamic programming language built around the matrix data type. It has been adopted in many engineering fields, such as in control systems, image and signal processing, and simulation. The popularity of the matrix-oriented programming model is reflected in the similar languages proposed, such as Octave [10] and SciLab [11].

Like most interpreted languages MATLAB does not require variable declarations or type definitions at compile time. The default numeric representation is the IEEE-754 floating-point format with double precision. Other supported numeric data types include integers (with 8, 16, 32 and 64 bits), single precision floating-point numbers, and other numeric representations by using specific toolboxes, which enable the assignment of specific data types and operation properties (e.g., overflow mode) to MATLAB variables. Useful features of MATLAB include operator overloading, function polymorphism, and dynamic type specialization. Matrices are the main data structure, but it also supports additional data types such as arrays of heterogeneous elements (known as cells), structures, strings, booleans, and function-handlers. MATLAB allows developers to specify matrix operations in very compact code when compared to the needed code when using the C programming language.

A MATLAB program comprises functions (known as M-files) and scripts. Functions have a name, parameters, and may have zero or more return variables. Functions can be called without passing all the arguments. All function arguments are passed by value. To save memory, MATLAB execution environments pass by value only those arguments modified by a function and by reference all the others. Scripts correspond to files with MATLAB code and without specifying input/outputs. Scripts can be also called in other scripts and in functions.

A characteristic of MATLAB is that a particular name (identifier) may refer to a variable or to a script or a function. This is extended to matrix accesses and function calling, which share the same syntax. E.g., in `a(3)`, `a` can be either a variable or a function. Additionally, an identifier can be redefined to be a variable or a function at virtually any point in the code. Due to this flexibility, MATLAB postpones name resolution of identifiers to runtime.

It is common in MATLAB to use indexing expressions to access matrix elements, ranges of matrix elements, and sections of matrices. Indexing can be numerical or logical and is an important way to manipulate matrices in MATLAB.

These features are useful for developers, but represent a challenge for multi-target static compilation. They require advanced static analysis (such as type and shape inference) and the possibility to use complementary information provided by developers and/or domain experts, in order to make less conservative decisions. This research aims to translate MATLAB code to C code in a way that enables architecture-specific optimizations, and minimizes the runtime overheads (e.g., type inference) associated with MATLAB.

3. THE MATISSE COMPILER

MATISSE is a MATLAB compiler framework being developed in Java and leveraging user knowledge. MATISSE consists of a MATLAB-to-C compiler targeting embedded systems, and a LARA-controlled MATLAB *weaver* which allows transformations over MATLAB code and communication of user information using specifications in LARA [6][8]. LARA is a domain-specific language inspired by both

AOP concepts [12] and JavaScript semantics and constructs. LARA uses a mix of imperative and declarative semantics to allow developers to specify strategies for actions over application source code and/or compiler IRs (e.g., instrument code, extract information, explore transformations, apply compiler optimizations). Although LARA aspects (and the language itself) are kept as much agnostic as possible to the language they are applied to, they require a *weaver* for communicating with the target language compiler.

MATISSE can be used as a source-to-source code transformation and instrumentation tool (e.g., one can use LARA to insert and transform arbitrary MATLAB code) allowing for quick and reliable generation of reference C implementations, a key step in the development of many embedded applications. All LARA aspects are executed statically, as a compilation step, and include actions such as insertion of code, definitions of types and shapes, and code specialization based on default values. The knowledge regarding data types and shapes, provided by LARA aspects, allows MATISSE to generate highly customized C code that conforms to the requirements of a specific target.

Figure 1 presents the overall flow of MATISSE. The input MATLAB files are translated to an abstract syntax tree (AST) based MATLAB IR. This IR is then input to *MWeaver*, which uses LARA aspects to modify and add information to the IR. MATISSE currently supports two back-ends, one for MATLAB and another one for C (including an experimental option to output ANSI C). The C code generation back-end has a Transformations stage that applies optimizations such as element-wise mapping operations and matrix views.

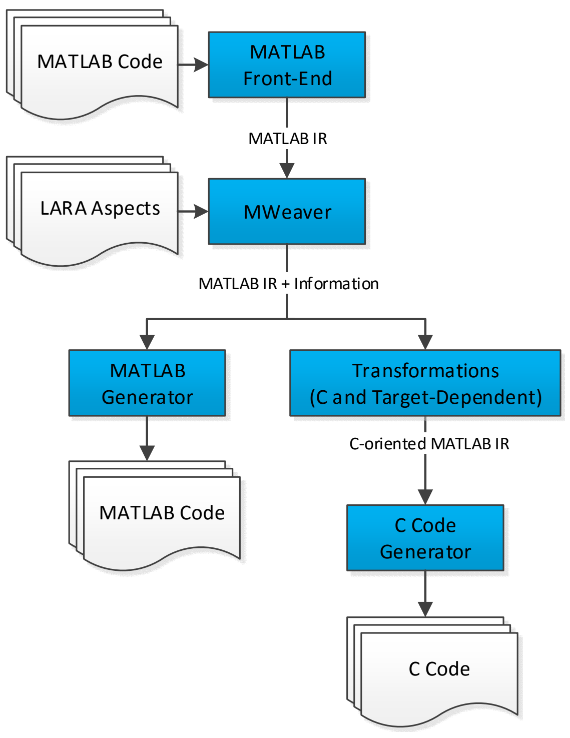

Figure 1. Overview of the MATISSE compiler framework (note that the framework is being enhanced with an OpenCL generation flow not visible in this block diagram).

The MATLAB back-end allows the generation of MATLAB code for validation, testing, monitoring, and specialization. With the C back-end, LARA aspects enable MATISSE to generate customized code for a particular target, to have fine-grained control over the generation of C code, and also to enable the generation of different implementations from the same original MATLAB code.

The knowledge regarding data types and shapes, provided by LARA aspects, allows MATISSE to generate both more efficient C code and stylized C code that conforms to the input requirements of specific tools of a given toolchain. A common example includes the restructuring of source code and the use of statically declared array variables to be compliant with the requirements of high-level synthesis tools (e.g., *Catapult C* and *Vivado HLS*).

3.1 Internal Representation

Given the differences between MATLAB and C, MATISSE uses a C specific AST-based IR to represent the C code (C-IR). This allows the compiler to separate concerns related to C generation (e.g., include files, variable declarations) that are not considered in the MATLAB IR, and simplifies the generation of C code. Having an IR for C generation allows the MATLAB IR to remain clean and independent of the specificities of the code generation to be applied.

To support translation of MATLAB code to C code, MATISSE needs to be able to specify the constraints of C. For instance, to support the statically typed notion of C, MATISSE uses the concept of *FunctionInstances* and *InstanceProviders* to generate specialized code for each function for different configurations, contexts, and input types. A configuration refers to the options available to the MATISSE compiler (e.g., enable/disable dynamic memory allocation), and a context is about information of the function call in a specific place in the code (e.g., function is called with a specific number of parameters; the expected output type is known). A *FunctionInstance* represents a particular implementation of a function. Each C function has a 1-to-1 relationship to a *FunctionInstance*, but a MATLAB function may have a 1-to-many relationship to *FunctionInstances*. An *InstanceProvider* creates *FunctionInstances*, according to the input types of the function, configurations and context. Each MATLAB function has a corresponding *InstanceProvider* that is able to create all possible *FunctionInstances* for that function.

MATISSE uses the *VariableType* to represent all the information needed about a data type (e.g., code needed to declare the variable, how to convert to another type, how to perform an addition between variables of this type). Figure 2 shows a subset of the hierarchy that starts with the *VariableType* interface. *Scalar* represents a single value, and *Matrix* represents a multi-dimensional array of elements of type *Scalar*. These two classes add contract methods that provide information specific to these types, such as number of bits, maximum and minimum values or signed/unsigned, for *Scalar*, or matrix shape and element type, for *Matrix*. *CNative* represents the native types of C (e.g., *int*, *float*, in the case of *CType*, and, e.g., *int32_t*, *uint8_t* in the case of *StdInt*). *VariableType* allows a seamless integration of several types in a modular way. For instance, by implementing the interfaces contract, MATISSE is able to automatically generate code which mixes uses of *StaticMatrix* and *DynamicMatrix*, two different matrix implementations.

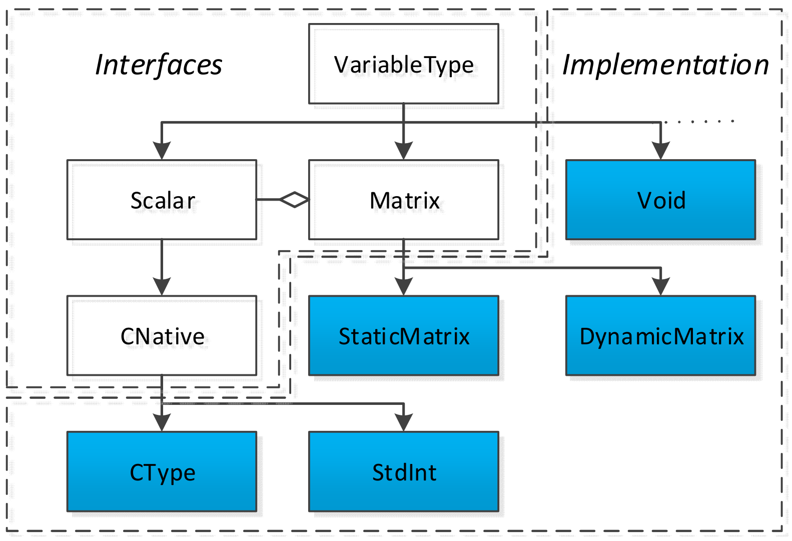

Figure 2. Subset of the *VariableType* hierarchy in MATISSE

3.2 Weak types

In our initial experiments, we observed that the performance of the generated code could be significantly improved by reducing the number of cast operations. For instance, consider the MATLAB expressions in Figure 3(a) and suppose that variable *s* has been previously inferred as a *single*. When inferring the type of constant *c* we need some kind of default type. In the case of MATLAB, it uses double as the type of all numeric constants (MATISSE distinguishes between integer and real constants, and uses user-defined defaults). According to MATISSE semantics, the result of an addition between a single and a double should return a double. Considering the type of *c* is a double, we obtain the code `r = c + (double) s;` where one cast is introduced. Note that if we generate the C code `c + s`, it still has an implicit cast operation introduced by the compiler.

In the previous case, it will be beneficial if the constant has the same type of *s*. According to the type of *s*, the compiler should generate the code in Figure 3(b) or in Figure 3(c). We introduce the concept of *weak type* to refer to types that can be changed by type inference to lower or higher precision according to the needs. In Figure 3(a), by marking the type of the constant *c* as *weak double*, the type inference system determines it is better to be a *single* (Figure 3(c)) or a *double* (Figure 3(b)), and back-propagate the type. In our current approach, the compiler coalesces the different types that might be inferred for each variable into a single type (e.g., *single* and *weak double* into *single*). Weak types are being used to help the compiler do type coalescing. Generally, the types of constants in MATISSE are considered to be *weak-types* (this includes the types of functions that return constants, such as *zeros* or *ones*).

`c = 1.5;`
`r = c + s;`
	`double c, r, s;`
	`c = 1.5;`
	`r = c + s;`
		`single c, r, s;`
		`c = 1.5f;`
		`r = c + s;`

(a)	(b)	(c)

Figure 3. Example to explain weak types: (a) MATLAB code; (b) C code resultant when *s* is inferred as a double; (c) C code resultant when *s* is inferred as a single.

However, *weak-types* should not be always ignored. For instance, consider variable *s* was inferred as an integer. If *c*, a constant of a floating-point type, is ignored, variable *r* will be inferred as an integer, which probably is not what is intended. Although a *weak type* does not provide guarantees about the type, it does have some information about what is expected in terms of the type, and when doing type inference that information is taken into account. For instance, in this case the type of *c* has the property *Real*.

To choose which types should be considered for type inference for each expression, MATISSE uses the algorithm presented in Figure 4. It starts with a list containing all the types of the expression (e.g., the input types of the addition, plus possibly a suggestion for the output type) and then splits the types into weak types and non-weak types. If one of the lists is empty, then all types are considered for type-inference. Otherwise, the algorithm collects the properties for all non-weak types. For each weak type, it then checks if it has properties that should be considered in comparison with the properties of the non-weak types. If the answer is affirmative, the weak type is used for type inference. MATISSE currently uses two properties, *Real* and *DynamicallyAllocated.* The same type can have more than one property, and the type inference is flexible enough to deal with additional properties.

```
// Input: list of types from an expression
List exprTypes
// Output: list of types to be used in type inference for the expression
List inferenceTypes

// Select non-weak types
List nonWeakTypes = exprTypes.NonWeakTypes
// Select weak types
List weakTypes = exprTypes.WeakTypes

// When there is no mix of weak and non-weak
if nonWeakTypes.isEmpty || weakTypes.isEmpty
  inferenceTypes.add(exprTypes)
  return
endif

// Properties of non-weak types
Map nonWeakPr = getProp(nonWeakTypes)

// All non-weak types are used in inference
inferenceTypes.add(nonWeakTypes)

for each weakType in weakTypes
  Map weakTypePr = getProp(weakType)
  if considerWeakType(weakTypePr, nonWeakPr)
    inferenceTypes.add(weakType)
  endif
endfor
```

Figure 4. Algorithm to choose which types to use in type inference.

3.3 Type Inference

The type inference mechanism in MATISSE uses a simple dataflow analysis [13], where type information is derived by processing each MATLAB statement, and is done bottom-up along the AST. This type inference is complemented with information provided by LARA aspects (e.g., it is possible to define the type to be used as the default real type), which is passed in a top-down manner, and we introduce several techniques to allow a less conservative inference and to be able to generate more efficient code.

There are two general situations where type-inference is applied in MATISSE: (1) in assignments, and (2) in function calls. In assignments, the variables on the left hand side are bound to the type inferred in the right hand side, unless the type of the variable has been explicitly defined using LARA.

For function calls, we took a more decentralized approach, and allow each function to define the types of its returns. When a *FunctionInstance* is created, it uses information, such as the input types of the function, to infer its output types. We show below three examples of how type-inference can change between functions:

- General scalar operations: Most scalar operations use the same inference rule in MATISSE. They consider the input types, and a possible output type if available, and determine the type with maximum rank (i.e., the type able to represent all the others with minimum precision loss). If the input types have information about the values of the variables at that point, and if constant propagation is possible, the result of the operation is added to the output type.
- Scalar division: Scalar division is a special case, where the output type sometimes needs to be of higher rank than the inputs (e.g., a division between integers can have a float as output). If all inputs are integers, the function will conservatively use a real as output, unless there is information about the input values and it can perform constant propagation (in this case it adapts the output to the result).
- Matrix creation: Certain functions, such as the ones that create matrices (e.g., zeros, ones), allow the type of the output to be explicitly passed as an input. Otherwise, it needs to use a default type, which in MATLAB is *double* (i.e., floating-point with *double* precision). In order to be able to specialize cases such as these, MATISSE uses information about the output type of the assignment, if available.

As previously mentioned, type-inference in MATISSE starts at the AST leafs. As MATISSE allows users to define the type of any variable in the code, doing type-inference while using this information poses additional challenges. Consider the example in Figure 5(a). If the user defines the type of variable `a` as `single`, this information is passed to the elements in the right hand. When inferring the type of the expression `2 + sin(b)`, the inference engine knows that the output should be a `single`, and infers that the constant and the value returned by the function should be represented as `single` (i.e., `2.0f + sinf(b)`) instead of double (i.e., `2.0 + sin(b)`). This means that MATISSE can avoid casts that can become very costly in tight loops. However, it is not always desirable to pass the information about the output type. Consider Figure 5(b), where `a` is defined as an `int`. If we propagate the `int` type to all elements, we will generate C code that performs an integer division (i.e., `a = 10 * (3 / 2)`) and lose a considerable amount of precision (the result will be 10, instead of 15). Instead, MATISSE can set arbitrary rules, per function, for the expected types for its inputs and outputs, instead of always "blindly" propagating the output type. In this case, the generated code by MATISSE is `a = (int)(10.0 * (3.0 / 2.0))`, which gives the correct result.

`a = 2 + sin(b)`	`a = 10 * (3 / 2)`
(a)	(b)

Figure 5. MATLAB examples to explain type-inference using top-down information.

3.4 MATISSE Intrinsics

When generating code for a given core functionality set, such as some MATLAB built-in and toolbox functions, MATISSE usually uses templates written in C, or more commonly, MATLAB code from which the C-IR is automatically generated. However, sometimes it is not possible to write code that will be translated efficiently to C using pure MATLAB.

For instance, consider we want to initialize a matrix, using the size of another matrix. When writing MATLAB we will possibly use the code in Figure 6(a). Also, due to compiler analysis, consider that we know that all elements of the matrix will be written before they are read (e.g., the output matrix of an element-wise operation). In this case, when creating the matrix in C we want to just allocate the memory, and avoid the initialization of the matrix values (e.g., to zero). Assuming we are using dynamically allocated matrices, the MATLAB code in Figure 6(a) can be naively translated to the C code in Figure 6(b). Note that it calls a function that initializes the values to zero, even if we know that in this case we do not need such initialization. Furthermore, there is a call to the function `size`, which creates the matrix `temp_m0`. This matrix could be avoided, since we can directly use the size of the matrix that is given as input.

To enable the generation of more efficient code from MATLAB templates, MATISSE provides *intrinsics* that can be called from MATLAB code. *Intrinsics* are MATLAB functions MATISSE understands and usually translate to more low-level C code. They are intended to be used internally by MATISSE, in templates, and we do not expect users to modify their MATLAB code to insert such functions. For instance, Figure 7(a) uses the intrinsic `matisse_new_array_from_matrix` instead of the function `zeros` to initialize a matrix with size information from another matrix, and without initialization of the matrix values (Figure 7(b)).

```
C = zeros(size(A));
```
(a)

```
zeros_double(*size_d(A, &temp_m0), C);
```
(b)

Figure 6. Code generation for *zeros* function: (a) original MATLAB code using function *zeros*; (b) C code generated.

```
C = matisse_new_array_from_matrix(A);
```
(a)

```
new_array(A->shape,
          A->dims, C);
```
(b)

Figure 7. Using MATISSE intrinsics: (a) MATLAB code representing initialization of a matrix with size information from another matrix, and without values initialization (intrinsic *matisse_new_array_from matrix*); (b) C code generated when using the MATISSE intrinsic.

Examples of other *intrinsics* are the *matisse_change_shape*, which changes the shape of a matrix directly, instead of creating a new matrix with the new shape; *matisse_to_real*, which casts an element to a default real type defined in MATISSE configuration; or *matisse_idivide*, which performs integer division (see Section 4). In addition, there are *intrinsics* for debugging and monitoring. For instance, *matisse_probe* accepts an arbitrary MATLAB expression and statically reports the output type inferred for that expression.

Although the main purpose of MATISSE *intrinsics* is to be able to generate the C code we need from internal MATLAB templates, we can also use LARA strategies to insert calls to *intrinsics* in user MATLAB code, before generating the C code (i.e., it is possible to use them directly in any M-file, for instance during transformations with LARA aspects). Note that we consider the use of *intrinsics* in MATLAB user code as an advanced technique to get further customization, and their use is not required to generate efficient C code.

Finally, MATISSE provides a compatibility library with MATLAB implementations of the *intrinsics*, so that MATLAB code which uses *intrinsics* can still run in a MATLAB environment (for instance, to test the outputs).

4. MATISSE TECHNIQUES

In this section we present some of the techniques MATISSE applies to generate customized C code.

4.1 LARA Aspects for MATLAB transformations

MATISSE includes a MATLAB weaver which allows transformations over MATLAB code and can be used as a source-to-source code transformation and instrumentation tool. The weaving stage of the compiler performs LARA specified actions such as insertion of code, definitions of types and shapes, code specialization based on default values, and code transformations.

For instance, consider the code for the benchmark *closure*, in Figure 8(a), where the input *B* is always a square matrix with a power of two side size. Under these conditions, the two divisions in the code (`ii = N/2` and `ii = ii/2`) will always be integer divisions, but MATISSE will automatically use a real division, to avoid possible loss of precision. One way to force an integer division is to use the MATLAB function *idivide* instead of the division operator. As *idivide* uses MATLAB integer types, we have to introduce a cast and obtain the code in Figure 8(b). However, this has several caveats: first, the code is now a bit more complex and might add up to the complexity if several transformations are needed; second, and more importantly, we are now forcing the program to always do an integer division. In the context of MATISSE, we propose to maintain the original MATLAB code as generic as possible, and any target-specific information be conveyed separately, in an aspect, and woven with the source code during a compilation step.

Figure 9(a) shows a LARA aspect which replaces the divisions in the original *closure* code with a call to the function *matisse_idivide* (Figure 9(b)), a MATISSE intrinsic (see Section 3.4) which indicates a division that can be performed as an integer division. The code calls a generic library aspect, shown in Figure 9(c), which can be used to replace arbitrary operators with a function, depending on their operands. This way, we avoid having two versions of the same reference code in order to pass information to MATISSE that, e.g., those divisions can be done as integer divisions.

4.2 Matrix Views

A *view* is a section of a matrix, which is accessed using the colon operator (e.g., `A(N:M)`). Such accesses are very common in MATLAB, and the most straightforward way to implement them is to create a temporary matrix with a copy of the values in the range specified by the operator. If the range has a step of 1, we can avoid copying the values and just pass a pointer to the beginning of the range (i.e., a *matrix view*).

```
% B is a square matrix whose side size is always a power of two
function [B_out] = closure(B)

N = size(B, 1);
...

ii = N/2; % This division will always be an integer division
while ii >= 1,
  B = B*B;
  ii = ii/2; % This division also
end;

B_out = B > 0;

...
```

(a)

```
ii = double(idivide(N, int32(2)));
```

(b)

Figure 8. Benchmark *closure*: (a) part of the original MATLAB code; (b) code of one of the divisions, when using the MATLAB integer division.

```
aspectdef closure_idivide
   select function{"closure"}.operator end
   apply
      call replace_operator($operator, "/", "N", "2", "matisse_idivide");
      call replace_operator($operator, "/", "ii", "2", "matisse_idivide");
   end
end
```

(a)

```
...
ii = matisse_idivide(N,2);
while ii >= 1,
   B = B*B;
   ii = matisse_idivide(ii, 2);
end;

...
```

(b)

```
aspectdef replace_operator
    input $operator, operatorSymbol, lOperand, rOperand, newFunction end
    check $operator.symbol == operatorSymbol end

    var leftValue = $operator.operands[0];
    if(leftValue != lOperand) {return;}

    var rightValue = $operator.operands[1];
    if(rightValue != rOperand) {return;}

    $operator.insert replace
         %{[[newFunction]]([[leftValue]], [[rightValue]])}%;
end
```

(c)

Figure 9. LARA and intrinsics: (a) LARA code that replaces division operators with intrinsic *matisse_idivide*; (b) code after weaving; (c) LARA code for replacing operators.

However, it is not always possible to use matrix views. One such example is when the matrix is modified before the view is read. A more subtle case is when the potential view is on the right hand of an assignment, over a local variable, and the recipient is an output of the function (e.g., `output_var = local_var(N:M)`). The problem with this case is that when using a view with pointers, `output_var` will point to the values in `local_var`. After the function ends, `local_var` is freed and output_var will be a dangling pointer. Although currently MATISSE detects this case and uses a view by copy, we are considering the use of a reference-counting mechanism to avoid copies in such cases.

4.3 Attributes

LARA aspects can be used to set attributes to elements in the code. For instance, MATISSE can assign to a variable the attribute "constant", which refers to variables whose values do not change during their lifetime, after the initial assignment. Consider the code `s = z(a:b)`. In the general case, we should use a view by copy (see Section 4.2) to obtain the range values, and then copy the contents of the view to matrix `s`. However, if we know both `z` and `s` are constant in the scope they are used, we can avoid both copies by 1) using a pointer view instead of a copy view, and 2) making variable `s` itself a view of `z`.

Although this information could possibly be extracted using source-code analysis, currently MATISSE relies exclusively on user-provided information to indicate the constant variables (see Figure 10). Note that for some cases, it might not be possible to use automatic analysis, and user information may still be needed.

```
select var{"z", "s"} end
apply
    $var.def constant = true;
end
```

Figure 10. LARA code for defining variables *z* and *s* as constant.

4.4 Compile-time Copy-on-Write

Even though MATLAB uses pass-by-copy semantics, for efficiency reasons MATISSE passes matrix pointers instead of matrix copies to functions). To avoid losing the pass-by-copy semantics, MATISSE does a preventive copy when an input of a function is altered inside the function.

First, MATISSE checks if any of the inputs appears on the left hand side of an assignment. If an input is found, then it checks if it is a reference type, such as a matrix, or a pointer to a scalar. After collecting all the inputs that can be changed during the execution of the function, it modifies the code as shown in Figure 11.

This transformation is conservative – the input variables might be written only on certain control-flow paths that are not always executed. Possible MATISSE extensions may include code for a runtime check every time an affected variable is written instead of doing always a preventive copy.

(a)	(b)

```
function out_var = f(input_var)
   input_var(3) = 0;
   out_var = input_var(2:end);
end
```
	```
	function out_var = f(input_var_original)
	   input_var = input_var_original;
	   input_var(3) = 0;
	   out_var = input_var(2:end);
	end
	```

Figure 11. Copy-on-Write transformation example: (a) original MATLAB code; (b) transformed MATLAB code.

4.5 Nargin and Nargout Removal

MATLAB allows the use of two system variables (`nargin` and `nargout`) that store the number of input/output arguments for a given function call. Figure 12 shows a common idiom in MATLAB, used to support a variable number of function inputs with default values. The `nargin` system variable is available at runtime and returns the number of input arguments passed in the call to the currently executing function. The MATISSE approach avoids the generation of generic C code to support a distinct number of input/outputs used in distinct calls. Instead, each call to a given function using a different number of input/outputs implies a specialized version of the function.

Our approach to this kind of concerns considers:

- Migration of these concerns to LARA aspects, in the case where we want to maintain a clean MATLAB code version;
- Generation of multiple versions of the function according to the number of input/output arguments of the function call.

MATISSE converts MATLAB functions when there is a call in the code to a specific function. This means that when converting all MATLAB functions but the topmost, MATISSE knows exactly with how many arguments the function being converted was called, and can specialize the code of the function according to the specific number of arguments.

```
function out_var = f(var1, var2, var3)

   if(nargin < 3)
      var3 = 0;
   end

…
```

Figure 12. MATLAB idiom for variable input arguments

4.6 Transforming Element-Wise Mapping Operations

Certain MATLAB operators and functions (e.g., addition, subtraction, square root) perform element-wise mapping operations. These operations calculate each output matrix element from one input matrix element at the same position (or from multiple elements from distinct matrices all at the same position).

In the presence of chained-operations, allocating a new matrix for each operation is inefficient. Thus, MATISSE rewrites chains of function calls consisting of element-wise mapping operations to *for* loops. Figure 13 shows an example where the original MATLAB code (see Figure 13(a)) includes a chain of element-wise operations. A naive implementation uses the chain of function calls shown in Figure 13(b) and Figure 13(d), when using static memory allocation and dynamic memory allocation, respectively. With the element-wise mapping transformation, MATISSE is able to obtain the codes in Figure 13(c) and Figure 13(e). These versions are advantageous, since the *for* loop is more efficient to implement in C, prevents the use of temporary variables, and exposes data-parallelism.

Note that some functions are element-wise mapping operations depending on the types of input. For instance, the multiplication operator (*) is element-wise if one of the operands is a scalar, but not if both operands are matrices.

```
R = sqrt((A .* A) + (B .* B));
```

(a)

```
sqrt_e((add_e(((mult_e(A, A, temp_m0))), ((mult_e(B, B, temp_m1))), temp_m2)), R);
```

(b)

```
for(i = 0; i < numel_A; i++){
  R[i] = sqrt(A[i]*A[i] + B[i]*B[i]);
}
```

(c)

```
sqrt_e((add_e(((mult_e(A, A, &temp_m0))),((mult_e(B, B, &temp_m1))), &temp_m2)), R);
```

(d)

```
new_array_helper_f(A->shape, A->dims, R);
for(i = 0; i < A->length; i++){
   (*R)->data[i] = sqrt(A->data[i]*A->data[i] + B->data[i]*B->data[i]);
}
```

(e)

Figure 13. Element-wise example: (a) a MATLAB statement of element-wise operations; (b) the equivalent C with static allocation when element-wise transformation is disabled and (c) enabled; (d) the equivalent C with dynamic allocation when element-wise transformation is disabled; (e) enabled.

4.7 Algebraic Analysis for Range Determination

Figure 14 shows a segment of code which uses arithmetic expressions to determine a range of values for matrix *A*. In order to be able to generate code using static memory allocation, we need to know the size of all matrices at compile time, which in the case of matrix B, implies knowing the number of elements in the range for matrix *A*.

The number of elements in a range of values is given by the equation end – start + 1, which for the range in matrix *A* corresponds to the expression (N×M×i) - (N×M×(i-1) + 1) + 1. MATISSE generates the expression and then applies an algebraic solver (it currently uses the library Symja [14]) to calculate the number of elements in the range (N×M, in this case). If *N* and *M* are statically known, MATISSE then substitutes the expressions to constants, and is able to generate a static version of the code.

```
for i=1:L
   B(N*M*(i-1)+1:N*M*i) = fft2d(A(N*M*(i-1)+1:N*M*i));
end
```

Figure 14. Example of code that uses expressions to determine ranges.

4.8 Using third Party Libraries

Using third party libraries is important for generating high-performance code. One example is the use of BLAS [15] for matrix multiplication operations. It is also important to control when libraries and which ones should be used or not. For instance, for small matrix sizes the use of BLAS can be less efficient than a simple matrix multiplication implementation. MATISSE addresses these aspects using LARA strategies and including support to interface to third-party libraries (at the moment the support allows the use of BLAS). Further improvements will focus on the specification of library interfaces to make MATISSE more flexible.

Currently, in the case of BLAS, the use of the library is controlled by a threshold over the size of the matrices used in a function call (which can be defined by the user). In the context of this paper, we do not intend to provide a scheme to find when a library should be used in a general case.

5. EXPERIMENTAL RESULTS

We present here an evaluation of the MATISSE compiler using a set of 31 MATLAB programs[1] obtained from several sources. Table 1 presents the benchmarks we consider, along with the size of inputs.

We tried to avoid modifications to the original MATLAB code as much as possible. The necessary modifications included the removal of code not related to the functionalities, such as code for time measurements (*capacitor*, *closure*) and renaming output names (*closure*), the adaptation of the code for each feature not currently supported by MATISSE, such as complex numbers (*diffraction*), and the inclusion of pre-allocation of matrices (*crnich*, *diffraction*). Note that some of these modifications, such as pre-allocation of matrices, can be expressed by LARA aspects. Most programs could be compiled without additional information from the user, e.g., including LARA aspects to optimize the generated code. The performance of the modified MATLAB code remained similar to the achieved by the original MATLAB code, with the exception of the code for *diffraction*, whose modifications resulted in a speedup of 1.5× over the original MATLAB code. The results where compared against the new, faster MATLAB version.

We consider here three platforms for our experiments:

- PC – a desktop PC with a 2.66 GHz Core 2 Quad processor, Windows 7 64-bit, 12 GB of RAM;
- ODROID – an ODROIDXU+E board, with an Exynos 5410 SoC with an 1.6 GHz ARM's big.LITTLE configuration, 1 GB of RAM;
- BeagleBoard – a BeagleBoard-XM revB running Ubuntu 12.10 32-bit, with a 1 GHz ARM Cortex-A8 and 512 MB of RAM.

We used GNU gcc 4.8 on all platforms, MATLAB R2014a on the desktop, and OpenBLAS v0.2.12 as the BLAS library. An online version of the MATISSE compiler is available [9].

Regarding the LARA aspects, they are executed statically, as a compilation stage, and compilation time is dominated by C compilation (i.e., by gcc), the LARA compilation and weaving represent a small fraction of the total compilation time. The use of LARA aspects to transform the original MATLAB code was limited to the insertion of the intrinsic *matisse_idivide* (as explain in Section 4.1) in the benchmark *closure*, and performing loop interchange in the benchmarks *capacitor*, *conv_2*, *editdist* and *seidel*.

Application speedups are calculated taking into account the total execution time of the binaries obtained by compiling the final C application. Since some C source files might contain some of the inputs (e.g., the source file with the *main* method), these files are compiled separately from the rest of the code, to avoid optimizations such as aggressive constant propagation from the C compiler.

[1] The MATLAB source code, the LARA aspects and the inputs of the benchmarks (as well as other elements such as the C code generated by MATISSE) can be found at http://specs.fe.up.pt/publications/spe2016.zip

Table 1. Benchmarks considered for results and corresponding input sizes.

Benchmark	Input Sizes	Benchmark	Input Sizes
adapt [16]	100,000	fft2d_v2 [7]	256×256
capacitor [16]	256×512	finediff [16]	4096×4096
cfd3_v1 [7]	256×256x3	fir_1d [7]	1,000,000×32
cfd3_v2 [7]	256×256×3	gauss [16]	257×513
closure [16]	4,096 vertices	grid_iterate [7]	32×64×16, 100
conv2 [7]	96, 11	hypotenuse [2]	512×1024
copy_vs_ptr [2]	2048×2048	latnrm [7]	32,000; 8
crnich [16]	8000×4000	monte_carlo [3]	200
diff [16]	61×1580	nbody1d [16]	4096 bodies
dilate [7]	2048×2048	nbody3d [16]	512 bodies
dirich [16]	2000×2000	rgb2yuv [2]	1024×600
dotprod [7]	512×512×10	rgb2yuv_v2 [2]	1024×600
dotprod_v2 [7]	256×256×3	seidel [16]	257×513
editdist [16]	4096 chars	subband_2d [7]	2048×2048×2048
fdtd [16]	512×512×15	tridiagonal [16]	1×2000
fft2d [7]	256×256		

The C code used herein for evaluation is related to C code generated and producing similar results to the ones achieved by the original MATLAB code. For all results, we tested if the values given by the C code were within 0.1% of the original values.

5.1 MATISSE vs Coder

We were unable to use the presented benchmarks to compare MATISSE with a commercial tool such as Mathworks' MATLAB Coder [2]. Our comparison is limited to the use of an averaging filter function[4] available at the Mathworks' website. For this example, which uses only statically-allocated memory, the C code generated by MATISSE was 2×, 2.2× and 1.5× faster than the C code generated by Coder, when running on the PC, ODROID, and BeagleBoard platforms, respectively.

The codes generated by Coder and MATISSE for this function are very similar. The most noticeable difference is that in the main loop, Coder generates two calls to the library function `memcpy`, while MATISSE generates only one call, being the functionality of the second `memcpy` implemented in a `for` loop.

The `memcpy` function is an optimized C library function which usually is significantly more efficient than using an equivalent `for` loop. However, the second call to `memcpy` inserted by Coder is done over a 16-byte unaligned address (i.e., `&buffer[1]`), which can impact performance. We observed that the gcc compiler is able to vectorize the `for` loop generated by MATISSE, by using instructions that move two values in a single operation (e.g., `movapd %xmm2,%xmm1` in the case of the PC platform). These instructions can only be used over 16-byte aligned addresses.

[2] Written by the authors

[3] Based on an example by MathWorks (http://bit.ly/1Y9u1KO)

[4] http://www.mathworks.com/help/coder/examples/averaging-filter.html (accessed on December 2015)

Additionally, the call to `memcpy` inserted by MATISSE is optional. It is possible to disable library functions on a per-function basis through LARA aspects and generate pure C implementations. This can be useful when targeting systems which do not support such functions, as is the case when targeting hardware using C to hardware compilers.

Although this single example cannot be used to make a strong comparison, it shows some of the features of MATISSE: (a) the possibility to generate code with or without calls to functions possibly not supported by the target toolchains (as the ones not supported by the C to hardware compilers); (b) the generation of compiler-friendly C code (important in this example to enable loop vectorization).

5.2 Results using LARA strategies

For each benchmark we consider three LARA strategies, according to the number of defined types for variables: *minimum*, where all inputs are considered to be *double* (in practice, this is accomplished by a single LARA statement setting the default real type to double); *inputs*, where in addition to the definitions included in *minimum*, we define the inputs as *single* or as integer types whenever possible (this strategy is similar with what we can do with MATLAB Coder); and *optimized*, where we set the default real type to *single*, and in addition to the types in *inputs*, we define any variable type to obtain a program that uses *single* types and integer types whenever possible.

Using the strategy *inputs*, we achieved a geometric mean speedup of 1.04×, 1.01× and 1.24× over the C code generated by the strategy *minimum*, for the PC, ODROID, and BeagleBoard, respectively. The BeagleBoard is the case that benefitted most from defining the input types, while for the other targets we saw virtually no difference. This happens because the BeagleBoard is the system where there is a greater difference in performance when using different types (e.g., integers, singles and doubles).

The *optimized* strategy achieved further improvements of 1.35×, 1.32× and 1.21× over the C code generated by the previous strategy *inputs*, for the same targets. Besides the default real being set to *single*, the *optimized* strategy defines types for additional variables in 12 benchmarks, being the maximum number of defined variables 6 (*fft2d*), and the minimum 1 (*editdist*). On average, the strategy defines 2.25 additional variables for each of the 12 benchmarks.

Figure 15 shows the individual speedups obtained for those 12 benchmarks, when compared with the *inputs* strategy (we consider to be a fairer comparison than using the strategy *minimum*). The source of the improvements are mostly from avoiding doing casts (with significant cost when inside loops) with the *optimized* strategy. The *inputs* strategy often infers certain types conservatively as *doubles*, including the outputs, even if the inputs are defined as *single*.

The performance of *editdist* is significantly higher when running on the BeagleBoard (3.6×). This is due to the specialization that replaces uses of floating-point data types with integers. The performance of integer operations on the BeagleBoard is much higher than floating-point performance (in the other two targets the performance is similar).

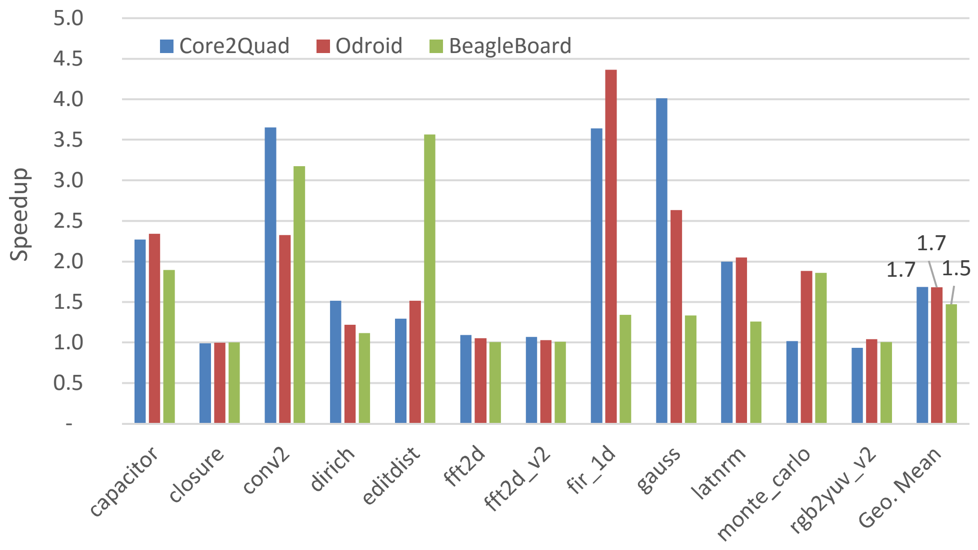

Figure 15. Speedups when using the strategy *optimized*. The baseline execution refers to the code generated using the strategy *inputs*.

5.3 MATLAB vs C Results

Figure 16 shows the speedups obtained by the C code generated with the *optimized* strategy, when compared with MATLAB executions on the PC. The absolute execution times are presented in Table 2. They are averages over 5 measurements, the variation in the majority of cases was 1% or lower, and was never higher than 3%. For all cases, the execution time is at least the same as in MATLAB (e.g., *closure*) or lower. On average, the C code generated by MATISSE has a speedup of 13× over the execution of the original code in MATLAB.

Figure 17 compares the performance between C code generated by MATISSE and C code generated by MEGHA [17] (see Table 3 for a detailed view of the values). Although MEGHA focuses on generating CUDA-enabled code, it is also able to generate efficient C code. We tried to replicate in MATISSE the experimental setup used by Prasad et al. [17], by using input matrices with the same sizes, and adjusting inputs in certain cases so that we could get approximate absolute MATLAB execution times for the same benchmarks. We were not able to obtain neither MEGHA nor its generated C code, so we use here the published speedup values which compare the performance of the C code generated by MEGHA with the execution of the original MATLAB. We are aware that the hardware and the software versions used in the experiments are different, and it is not possible to make a quantitative comparison, even if the results are taken to a relative reference (the MATLAB execution). Note that the C code currently generated by MATISSE does not target multicore architectures, and one of the planned steps is to generate C code with OpenMP directives. Also, the more recent version of the MATLAB environment used in our experiments might be more advanced and thus might achieve better performance when running MATLAB code. In this case, the speedups achieved by the code generated by MATISSE would highlight even more the efficiency of MATISSE.

Experiments suggest that the performance obtained by MATISSE is comparable to the performance of MEGHA for C code (8.1× and 5.6× of geometric mean for MATISSE and MEGHA, respectively). The case where there was a greater difference between MEGHA and MATISSE, when considering the biggest inputs, was *editdist* (2.8× difference). We think this might have to do with *editdist* being the only example that can be implemented using only integers, a case that MATISSE can optimize aggressively using aspects.

Table 2. Execution time average of 5 measurements when running the MATLAB code and the C code generated by MATISSE with optimize strategy, on the PC.

Benchmark	**Time (s)**		**Benchmark**	**Time (s)**	
	MATLAB code	**C code generated by MATISSE**		**MATLAB code**	**C code generated by MATISSE**
adapt	0.12	0.042	fft2d_v2	0.13	0.0066
capacitor	16.6	0.92	finediff	1.00	0.16
cfd3_v1	3.38	0.037	fir_1d	3.37	0.044
cfd3_v2	0.55	0.033	gauss	258E-07	9.23E-07
closure	44.3	43.4	grid_iterate	3.29	0.029
conv2	0.044	0.0049	hypotenuse	0.0052	0.0026
copy_vs_ptr	0.026	0.0053	latnrm	0.23	0.0010
crnich	9.4	1.59	monte_carlo	2.64	0.039
diff	0.23	0.045	nbody1d	19.7	5.64
dilate	26.7	2.97	nbody3d	42.8	10.7
dirich	59.6	7.16	rgb2yuv	0.072	0.022
dotprod	0.19	0.025	rgb2yuv_v2	4.62	0.052
dotprod_v2	0.00918	0.0015	Seidel	0.016	0.0014
editdist	37.8	0.22	subband_2d	0.036	0.026
fdtd	189	20.0	tridiagonal	38.4E-05	4.26E-05
fft2d	0.85	0.0066			

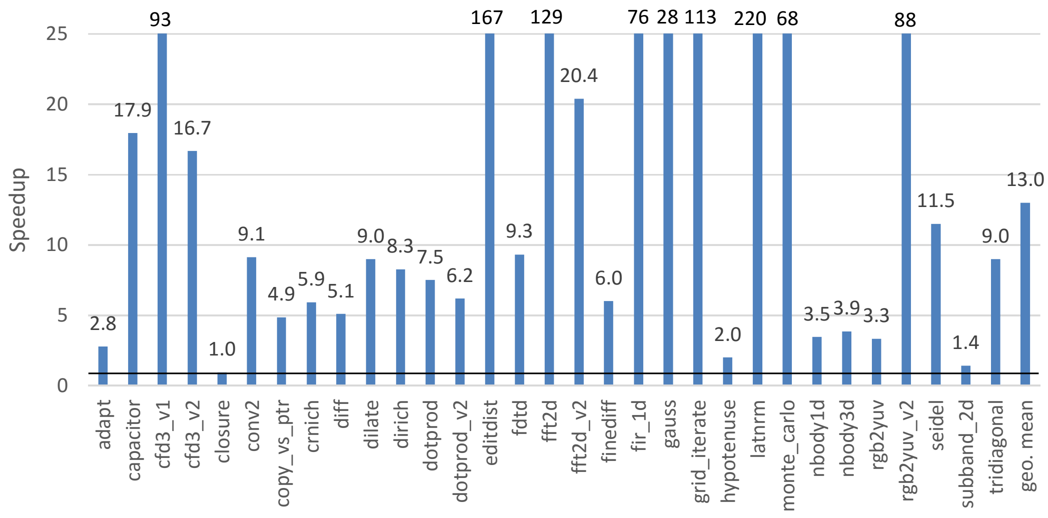

Figure 16. Speedups of C code generated by MATISSE vs MATLAB code when running on the PC.

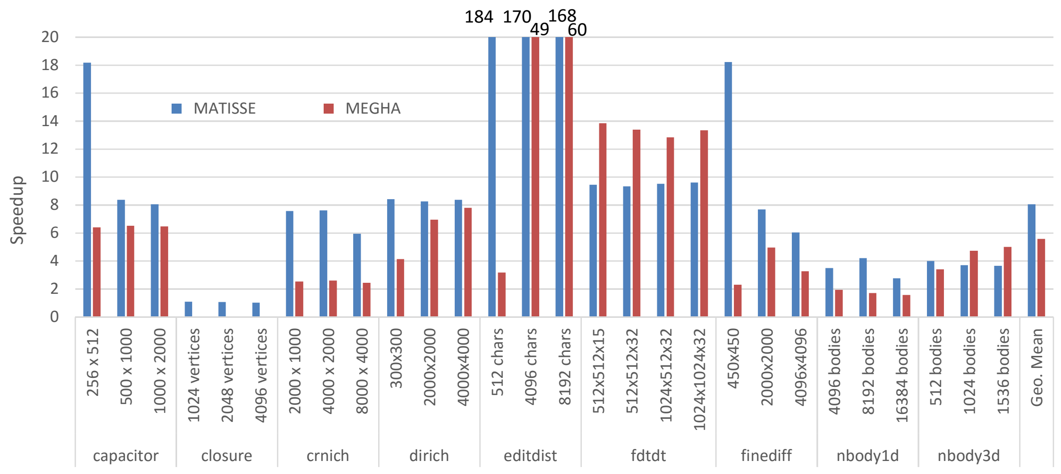

Figure 17. Speedups of the performance of the C code generated by MATISSE and MEGHA [17] vs. MATLAB execution when considering the execution of the benchmarks in a desktop PC.

The *closure* example is dominated by matrix multiplication and as both MATLAB and MEGHA use BLAS to perform matrix multiplications, the performance is roughly the same (Prasad et al. [17] did not present results for *closure*, but mentioned in the paper that the execution time was the same as MATLAB). As another use case, we ran the same benchmarks using a modern PC (with an AMD A10-7850K, 4.10 GHz, from 2014). There were differences in speedups of more than 160% on some of the individual benchmarks with larger datasets (both ways, either faster or slower), showing that the speedups can change significantly between CPUs. However, we obtained a similar geometric mean speedup (8.0×).

5.4 MATISSE Techniques

We studied the impact of four techniques used in MATISSE. Table 4 presents the techniques and identifies in which benchmarks there was a larger impact. We present the results for each technique in the next subsections.

Matrix Views Results

The benchmark *copy_vs_ptr* is a synthetic benchmark used to test the potential of using views with pointers instead of views by copy (see Section 4.2). The use of views with pointers allowed MATISSE to generate C code with performance improvements of 1.8×, 1.4× and 1.5×, for the PC, ODROID, and BeagleBoard, respectively. We believe the higher improvements on the PC are due to a higher overhead for copying data in this system. After enabling views with pointers, the PC also achieved better performance for *closure* (1.6×), *dotprod* (1.3×), *fdtd* (1.3×) and *nbody3d* (1.3×). However, the improvements were negligible for the other benchmarks.

Third-Party Libraries and Target-Specific Parameters

The use of optimized third party libraries is essential to generate competitive code, e.g., in terms of performance. For instance, the benchmark *closure* spends most of its execution time in matrix multiplication operations, and can run orders of magnitude slower if instead of BLAS we use a simple, straightforward, version of matrix multiplication (e.g., 94× times slower on PC when *N* is 1,024).

Table 3. Execution time for MATLAB programs (running with the Mathworks MATLAB 2014a environment) and for the binary code obtained by compiling the C programs generated by MATISSE with GNU gcc –O3, all executed in a desktop PC.

Benchmark	**Size**	**Execution Time (s)**	
		MATLAB code	**C code generated by MATISSE**
capacitor	256×512	16.6	0.92
	500×1,000	66.1	7.9
	1,000×2,000	270	33.6
closure	1,024 vertices	0.8	0.68
	2,048 vertices	5.6	5.3
	4,096 vertices	44.3	43.4
crnich	2,000×1,000	0.6	0.08
	4,000×2,000	2.4	0.31
	8,000×4,000	9.4	1.6
dirich	300×300	1.3	0.16
	2,000×2,000	59.6	7.2
	4,000×4,000	238	28.4
editdist	512 chars	0.6	0.0033
	4,096 chars	37.8	0.22
	8,192 chars	150	0.90
fdtdt	512×512×15	189	20.0
	512×512×32	404	43.3
	1,024×512×32	810	85.0
	1,024×1,024×32	1629	170
finediff	450×450	0.01	0.00066
	2,000×2,000	0.2	0.030
	4,096×4096	1.0	0.16
nbody1d	4,096 bodies	19.7	5.6
	8,192 bodies	80.2	19.1
	16,384 bodies	249	90.2
nbody3d	512 bodies	42.8	10.7
	1,024 bodies	171	46.1
	1,536 bodies	379	103

Table 4. Techniques whose impact was studied, and the benchmarks mostly impacted by them.

Technique	**Benchmarks**
Matrix Views	copy_vs_ptr, closure, dotprod, fdtd, nbody3d
3rd Party Libraries (BLAS)	closure, nbody3d
Element-Wise Mapping	capacitor, cfd3_v2, hypotenuse, nbody1d
Intrinsics	capacitor, crnich, hypotenuse, nbody1d

Figure 18 shows the impact on speedup of the BLAS version of the *closure* benchmark over a simple implementation for different input sizes. Using the library for certain input sizes can cause slowdowns. However, the threshold to decide if it is more advantageous to use BLAS changes with the target platform: on the PC, the use of BLAS provides clear speedups when the input size is greater or equal than 16, but on the BeagleBoard, for an input size of 32 it is still slower to use BLAS. The speedup using BLAS in the PC decreases from 2.6× to 1.3× for values of N between 16 and 32. When we observe the behavior of both implementations separately, the behavior of the simple algorithm is more predictable, while the behavior using BLAS is more dependent on the data sizes. BLAS is a very complex and highly optimized package, whose internals can change from version to version.

Another benchmark which heavily uses matrix multiplications is *nbody3*. For this benchmark, if we disable the threshold and only use BLAS (i.e., disable the use of the simple algorithm for small matrices), we have a 9% performance degradation for the tested input sizes as result.

Our approach allows the specification of strategies in LARA to guide the compiler about the use of a certain implementation. In addition, one can specify a LARA strategy that inserts code rules to decide at runtime about the implementation to use.

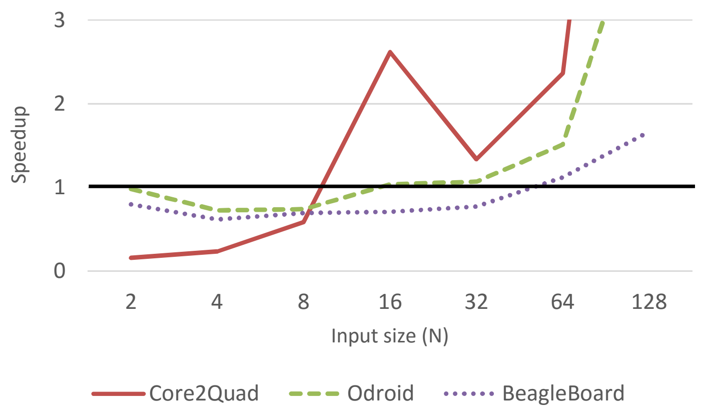

Figure 18. Speedup of *closure* using BLAS over the use of simple implementations for matrix multiplication according to data sizes.

Element-Wise Mapping Operations

If a program spends a significant amount of time using element-wise operations, transforming such operations to *for* loops can yield performance improvements (see Table 5). The benchmark *hypotenuse* (see Figure 13) spends most of its execution time doing element-wise operations, and can be used as an indicator of the potential improvement we can obtain from such transformation. In the case of the PC and ODROID, we obtained a speedup of 3.7× and 3.2×, respectively. In the BeagleBoard, the benchmark achieved a significantly lower gain, 1.6×. Inspecting the assembly of this benchmark, we concluded that the higher speedups in the PC and ODROID come from automatic vectorization of the generated *for* loops, using special instructions available in the processors (SSE in the case of PC, and NEON in the ARM Cortex-A15 used in the Exynos 5410 of the ODROID board). As the BeagleBoard processor does not have such instructions, the speedup is lower, and comes mostly from the lower overhead in function calls and avoiding the creation of temporary variables.

Table 5. Speedups obtained when applying the element-wise to for transformation.

Benchmark	**Speedups**		
	PC	**ODROID**	**BeagleBoard**
capacitor	1.3	1.0	1.0
cfd3_v2	1.3	1.0	1.0
hypotenuse	3.7	3.2	1.6
nbody1d	1.8	1.1	1.2

MATISSE Intrinsics

On the PC, the use of the MATISSE *intrinsics* has noticeable impact in the performance, while in the other platforms there are virtually no difference (see Table 6). We believe this happens because current *intrinsics* are related to avoiding setting newly allocated memory (*matisse_new_array_from_matrix*). As in the PC the cost of memory operations is higher when compared with the other tested platforms, this kind of optimizations are more noticeable.

Table 6. Speedups obtained when using MATISSE *intrinsics*.

Benchmark	**Speedups**		
	PC	**ODROID**	**BeagleBoard**
capacitor	1.5	1.04	1.04
crnich	1.3	1.02	1.02
hypotenuse	1.3	1.14	1.01
nbody1d	1.3	1.06	1.02

5.5 Overview

The evaluation and results presented in this section show the status of the MATISSE compiler framework, and the advantages of the implemented techniques. We have shown that having the flexibility to specify the type of any variable (as opposed to only the inputs) can have significant impact in some cases, and can aid the type inference system. Each of the tested techniques provide benefits, that when used together allow MATISSE to generate efficient code while providing flexibility in the code generation.

6. RELATED WORK

Given the importance of the MATLAB runtime support there have been research efforts to improve the execution of JIT MATLAB compilers. A recent example is the McVM [18] which performs function specialization based on the runtime knowledge of the types in function calls. Yet, the need to avoid a MATLAB runtime system in most embedded systems has led to efforts on how to best translate MATLAB programs into equivalent C code, see, e.g., [19][20].

The translation of MATLAB to other programming languages is not recent. DeRose and Padua proposed the FALCON environment [21][22][23][24][25] to translate MATLAB to FORTRAN90 code. They leverage on an aggressive use of static and type inference for base types (doubles and complex) as well as shape (or rank) for the matrices. Other researchers have explored the reuse of storage for array variables across a MATLAB code thus reducing the memory footprint of the corresponding C reference code [19]. Joisha et al. [16][26] and Olmos et al. [27] present additional type and shape inference techniques. Researchers have also relied on a mix of type inference approaches and user's provided information. For instance, [28][29] use annotations to specify data types and shapes and simple type inference analysis and target VHDL code specification for hardware synthesis onto FPGAs.

The Sable Lab at McGill University have done extensive work around the MATLAB language [30], including an AOP-extension [31], a virtual machine [32] and code generators for several languages, such as Fortran95 [33] and X10 [34]. They present several analysis and transformations relevant to our work, some of them, e.g., kind analysis [35], and language simplification [36], we plan to integrate in MATISSE in a near future.

Sharma et al. [37] explore the parallel capabilities MATLAB currently offers out-of-the-box (e.g., *parfor*). They focus on using MATLAB on a cluster environment and on the use of MPI, and do not target embedded systems.

We specifically note that our approach is focused on embedded implementations of the MATLAB programs. In this context, an efficient translation to an implementation language (mainly C) is needed. One of the possibilities is to consider a subset of MATLAB allowing feasible and efficient static compilation. One example using such a subset is the embedded MATLAB (a subset of MATLAB) to C code translation previously existent in the MathWorks Real-Time Workshop [38], which supported embed annotations with MATLAB code to achieve C code implementations for embedded systems [39]. The MATLAB Coder [2] is a more recent Mathworks solution for MATLAB to C compilation. They support a large subset of MATLAB, and allow the code to be customized through directives and options. However, the customization is fairly coarse-grained, and finer control (e.g., addressing types, shapes, and target) is supported by direct modification of the MATLAB source code. When dealing with the myriad of target architectures and toolchains used in/for embedded systems, this approach presents a low level of flexibility, e.g., as the style of the C code generator might need to be tuned to the toolchain as is the case when targeting C to hardware compilers.

Languages similar to MATLAB have been proposed, being *Scilab* [11] and *Octave* [10] possibly the most used ones. Related to our goals is *Sci2C* [40], which translates Scilab to C and focus entirely on embedded systems. It is completely dependent on annotations embedded in the *Scilab* code to specify data sizes and precisions. MATISSE distinguishes from *Sci2C* as it is able to generate C code even without annotations, and specialization of the generated C code can be achieved without modifying the original code (MATLAB, in our case). Furthermore, *Sci2C* requires that the size of all arrays is fixed and statically known, while MATISSE allows dynamically allocated arrays whose sizes are unknown at compile time.

Recently and orthogonally to this work, there have been approaches to translate MATLAB (and its main clones) to languages suitable to multicore and/or GPU architectures. For instance, MiX10 [41] compiles MATLAB to IBM's X10 language, designed for high performance computing. Their work takes advantage of MATLAB's *parfor* and X10 parallel features. MEGHA [17] is a compiler which processes MATLAB/Octave scripts and generates CUDA and C/C++ code. MEGHA uses heuristics to decide which portions of the code should be executed on the CPU and which should be offloaded to the GPU. Chun-Yu Shei et al. [42] presents another MATLAB/Octave to CUDA compiler, which generates both C++ and CUDA code. However, unlike MEGHA, portions of the resulting code remain in MATLAB. In the context of the MATISSE compiler, Reis et al. [43][44] present the first steps of an OpenCL generator for MATISSE which relies on OpenACC-based directives. Although their focus is on OpenCL generation aware of the target architecture (e.g., CPU, GPU, FPGA), in particular in embedded computing domains, the current version of the OpenCL code generator is not aware of the target architecture.

The ALMA project [20] developed a toolchain for compiling annotated Scilab [11] code to MPSoC (Multi-Processor System-on-Chip) architectures. They consider the exploitation of coarse- (task-level) and fine-grained (instruction-level) parallelism. A frontend of the toolchain compiles subsets of Scilab extended with a preprocessing language allowing declaration of variables, specification of static types, and maximum sizes of vectors and matrices, and with annotations to support the extraction of parallelism. The result is annotated C code which is then compiled by other tools to the target architectures. The Scilab compiler includes high-level, platform independent, optimizations (e.g., function inlining/outlining, algebraic optimizations, operator strength reduction, loop invariant code motion, constant propagation and folding, copy propagation, and common subexpression elimination). However, the approach postpones to C compilers some of the optimizations specific to the target architecture, such as loop parallelization and data layout optimizations. Our approach differs from the one proposed by ALMA as it relies on a separation of concerns provided by LARA specifications and on the multi-target exploration at higher-levels and our intention is to generate code (e.g., C code) as much as possible tuned to the target architectures and following toolchains.

In this work we described a mechanism for conveying information about types and shape/rank similar in spirit with the notion of Aspects [6]. Previous work has proposed aspect-oriented extensions to MATLAB [45] and an aspect-oriented code transformation language for MATLAB [46]. Other authors have explored aspect-oriented approaches for MATLAB [31][47], being AspectMatlab++ one of the most relevant approaches. AspectMatlab++ is an extension to AspectMatlab and provides an AOP approach for MATLAB in the context of scientific programming. It provides support to patterns such as loop bodies, function calls, patterns embedded in comments, types, dimension, etc. AspectMatlab++ provides MATLAB code woven with the aspects and it is not focused on assisting the generation of other target languages.

To the best of our knowledge the AOP extensions to MATLAB do not consider the use of aspects to specify complementary information that can be used by compilers to produce more efficient implementations. Some of the benefits of using AOP extensions to MATLAB have been already exposed by a number of software metrics [48]. However, the inclusion of the versatility of LARA [6][7][8] on providing strategies for MATLAB tools adds further suitable dimensions to the use of AOP approaches in the context of the MATLAB programming language, for instance, on the application of source-to-source transformations as the ones presented by Birkbeck et al. [49].

MATISSE differs from the previous approaches in several aspects. First, it focuses on exploring several C implementations for multitarget solutions, unlike other approaches which have a larger focus on the MATLAB language itself and how to generate a single correct/efficient implementation [30]. In the multitarget context, it is very important to have a high degree of customization of the generated code. As an example, MATISSE allows users to define the type of any variable in the code, unlike Mathworks' MATLAB Coder, which only allows to set the types of the function inputs [2]. MATISSE main goal is to customize the output code without modifying the source code, hence the use of AOP concepts to control the customization. Furthermore, since MATISSE mainly targets embedded systems, it considers that the generated code will be run in environments where a MATLAB-compatible runtime might not be available, unlike hybrid approaches [42]. Orthogonally to the work presented in this paper, the MATISSE compiler is being extended with OpenCL generation [50], also leveraging the complementary information that can be provided by LARA aspects.

7. CONCLUSION

This paper presented MATISSE, a multitarget/multitoolchain framework for compiling MATLAB to lower level programming languages such as C. MATISSE relies on LARA aspects for specifying data types, shapes, and code instrumentation and specialization. The compiler includes a type and shape inference stage and is able to generate MATLAB and C code. We presented the general flow of the tool and described a number of transformations and optimizations performed by the compiler. The experiments reveal the flexibility of our approach and promising performance results, e.g., achieving a geometric mean speedup of 13× over execution in MATLAB when considering 31 benchmarks. Furthermore, transformations such as detecting element-wise operations and operations such as matrix multiplications highlight the efficiency of using a higher abstraction level such as the one provided by MATLAB as a starting point for specialized implementations in C.

Ongoing work is focused on further optimizations for the C generator and on the development of additional targets for the C code, such as C tailored for hardware generation through High-Level Synthesis tools, and on enhancing the OpenCL backend for multi-CPUs, GPUs and FPGAs.

ACKNOWLEDGMENTS

This work was partially funded by project REAL TIME NORTE-07-0124-FEDER-000062 which was financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, *Fundação para a Ciência e a Tecnologia* (FCT). The authors would like to acknowledge Luís Reis and Tiago Carvalho regarding their help on topics addressing the code generation and the LARA language, respectively.

REFERENCES

[1] *MATLAB – the Language of Technical Computing*. The MathWorks, Inc., http://www.mathworks.com/.

[2] *MATLAB Coder: Generate C and C++ code from MATLAB code*. The MathWorks, Inc, 2012.

[3] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. Cardoso, and P. C. Diniz, "The MATISSE MATLAB compiler," in *11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 602–608.

[4] J. Bispo, L. Reis, and J. M. Cardoso, "Techniques for efficient MATLAB-to-C compilation," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2015, pp. 7–12.

[5] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ: aspect-oriented programming in Java*. NY, USA: John Wiley & Sons, 2003.

[6] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, Potsdam, Germany, 2012, pp. 179–190.

[7] J. M. P. Cardoso, P. Diniz, J. G. Coutinho, and Z. Petrov, *Compilation and Synthesis for Embedded Reconfigurable Systems*. Springer, 2013.

[8] J. M. Cardoso, J. G. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves, "Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach," *Software: Practice and Experience*, 2014.

[9] *MATISSE*. http://specs.fe.up.pt/tools/matisse/.

[10] *Octave*. http://www.gnu.org/software/octave/.

[11] *Scilab*. Scilab Enterprises S.A.S, http://www.scilab-enterprises.com/.

[12] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, p. 154, Dec. 1996.

[13] M. Lam, R. Sethi, J. D. Ullman, and A. Aho, *Compilers: principles, techniques, and tools*. Addison-Wesley, 2006.

[14] *Symja - Java Computer Algebra Library*. .

[15] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, and others, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[16] P. G. Joisha and P. Banerjee, "A translator system for the MATLAB language," *Software: Practice and Experience*, vol. 37, no. 5, pp. 535–578, 2007.

[17] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," in *ACM Sigplan Notices*, 2011, vol. 46, pp. 152–163.

[18] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge, "Optimizing MATLAB through just-in-time specialization," in *Compiler Construction*, 2010, pp. 46–65.

[19] P. G. Joisha and P. Banerjee, "Static array storage optimization in MATLAB," in *ACM SIGPLAN Notices*, San Diego, CA, USA, 2003, vol. 38, pp. 258–268.

[20] J. Becker, T. Stripf, O. Oey, M. Huebner, S. Derrien, D. Menard, O. Sentieys, G. Rauwerda, K. Sunesen, N. Kavvadias, and others, "From Scilab to high performance embedded multicore systems: the ALMA approach," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, 2012, pp. 114–121.

[21] G. Almasi and D. A. Padua, *MaJIC: A MATLAB just-in-time compiler*. Springer, 2001.

[22] L. De Rose and D. Padua, "Techniques for the translation of MATLAB programs into Fortran 90," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 2, pp. 286–323, Mar. 1999.

[23] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua, "FALCON: A MATLAB interactive restructuring compiler," in *Languages and Compilers for Parallel Computing*, Springer, 1996, pp. 269–288.

[24] L. De Rose and D. Padua, "A MATLAB to Fortran 90 translator and its effectiveness," in *Proceedings of the 10th international conference on Supercomputing*, 1996, pp. 309–316.

[25] L. A. De Rose, "Compiler techniques for MATLAB programs," Citeseer, 1996.

[26] P. G. Joisha and P. Banerjee, "An algebraic array shape inference system for MATLAB®," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 5, pp. 848–907, 2006.

[27] K. Olmos and E. Visser, "Turning dynamic typing into static typing by program specialization in a compiler front-end for Octave," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, 2003, pp. 141–150.

[28] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Parallelization of Matlab applications for a multi-FPGA system," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, Rohnert Park, CA, USA, 2001, pp. 1–9.

[29] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, "Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design," in *Field-Programmable Custom Computing Machines, 2003. 11th Annual IEEE Symposium on*, Napa, CA, USA, 2003, pp. 263–264.

[30] A. Casey, J. Li, J. Doherty, M. Chevalier-Boisvert, T. Aslam, A. Dubrau, N. Lameed, A. Aslam, R. Garg, S. Radpour, and others, "McLab: an extensible compiler toolkit for MATLAB and related languages," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 2010, pp. 114–117.

[31] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, "AspectMatlab: An aspect-oriented scientific programming language," in *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, NY, USA, 2010, pp. 181–192.

[32] N. A. Lameed, "Dynamic compiler optimization techniques for MATLAB," McGill University, 2013.

[33] X. Li, "Mc2For: a MATLAB to FORTRAN 95 compiler," McGill University, 2014.

[34] V. Kumar and L. Hendren, "MiX10: Compiling MATLAB to X10 for high performance," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 617–636.

[35] J. Doherty, L. Hendren, and S. Radpour, “Kind analysis for MATLAB,” in *ACM SIGPLAN Notices*, 2011, vol. 46, pp. 99–118.

[36] A. W. Dubrau and L. J. Hendren, *Taming Matlab*, vol. 47. ACM, 2012.

[37] G. Sharma and J. Martin, “MATLAB®: a language for parallel computing,” *International Journal of Parallel Programming*, vol. 37, no. 1, pp. 3–36, 2009.

[38] *Real-Time Workshop: Generate C code from Simulink models and MATLAB code*. .

[39] “Best Practices for a MATLAB to C Workflow Using Real-Time Workshop.” Nov-2009.

[40] *Scilab 2 C - Translate Scilab code into C code*. .

[41] V. Kumar and L. Hendren, “First steps to compiling Matlab to X10,” in *Proceedings of the third ACM SIGPLAN X10 Workshop*, New York, NY, USA, 2013, pp. 2–11.

[42] C.-Y. Shei, A. Yoga, M. Ramesh, and A. Chauhan, “MATLAB parallelization through scalarization,” in *15th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, 2011, pp. 44–53.

[43] L. Reis, J. Bispo, and J. M. Cardoso, “Towards a multitarget C and OpenCL generation from MATLAB,” presented at the Compilers for Parallel Computing (CPC’15), Imperial College, London, UK, 2015.

[44] J. Bispo, L. Reis, and J. M. Cardoso, “Multi-target C code generation from MATLAB,” in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, UK, 2014, p. 95.

[45] J. M. Cardoso, J. Fernandes, and M. Monteiro, “Adding aspect-oriented features to MATLAB,” in *workshop on Software Engineering Properties of Languages and Aspect Technologies*, Bonn, Germany, 2006.

[46] J. M. Cardoso, P. Diniz, M. P. Monteiro, J. M. Fernandes, and J. Saraiva, “A domain-specific aspect language for transforming MATLAB programs,” in *Domain-Specific Aspect Language Workshop, part of AOSD*, Rennes & Saint Malo, France, 2010, pp. 15–19.

[47] A. Bodzay and L. Hendren, “AspectMatlab++: annotations, types, and aspects for scientists,” in *Proceedings of the 14th International Conference on Modularity*, 2015, pp. 41–54.

[48] P. Martins, P. Lopes, J. P. Fernandes, J. Saraiva, and J. M. Cardoso, “Program and aspect metrics for MATLAB,” in *Computational Science and Its Applications–ICCSA 2012*, Springer-Verlag, 2012, pp. 217–233.

[49] N. Birkbeck, J. Levesque, and J. N. Amaral, “A dimension abstraction approach to vectorization in Matlab,” in *Code Generation and Optimization, 2007. CGO’07. International Symposium on*, 2007, pp. 115–130.

[50] João Bispo, Luís Reis, and João M. P. Cardoso, “C and OpenCL generation from MATLAB,” in *30th ACM Symposium on Applied Computing (SAC 2015)*, Salamanca, Spain, 2015.